

Los errores no críticos

Hay que saber que PowerShell permite definir su comportamiento frente a los errores de diversas maneras:

- Globalmente: es decir para todo el script o para la extensión en curso (véase el capítulo Fundamentos) gracias a la variable de preferencia `$ErrorActionPreference`.
- Selectivamente: es decir que para cada commandlet el comportamiento puede cambiar. Esto es posible gracias a la utilización de un parámetro común a todos los commandlets. Este parámetro se denomina `ErrorAction`.


1. Variable de preferencia: `$ErrorActionPreference`

Centrémonos por ahora en la «variable de preferencia» (así es como se denominan las variables intrínsecas que almacenan las preferencias de los usuarios) `$ErrorActionPreference`.

Podrán tomar los valores siguientes:

Valor	Descripción
SilentlyContinue	El script se ejecuta sin mostrar error incluso si lo encuentra.
Continue	El script sigue si encuentra un error y lo muestra (valor por defecto).
Stop	El script se interrumpe si encuentra un error. En este caso todos los errores se consideran errores críticos.
Inquire	Cuando aparece un error, un prompt preguntará al usuario lo que debe hacer (seguir, seguir en modo silencioso, detener o suspender).

Como por defecto `$ErrorActionPreference` contiene el valor `continue`, los errores no críticos no bloquean la ejecución; sólo serán consignados por PowerShell y se mostrarán en pantalla.

 Cuando `$ErrorActionPreference` toma el valor `stop`, todos los errores encontrados pasan a ser errores críticos. De este modo, si un error no crítico aparece durante la ejecución de un script, lo interrumpirá, exactamente como si fuera un error crítico. Para cambiar el valor de `$ErrorActionPreference`, podrá hacer lo siguiente `$ErrorActionPreference = 'Stop'` (¡no olvide las comillas simples!) o `Set-Variable ErrorActionPreference Stop`

Ejemplo:

Cuando esté en Windows Vista o Windows 7 e inicie normalmente PowerShell, éste se ejecutará en modo de usuario aunque usted esté conectado con su cuenta administrador. Por este hecho, es muy posible que usted no tenga los permisos de acceso suficientes para ver el contenido de un directorio en particular.

```
PS > $ErrorActionPreference
Continue

PS > Get-ChildItem 'C:\documento privado'
Get-ChildItem: El acceso a la ruta de acceso 'C:\documento privado' está
denegado.
En línea: 1 Carácter: 14
+ Get-ChildItem << 'C:\documento privado'
```

Ahora modificamos la variable `$ErrorActionPreference` con el valor `silentlyContinue` y volvemos a ejecutar el mismo comando:

```
PS > $ErrorActionPreference = 'SilentlyContinue'
PS > Get-ChildItem 'C:\documento privado'
```

Esta vez, aunque el error sigue presente, no se muestra. Por otra parte, los errores dejarán de mostrarse, sea cual sea el comando que tecleemos y esto será así mientras no regresemos al modo `Continue`.

```
PS > $ErrorActionPreference = 'SilentlyContinue'
PS > Get-ChildItem 'C:\documento privado'
PS > Get-CommandQueNoExiste
```

En lugar de estar jugando todo el rato con la extensión actual (la del script), y con el riesgo de no saber en qué modo nos encontramos, podríamos utilizar un bloque de script para hacer nuestras pruebas. Ya habrá comprendido que el alcance de `$ErrorActionPreference`, al igual que para todas las demás variables, se limita al bloque. Para retomar nuestro ultimo ejemplo, podemos escribir lo siguiente:

```
PS > &{
>> $ErrorActionPreference = 'SilentlyContinue'
>> Get-ChildItem 'C:\documento privado'
>> Get-CommandQueNoExiste
>> }
>>
```

De este modo, sea cual sea la modalidad de ejecución actual de nuestra Shell, el bloque se ejecutará siempre de la misma forma.

2. El parámetro -ErrorAction y los parámetros comunes

Otra técnica consiste en utilizar los parámetros «comunes» de los commandlets. Encontraremos muy a menudo el término inglés «Common Parameters» para designarlos.

Éstos constituyen una de las grandes fuerzas de PowerShell: la homogeneidad. En efecto, estos parámetros están presentes en todo el conjunto de comandos de PowerShell.

Estos parámetros son los siguientes:

Parámetro	Descripción
ErrorAction (SilentlyContinue Continue Inquire Stop)	Determina el comportamiento del commandlet en caso de error.
ErrorVariable <nombre de la variable>	El error resultante se almacena en la variable pasada por parámetro, además de <code>\$Error</code> .
Debug {\$true \$false}	Indica al commandlet de pasar al modo depuración. Este modo no siempre existe para todos los commandlets.
Verbose {\$true \$false}	Indica al commandlet de pasar al modo verbose. Este modo no siempre existe para todos los commandlets.
OutVariable <nombre de la variable>	La salida resultante del comando durante el tratamiento se almacena en la variable pasada por parámetro.
OutBuffer <Int32>	Determina el número de objetos a guardar en la memoria tampón antes de llamar al commandlet siguiente de la tubería.

Para dar un poco más de flexibilidad a nuestros scripts, y para evitar utilizar siempre la variable `$ErrorActionPreference`,

podemos utilizar el parámetro `-ErrorAction`. Este parámetro nos permitirá actuar sobre el comportamiento de cada commandlet ejecutado y no a nivel global como hemos visto antes.

Ejemplo:

```
PS > $ErrorActionPreference = 'Continue'
PS > Get-ChildItem 'C:\documento privado'
Get-ChildItem: El acceso a la ruta de acceso 'C:\documento privado' está
denegado.
En línea: 1 Carácter: 14
+ Get-ChildItem <<<< 'C:\documento privado'

PS > gci 'C:\documento privado' -ErrorAction SilentlyContinue
```



Hemos utilizado el alias «gci» de `Get-ChildItem` a fin de garantizar que el comando permanezca en una única línea para una mejor comprensión. Hubiéramos podido también utilizar el parámetro abreviado «-EA» en lugar de `ErrorAction`. Gracias al empleo de `-ErrorAction SilentlyContinue` hemos impedido la visualización de un mensaje de error cuando que `$ErrorActionPreference` se encontraba «en modo `Continue`».

Acabamos de ilustrar los valores `Continue` y `SilentlyContinue` de `$ErrorActionPreference`, pero no hemos hablado aún de `stop` y de `Inquire`.

`stop` permite interrumpir la ejecución de un script cuando encontramos un error incluso si se trata de un error no crítico. Es un medio de asegurarse que un script no podrá desarrollarse si se produce un error.

En cuanto a `Inquire`, éste indica a PowerShell que debe preguntar al usuario lo que hay que hacer, como en el ejemplo siguiente:

```
PS > gci 'C:\documento privado' -ErrorAction Inquire

Confirmar
El acceso a la ruta de acceso 'C:\documento privado' está denegado.
[O] Si [T] Si para todo [I] Interrumpir el comando
[S] Suspender [?]
Ayuda (el valor por defecto es «0»): i
Get-ChildItem: La ejecución del comando se detiene ya
que el usuario ha seleccionado la opción Halt.
En línea: 1 Carácter: 4
+ gci << 'C:\documento privado' -ErrorAction Inquire
```



Podemos pasar un valor a un parámetro como se muestra en estos ejemplos: `gci 'c:\documento privado' -ErrorAction Inquire`; `gci 'c:\documento privado' -ErrorAction:Inquire`. Ambas formas de sintaxis son posibles.

3. Consignación de los errores

Cualquiera que sea el modo en el que se encuentre (`Continue`, `SilentlyContinue`, `stop` o `Inquire`), existe una variable «automática» llamada `$Error` que contiene, en forma de tabla (o más concretamente un `ArrayList`, se trata en este caso de una tabla de objetos de tipo `ErrorRecord`), los últimos 256 mensajes de error encontrados. La cifra de 256 corresponde a la variable `$MaximumErrorCount`. Sin embargo, si usted tiene necesidad de almacenar mayor cantidad de errores, puede modificar dicho valor.

El último error se encuentra siempre en `$Error[0]`, el penúltimo en `$Error[1]` y así sucesivamente...

Por ejemplo:

```
PS > $ErrorActionPreference = 'Continue'
PS > gci 'C:\documento privado' -ErrorAction SilentlyContinue
```

Gracias al parámetro **ErrorAction** hemos podido impedir la visualización de un mensaje de error cuando todavía estábamos a nivel del script en modo **Continue**.

Veamos ahora el contenido de la variable **\$Error[0]**:

```
PS > $ErrorActionPreference = 'Continue'
PS > gci 'C:\documento privado' -ErrorAction SilentlyContinue
PS > $Error[0]
Get-ChildItem: El acceso a la ruta de acceso 'C:\documento privado' está
denegado.
En línea: 1 Carácter: 4
+ gci << 'C:\documento privado' -ErrorAction SilentlyContinue
```

Además de utilizar **\$Error[0]**, existe otro medio para recuperar un mensaje de error. Aunque **\$Error[0]** es muy práctico, con el uso usted se dará cuenta de que en algunos casos no encontramos siempre el error previsto. Supongamos que tengamos algunas líneas de código que se suceden unas a otras y que luego llega nuestra verificación de error con **\$Error[0]**.


El registro **\$Error[0]** contiene información suficiente para poder identificar fácilmente la línea que ha provocado un error. En este contexto, el problema es que es posible perder un error y registrar el error generado por otro comando posterior del script. Es preciso entonces repasar «a mano» la tabla **\$Error** para encontrar la línea que nos interesa, lo que puede ser laborioso y hace complicada la automatización del tratamiento del error.

De ahí la ventaja de «fijar» el almacenamiento del error a nivel del comando en sí.

Gracias al parámetro **-ErrorVariable** vamos a poder almacenar el mensaje de error en una variable elegida por nosotros, y para cada commandlet o sólo para los que nos interesen; exactamente igual que con el parámetro **-ErrorAction**.

En el ejemplo siguiente, procederemos a enviar el error a la variable **\$MiVariable**. Tenga en cuenta que no hay que poner el símbolo del dólar antes del nombre de la variable con **-ErrorVariable**.

```
PS > $ErrorActionPreference = 'SilentlyContinue'
PS > gci 'C:\documento privado' -ErrorVariable MiVariable
PS >
PS > $MiVariable
Get-ChildItem: El acceso a la ruta de acceso 'C:\documento privado' está
denegado.
En línea: 1 Carácter: 4
+ gci << 'C:\documento privado' -ErrorVariable MiVariable
```

 Si es de los que le gusta utilizar parámetros abreviados, podrá entonces usar el parámetro abreviado **-ea** en lugar de **-ErrorAction** y **-ev** en lugar de **-ErrorVariable**.

4. El tipo **ErrorRecord**

Examinemos con más detalle nuestra variable **\$MiVariable** dado que ésta resulta ser particularmente interesante. Teclee:

```
PS > $MiVariable | Get-Member -Force
      TypeName: System.Management.Automation.ErrorRecord

Name                MemberType      Definition
----                -
pstypenames         CodeProperty    System.Collections.ObjectModel...
```

psadapted	MemberSet	psadapted {Exception, TargetObj...
PSBase	MemberSet	PSBase {Exception, TargetObject...
psextended	MemberSet	psextended {PSMessageDetails}...
psobject	MemberSet	psobject {Members, Properties, ...
Equals	Method	bool Equals(System.Object obj) ...
GetHashCode	Method	int GetHashCode()...
GetObjectData	Method	System.Void GetObjectData(Syste...
GetType	Method	type GetType()
get_CategoryInfo	Method	System.Management.Automation.Er...
get_ErrorDetails	Method	System.Management.Automation.Er...
get_Exception	Method	System.Exception get_Exception(...
get_FullyQualifiedErrorId	Method	string get_FullyQualifiedErrorI...
get_InvocationInfo	Method	System.Management.Automation.In...
get_PipelineIterationInfo	Method	System.Collections.ObjectModel. ...
get_TargetObject	Method	System.Object get_TargetObject(...
set_ErrorDetails	Method	System.Void set_ErrorDetails(Sy...
ToString	Method	string ToString()
CategoryInfo	Property	System.Management.Automation.Er...
ErrorDetails	Property	System.Management.Automation.Er...
Exception	Property	System.Exception Exception {get...
FullyQualifiedErrorId	Property	System.String FullyQualifiedErr...
InvocationInfo	Property	System.Management.Automation.In...
PipelineIterationInfo	Property	System.Collections.ObjectModel. ...
TargetObject	Property	System.Object TargetObject {get...
PSMessageDetails	ScriptProperty	System.Object PSMessageDetails


Podemos apreciar que el tipo es **ErrorRecord**, el mismo que el de los registros de la tabla **\$Error**; y este tipo posee las características siguientes (las propiedades señaladas con un asterisco sólo están disponibles con PowerShell v2):

Propiedad	Descripción
Exception	Se trata de un mensaje de error tal como se muestra en pantalla. En realidad es algo más complejo que esto ya que esta propiedad devuelve de hecho un objeto cuyo tipo varía en función del error. Para comprobarlo, bastará con observar el tipo y los miembros de \$Error[0] , y después de \$Error[1] , se sorprenderá.
ErrorDetails	Contiene información adicional sobre el error encontrado. Esta propiedad puede ser nula. Si no es nula, es preferible mostrar ErrorDetails.message en lugar de Exception.message ya que el mensaje es mucho más preciso.
FullyQualifiedErrorId	Esta propiedad identifica el error de la forma más precisa posible. Se suele utilizar para aplicar un filtro en un error concreto.
CategoryInfo	Devuelve la categoría del error.
TargetObject	Objeto que ha provocado el error. Esta propiedad puede ser nula.
PipelineIterationInfo (*)	Devuelve el estado de la tubería cuando se ha creado el error.
InvocationInfo	Devuelve el contexto en el que se ha producido el error (véase la figura siguiente), como la posición y el número de línea.

Observamos ahora las propiedades de nuestro error:

```
PS > $MiVariable | Format-List -Force
```

```
Exception          : System.UnauthorizedAccessException: El acceso a la
ruta de acceso 'C:\documento privado' está denegado.
                  en System.IO.__Error.WinIOError(Int32 errorCode,
String maybeFullPath)
                  en System.IO.Directory.InternalGetFileDirectoryNames
(String path, String userPathOriginal, String searchPattern, Boolean
includeFiles, Boolean includeDirs, SearchOption searchOption)
                  en System.IO.DirectoryInfo.GetDirectories(String
searchPattern, SearchOption searchOption)
                  en System.IO.DirectoryInfo.GetDirectories()
                  en Microsoft.PowerShell.Commands.FileSystemProvider.
Dir(DirectoryInfo directory, Boolean recurse, Boolean nameOnly,
ReturnContainers returnContainers)
TargetObject       : C:\documento privado
CategoryInfo       : PermissionDenied: (C:\documento privado:String)
[Get-ChildItem], UnauthorizedAccessException
FullyQualifiedErrorId : DirUnauthorizedAccessError,Microsoft.PowerShell.
Commands.GetChildItemCommand
ErrorDetails       :
InvocationInfo      : System.Management.Automation.InvocationInfo
PipelineIterationInfo: {0, 1}
PSMessageDetails    :
```

 Para poder ver todas las propiedades necesitaremos utilizar el parámetro **-Force** del commandlet **Format-List**. Esto es debido a una vista personalizada definida por los creadores de PowerShell. Se decidió no mostrar más que lo estrictamente necesario, con el fin de no bombardear al usuario con un montón de mensajes superfluos.

5. Direccionar la visualización de los mensajes de error

Acabamos de ver el primer método para acabar con la visualización de mensajes de error. Se trata de cambiar el modo de ejecución a **silentlyContinue**, bien de forma global (con **\$ErrorActionPreference**), bien de forma selectiva commandlet por commandlet (con el parámetro **-ErrorAction**).

El otro método consiste en dirigir el flujo de escritura de los mensajes de error, no hacia el flujo de error, sino hacia el flujo de visualización estándar. Será por tanto posible enviar los errores bien a un archivo de texto, bien a una variable.

a. Direccionar los errores a un archivo de texto

Esto se hace mediante el empleo del operador **«2>»**.

Ejemplo:

Direccionar errores a un archivo de log.

```
PS > Get-ChildItem 'C:\documento privado' 2> Error.log
```

Ejecutando este comando hemos creado un archivo. Sin embargo, tenga en cuenta que si existe un archivo con el mismo nombre, se borrará.



Si lo que queremos es añadir contenido a un archivo, es preciso esta vez utilizar el operador **«2>>»**.

b. Direccionar los errores a una variable

Hemos visto anteriormente que esto es posible con el empleo del parámetro `-ErrorVariable`. Ahora bien, existe una segunda vía gracias al operador `<2>&1`.

Este operador indica al intérprete que tiene que enviar los errores hacia el flujo estándar, por tanto a la pantalla, y no hacia el flujo de errores. Por tanto necesitamos utilizar una variable para almacenar nuestro error.



Empleamos los términos «almacenar nuestro error» en lugar de «almacenar nuestro mensaje de error» ya que la variable de error es de tipo `ErrorRecord` y no de tipo `String`.

Ejemplo:

Direccionar los errores a una variable.

```
PS > $Error = Get-ChildItem 'C:\documento privado' 2>&1
```



Observará que cuando se produce un mensaje de error éste es de color rojo. Cuando utilice el operador `<2>&1` sin una variable para recuperar el error, el mensaje que aparece en pantalla es de color estándar. Esta es la prueba de que el mensaje de error ha sido direccionado hacia el flujo estándar.



Si el color rojo de los mensajes de error no le gusta, puede cambiarlo, por ejemplo por el verde: `$host.PrivateData.set_ErrorForegroundColor('green')`. Para ver la lista de colores, consulte el capítulo Control del Shell - Personalizar PowerShell modificando su perfil.

c. Direccionar los errores hacia \$null

También se puede direccionar los mensajes hacia `$null` utilizando la sintaxis `<2>$null`. Todo y que el error está siempre consignado a `$Error`, el flujo de error se direcciona y se muestra en pantalla.

Ejemplo:

Direccionar hacia \$null.

```
PS > Get-ChildItem 'C:\documento privado' 2>$null
```

6. Interceptación de los errores no críticos

Hay varias maneras de interceptar los errores en un script. La más sencilla consiste en verificar el resultado booleano contenido en la variable `$?`. Esta variable contiene el resultado de la ejecución de la última operación.

Para comprender su funcionamiento emplearemos el ejemplo siguiente:

```
PS > &{  
>> $ErrorActionPreference = 'SilentlyContinue'  
>> [int]$i='ABC' # Origen del error  
>>  
>> if ($?)  
>> {  
>>     Write-Host 'Todo va bien'  
>> }  
>> Else
```

```
>> {
>>     Write-Host "Ha ocurrido un error: $($Error[0].exception.message)"
>> }
>> Write-Host 'Continuación del script.'
>> }
>>
Ha ocurrido un error: No se puede convertir el valor "ABC" al tipo "System.
Int32". Error: "La cadena de entrada no tiene el formato correcto."
Continuación del script.
```

En el ejemplo anterior, si `$?` contiene el valor falso es que se ha producido un error. Es normal que se produzca ya que estamos intentando introducir una cadena en una variable de tipo entero (Int) lo que nos planteará problemas. Seguidamente nos muestra un mensaje de error personalizado seguido del error en cuestión.

Para interceptar los errores de esta manera es preciso estar en modo `Continue` o `silentlyContinue` ya que si nos encontramos en modo `stop` el script se interrumpe justo a la altura del error y no supera por tanto la verificación. Por tanto, podemos concluir que este método no nos permitirá gestionar los errores críticos.



Será mejor estar en modo `silentlyContinue` sino PowerShell visualizará el error estándar además del mensaje personalizado, lo que puede generar algo de desorden...

Existe otro medio para saber si un programa se ha llevado a cabo correctamente con `$LastExitCode`. Esta variable contiene el código de error del último comando ejecutado pero sólo se aplica a los archivos ejecutables externos a PowerShell. En otras palabras, no se puede utilizar con los commandlets. En Windows, cuando un proceso Win32 finaliza, devuelve siempre un código de salida en forma de entero. Por convenio, este código vale `cero` si el proceso se ha desarrollado sin error, y `otro valor` en caso contrario.

Ejemplo:

```
PS > ping miMaquina.dominio
La solicitud de ping no pudo encontrar el host miMaquina.dominio.
Compruebe el nombre y vuelva a intentarlo.
PS > $?
False
PS > $LastExitCode
1
PS > ping 127.0.0.1 -n 1

Haciendo ping a 127.0.0.1 con 32 bytes de datos:

Respuesta desde 127.0.0.1: bytes=32 tiempo<1m TTL=128

Estadísticas de ping para 127.0.0.1:
    Paquetes: enviados = 1, recibidos = 1, perdidos = 0 (0% perdidos),
    Tiempos aproximados de ida y vuelta en milisegundos:
        Mínimo = 0ms, Máximo = 0ms, Media = 0ms
PS > $LastExitCode
0
PS > $?
True
```

El comando «ping» es un ejecutable externo (ping.exe), la variable `$LastExitCode` contiene un valor al ejecutarse. Observe que incluso en esta situación, `$?` puede sernos útil. De hecho, el principio de `$?` consiste en verificar si el último código de salida de un ejecutable (`$LastExitCode`) ha sido 0 o no.