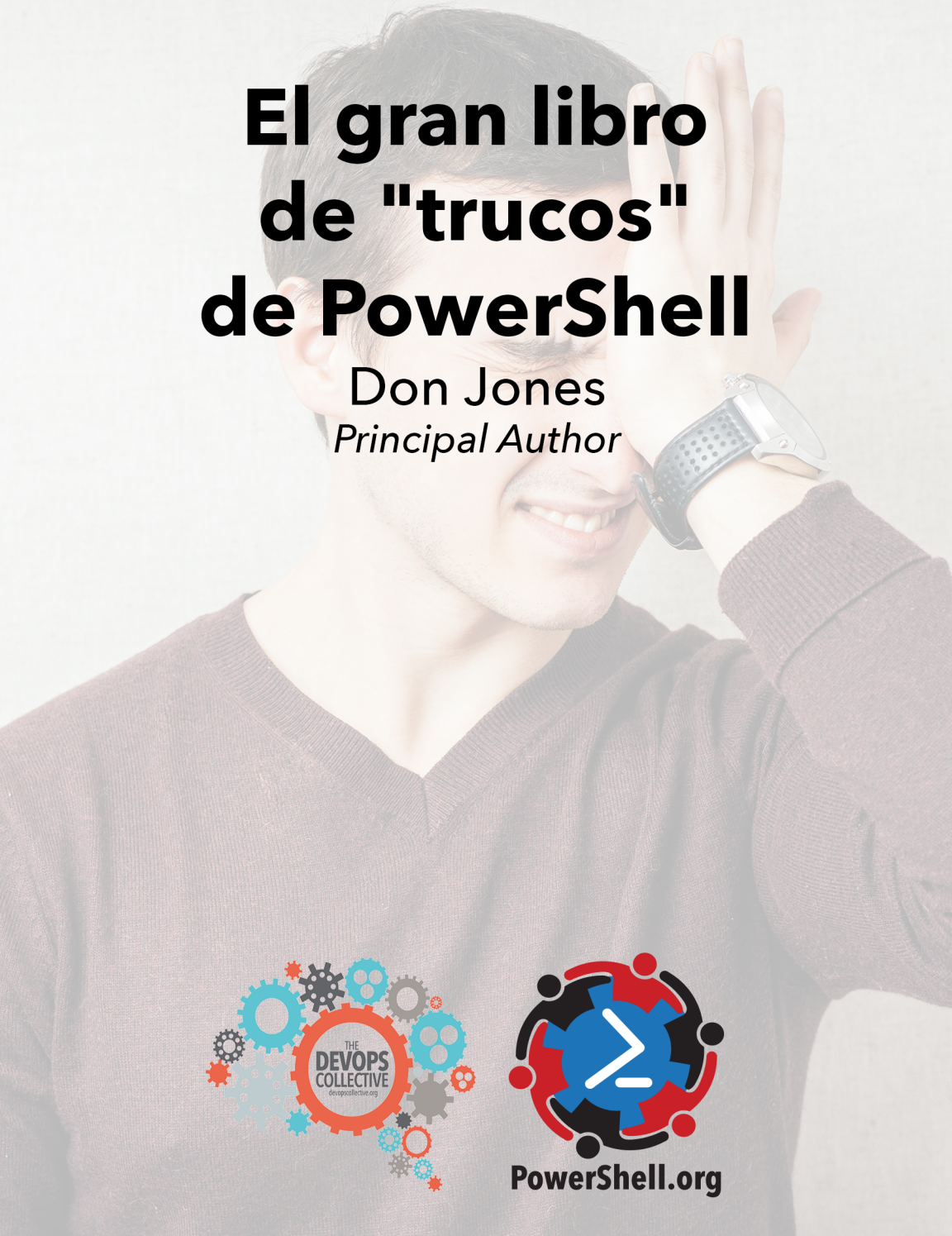
Es

El gran libro de "trucos" de PowerShell

Don Jones
Principal Author



THE DEVOPS COLLECTIVE
devopscollective.org

PowerShell.org

The Big Book of PowerShell Gotchas (Spanish)

The DevOps Collective, Inc.

Este libro está a la venta en
http://leanpub.com/big-book-of-powershell-gotchas-spanish

Esta versión se publicó en 2018-10-28

Este es un libro de Leanpub. Leanpub anima a los autores y publicadoras con el proceso de publicación. Lean Publishing es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener feedback del lector hasta conseguir tener el libro adecuado.

También por The DevOps Collective, Inc.

Creating HTML Reports in Windows PowerShell

A Unix Person's Guide to PowerShell

The Big Book of PowerShell Error Handling

DevOps: The Ops Perspective

Ditch Excel: Making Historical and Trend Reports in PowerShell

Secrets of PowerShell Remoting

The Big Book of PowerShell Gotchas

The Monad Manifesto, Annotated

Why PowerShell?

Windows PowerShell Networking Guide

The PowerShell + DevOps Global Summit Manual for Summiteers

Why PowerShell? (Spanish)

Secrets of PowerShell Remoting (Spanish)

DevOps: The Ops Perspective (Spanish)

The Monad Manifesto: Annotated (Spanish)

Creating HTML Reports in PowerShell (Spanish)

The Big Book of PowerShell Error Handling (Spanish)

DevOps: WTF?

PowerShell.org: History of a Community

Índice general

El gran libro de “trucos” de PowerShell

Por Don Jones (mayormente)

PowerShell está lleno de “trucos” - pequeñas cosas que a veces se interponen en su camino y son difíciles de averiguar por su cuenta. Este breve libro está destinado a ayudarle a resolverlos y evitarlos.

Obteniendo el código El módulo EnhancedHTML2 mencionado en este libro puede encontrarse en PowerShell Gallery[1]. Esa página incluye las instrucciones de descarga. PowerShellGet es necesario y se puede obtener de PowerShellGallery.com

¿Ha sido útil este libro? El (los) autor (es) le pide (n) que haga una donación deducible de impuestos (en los EE.UU., consulte sus leyes si vive en otro lugar) de cualquier cantidad a The DevOps Collective[2] para apoyar su trabajo.

** Revise las actualizaciones! ** Nuestros ebooks se actualizan a menudo con contenido nuevo y corregido. Los hacemos disponibles de dos maneras:

[1] https://www.powershellgallery.com/packages/EnhancedHTML2
[2] https://devopscollective.org/donate

- Nuestra rama principal GitHub organization[3], con un repositorio para cada libro. Visite https://github.com/devops-collective-inc/
- En LeanPub[4], donde se pueden descargar como PDF, EPUB, o MOBI (login requerido), y “comprar” los libros haciendo una donación a DevOps. También puede elegir recibir notificaciones de actualizaciones. Visite https://leanpub.com/u/devopscollective

GitBook y LeanPub generan la salida del formato PDF ligeramente diferente, por lo que puede elegir el que prefiera. LeanPub también le puede notificar cada vez que liberamos alguna actualización. Nuestro repositorio de GitHub es el principal; los repositorios en otros sitios suelen ser sólo espejos utilizados para el proceso de publicación. LeanPub siempre contiene la más reciente “publicación liberada” de cualquier libro.

[3] https://github.com/devops-collective-inc
[4] https://leanpub.com/u/devopscollective

Formato a la derecha

Todo el mundo se encuentra con esto. Comienza escribiendo un comando verdaderamente impresionante.

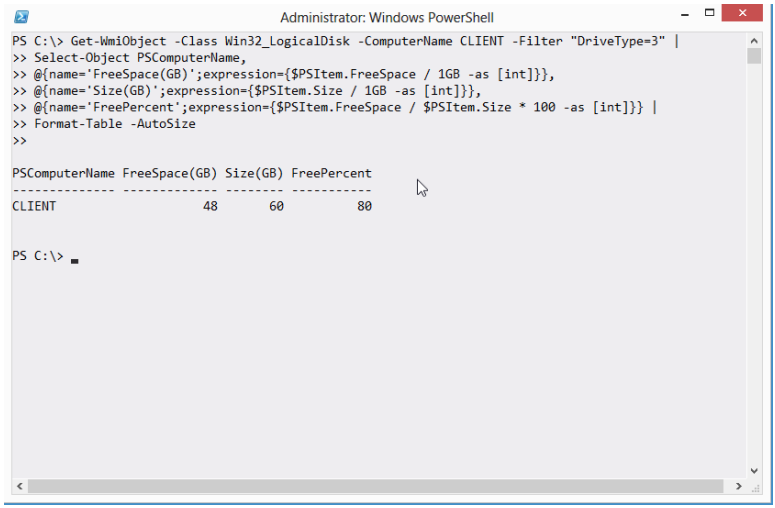


image005.png

Y luego piensa, “Wow, esto quedaría muy bien en un archivo HTML.”

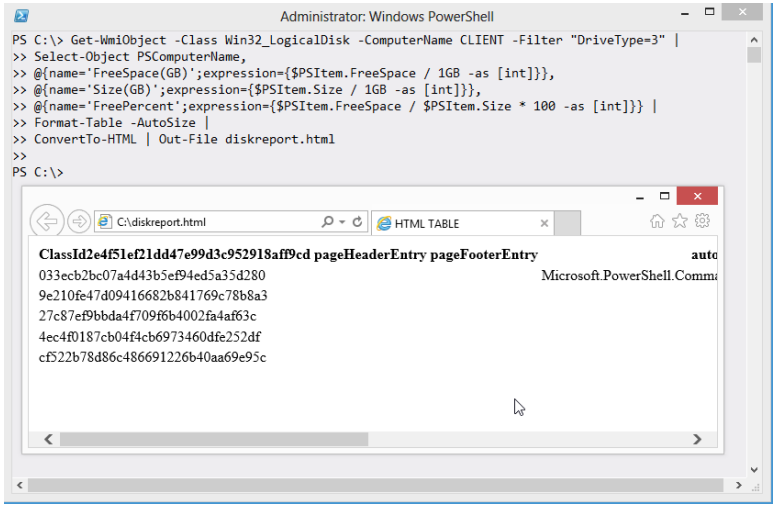


image007.png

¿¡¿¡¿Un momento QUÉ?!?!?

Sucede todo el tiempo. Si desea una manera fácil de recordar lo que no se debe hacer, es esto: nunca canalice (enviar al pipeline) la salida de un comando de formato. Esa no es toda la verdad (llegaremos a toda la verdad en un momento), pero si sólo quiere una respuesta rápida, eso es todo. En la comunidad, lo llamamos la regla del "formato a la derecha", porque tiene que ver con mover su comando Format al extremo derecho de la línea de comandos. Es decir, el comando Format va al final, y nada más viene después de él.

La razón es que todos los comandos de formato producen códigos de salida internos especiales, que están destinados a generar una visualización en pantalla. Canalizar esos códigos (enviarlos al pipeline) a cualquier otro comando - ConvertTo-HTML, Export-CSV, lo que sea – solo hará que se obtenga una salida ilegible.

De hecho, hay algunos comandos que pueden venir después de un comando de formato en la canalización (pipeline):

1. Out-Default. Técnicamente siempre está al final de la canalización (pipeline), aunque sea "invisible". Es el encargado de redirigir la salida al Host. Por eso es que vemos siempre la salida en pantalla.
2. Out-Host también entiende la salida de los comandos de formato, porque Out-Host es la forma en la que los códigos de formato obtienen la información de lo que se debe mostrar en pantalla.
3. Out-Printer también entiende los códigos de formato de salida y además, construye una página impresa que se vería exactamente como la salida normal en pantalla.
4. Out-File, como Out-Printer, redirecciona la salida en pantalla, pero esta vez a un archivo de texto en disco.
5. Out-String utiliza los códigos de formato de salida y produce una cadena simple que contiene el texto que de otro modo habría aparecido en pantalla.

Aparte de esas excepciones -y de ellas, usualmente sólo se utiliza Out-File- no se puede canalizar la salida de un comando Format a otro comando si desea obtener cualquier cosa que parezca útil.

¿Dónde está el comando <SuNombreAqui>? He instalado la última versión de PowerShell y no puedo encontrarlo!

Una cosa difícil es entender que hay un cierto número de comandos que *vienen con PowerShell* y otros que simplemente no vienen.

Cada nueva versión de PowerShell incluye al menos algunos nuevos comandos. Por ejemplo, Start-Job apareció por primera vez en PowerShell v2, mientras que Invoke-AdWorkflow fue introducido en PowerShell v3.

Lo que confunde a la gente es que una nueva versión de PowerShell también tiende a corresponder con una nueva versión del sistema operativo Windows. Y el Sistema Operativo viene con cientos de comandos. Por ejemplo, puede haber utilizado Get-SmbShare por primera vez en Windows Server 2012, que incluye PowerShell v3. Pero Get-SmbShare es parte del sistema operativo, no parte de PowerShell. Es decir, no tendrá Get-SmbShare en cada sistema que tenga PowerShell v3 o posterior, porque el comando no es una "característica de PowerShell ", es una "característica de Windows".

Así que… ¿De dónde se obtienen los comandos?

Normalmente, los comandos son parte de algún producto. ¿Necesita los comandos de Exchange Server? Instale las herramientas de administración de Exchange Server. ¿Necesita los comandos de Windows Server 2012? Instale el kit de herramientas de adminis-

tración remota del servidor (RSAT), que contiene las herramientas de administración del servidor.

PowerShell.exe no es PowerShell

Es importante entender que Windows PowerShell, detrás de escenas es en realidad un motor. Usted como un simple ser humano no puede interactuar directamente con PowerShell.

En su lugar, necesita una aplicación Host. Un Host incrusta el motor internamente, y luego le da una manera de interactuar con él. Por ejemplo, powershell.exe es una aplicación Host. Se construye alrededor de la misma consola de consola de Windows (conhost.exe) a través de la antigua shell de línea de comandos cmd.exe, pero incrustando el motor PowerShell. Se escriben los comandos y el Host los envía al motor para su ejecución. El Host también es responsable de mostrar cualquier resultado. En este caso, en pantalla.

¿Por qué es importante esta distinción?

Porque diferentes Hosts pueden comportarse de diferentes maneras. Por ejemplo, el PowerShell ISE se comporta un poco diferente que el Host de la consola, y ambos se comportan de manera muy diferente de Active Directory Administration Center, otro host de PowerShell.

Acumulando la salida en una función

Esto es un truco un poco "avanzado", pero es uno en que muchos desarrolladores experimentados caen. Aquí hay un ejemplo, sólo para demostrar el punto (no es funcional, ya que el comando utilizado es ficticio):

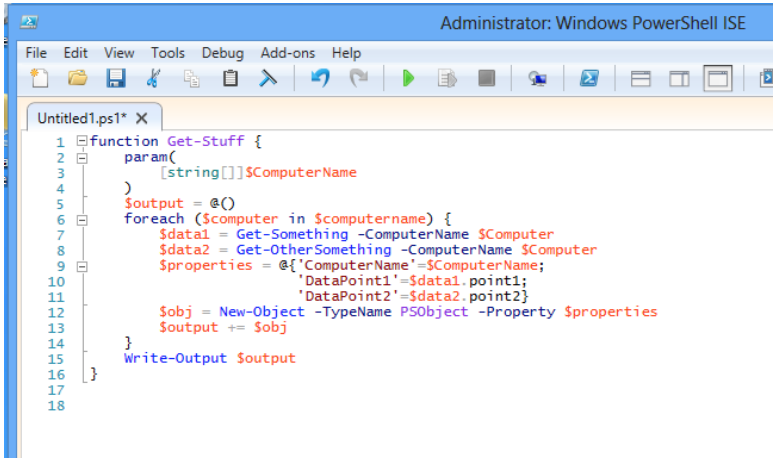


image009.png

El problema es que la función puede generar múltiples objetos de salida, y el programador está acumulándolos en la variable de $output. Esto significa que esta función no emitirá nada hasta que su ejecución esté completamente terminada. No es así como los comandos PowerShell (y las funciones) suelen estar diseñados para funcionar.

Los comandos de PowerShell *normalmente* deben enviar cada objeto a la canalización (pipeline), uno a la vez, apenas esos objetos

estén listos. Esto permite que la canalización (pipeline) acumule la salida, e inmediatamente la pase a lo largo de la siguiente función o comando en la canalización (pipeline). Así funcionan los comandos en PowerShell. Ahora, siempre hay excepciones. Sort-Object, por ejemplo, *tiene* que acumular su salida, porque en realidad no puede ordenar nada hasta que tenga *todos* los elementos. Es por esto que se le llama un comando _blocking, porque "bloquea" la canalización (pipeline) completamente hasta que se produce su salida. Pero eso es una excepción.

Normalmente esto es muy fácil de solucionar, simplemente enviando a la canalización (pipeline) directamente en lugar de acumular:

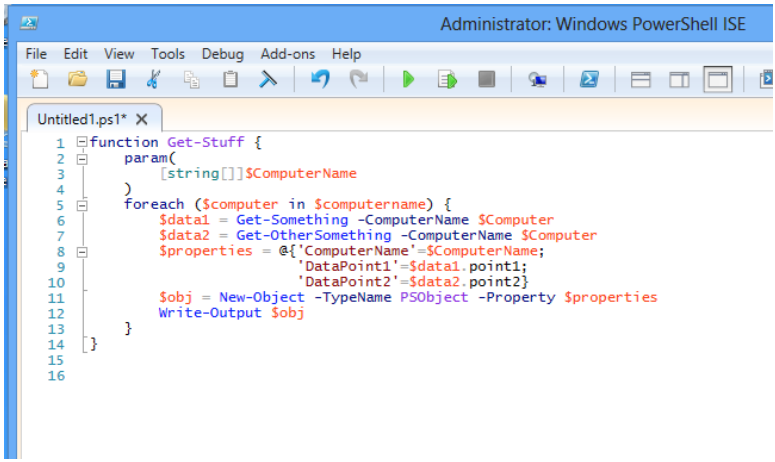


image011.png

ForEach vs ForEach vs ForEach

PowerShell tiene comandos de aspecto similar que pueden confundir, especialmente a los recién llegados. Por ejemplo, usted tiene dos entidades ForEach:

- El Cmdlet ForEach-Object, que tiene un alias ForEach (también tiene el alias %). Está destinado a funcionar en la canalización (pipeline), y utiliza un parámetro de proceso que acepta un ScriptBlock.
- La declaración ForEach. Tiene una sintaxis específica, no está destinado a ser utilizado en la canalización (pipeline) y no tiene un alias.

Aquí están los tres en acción, en un ejemplo muy simple:

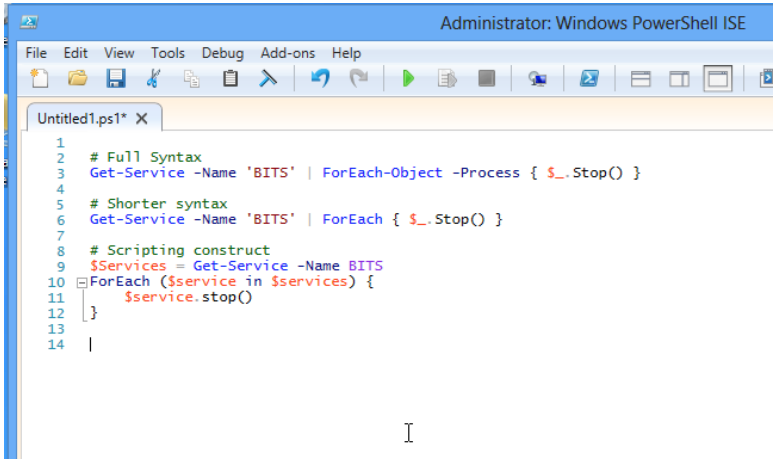


image013.png

La gran diferencia es que, en la canalización (pipeline), ForEach-Object procesa un objeto a la vez. *Esto significa que puede ser más lento,* ya que ese ScriptBlock debe interpretarse en cada iteración. También tiende a usar menos memoria, ya que los objetos fluyen por la canalización (pipeline) uno a la vez y no tienen que ser agrupados en una variable primero.

La declaración ForEach tiende a ser más rápida, pero a menudo tiene más sobrecarga de memoria, ya que tiene que iterar sobre toda la colección de objetos a la vez, en lugar de transmitir objetos de uno en uno cada vez.

Ambos usan una sintaxis parecida, pero hay diferencias. Es importante entender que no son los mismos comandos, y que se ejecutan de manera diferente. Es confuso porque "ForEach" es tanto un alias como una declaración de Scripting. El Shell determina qué se está utilizando mirando el contexto en el que lo está utilizando.

Finalización con Tab

Es triste y sorprendente ver cómo pocas personas confían en la terminación con la tecla Tab, tanto en el PowerShell ISE como en la ventana de la consola.

- Cuando se completa con Tab, nunca digitara comandos o nombres de parámetros incorrectos
- Para muchos valores de parámetros que son listas estáticas o listas de fácil consulta, la terminación con Tab (especialmente en v3 y posteriores) puede completar los valores de dichos parámetros
- La terminación con Tab hace que los nombres de Cmdlet largos sean mucho más fáciles de escribir, sin necesidad de conocer un alias de difícil o tener que memorizar el nombre completo.

Mantenga el hábito de usar la terminación con Tab todo el tiempo que sea posible. Le garantizará cometer menos errores.

-Contains y -Like son diferentes

Si tuviera un centavo por cada vez que he visto esto:

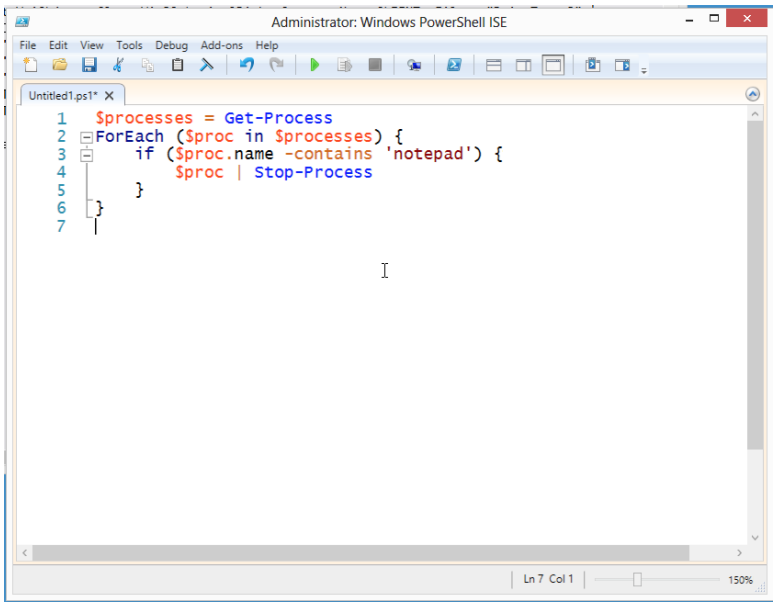


image015.png

Entiendo cómo sucede. El operador -Contains parece que debería comprobar si el nombre de un proceso contiene las letras “notepad”. Pero eso no es lo que hace.

El enfoque correcto es utilizar el operador -Like, que de hecho hace una comparación de cadena con comodines:

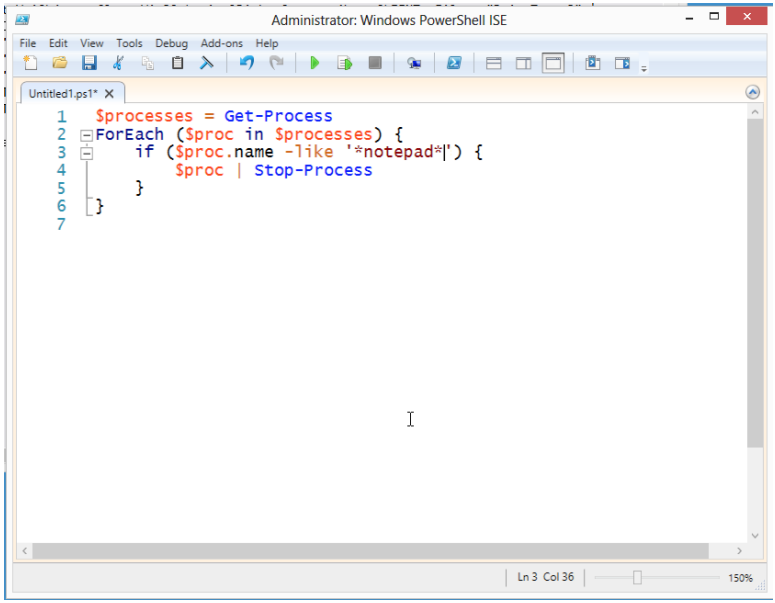


image017.png

Voy a dejar pasar la idea de que la respuesta realmente correcta es ejecutar Stop-Process -Name *notepad *, porque estaba apuntando a un ejemplo simple aquí. Pero ... no piense demasiado. A veces un script en un bucle foreach no es el mejor enfoque.

Así que de todos modos, ¿qué hacen -Contains (y su amigo, -NotContains) en realidad? Son similares a los operadores -In y -NotIn introducidos en PowerShell v3. Estos operadores pueden causar un poco de confusión. Lo que hacen es comprobar si una colección de objetos contiene un único objeto dado. Por ejemplo:

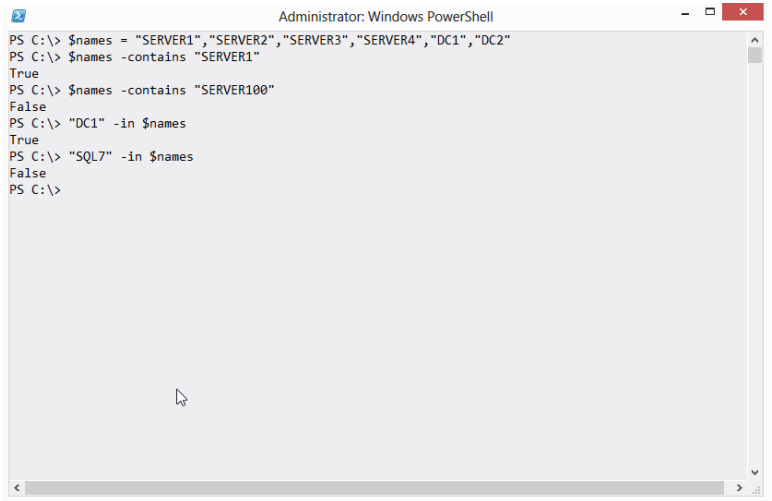


image019.png

De hecho, este ejemplo es probablemente la mejor manera de verlo funcionar. El truco es que, cuando se utiliza un objeto complejo en lugar de un valor simple (como lo hice en ese ejemplo), -Contains e -In buscan en todas las propiedades del objeto para encontrar una coincidencia. Si piensa en algo como un proceso, ellos siempre estarán cambiando. De cuando en cuando, la CPU y la memoria de un proceso, pueden ser diferentes.

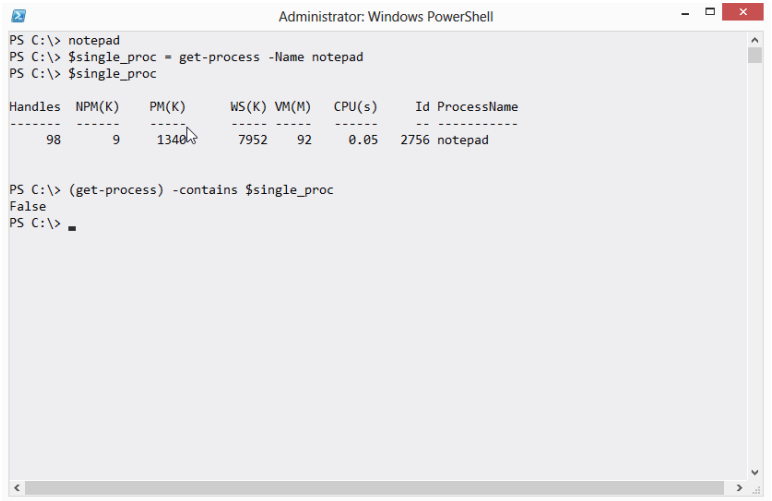


image021.png

En este ejemplo, he iniciado el bloc de notas. He puesto su objeto de proceso en $single_proc, y se puede ver que he verificado que estaba allí. Pero cuando ejecuto Get-Process para comprobar si la colección contenía mi Notepad, el resultado fue falso. Eso es porque el objeto en $single_proc está desactualizado. Notepad está en ejecución, pero ahora se ve diferente, por lo que -Contains no puede encontrarlo.

Los operadores -in y -contains son mejores con valores simples, o con objetos que no tienen valores de propiedad que cambian constantemente. Pero no son operadores de coincidencia de cadenas de caracteres comodines. Use-like (o -notlike) para eso.

No puede tener lo que no se tiene

¿Puede ver lo que está mal?

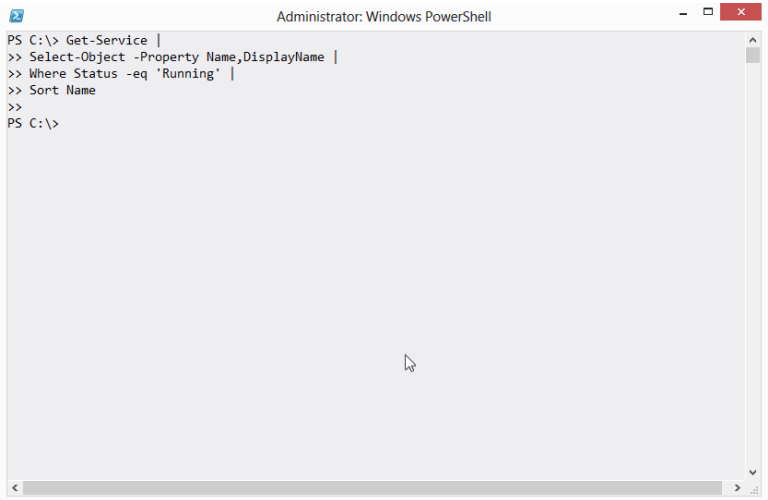


image023.png

Quiero decir, estoy bastante seguro de que tengo algunos servicios en ejecución. Se supone que algo se debía mostrar.

Si no ve la respuesta de inmediato - o no la ve - es un buen momento para hablar acerca de cómo solucionar problemas con algunas líneas de comandos. Para empezar, como siempre digo, retrocediendo un paso. Elimine el último comando, y vea si eso hace alguna diferencia.

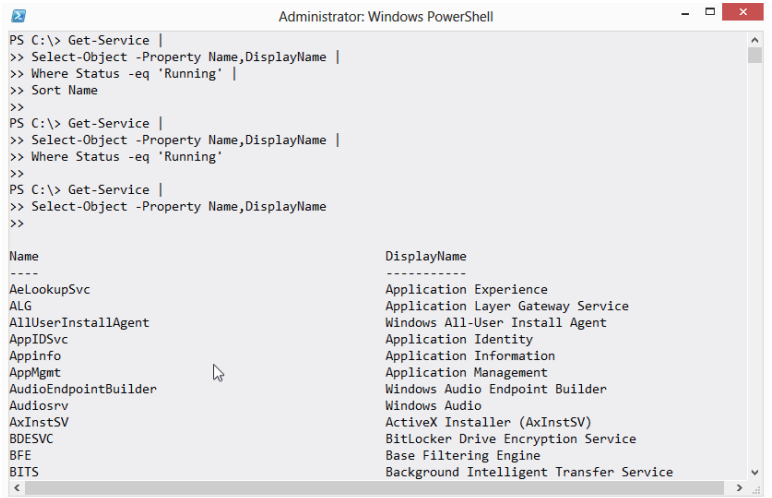


image025.png

En este caso, quité el comando Sort-Object (Sort) y no ocurrió nada diferente, así que eso no era la causa del problema. A continuación, eliminé el comando Where-Object (Where, en la sintaxis corta de v3), y ah-ha! Apareció la salida. Así que el comando Where-Object está "rompiendo" algo. Vamos a revisar lo que funcionó y a canalizarlo a Get-Member, para ver qué hay en la canalización (pipeline) después de ejecutar Select-Object.

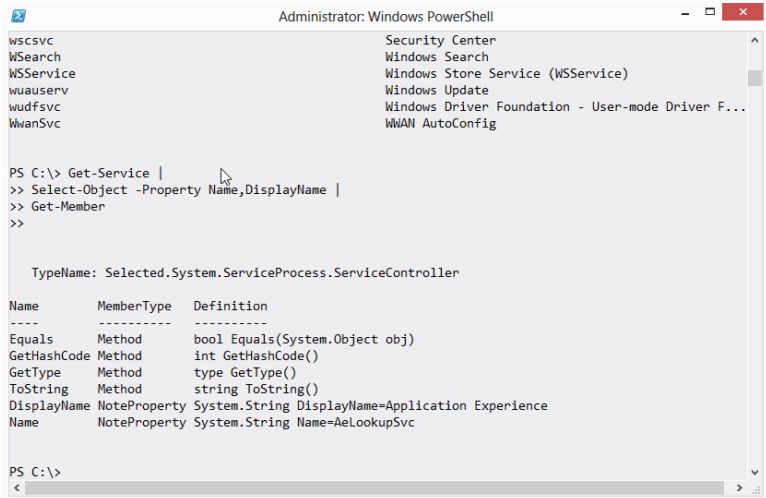


image027.png

OK, tengo un objeto que tiene una propiedad DisplayName y una propiedad Name.

Y mi comando Where-Object estaba comprobando la propiedad Status. ¿Ve una propiedad Status? No, no se ve. Mi error es que quité la propiedad Status cuando no la incluí en la lista de salida del comando Select-Object. Así que el objeto no tenía nada contra qué trabajar y no devolvió nada.

(Sí, sería mejor si PowerShell lanzara un error - "hey, pidio filtrar la propiedad Status, y no hay una!" - pero eso no así cómo funciona).

Moraleja de la historia: prestar atención a lo que está en la canalización (pipeline). No se puede trabajar con algo que no se tiene. No siempre obtendrá un mensaje de error útil, por lo que a veces tendrá que escarbar y averiguarlo de otra manera - como retrocediendo un paso.

-Filter y la diversidad de valores

Esta es una de las cosas más difíciles de acostumbrarse en PowerShell:

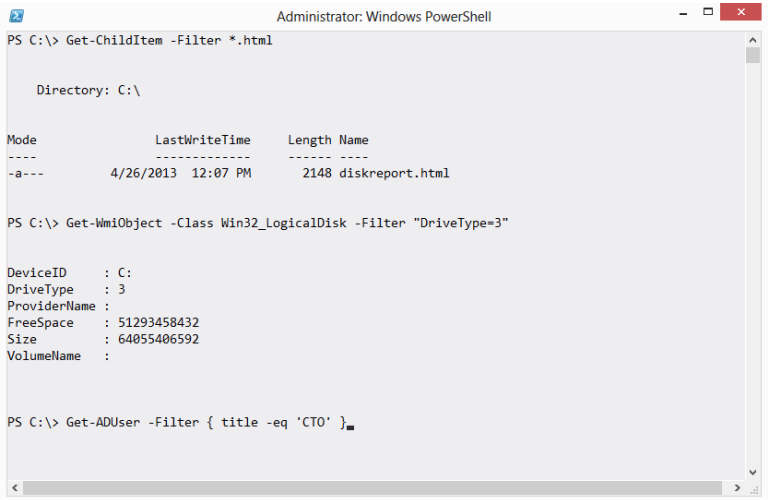


image029.png

Aquí vemos tres comandos, cada uno usando un parámetro -Filter. Cada uno de esos filtros es diferente.

1. Con Get-ChildItem, -Filter acepta los comodines del sistema de archivos como *.
2. Con Get-WmiObject, -Filter requiere una cadena, y utiliza operadores de estilo de programación (como = para la igualdad).
3. Con Get-ADUser, -Filter requiere un bloque de script, y acepta operadores de comparación de estilo PowerShell (como -eq

para la igualdad)

Esto es lo que pienso cuando se utiliza un parámetro –Filter. PowerShell no está procesando el filtrado. En su lugar, los criterios de filtrado se están transmitiendo a la tecnología subyacente, como el sistema de archivos, o WMI, o al directorio activo. Es esta tecnología la que decide qué tipo de criterios de filtro se van a aceptar. PowerShell es sólo el intermediario. Así que es mejor leer cuidadosamente la ayuda, y tal vez buscar ejemplos, para entender cómo la tecnología subyacente necesita que especifique su filtro.

Sí, sería bueno si PowerShell tradujera para usted (que es realmente lo que hace Get-ADUser - el comando traduce eso en un filtro de LDAP tras bambalinas). Pero, por lo general, no lo hace.

No todo produce una salida

Veo esto a menudo:

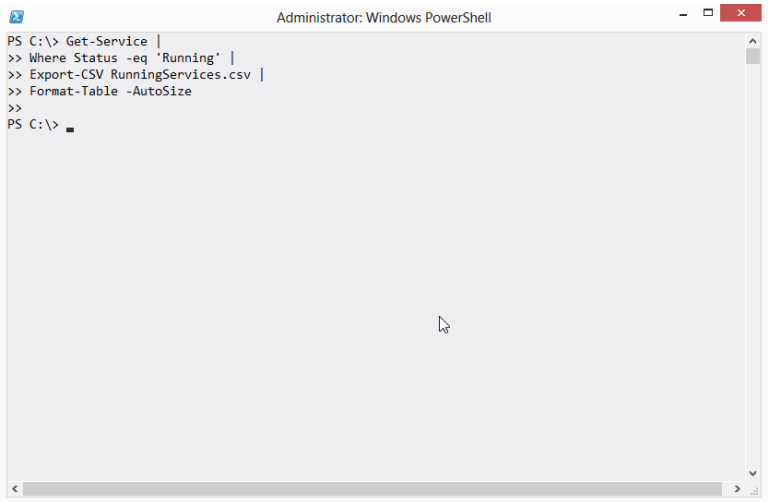


image031.png

Si esperaba algo en la pantalla en términos de salida, estará decepcionado. El truco aquí es hacer un seguimiento de lo que cada comando produce como salida, y es allí donde hay un posible punto de confusión.

En el mundo de PowerShell, la salida es lo que aparecería en la pantalla si ejecutamos el comando y no lo canalizamos (enviar al pipeline) a nada más. Sí, Export-CSV hace algo - crea un archivo en disco - pero en el mundo de PowerShell ese archivo no se ve en pantalla. Export-CSV no produce ninguna salida, hablando de algo que aparecería en la pantalla. Por ejemplo:

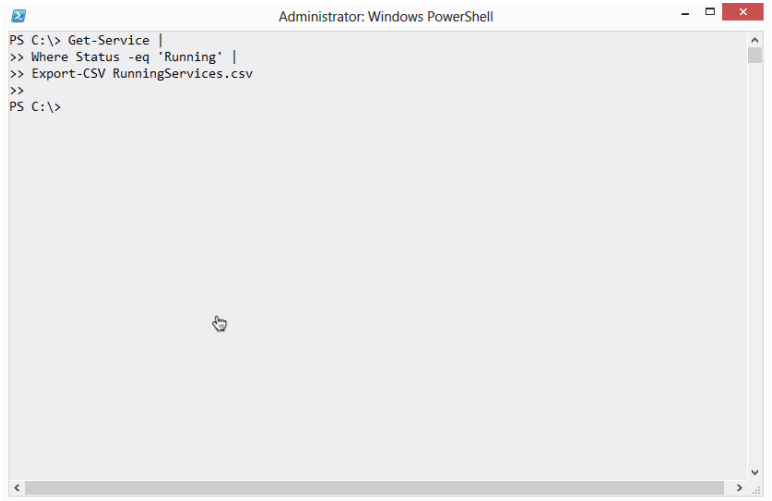


image033.png

¿Lo ve? nada. Ya que no hay nada en la pantalla, no hay nada en la canalización (pipeline). No puede canalizar Export-CSV a otro comando, porque no hay nada que canalizar.

Algunos comandos pueden incluir un parámetro -PassThru. Cuando lo tienen y se utiliza, harán lo que hagan normalmente, pero también pasarán sus objetos de entrada a través de la canalización (pipeline), para que luego se puedan canalizar a otra cosa. Export-CSV no es uno de esos comandos, - nunca produce una salida, por lo que nunca tendrá sentido para canalizarlo a otra cosa.

Una página HTML a la vez, por favor

Esto me vuelve loco:

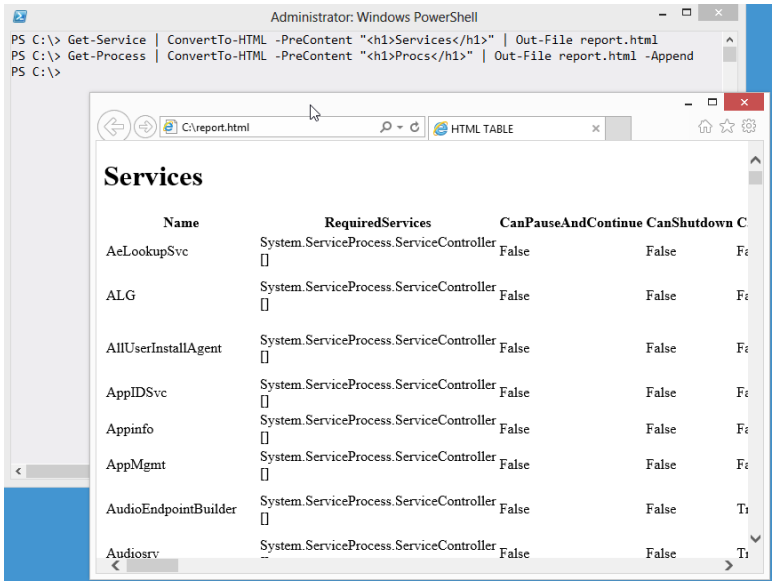


image037.png

Lo que está pasando es que alguien ejecutó dos comandos, canalizando la salida de cada uno a ConvertTo-HTML, y esencialmente combinando ambas páginas HTML en un solo archivo. Lo que me realmente me vuelve loco es que Internet Explorer está bien con esa tontería.

Los archivos HTML pueden empezar con una etiqueta de nivel superior, pero si se echa un vistazo a ese archivo verá que contiene dos:

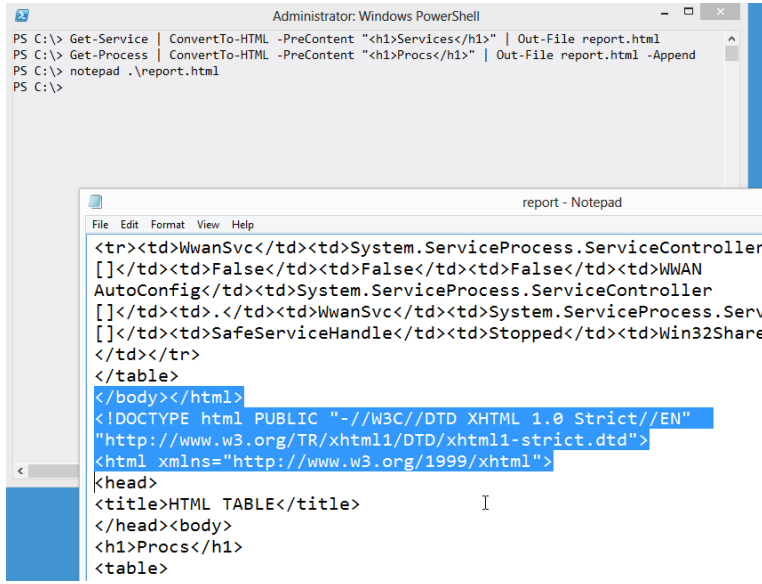


image039.png

He resaltado las líneas que terminan una página HTML y comienzan la siguiente. Esto es técnicamente un archivo HTML malformado. Algunos navegadores Web lo admiten (unos si, otros no), difícil de analizar si alguna vez necesita para manipular el contenido mediante programación, y … bueno, es esta mal. Es como el incesto o algo así. Inaceptable.

Si necesita combinar varios elementos en un único archivo HTML, utilice el parámetro -Fragment de ConvertTo-HTML. Produzca sólo una parte del HTML o varias porciones de ese tipo y luego combínelas en una sola página completa. Ahhh bien. Todo el proceso al respecto de la creación de informes HTML en PowerShell lo encuentra en nuestro otro libro electrónico gratuito que viene con este.

{

Horrible} (Puntuación)][Sangriento] {Horrible} (Puntuación)

Esto no un “truco” pero vale la pena revisarlo para que no resulte confuso. Las tuercas de PowerShell con la puntuación.

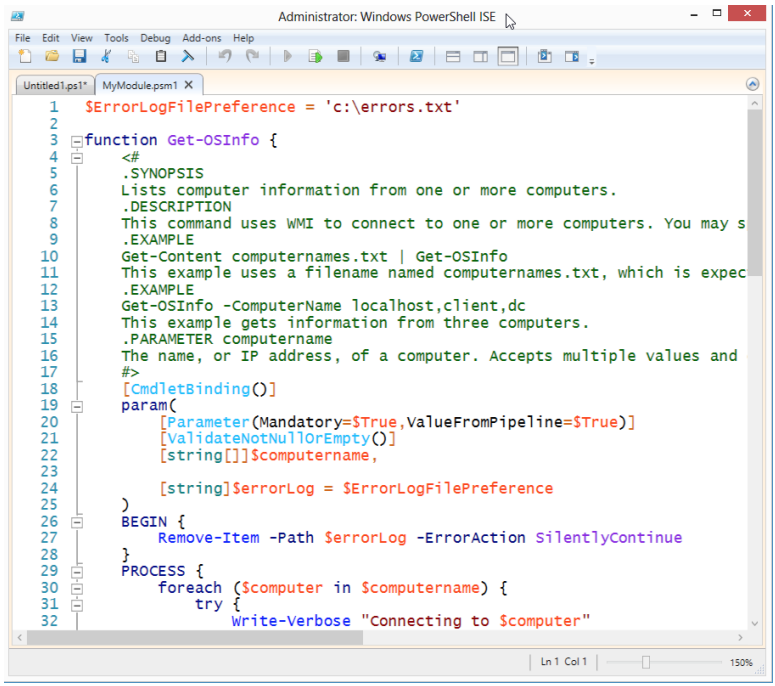


image041.png

(Paréntesis) se utilizan para encerrar expresiones, como la expresión foreach() y en ciertos casos para resaltar alguna sintaxis declarativa. Por ejemplo el bloque param() y en el atributo [parameter()].

[Corchetes cuadrados] se utilizan alrededor de algunos atributos, como en [CmdletBinding()], y alrededor de tipos de datos co-

mo [string]. También se utilizan para indicar arrays - como en [string[]]. Pueden aparecer en otros lugares.

{Corchetes} casi siempre contienen código ejecutable, como en el bloque try{}, el bloque begin{} y la función en sí. También se utilizan para expresar literales de tablas hash (como @{}).

Si el teclado tuviera algunos botones más, PowerShell no habría tenido que tener todos estos usos "incorporados" de caracteres de puntuación. Pero lo hace. En este punto, son casi una parte del "coste de entrada" del Shell, por lo que tendrá que acostumbrarse a ellos.

No+Concatene+Strings

Realmente me disgusta la concatenación de cadenas. Es como obligar a alguien a acurrucarse con alguien que ni siquiera conocen. Grosero.

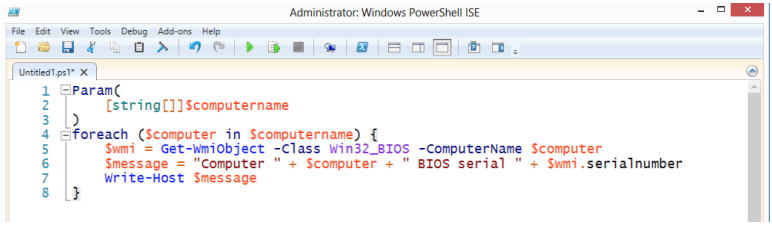


image043.png

Y completamente innecesario, cuando se utilizan comillas dobles.

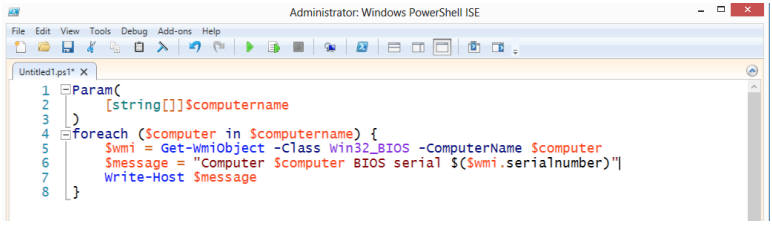


image045.png

Mismo efecto al final. Usando comillas dobles, PowerShell buscará el carácter $. Cuando lo encuentre:

1. Si el siguiente carácter es { entonces PowerShell llevará todo a la concordancia } como un nombre de variable, y reemplazará todo con el contenido de esa variable. Por ejemplo, poner $ {mi variable} dentro de comillas dobles reemplazará con el contenido de $ {mi variable}.

2. Si el siguiente carácter es un (entonces PowerShell llevará todo a la coincidencia) y lo ejecutara como si fuera código. Por lo que, ejecuté $wmi.serialnumber para acceder a la propiedad serialnumber del objeto que se encontraba en la variable $wmi.
3. De lo contrario, PowerShell tomará todos los caracteres que sean legales para un nombre de variable, hasta el primer carácter de nombre de variable ilegal, y lo reemplazará con esa variable. Así es como funciona $computer en mi ejemplo. El espacio después de la r no es legal para un nombre de variable, por lo que PowerShell sabe que el nombre de la variable se detiene en r.

Una cosa para resaltar aquí:

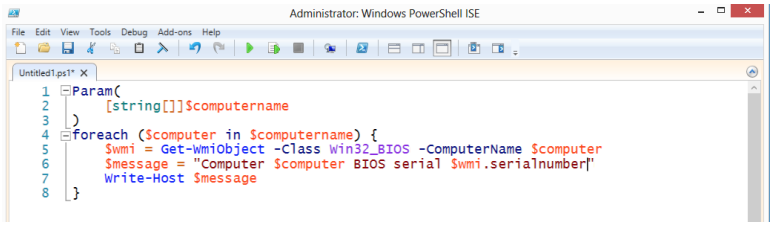


image047.png

Esto no funcionará como se esperaba. En la mayoría de los casos, $wmi será reemplazado por un nombre de tipo de objeto y .serialnumber seguirá allí. Eso ocurre porque . no es un nombre de variable legal, por lo que PowerShell deja de "observar" la variable con la letra i. Entonces, reemplaza $wmi con su contenido. Usted vio en el ejemplo anterior, el uso de $($wmi.serialnumber), que es una subexpresión y funciona. Los paréntesis hacen que su contenido se ejecute como código..

$ no forma parte del nombre de la variable

Gran “trampa”.

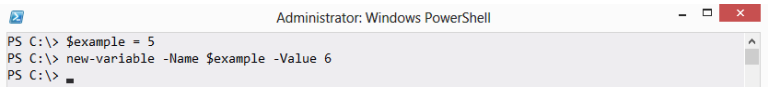


image049.png

¿Puede predecir el resultado?

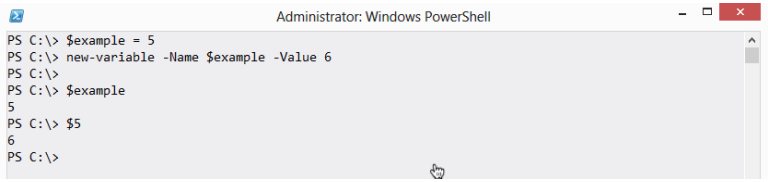


image051.png

Observe que el símbolo de moneda $ no forma parte del nombre de la variable. Si tiene una variable llamada example, que es como tener una caja con la palabra “ example” escrito al costado. Cuando se refiere a example significa que está hablando de la caja misma. Cuando se refiere a $ example significa que está haciendo referencia al contenido de la caja.

Así que en mi ejemplo, he utilizado $example = 5 para poner un 5 en la caja. Luego, cree una nueva variable. El nombre de la nueva variable fue $example – que como lleva el símbolo de moneda, en realidad hace referencia al valor de la variable $example que es 5. Así que lo que ocurrió en realidad fue que se creó una variable llamada 5, que tiene asignado el valor 6, a la que se puede hacer referencia por el nombre $5.

Difícil, ¿verdad? Lo es:

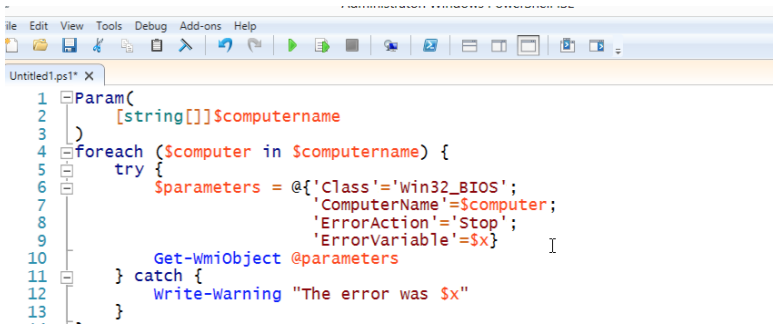


image053.png

En ese ejemplo, utilicé el parámetro -ErrorVariable para especificar una variable en la que se almacenaría cualquier error que se produzca. El problema es, he utilizado $x cuando debería haber utilizado solo x (sin el símbolo de moneda):

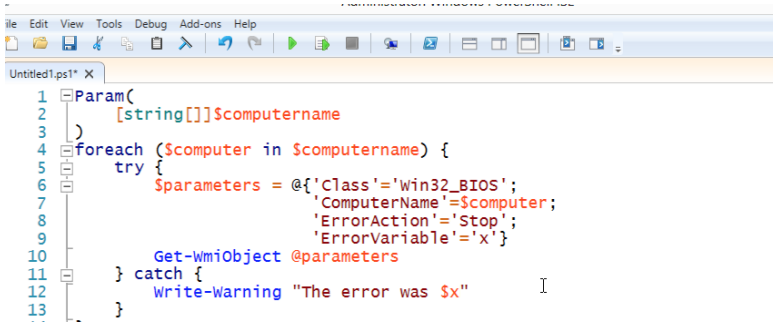


image055.png

Ahora la forma correcta. Utilizando solo x, a la que más tarde se puede acceder usando $x para obtener su contenido, es decir, cualquier error que haya sido almacenado allí.

Utilizar la canalización (pipeline), no una matriz

Un error muy común cometido por programadores tradicionales que recién llegan a PowerShell:

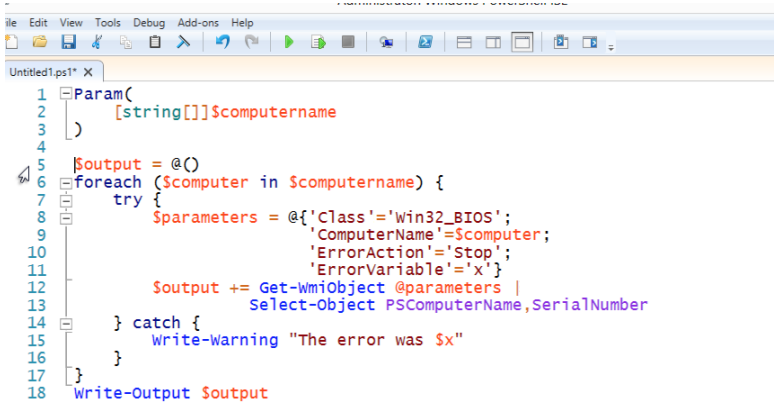


image057.png

Esta persona ha creado una matriz vacía en $output, y mientras recorre la lista de ordenadores y ejecuta consultas WMI, están agregando nuevos objetos de salida al contenido de la matriz. Finalmente, envía la matriz a la canalización (pipeline).

Mala práctica. Como se ve, esto obliga a PowerShell a esperar mientras se completa la ejecución del comando. Cualquier comando subsecuente en la canalización (pipeline) se sentará a esperar con los brazos cruzados. ¿Un mejor enfoque? Utilizar la canalización (pipeline), cuyo propósito es acumular la salida por usted - sin necesidad de que usted mismo la acumule en una matriz.

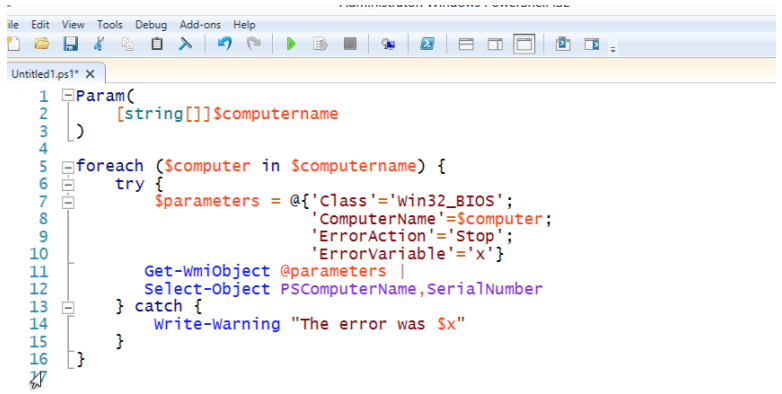


image059.png

Ahora, los comandos posteriores recibirán la salida, dejando que varios de esos comandos se ejecuten más o menos simultáneamente en la canalización (pipeline).

Backtick, Grave Accent, Escape

A menudo va a encontrarse con esto

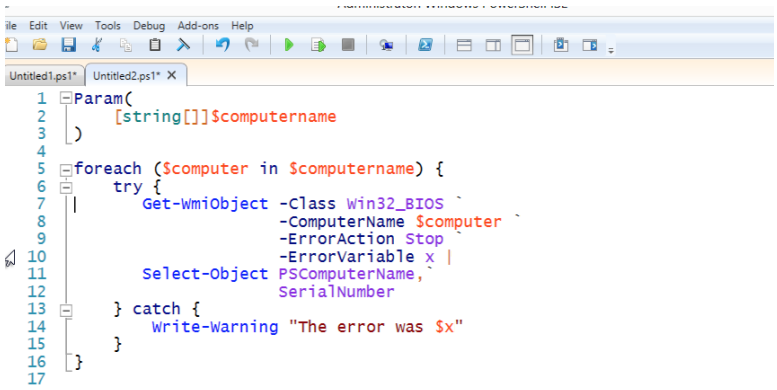


image061.png

No, no es un píxel muerto en el monitor o un trozo de tóner perdido en la página, es la marca de acento grave o backtick. ‘ Es el carácter de escape de PowerShell. En este ejemplo, está “escapando” el retorno de carro invisible al final de la línea, eliminando su propósito especial como final de línea lógica, simplemente haciendo que sea un retorno de carro literal.

No me gusta el backtick utilizado de esta manera.

Primero, es difícil de ver. Segundo, si se deja un espacio en blanco extra después de él, ya no estará escapando el retorno de carro, y el script se romperá:

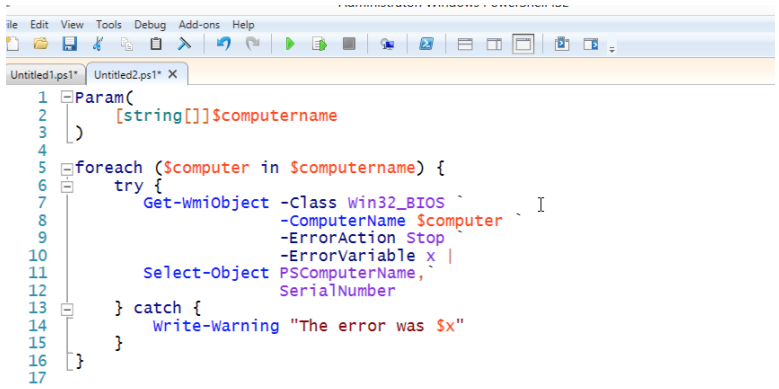


image063.png

Observe cuidadosamente el parámetro -computername - en este segundo ejemplo. Fíjese como se muestra un color incorrecto para un nombre de parámetro. Ocurre porque he añadido un espacio después del backtick en la línea anterior. IMPOSIBLE de rastrear.

Y el backtick es innecesario como carácter de continuación de línea. Permítanme explicar por qué:

PowerShell ya le permite agregar un "Enter" en ciertas situaciones. Usted solo tiene que aprender cuáles son esas situaciones, y luego tomar ventaja de ellas. Entiendo totalmente el deseo de tener código perfectamente formateado - predico sobre eso todo el tiempo - pero no tiene que confiar en un personaje como el backtick para obtener código bien formateado.

Sólo tiene que ser más listo.

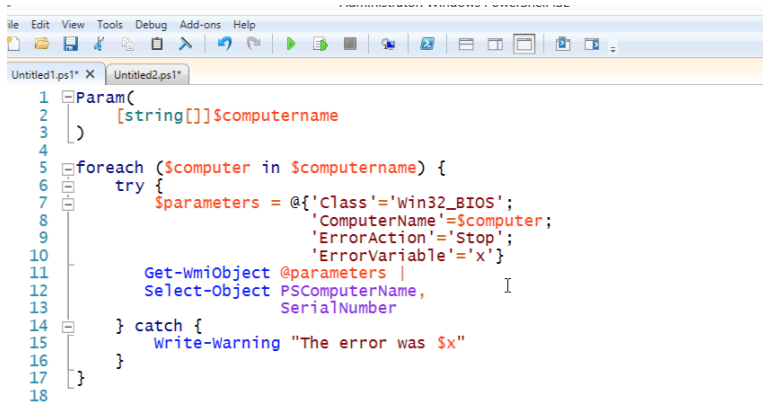


image065.png

Para empezar, he puesto mis comandos Get-WmiObject en una tabla hash, por lo que ahora puedo dar un formato agradable y bonito. Cada línea termina en un punto y coma, y PowerShell me permite romper la línea después de cada punto y coma. Incluso si agrego un espacio adicional o un Tab después del punto y coma, funcionará bien. Entonces hago "Splat" de esos parámetros al comando Get-WmiObject.

Después de Get-WmiObject, tengo un carácter Pipe, y PowerShell admite un "Enter" luego de un carácter Pipe.

Usted notará al final de Select-Object que se puede utilizar una coma también.

Así termino con un formato que parece al menos tan bueno, si no mejor, porque no tiene un backtick ‘ flotando por todas partes.

Una multitud no es un individuo

Un error muy común de novato:

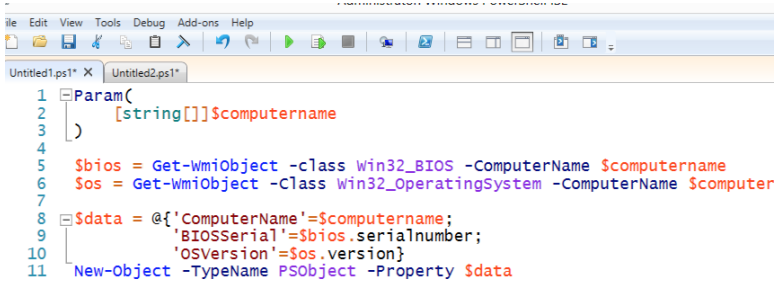


image067.png

Aquí, el problema es que se está tratando todo como si estuviera compuesto de un sólo un valor. Pero aqupi $computername puede contener varios nombres de equipo (eso es lo que significa ([string[]]), lo que significa que tanto $bios como $os podrían contener también varios elementos. El truco está en enumerar $computername para conseguir el resultado deseado:

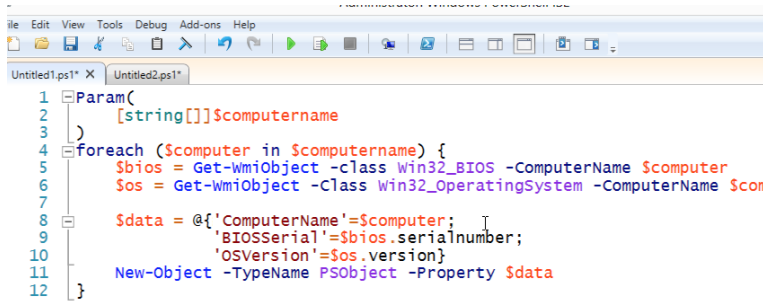


image069.png

Algunas veces también se encontrará con esto, incluso en situaciones sencillas. Por ejemplo:

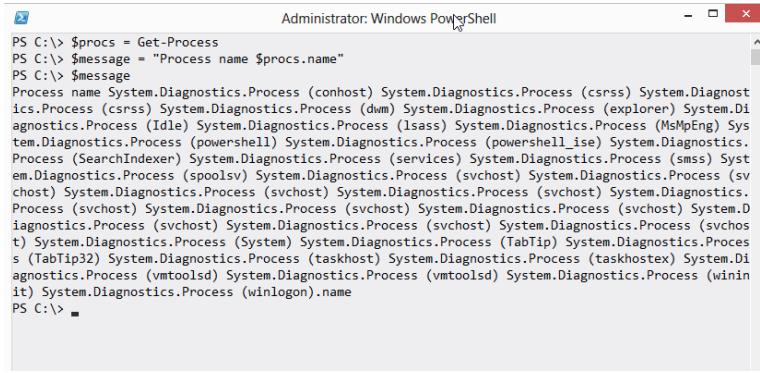


image071.png

PowerShell v2 no reaccionará tan bien, pero en PowerShell v3, la variable dentro de comillas dobles $procs es una variable que contiene varios objetos. PowerShell los enumera implícitamente, además de buscar una propiedad llamada name. Fíjese en ".name" al final de la cadena - PowerShell no hizo nada con eso.

Es probable que mejor desee enumerar así:

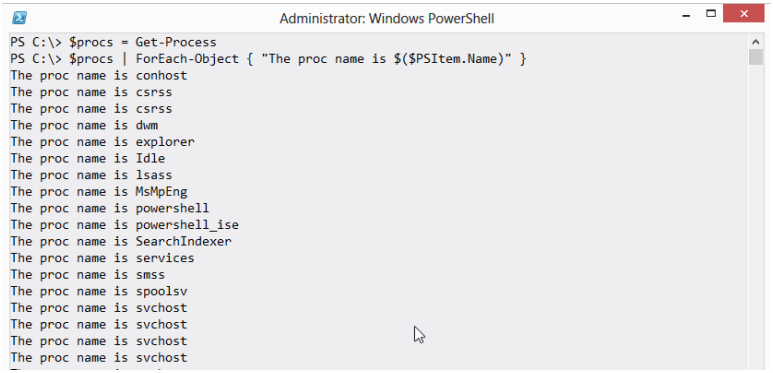


image073.png

Comandos de la vieja escuela

Siempre tenga en cuenta que mientras PowerShell tiene comandos llamados dir y cd, no son los viejos comandos de MS-DOS. Son simplemente alias o apodos, a comandos de PowerShell. Eso significa que tienen una sintaxis diferente.

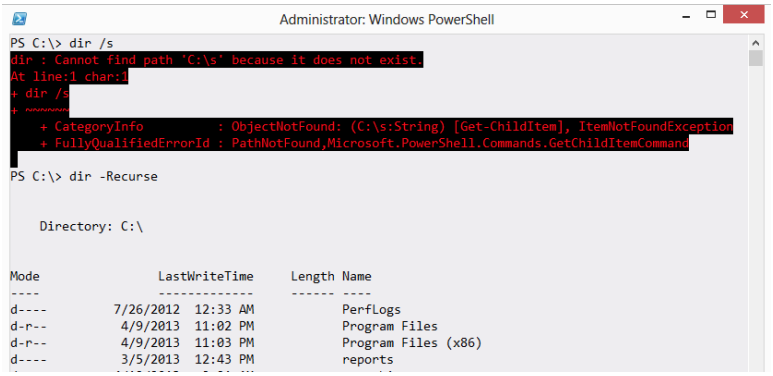


image075.png

Puede ejecutar la ayuda para el comando dir (o cualquier otro alias) para ver el nombre del comando real y su sintaxis adecuada.

Propiedades vs. Valores

```
$names = Get-ADComputer -filter * |
 Select-Object -Property Name

 Get-CimInstance -Class Win32_BIOS -ComputerName $names
```

¿Sabe por qué esto no funcionará? Porque el resultado de Get-ADComputer es un objeto que tiene propiedades. Usted probablemente sabía eso. Pero el resultado de Select-Object es también un objeto que tiene propiedades. Específicamente, en este caso, es un objeto "ADComputer " seleccionado, que tiene una sola propiedad: Name.

Observe la ayuda del comando Get-CimInstance. El parámetro -ComputerName acepta objetos de tipo String. Así lo la ayuda. Pero un objeto ADComputer no es lo mismo que una cadena. La propiedad Name que se ha seleccionado contiene cadenas, pero no es una cadena en sí. Esto es una distinción enorme y es mejor no olvidarse de ello.

Piense en una propiedad como una caja. Esa caja puede contener cosas, pero es una cosa en y por sí misma, también. En este caso, la caja se denomina Name y contiene cadenas. Pero no se puede "empujar" toda la caja en algo que sólo estaba esperando Strings. "Hey, quería un String, no toda la caja"

Ahora piense en un Fax. ¿Recuerda esas máquinas? Recibían y transmitían páginas. Ahora suponga que tiene un sobre lleno de páginas. No se puede "empujar" el sobre en la máquina de fax y esperar resultados correctos. En esa analogía, el sobre es una propiedad, y las páginas dentro de ella son valores. Para lograr transmitir las páginas primero debe sacarlas del sobre.

Lo que quiere hacer en este caso es extraer las cadenas (Strings) de la caja, y Select-Object ofrece una manera de hacer eso:

```
$names = Get-ADComputer -filter * |
 Select-Object -ExpandProperty Name

 Get-CimInstance -Class Win32_BIOS -ComputerName $names
```

¿Ve la diferencia? -ExpandProperty obtiene sólo el contenido de la propiedad especificada, en lugar de devolver un objeto que sólo tiene esa propiedad. ¿Quiere una manera sencilla de probar esto en el shell? Ejecute este par de comandos:

```
Get-Service | Select -Property Name | Get-Member
Get-Service | Select -ExpandProperty Name | Get-Member
```

Variables Remotas

Cuando utilice PowerShell Remoting, debe recordar que se trata de dos o más equipos que no comparten información entre ellos. Por ejemplo, el siguiente comando funcionará correctamente en su equipo local:

```
$f1 = 'D:\Scripts\folder1'
$f2 = 'D:\Scripts\folder2'
Copy-Item -Path $f1 -Recurse -Destination $f2 -Verbose -F\
orce
```

Sin embargo, si intenta ejecutar el comando Copy-Item en un equipo remoto, se producirá un error:

```
 $f1 = "D:\Scripts\folder1"
 $f2 = "D:\Scripts\folder2"

 Invoke-Command -ComputerName MemberServer -ScriptBlock {\
Copy-Item -Path $f1 - Recurse -Destination $f2 -Verbose -\
Force}

 Cannot bind argument to parameter 'Path' because it is n\
ull.
 + CategoryInfo : InvalidData: [:] [Copy-Item], Parameter\
BindingValidationException
 + FullyQualifiedErrorId : ParameterArgumentValidationErr\
orNullNotAllowed,Microsoft.PowerShell.Commands.CopyItemCo\
mmand
 + PSComputerName : MemberServer
```

El problema es que $f1 y $f2 se definen en su equipo local, pero no en el equipo remoto. El bloque de secuencia de comandos enviado

a Invoke-Command no se evalúa en su computadora, simplemente se pasa como está (as-is).

Hay dos posibles soluciones. La primera es simplemente incluir las definiciones de variables en el bloque de secuencia de comandos:

```
1  Invoke-Command -ComputerName MemberServer -ScriptBlock {
2  $f1 = "D:\Scripts\folder1"
3  $f2 = "D:\Scripts\folder2"
4  Copy-Item -Path $f1 -Recurse -Destination $f2 -Verbose -\
5 Force
6  }
```

Otra técnica, disponible en PowerShell v3 y posterior, es utilizar el designador de variable $using. PowerShell pre-escanea el bloque de secuencia de comandos y pasará los valores de la(s) variable(s) local(es) al (los) equipo(s) remoto(s).

```
1  $f1 = "D:\Scripts\folder1"
2  $f2 = "D:\Scripts\folder2"
3
4  Invoke-Command -ComputerName MemberServer -ScriptBlock {
5  Copy-Item -Path $using:f1 -Recurse -Destination $using:f\
6 2 -Verbose -Force}
```

El uso de la sintaxis especial $using: es lo que hace que esta versión del comando funcione.

New-Object PSObject vs. PSCustomObject

A menudo hay cierta confusión en lo que respecta a las diferencias entre el uso de nuevo objeto PSObject y PSCustomObject, así como el funcionamiento de ambos.

Cualquiera de los dos se puede utilizar para formar un conjunto de valores en una colección de objetos PowerShell y agruparlos en una sola entidad. Asimismo, ambas formas darán salida a los datos como NoteProperties en los tipos de objeto System.Management.Automation.PSCustomObject. Así que ¿cuál es la gran diferencia entre ellos?

Para empezar, el Cmdlet New-Object fue introducido en PowerShell v1.0 y ha pasado por una serie de cambios, mientras que el uso de la clase PSCustomObject vino más tarde en la versión 3.0. Para los sistemas que utilicen PowerShell v2.0 o anterior, se debe utilizar New-Object. La diferencia clave entre la versión 2.0 y la versión 1.0 desde un punto de vista administrativo es que 2.0 permite el uso de tablas hash. Por ejemplo:

New-Object PSObject en v1.0

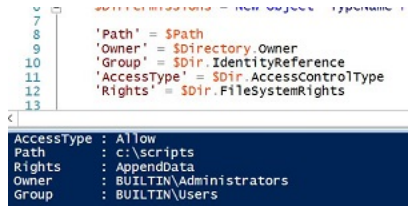
```
$Path = "c:\scripts"
$Directory = Get-Acl -Path $Path

ForEach ($Dir in $Directory.Access){

    $DirPermissions = New-Object -TypeName PSObject
    $DirPermissions | Add-Member -MemberType NoteProperty\
 -Name Path -Value $Path
    $DirPermissions | Add-Member -MemberType NoteProperty\
 -Name Owner -Value $Directory.Owner
    $DirPermissions | Add-Member -MemberType NoteProperty\
 -Name Group -Value $Dir.IdentityReference
    $DirPermissions | Add-Member -MemberType NoteProperty\
 -Name AccessType -Value $Dir.AccessControlType
    $DirPermissions | Add-Member -MemberType NoteProperty\
 -Name Rights -Value $Dir.FileSystemRights

    $DirPermissions
}
```

Con el método New-Object en PowerShell v1.0, tiene que declarar el tipo de objeto que desea crear y agregar miembros a la colección en comandos de forma individual. Sin embargo en la versión 2.0 con la capacidad de utilizar hashtables:

New-Object en PS 2.0

```
$Path = "c:\scripts"
$Directory = Get-Acl -Path $Path

ForEach ($Dir in $Directory.Access){

    $DirPermissions = New-Object -TypeName PSObject -Prop\
erty @{

    'Path' = $Path
    'Owner' = $Directory.Owner
    'Group' = $Dir.IdentityReference
    'AccessType' = $Dir.AccessControlType
    'Rights' = $Dir.FileSystemRights

    }

    $DirPermissions
}
```

Aquí está la salida:

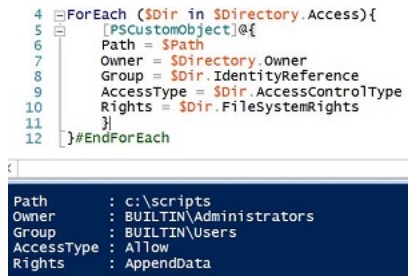


Note theorder of output vs. order in the hash table

Esta forma nos ahorra una gran cantidad de escritura al mismo tiempo que permite un script más limpio. Sin embargo ambos métodos tienen el mismo problema. La salida no está necesariamente en el mismo orden en que se ha declarado, así que si está buscando un formato determinado, puede que no funcione. PSCustomObject corrigió esto cuando fue introducido en la versión 3.0.

PSCustomObject en PowerShell v3.0

```
$Path = "c:\scripts"
$Directory = Get-Acl -Path $Path

ForEach ($Dir in $Directory.Access){
    [PSCustomObject]@{
    Path = $Path
    Owner = $Directory.Owner
    Group = $Dir.IdentityReference
    AccessType = $Dir.AccessControlType
    Rights = $Dir.FileSystemRights
    }#EndPSCustomObject
}#EndForEach
```



Note the order of the properties

Como se puede observar, la salida siempre coincidirá con lo que se ha definido en el Hashtable. Otra ventaja de usar PSCustomObject es que la enumeración de los datos se hace más rápidamente que su contraparte New-Object. Lo único que debe tener en cuenta con PSCustomObject es que no funcionará con los sistemas que ejecutan PowerShell v2.0 o anteriores.

Ejecutando algo como el "usuario actualmente conectado"

Una solicitud de PowerShell común es poder iniciar de forma remota algún código que se ejecuta bajo la cuenta del usuario que está conectado actualmente a una máquina remota o el usuario que más a menudo utiliza la máquina remota.

Esto es realmente difícil, y generalmente impráctico.

Primero, entender que Windows es inherentemente un sistema operativo multiusuario. No tiene un concepto para "el usuario actualmente conectado" porque puede haber muchos usuarios conectados. Aunque las versiones cliente de Windows no permiten técnicamente múltiples inicios de sesión interactivos, el sistema operativo base actúa como si pudiera.

Segundo, como un sistema operativo multiusuario, el trabajo de Windows es mantener un estricto aislamiento alrededor del espacio de proceso de cada usuario. Usted no quiere que un usuario salte en el espacio de trabajo a otro, porque eso sería un gran riesgo para la seguridad y la estabilidad. Es por esto que no puede iniciar sesión como un usuario y ejecutar algo que otro usuario puede "ver".

Por ejemplo, una versión común de esta solicitud es para que un administrador de manera remota abra el Bloc de Notas como una ventana (pop up) en frente de los usuarios, para presentar de forma remota mensajes importantes. Por desgracia, el Bloc de Notas no es una buena aplicación de mensajería instantánea y Windows no hace que esto sea fácil. Si lo piensa con más detalle, ¿se imagina que podría hacer el malware si esto fuera posible? Sería horrible!

Con muy pocas excepciones, realmente no se puede ejecutar algo "como otro usuario en una máquina remota". Una excepción es si conoce el nombre de usuario y la contraseña del usuario remoto. Si lo conoce, puede iniciar una sesión de acceso remoto en la computadora mediante sus credenciales y, potencialmente, ejecutar aplicaciones en el espacio de proceso de ese usuario. Aunque eso es muy poco práctico en la mayoría de situaciones.

Comandos que necesitan un perfil de usuario pueden fallar cuando se ejecuta de forma remota

Muchos comandos actúan utilizando el perfil del usuario que ha iniciado sesión actualmente. Estos comandos a veces pueden fallar cuando los ejecuta a través de una conexión remota, como con Invoke-Command o Enter-PSSession. Por ejemplo, muchos instaladores predeterminan la creación de iconos por usuario y pueden fallar cuando se ejecutan remotamente, incluso cuando se ejecutan en un modo de "instalación silenciosa".

El problema es que, cuando se conecta a un equipo remoto, no está generando un entorno de usuario completo. Técnicamente no está "conectándose" a la máquina en el sentido usual. Se está autenticando, sí, pero de la misma manera que si se autenticara a una carpeta compartida. Su conexión remota no tiene un perfil de usuario completo, por lo que cualquier cosa que se espere puede obtener errores y fallar (incluso si no muestran esos errores).

No hay solución fácil para esto, por desgracia.

Escribiendo en SQL Server

Guardar datos en un servidor SQL - frente a Excel o algún otro formato - es muy fácil.

Suponga que tiene SQL Server Express instalado localmente. Ha creado una base de datos llamada mydb y una tabla llamada mytable. La tabla tiene dos columnas ColumnA y ColumnB, y ambas son campos de cadenas (varchar). El archivo de base de datos está ubicado en c:\myfiles\mydb.mdf. Esto es muy fácil de configurar en un GUI si descarga la versión de SQL Server Express "con herramientas". Es gratis!

```
$cola = "Data to go into ColumnA"
$colb = "Data to go into ColumnB"

$connection_string = "Server=.\SQLExpress;AttachDbFilenam\
e=C:\Myfiles\mydb.mdf;Database=mydb;Trusted_Connection=Ye\
s;"
$connection = New-Object System.Data.SqlClient.SqlConnect\
ion
$connection.ConnectionString = $connection_string
$connection.Open()
$command = New-Object System.Data.SqlClient.SqlCommand
$command.Connection = $connection

$sql = "INSERT INTO MYTABLE (ColumnA,ColumnB) VALUES('$co\
la','$colb')"
$command.CommandText = $sql
$command.ExecuteNonQuery()
```

```
18
19 $connection.close()
```

Puede insertar una gran cantidad de valores simplemente haciendo un bucle a través de las tres líneas que definen la sentencia SQL y ejecutarla:

```
1  $cola = @('Value1','Value2','Value3')
2  $colb = @('Stuff1','Stuff2','Stuff3')
3
4  $connection_string = "Server=.\SQLExpress;AttachDbFilenam\
5  e=C:\Myfiles\mydb.mdf;Database=mydb;Trusted_Connection=Ye\
6  s;"
7  $connection = New-Object System.Data.SqlClient.SqlConnect\
8  ion
9  $connection.ConnectionString = $connection_string
10 $connection.Open()
11 $command = New-Object System.Data.SqlClient.SqlCommand
12 $command.Connection = $connection
13
14 for ($i=0; $i -lt 3; $i++) {
15   $sql = "INSERT INTO MYTABLE (ColumnA,ColumnB) VALUES('$\
16 ($cola[$i])','$($colb[$i])')"
17   $command.CommandText = $sql
18   $command.ExecuteNonQuery()
19 }
20
21 $connection.close()
```

Es igual de fácil ejecutar consultas de actualización o eliminación. Las consultas de selección usan ExecuteReader() en lugar de ExecuteNonQuery() y devuelven un objeto SqlDataReader que se puede utilizar para leer datos de cada columna o avanzar a la siguiente fila en el conjunto de resultados.

Obtener tamaños de carpetas

La gente suele preguntar cómo usar PowerShell para obtener el tamaño de una carpeta. Por ejemplo la carpeta de documentos de un usuario.

El problema es que *las carpetas no tienen un tamaño*. Windows literalmente no rastrea el tamaño de los objetos de carpeta. El tamaño de una carpeta es simplemente la suma de los tamaños de los archivos en dicha carpeta, lo que significa que para obtener el tamaño de la carpeta se tiene que sumar el tamaño de los archivos.

```
1 Get-ChildItem -Path <whatever> -File -Recurse |
2 Measure-Object -Property Length -Sum
```

Veamos un ejemplo. Primero, es necesario obtener todos los archivos, y luego sumar sus propiedades de tamaño.