Es

El gran libro de manejo de errores en PowerShell

Dave Wyatt
Principal Author

The Big Book of PowerShell Error Handling (Spanish)

The DevOps Collective, Inc.

Este libro está a la venta en http://leanpub.com/big-book-of-powershell-error-handling-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 Gotchas (Spanish)

DevOps: WTF?

PowerShell.org: History of a Community

Índice general

El gran libro de manejo de errores en PowerShell

Escrito por Dave Wyatt

A pesar del título, este es en realidad un pequeño y conciso libro diseñado para ayudarle a entender cómo PowerShell genera y maneja errores. Intentará ayudarle a crear el mejor manejo posible de errores para sus propios scripts y funciones, en tan solo unas pocas lecciones.

Obteniendo el código El código de ejemplo, junto con un archivo que documenta nombres de clases de excepción conocidos, se puede encontrar en https://github.com/devops-collective-inc/big-book-of-powershell-error-handling/tree/master/attachments.

¿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[1] para apoyar su trabajo.

[1] https://devopscollective.org/donate/

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

- Nuestra rama principal GitHub organization[2], con un repositorio para cada libro. Visite https://github.com/devops-collective-inc/
- Nuestra GitBook page[3], donde puede navegar por los libros en línea, o descargarlos en formato PDF, EPUB o MOBI. Utilizando el lector en línea, puede saltar a capítulos específicos. Visite https://www.gitbook.com/@devopscollective
- 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. GitBook normalmente contendrá nuestra última versión, incluyendo algunos bits no terminados; LeanPub siempre contiene la más reciente "publicación liberada" de cualquier libro.

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

Introducción

El manejo de errores en Windows PowerShell puede ser un tema complejo. El objetivo de este libro -que afortunadamente no es tan "grande" como su nombre lo indica- es ayudar a aclarar algo de esa complejidad y ayudarle a hacer un trabajo mejor y más conciso para manejar errores en sus scripts.

¿Qué es el manejo de errores?

Cuando decimos que un script "maneja" un error, significa que reacciona al error haciendo algo distinto del comportamiento predeterminado. En muchos lenguajes de programación y de secuencias de comandos, el comportamiento predeterminado es simplemente mostrar un mensaje de error y fallar inmediatamente. En PowerShell, también se emitirá un mensaje de error, pero a menudo se seguirá ejecutando el código después de que se produzca el error.

El manejo de errores requiere que el autor de la secuencia de comandos anticipe dónde pueden ocurrir y que escriba código para interceptar y analizar dicho errores cuando ocurren. Esto puede ser un tema complejo y a veces frustrante, particularmente en PowerShell. El propósito de este libro es mostrarle las herramientas de manejo de errores que PowerShell pone a su disposición y la mejor forma de usarlas.

¿Cómo está organizado este libro?

Después de la introducción, el libro se divide en cuatro secciones. Las dos primeras secciones están escritas asumiendo que usted

no sabe nada sobre el manejo de errores de PowerShell, y para proporcionar un sólido contexto sobre el tema. Sin embargo, no hay nada nuevo en estas secciones que no esté cubierto por los archivos de ayuda de PowerShell. Si está bastante familiarizado con el objeto ErrorRecord y los diversos parámetros / variables / declaraciones relacionados con la generación de errores, de informes y de manejo, puede pasar directamente a las secciones 3 y 4.

La sección 3 es una mirada objetiva a las características de manejo de errores de PowerShell, basada en los resultados de algún código de prueba que escribí para entender su funcionamiento. La idea era determinar si existían diferencias funcionales entre enfoques similares para manejar errores ($Error versus ErrorVariable, el uso o no de $_ en un bloque catch, etc.), lo que generó fuertes opiniones, durante y después los Scripting Games en 2013.

Estas pruebas revelan un par de dificultades, en particular, al hacer uso de ErrorVariable.

La sección 4 resume las cosas dándole una visión más orientada a las tareas de manejo de errores, teniendo en cuenta los hallazgos de la sección 3.

Fundamentos para el manejo de errores en PowerShell

Empecemos por revisar algunos de los conceptos básicos.

ErrorRecords y Exceptions

En .NET Framework, sobre el que se construye PowerShell, el reporte de errores se realiza en gran medida lanzando excepciones. Las excepciones son objetos .NET que tienen como tipo base System.Exception[5]. Estos objetos de excepción contienen suficiente información para comunicar todos los detalles del error a una aplicación de .NET Framework (el tipo de error que ocurrió, un seguimiento de pila de llamadas del método que condujo al error, etc.) que por sí solo no es suficiente información para proporcionar a un script de PowerShell. Por eso, PowerShell tiene su propio seguimiento de la pila de scripts y de llamadas de función de las que .NET Framework no sabe nada. También es importante saber qué objetos generaron errores, ya que una única sentencia o tubería (pipeline) es capaz de producir múltiples errores.

Por estas razones, PowerShell expone el objeto ErrorRecord. ErrorRecord contienen una excepción .NET, junto con varias otras piezas de información específica de PowerShell. Por ejemplo, la figura 1.1 muestra cómo acceder a las propiedades TargetObject, CategoryInfo e InvocationInfo de un objeto ErrorRecord; que pro-

[5] http://msdn.microsoft.com/en-us/library/system.exception(v=vs.110).aspx

porcionan información útil para la lógica de manejo de errores de su script.

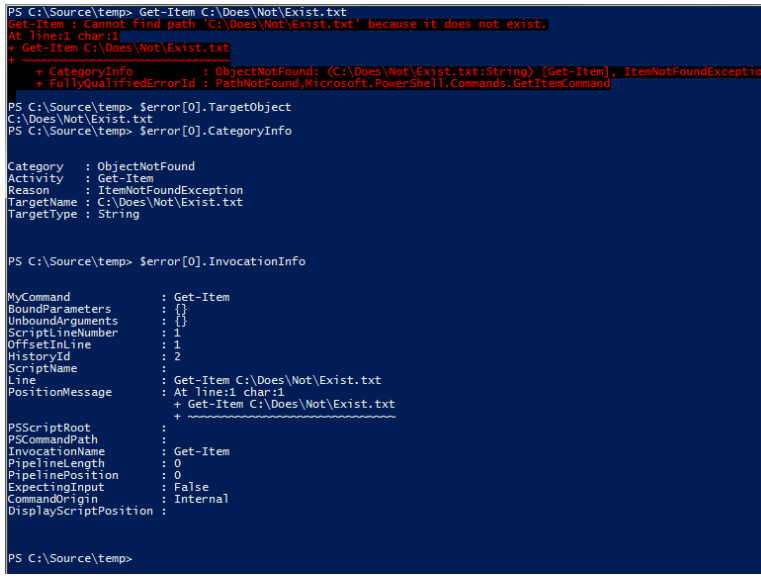

image003.png

Figura 1.1: Algunas de las propiedades más útiles del objeto ErrorRecord.

Terminating versus Non-Terminating Errors

PowerShell es un lenguaje extremadamente *expresivo*. Esto significa que una sola sentencia o pipeline de código PowerShell puede realizar el trabajo de cientos, o incluso miles de instrucciones crudas de CPU. Por ejemplo:

```
Get-Content .\computers.txt | Restart-Computer
```

Este pequeño script de PowerShell de tan solo 46 caracteres abre un archivo en disco, detecta automáticamente su codificación, lee el texto una línea a la vez, se conecta a cada computadora remota nombrada en el archivo, se autentica en ese equipo y, si tiene éxito, reinicia la computadora. Varios de estos pasos pueden encontrar errores, como en el caso del comando Restart-Computer, que puede tener éxito para algunos equipos y fallar para otros.

Por esta razón, PowerShell introduce el concepto de un error Non-Terminating. Un error Non-Terminating es aquel que no impide que el comando avance y pruebe el siguiente elemento en una lista de entradas. Por ejemplo, si uno de los equipos del archivo computers.txt está desconectado, eso no detendrá a PowerShell que seguirá intentando reiniciar el resto de los equipos del archivo.

Por el contrario, un error Terminating es uno que hace que el script o tubería (pipeline) falle. Por ejemplo, este comando busca las direcciones de correo electrónico asociadas con las cuentas de usuario en un Active Directory:

```
Get-Content .\users.txt |
Get-ADUser -Properties mail |
Select-Object -Property SamAccountName,mail
```

En este script, si el comando Get-ADUser no puede comunicarse con el Active Directory, no hay razón para seguir leyendo las líneas del archivo de texto o intentando procesar registros adicionales, por lo que se producirá un error Terminating. Cuando se encuentra este error Terminating, todo el script o tubería (pipeline) es abortado inmediatamente. Get-Content detendrá la lectura y cerrará el archivo.

Es importante conocer la distinción entre estos tipos de errores, ya que los scripts utilizarán diferentes técnicas para interceptarlos.

Como regla general, la mayoría de los errores producidos por los Cmdlets no Non-Terminating (con algunas excepciones, aquí y allá).

Controlando el comportamiento de los errores

Esta sección muestra brevemente cómo usar cada una de las declaraciones, variables y parámetros de PowerShell que están relacionados con el reporte o manejo de errores.

La variable $Error

$Error es una variable global automática en PowerShell que siempre contiene un ArrayList de cero o más objetos ErrorRecord. A medida que se producen nuevos errores, se agregan al principio de esta lista, por lo que siempre se puede obtener información sobre el error más reciente utilizando $Error[0]. Los errores Terminating y Non-Terminating se incluirán en esta lista.

Aparte de acceder a los objetos de la lista con la sintaxis de matriz, hay otras dos tareas comunes que se realizan con la variable $Error: Se puede comprobar cuántos errores están actualmente en la lista utilizando la propiedad $Error.Count y puede eliminar todos los errores de la lista con el método $Error.Clear(). Por ejemplo:

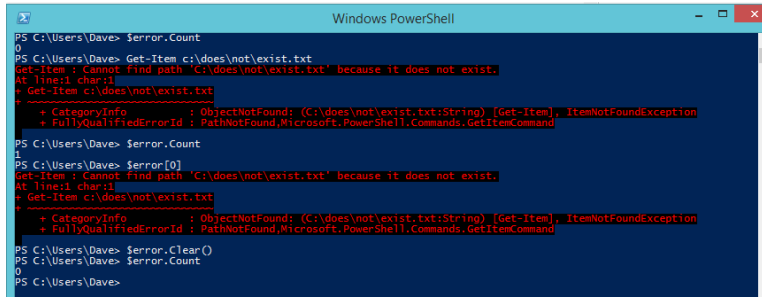

image004.png

Figura 2.1: Utilizando $Error para acceder a la información de error, verificar el recuento y borrar la lista.

Si está planeando hacer uso de la variable $Error en sus scripts, tenga en cuenta que puede contener información sobre errores que ocurrieron en la sesión actual de PowerShell, pero antes de que se iniciara la ejecución de su secuencia de comandos. Algunas personas consideran una mala práctica borrar la variable $Error dentro de un script. Como se trata de una variable global para la sesión de PowerShell, la persona que llamó a su secuencia de comandos podría revisar el contenido de $Error después de que su comando haya terminado la ejecución..

ErrorVariable

El parámetro común ErrorVariable proporciona una alternativa al uso de la colección $Error anterior. A diferencia de $Error, ErrorVariable sólo contendrá los errores que se produjeron desde el comando que se está llamando, en lugar de tener potencialmente errores de otras partes en la sesión PowerShell. Esto también evita tener que borrar el contenido de $Error (con los problemas que esto podría ocasionar).

Cuando se utiliza ErrorVariable, si desea anexar a la variable de error en lugar de sobrescribirla, coloque un signo + delante del

nombre de la variable. Tenga en cuenta que no se utiliza un signo de moneda cuando pasa un nombre de variable al parámetro ErrorVariable, pero si utiliza el signo de moneda más adelante cuando comprueba su valor.

La variable asignada al parámetro ErrorVariable nunca será nula. Si no se produjeron errores, contendrá un objeto ArrayList con un recuento de 0, como se ve en la figura 2.2:

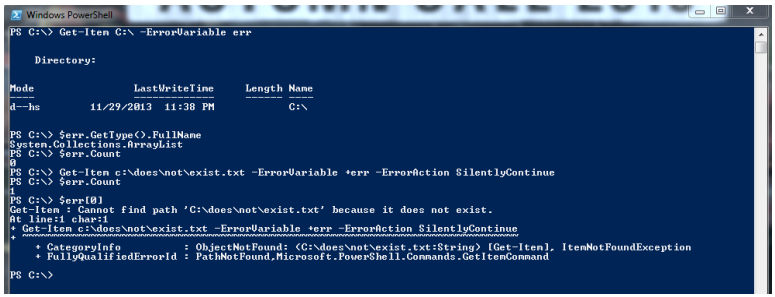

image005.png

Figura 2.2: Demostración del uso del parámetro ErrorVariable.

$MaximumErrorCount

De forma predeterminada, la variable $Error sólo puede contener un máximo de 256 errores antes de que comience a desechar los elementos más antiguos de la lista. Puede ajustar este comportamiento modificando la variable $MaximumErrorCount.

ErrorAction y $ErrorActionPreference

Hay varias maneras en las que puede controlar el comportamiento de PowerShell. Las que probablemente utilizará con más frecuencia

son los parámetro ErrorAction y la variable $ErrorActionPreference.

El parámetro ErrorAction se puede pasar a cualquier Cmdlet o función avanzada y puede tener uno de los siguientes valores: Continue (el valor por defecto), SilentlyContinue, Stop, Inquire, Ignore (sólo en PowerShell 3.0 o posterior), y Suspend (sólo para workflows, pero no se discutirá aquí). Este valor afecta el cómo se comporta el Cmdlet cuando produce un error Non-Terminating.

- El valor predeterminado Continue provoca que el error se escriba en la secuencia de errores y se agregue a la variable $Error. Entonces, el Cmdlet continuara su ejecución.
- El valor SilentlyContinue sólo agrega el error a la variable $Error. No escribe el error en la secuencia de errores (por lo que no se mostrará en la consola).
- El valor Ignore suprime el mensaje de error y no lo agrega a la variable $Error. Esta opción se agregó con PowerShell 3.0.
- El valor Stop hace que los errores Non-Terminating se traten como errores Terminating, deteniendo inmediatamente la ejecución del Cmdlet. Esto también permite interceptar estos errores en una sentencia try / catch o trap, como se describe más adelante.
- El valor Inquire provoca que PowerShell pregunte al usuario si el script debe continuar o no cuando se produce un error.

La variable $ErrorActionPreference se puede utilizar igual que el parámetro ErrorAction, con un par de excepciones: no puede establecer $ErrorActionPreference en Ignore o Suspend. Además, $ErrorActionPreference afecta su alcance actual además de cualquier comando secundario que se llame. Esta sutil diferencia tiene el efecto de permitirle controlar el comportamiento de los errores producidos por los métodos .NET, u otras causas como cuando PowerShell se encuentra con un error del tipo “comando no encontrado”.

La Figura 2.3 muestra los efectos de las tres configuraciones de ErrorActionPreference más utilizadas.

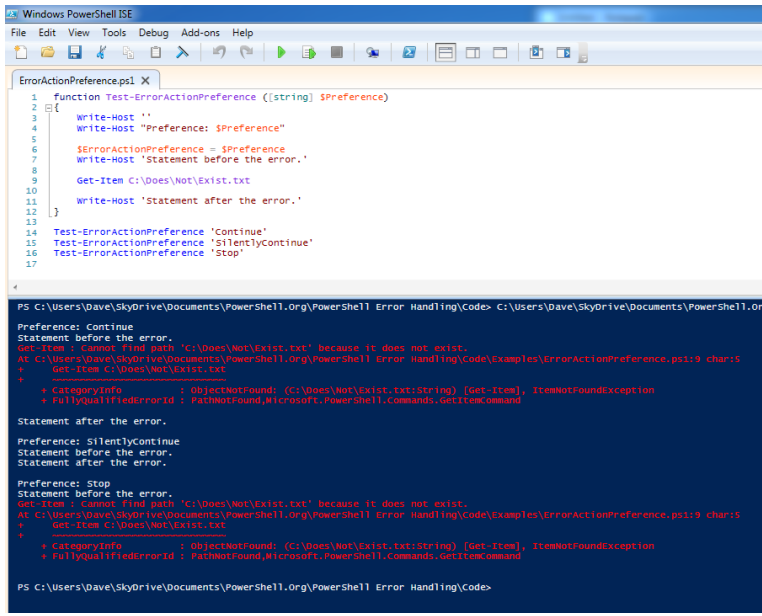

image006.png

Figura 2.3: Comportamiento de $ErrorActionPreference

Try/Catch/Finally

Las sentencias Try/ Catch/ Finally, agregadas en PowerShell 2.0, son la forma preferida de manejar los errores *Terminating*. No se pueden utilizar para manejar errores Non-Terminating, a menos que fuerce esos errores a convertirse en errores Terminating con ErrorAction o $ErrorActionPreference establecido en Stop.

Para usar Try/Catch/ Finally, comience con la palabra clave “Try” seguida de un solo bloque de secuencia de comandos de PowerShell. Después del bloque Try puede haber cualquier número de bloques

Catch y cero o un bloque Finally. Debe haber un mínimo de un bloque Catch o un bloque Finally. Un bloque Try no puede ser utilizado por sí mismo, debe tener al menos un bloque Catch.

El código dentro del bloque Try se ejecuta hasta que se completa o se produce un error Terminating. Si se produce un error Terminating, se detiene la ejecución del código en el bloque Try. PowerShell escribe el error Terminating en la lista $Error y busca un bloque Catch coincidente (ya sea en el ámbito actual o en cualquier ámbito superior). Si no existe un bloque Catch para manejar el error, PowerShell escribe el error en la secuencia Error, lo mismo que habría hecho si el error hubiera ocurrido fuera de un bloque Try.

Los bloques Catch se pueden escribir para capturar sólo tipos específicos de excepciones, o para capturar todos los errores Terminating. Si define varios bloques de captura para diferentes tipos de excepciones, asegúrese de colocar los bloques más específicos en la parte superior de la lista. Las búsquedas de PowerShell analizan los bloques de arriba abajo, y se detienen tan pronto como encuentran la primera coincidencia.

Si se incluye un bloque Finally, ese código se ejecuta después de que los bloques Try y Catch estén completos (se hayan ejecutado), independientemente de si se ha producido o no un error. Esto está destinado principalmente a realizar una limpieza de los recursos (liberar memoria, llamar a métodos Close () o Dispose (), etc.)

La Figura 2.4 muestra el uso de un bloque Try/Catch/Finally:

TryCatchFinally.ps1

```
try
{
    $testVariable = 'This is a test.'

    Write-Host 'Statement before the error.'

    [System.IO.File]::ReadAllText('C:\does\not\exist.txt')

    Write-Host 'Statement after the error.'
}
catch [System.IO.IOException]
{
    Write-Host 'An IOException was caught.'
    Write-Host "Exception type: $($_.Exception.GetType().FullName)"
}
catch
{
    Write-Host 'Some other type of error was caught.'
}
finally
{
    $testVariable = 'The finally block was executed.'
}

Write-Host "`$testVariable = '$testVariable'."
```

```
PS C:\Users\dwyatt> C:\Source\PowerShell\ErrorHandling\TryCatchFinally.ps1
Statement before the error.
An IOException was caught.
Exception type: System.IO.DirectoryNotFoundException
$testVariable = 'The finally block was executed.'.

PS C:\Users\dwyatt>
```

image007.png

Figura 2.4: Ejemplo del uso de Try/Catch/Finally.

Observe que el texto “Statement after the error” nunca se muestra, porque se produjo un error Terminating en la línea anterior. Dado que el error se produjo por una excepción IOException, se ejecutó ese bloque Catch, en lugar del bloque general “catch-all” que aparece al final. Después, el bloque Finally se ejecuta, cambiando el valor de $testVariable.

Fíjese también que mientras el bloque Catch especificaba un tipo [System.IO.IOException], el tipo de excepción real fue [System.IO.DirectoryNotFoundException]. Esto funciona porque DirectoryNotFoundException hereda de IOException, de la misma manera que todas las excepciones comparten el mismo tipo base [System.Exception]. Puede ver esto en la figura 2.5:

```
PS C:\Source\temp> $test = New-Object System.IO.DirectoryNotFoundException
PS C:\Source\temp> $test -is [System.IO.IOException]
True
PS C:\Source\temp> $test.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     DirectoryNotFoundException               System.IO.IOException


PS C:\Source\temp>
```

image008.png

Figura 2.5: Mostrando que IOException es el tipo base para DirectoryNotFoundException.

Trap

Las sentencias Trap fueron el método para manejar los errores Terminating en PowerShell 1.0. Al igual que con Try/Catch/Finally, la instrucción Trap no tiene ningún efecto en los errores Non-Terminating.

Trap es un poco incómodo de usar, ya que se aplica a todo el ámbito donde se define (y los ámbitos hijos también), en lugar de tener la lógica de manejo de errores cerca del código que podría producir el error como cuando se utiliza Try/Catch/Finally. Para aquellos de ustedes familiarizados con Visual Basic, Trap es parecido a "On Error Goto". Por eso, las sentencias Trap no ven mucho uso en los scripts de PowerShell modernos, y no los incluí en los scripts de prueba ni en el análisis de la Sección 3 de este libro.

En aras de mantener la integridad, he aquí un ejemplo de cómo usar Trap:

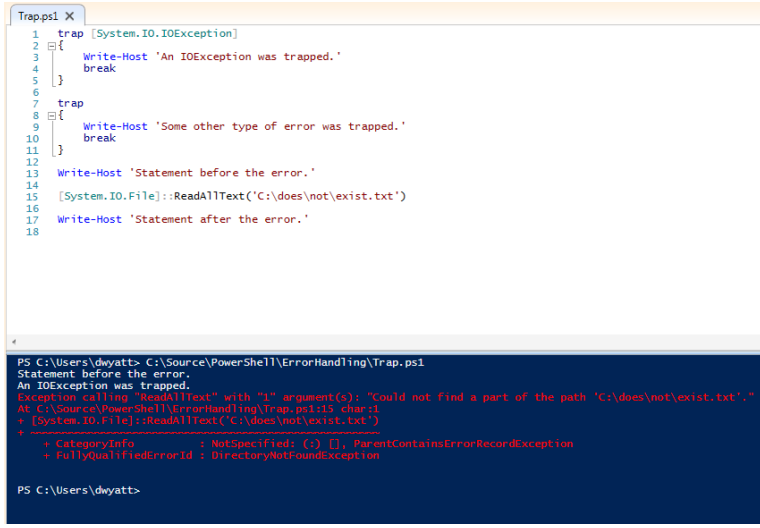

image009.png

Figura 2.6: Uso de la sentencia Trap

Como puede ver, los bloques Trap se definen de la misma forma que los bloques Catch, especificando opcionalmente un tipo Exception. Los bloques Trap pueden terminar opcionalmente con una instrucción Break o Continue. Si no se utiliza ninguno de estos, el error se escribe en la secuencia Error (Error Stream) y el bloque de secuencia de comandos actual continúa con la siguiente línea después del error. Si utiliza Break, como se ve en la figura 2.5, el error se escribe en la secuencia Error (Error Stream) y el resto del bloque de secuencia de comandos actual no se ejecuta. Si utiliza Continue, el error no se escribe en la secuencia de errores y el bloque de secuencia de comandos continúa la ejecución con la siguiente instrucción.

La variable $LASTEXITCODE

Cuando llama a un programa ejecutable externo en lugar de un Cmdlet, un Script o una función de PowerShell, la variable $LASTEXITCODE contiene automáticamente el código de salida de dicho proceso. La mayoría de los procesos utilizan por convención un código de salida con valor cero cuando el proceso finaliza con éxito y un valor diferente a cero si se produce un error, pero esto no está garantizado. Depende del desarrollador del ejecutable determinar qué significan sus códigos de salida.

Tenga en cuenta que la variable $LASTEXITCODE sólo se establece cuando llama a un ejecutable directamente o a través del operador de llamadas de PowerShell (&) o del Cmdlet Invoke-Expression. Si utiliza otro método, como Start-Process o WMI para iniciar el ejecutable, estos tienen sus propias maneras de comunicar su código de salida, por lo que no se afectará el valor actual de $LASTEXITCODE.

```
PS C:\> cmd.exe /c "dir c:\*.bat"
 Volume in drive C is Local Disk
 Volume Serial Number is 7859-839E

 Directory of c:\

06/10/2009  04:42 PM                24 autoexec.bat
               1 File(s)             24 bytes
               0 Dir(s)  91,893,669,888 bytes free
PS C:\> $LASTEXITCODE
0
PS C:\> ping.exe 10.0.0.2 -n 1

Pinging 10.0.0.2 with 32 bytes of data:
Request timed out.

Ping statistics for 10.0.0.2:
    Packets: Sent = 1, Received = 0, Lost = 1 (100% loss),
PS C:\> $LASTEXITCODE
1
PS C:\>
```

image010.png

Figura 2.7: Uso de $ LASTEXITCODE

La variable $?

La variable $? es un valor booleano que se establece automáticamente después de cada instrucción PowerShell o tubería (pipeline) finaliza la ejecución. Estará establecida en True si el comando anterior se ha ejecutado correctamente o en False si se produjo un error. Si el comando anterior era una llamada a un exe nativo, $? se establecerá en True si la variable $ LASTEXITCODE es igual a cero, de lo contrario, False. Cuando el comando anterior era una sentencia de PowerShell, $? Se establecerá en False si se han producido errores (incluso si ErrorAction se estableció en SilentlyContinue o Ignore).

Sólo tenga en cuenta que el valor de esta variable se restablece después de cada instrucción. Debe comprobar su valor inmediatamente después del comando que le interesa o se sobrescribirá (probablemente en True). La Figura 2.8 muestra este comportamiento. La primera vez $? se establece en False, porque el Get-Item encontró un error. La segunda vez $? Se comprobó y se estableció en True, porque el comando anterior finalizo correctamente. En este caso, el comando anterior fue "$?" cuando se visualizó el valor de la variable.

```
PS C:\> Get-Item c:\does\not\exist.txt -ErrorAction Ignore
PS C:\> $?
False
PS C:\> $?
True
PS C:\>
```

image011.png

Figura 2.8: Demostración del comportamiento de la variable $?

La variable $? no da ningún detalle sobre el error que ocurrió. Simplemente una bandera que indica que algo salió mal. En el caso de llamar a programas ejecutables, debe asegurarse de que devuelven un código de salida de 0 para indicar una operación exitosa y un valor distinto de cero para indicar un error antes de

poder confiar en el contenido de $?.

Resumen

Esto cubre todas las técnicas que puede utilizar para controlar, interceptar o manejar errores en un script de PowerShell. Resumiendo:

- Para interceptar y reaccionar ante errores Non-Terminating, comprueba el contenido de la colección automática $Error o de la variable que ha especificado como ErrorVariable. Esto se hace después de que el comando se completa. No puede reaccionar a un error Non-Terminating antes de que el Cmdlet o Función termine su trabajo.
- Para interceptar y reaccionar a la terminación de errores, utilice Try/Catch/Finally (preferido) o Trap (antiguo y en desuso). Ambos le permiten especificar diferentes bloques de secuencias de comandos para reaccionar a diferentes tipos de excepciones.
- Mediante el parámetro ErrorAction, puede cambiar la forma en que los Cmdlets y las funciones de PowerShell informan de errores Non-Terminating. Establecer a un valor de Stop hace que se conviertan en errores Terminating y entonces pueden interceptarse con Try/Catch/Finally o Trap.
- $ErrorActionPreference funciona como ErrorAction, excepto que también puede afectar al comportamiento de PowerShell cuando se produce un error Terminating, incluso si esos errores fueron ocasionados por el llamado a un método .NET en lugar de un Cmdlet.
- $LASTEXITCODE contiene el código de salida de ejecutables externos. Un código de salida cero normalmente indica una operación exitosa, pero eso depende del autor del programa.

- $? puede decirle si el comando anterior finalizo de forma exitosa, aunque debe tener cuidado al utilizarlo con comandos externos, si no siguen la convención de usar un código de salida con valor cero como indicador de éxito. También necesita asegurarse de comprobar el contenido de $? inmediatamente después del comando que le interesa.

Análisis de los resultados de las pruebas de manejo de errores

Como se mencionó en la introducción, el código de prueba y sus archivos de salida están disponibles para su descarga. Vea la sección "acerca de", al comienzo de este libro, para conocer la ubicación. Son un montón de datos, no muy bien formateados en un documento de Word, por lo que no serán incluidos en el contenido de los archivos en este libro. Si te cuestionas acerca de cualquiera de los análisis o conclusiones que he presentado en esta sección, te animo a descargar y revisar tanto el código como los archivos de resultados.

El código de prueba consta de dos archivos. El primero es un módulo de PowerShell (ErrorHandlingTestCommands.psm1) que contiene un Cmdlet, una clase .NET y varias funciones avanzadas para producir errores Terminating y Non-Terminating a demanda, o para probar el comportamiento de PowerShell cuando se producen tales errores. El segundo archivo es el script ErrorTests.ps1, que importa el módulo, llama a sus comandos con varios parámetros y produce la salida que fue redirigida (incluyendo la secuencia de errores) a los tres archivos de resultados: ErrorTests.v2.txt, ErrorTests.v3. Txt y ErrorTests.v4.txt.

Hay tres secciones principales en el script ErrorTests.ps1. La primera sección llama a los comandos para generar errores Terminating y Non-Terminating, y envía información sobre el contenido de $_ (en bloques Catch solamente), $Error y ErrorVariable. Estas pruebas tenían como objetivo responder a las siguientes preguntas:

- Cuando se trata sólo de errores Non-Terminating, ¿hay diferencias entre cómo $Error y ErrorVariable presentan la información acerca de los errores que ocurrieron? ¿Hay alguna diferencia si los errores provienen de un Cmdlet o función avanzada?
- Cuando se utiliza un bloque Try/Catch, ¿Hay diferencias en el comportamiento entre la forma en como $Error, ErrorVariable y $_ proporcionan información sobre el error Terminating que se produjo? ¿Hay alguna diferencia si los errores proceden de un Cmdlet, función avanzada o un método .NET?
- Cuando se producen errores Non-Terminating además del error, ¿Hay diferencias entre cómo $Error y ErrorVariable presentan la información? ¿Hay alguna diferencia cuando los errores provienen de un Cmdlet o función avanzada?
- En las pruebas anteriores, ¿Hay alguna diferencia entre un error Terminating que se produjo normalmente, en comparación con un error Non-Terminating que se produjo cuando ErrorAction o $ErrorActionPreference se establecieron a Stop?

La segunda sección consiste en algunas pruebas para determinar si ErrorAction o $ ErrorActionPreference afectan a los errores Terminating, o sólo a los errores Non-Terminating.

La sección final prueba cómo se comporta PowerShell cuando encuentra errores Terminating no controlados de cada origen posible (un Cmdlet que utiliza PSCmdlet.ThrowTerminatingError(), una función avanzada utiliza la sentencia Throw de PowerShell, un método .NET que genera una excepción, un Cmdlet o una Función avanzada que produce errores Non-Terminating cuando ErrorAction se establece en Stop en un comando desconocido).

Los resultados de todas las pruebas fueron idénticos en PowerShell 3.0 y 4.0. Powershell 2.0 tuvo un par de diferencias, que veremos en el análisis.

Interceptando errores Non-Terminating

Comencemos hablando de errores Non-Terminating.

ErrorVariable versus $Error

Cuando se trata de errores Non-Terminating, sólo hay una diferencia entre $Error y ErrorVariable: el orden de los errores en las listas se invierte. El error más reciente que se produce siempre se encuentra al principio de la variable $Error (índice cero) mientras que el error más reciente se encuentra al final de ErrorVariable.

Interceptando errores Terminating

Esta es la verdadera "carne de la tarea": Trabajar con errores Terminating, o excepciones.

$_

Al principio de un bloque Catch, la variable `$_` siempre se refiere a un objeto ErrorRecord para el error Terminating, independientemente de cómo se produjo ese error.

$Error

Al principio de un bloque Catch, `$Error[0]` siempre se refiere a un objeto ErrorRecord para el error Terminating, independientemente de cómo se produjo ese error.

ErrorVariable

Aquí, las cosas empiezan a complicarse. Cuando un error Terminating se produce por un Cmdlet o una función y está utilizando ErrorVariable, la variable contendrá algunos elementos inesperados y los resultados son bastante diferentes en las distintas pruebas realizadas:

- Cuando se llama a una función avanzada que genera un error Terminating, ErrorVariable contiene dos objetos de ErrorRecord idénticos para el error. Además, si está ejecutando PowerShell 2.0, estos registros de errores son seguidos por dos objetos idénticos de tipo System.Management.Automation.RuntimeException. Estos objetos RuntimeException contienen una propiedad ErrorRecord, que hace referencia a los objetos ErrorRecord idénticos al par que también figuraba en la lista ErrorVariable. Los objetos adicionales RuntimeException no están presentes en PowerShell 3.0 o posterior.
- Cuando se llama a un Cmdlet que genera un error Terminating, ErrorVariable contiene un solo registro, pero no es un objeto ErrorRecord. En su lugar, es una instancia de System.Management.Automation.CmdletInvocationException. Como los objetos RuntimeException mencionados en el último punto, CmdletInvocationException tiene una propiedad ErrorRecord y esa propiedad se refiere al objeto ErrorRecord que se esperaba que estuviera contenido en la lista ErrorVariable.
- Cuando se llama a una función avanzada con ErrorAction establecido a Stop, ErrorVariable contiene un objeto del tipo System.Management.Automation.ActionPreferenceStopException, seguido por dos objetos de ErrorRecord idénticos. Como con los tipos RuntimeException y CmdletInvocationException, ActionPreferenceStopException todos contiene una propiedad ErrorRecord, que se refiere a un objeto ErrorRecord que es idéntico a los dos que se incluyeron directamente en la lista ErrorVariable. Además, si se ejecuta PowerShell 2.0, hay dos

objetos más idénticos al tipo ActionPreferenceStopException, para un total de 5 entradas relacionadas con el mismo error de Terminating.

- Cuando se llama a un Cmdlet con ErrorAction establecido a Stop, ErrorVariable contiene un único objeto del tipo System.Management.Automation.ActionPreferenceStopException. La propiedad ErrorRecord de este objeto ActionPreferenceStopException contiene el objeto ErrorRecord que se esperaba que estuviera directamente en la lista ErrorVariable.

Efectos de establecer ErrorAction o $ErrorActionPreference

Cuando se ejecuta un Cmdlet o una función avanzada y establece el parámetro ErrorAction, se afecta el comportamiento de todos los errores Non-Terminating. Sin embargo, también parece afectar a los errores Terminating producidos por la sentencia Throw en una función avanzada (aunque no afecta los procedentes de los Cmdlets a través del método PSCmdlet.ThrowTerminatingError())

Si establece la variable $ErrorActionPreference antes de llamar al comando, su valor afecta a los errores Terminating and Non-Terminating.

Esto es comportamiento no se encuentra documentado. Los archivos de ayuda de PowerShell indican que tanto la variable de preferencia como el parámetro sólo deberían afectar a los errores Non-Terminating.

Cómo se comporta PowerShell cuando se encuentra errores Terminating no controlados

Esta sección del código demostró ser un poco molesta de probar, porque el manejó de los errores en el alcance del padre (el script), afectó el comportamiento del código dentro de las funciones. Si el ámbito de la secuencia de comandos no tenía ningún tratamiento de errores, en muchos casos, el error no controlado abortó el script también. Como resultado, el script ErrorTests.ps1 y los archivos de texto que contienen su salida se escriben para mostrar sólo los casos en que se produce un error Terminating, pero la ejecución de la función continua y pasa al siguiente comando.

Si desea ejecutar la batería completa de pruebas para este comportamiento, importe el módulo ErrorHandlingTests.psm1 y ejecute manualmente los siguientes comandos en una consola de PowerShell. Como los va a ejecutar uno a la vez, no encontrará problemas con que algunos de los comandos fallen su ejecución debido a un error no controlado anterior. Caso distinto seria si estuvieran todos en un script.

```
Test-WithoutRethrow -Cmdlet -Terminating

Test-WithoutRethrow -Function -Terminating

Test-WithoutRethrow -Cmdlet -NonTerminating

Test-WithoutRethrow -Function -NonTerminating

Test-WithoutRethrow -Method

Test-WithoutRethrow -UnknownCommand
```

También hay una función Test-WithRethrow que se puede llamar con los mismos parámetros, para comprobar que los resultados son consistentes en los 6 casos cuando se maneja cada error y se elige si se aborta la función.

PowerShell continúa la ejecución después de producirse un error Terminating, cuando:

- Un Cmdlet genera un error Terminating
- Un método .NET genera una excepción
- PowerShell encuentra un comando desconocido

PowerShell detiene la ejecución después de producirse un error Terminating, cuando:

- Una función utiliza la sentencia Throw
- Cualquier error Non-Terminating en conjunto con ErrorAction establecido a Stop
- En cualquier momento cuando $ErrorActionPreference se establece a Stop en el ámbito del llamador

Con el fin de lograr un comportamiento coherente entre estas diferentes fuentes de errores Terminating, puede colocar los comandos que potencialmente podrían producir un error de terminación en un bloque try. En el bloque catch, puede decidir si aborta o no la ejecución del bloque de secuencia de comandos actual. La figura 3.1 muestra un ejemplo de cómo forzar una función a abortar cuando se genera una excepción de terminación desde un Cmdlet (una situación en la que PowerShell normalmente solo continuaría y ejecuta la sentencia "after terminating error"), volviendo a lanzar el error del bloque Catch. Cuando se usa Throw sin argumentos

dentro de un bloque Catch, se pasa el mismo error hacia el ámbito padre.

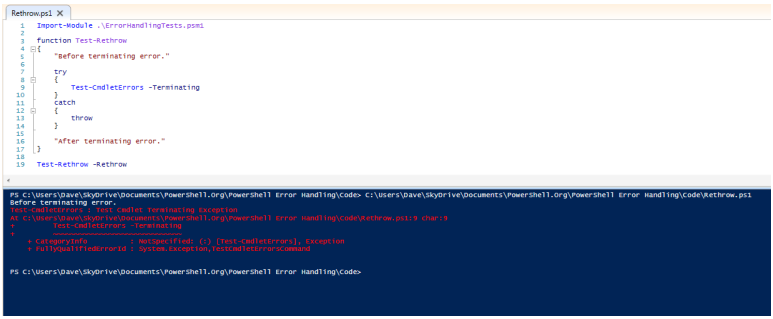

image013.png

Figura 3.1: Volviendo a lanzar un error Terminating para forzar a una función a detener la ejecución.

Conclusiones

Para errores Non-Terminating, puede utilizar $Error o ErrorVariable sin distinción. Solo debe tener presente en que el orden de los ErrorRecords se invierte, pero usted puede fácilmente controlar eso en su código, suponiendo que considere que eso sea un problema. Sin embargo, tan pronto como los errores Terminating entran en juego, ErrorVariable tiene un comportamiento muy molesto: a veces contiene objetos de excepción en lugar de ErrorRecords, y en otros casos, tiene uno o más objetos duplicados, todos relacionados con el error Terminating. Si bien es posible codificar alrededor de estas peculiaridades, realmente no parece que valga la pena el esfuerzo cuando se puede utilizar fácilmente $_ o $Error[0].

Cuando está llamando a un comando que puede producir un error Terminating y no maneja ese error dentro una sentencia Try/-Catch o Trap, el comportamiento de PowerShell es inconsistente, dependiendo de cómo se generó el error Terminating. Para lograr

resultados consistentes, independientemente de los comandos que esté llamando, coloque dichos comandos en un bloque Try y elija si desea volver a lanzar el error en el bloque Catch.

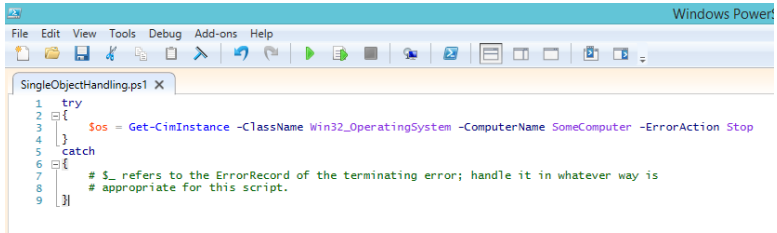

image018.png

Poniéndolo todo junto

Ahora que hemos examinado todas las herramientas de manejo de errores e identificado algunos posibles escenarios de "engañosos", he aquí algunos consejos y ejemplos de cómo abordar el manejo de errores en sus propios scripts.

Supresión de errores (no haga esto)

Hay ocasiones en las que puede "procesar" un error sin la intención de manejarlo. En realidad, las situaciones válidas para estos escenarios son pocas. Procure no establecer ErrorAction o $ErrorActionPreference en SilentlyContinue a menos que tenga la intención de examinar y verificar cada posible error, usted mismo más adelante en su código. Utilizar un bloque Try/Catch con un bloque Catch vacío equivale a la misma cosa. Por lo general esto no es lo correcto.

Es mejor al menos mostrar al usuario la salida de error por defecto en la consola, que tener un comando que falle sin indicación alguna de que algo salió mal.

Uso de la variable $? (úselo bajo su propio riesgo)

La variable $? parece una buena idea al principio, pero hay muchas cosas que podrían salir mña como para simplemente confiar en esta variable en un script de producción. Por ejemplo, si el error es generado por un comando que está entre paréntesis o una subexpresión, la variable $? se establecerá en true en lugar de false:

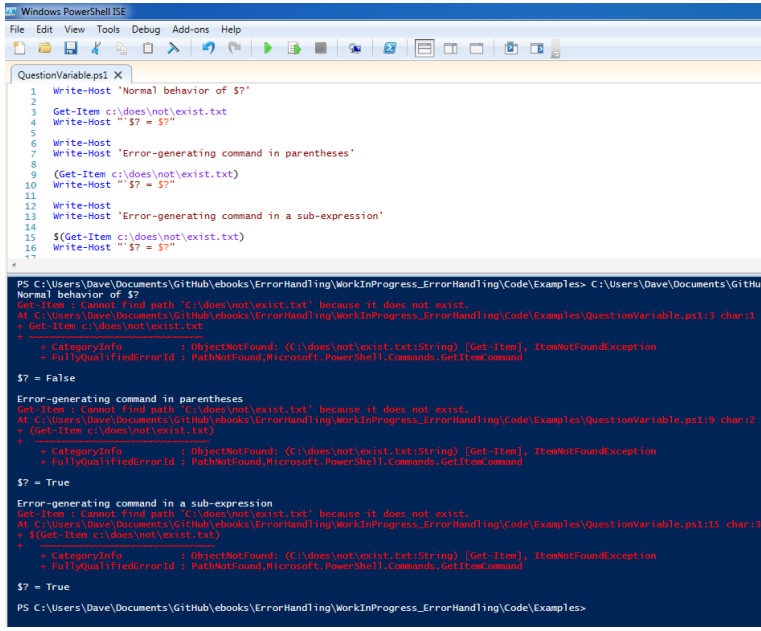

image015.png

Figura 4.1: Falsos positivos con la variable $?

Determinar qué tipos de errores puede producir un comando

Antes de que pueda decidir la mejor manera de manejar los errores de un comando en particular, a menudo necesitará saber qué tipo de errores puede producir. ¿Terminating o Non-Terminating? ¿Cuáles son los tipos de excepción que se pueden producir? Desafortunadamente, la documentación del Cmdlet de PowerShell no proporciona esta información, por lo que necesita recurrir a algún tipo de prueba y error. Aquí hay un ejemplo de cómo puede averiguar si los errores de un Cmdlet son Terminating or Non-Terminating:

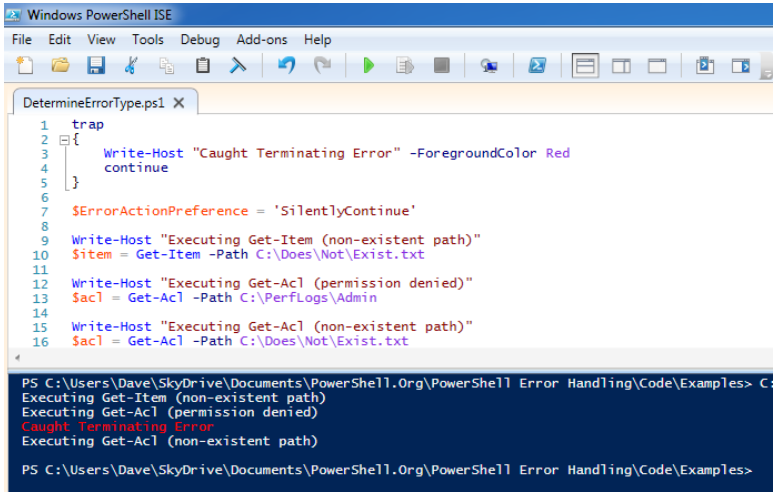

image016.png

Figura 4.2: Identificación de errores

Irónicamente, este era un lugar práctico tanto para usar la sentencia Trap como para establecer $ErrorActionPreference a SilentlyContinue, cosas que casi nunca haría en un script de producción. Como se puede ver en la figura 4.2, Get-Acl produce excepciones Terminating cuando el archivo existe, pero el Cmdlet no puede leer el ACL. Get-Item y Get-Acl producen errores Non-Terminating si el archivo no existe.

Pasar por este tipo de ensayo y error puede ser un proceso que consume mucho tiempo, sin embargo, es necesario que conozca las diferentes formas en que un comando puede fallar y, a continuación, reproducir esas condiciones para ver si el error resultante era Terminating o Non-Terminating. Como resultado de lo molesto que puede ser, además de este libro electrónico, el repositorio de Github contendrá una hoja de cálculo con una lista de errores Terminating conocidos de algunos Cmdlets. Será un documento "vivo", posiblemente convertido en un wiki en algún momento, pero probablemente nunca será una referencia completa, debido a la gran cantidad de Cmdlets de PowerShell que existen por ahí,

aunque esto es mucho mejor que nada.

Además de saber si los errores son Terminating o Non- Terminating, es posible que también desee conocer qué tipos de excepciones se producen. La figura 4.3 muestra cómo puede enumerar los tipos de excepción que están asociados con diferentes tipos de errores. Cada objeto de excepción puede contener opcionalmente una InnerException, y puede usar cualquiera de ellos en un bloque Catch o Trap:

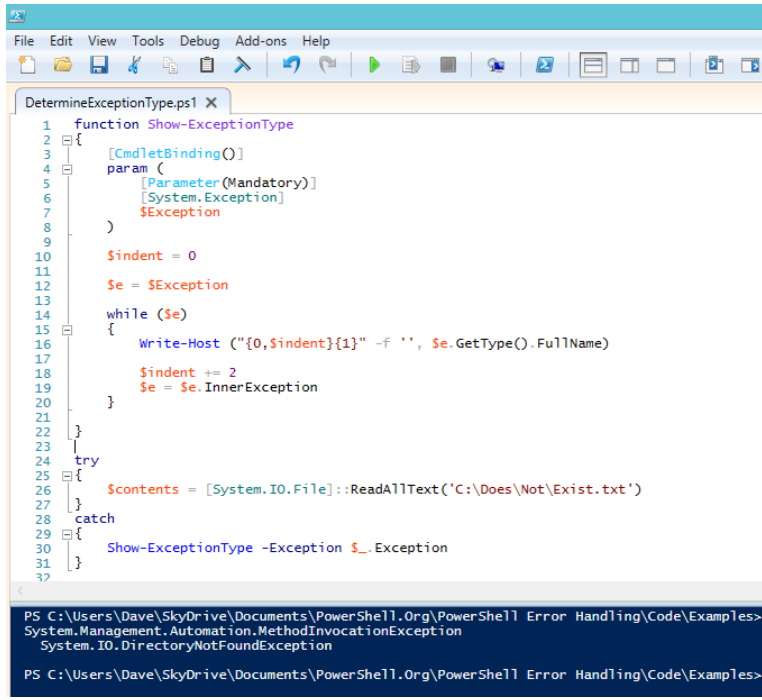

image017.png

Figura 4.3: Visualización de los tipos de Excepciones y de cualquier InnerException.

Tratamiento de errores Terminating

Esta es la parte fácil. Sólo use try/catch, y consulte $_ o $Error[0] en sus bloques Catch para obtener información sobre el error.

Tratamiento de errores Non-Terminating

Tiendo a clasificar los comandos que pueden producir errores Non-Terminating (Cmdlets, funciones y secuencias de comandos) de una de tres maneras: comandos que necesitan procesar un solo objeto de entrada, comandos que sólo pueden producir errores Non-Terminating y comandos que podrían producir errores Terminating o Non-Terminating. Suelo manejar cada una de estas categorías de las siguientes formas:

Si el comando sólo necesita procesar un único objeto de entrada, como en la figura 4.4, uso ErrorAction en Stop y manejo los errores en un bloque Try /Catch. Debido a que el Cmdlet sólo trata con un único objeto de entrada, el concepto de un error Non-Terminating no es terriblemente útil de todos modos.

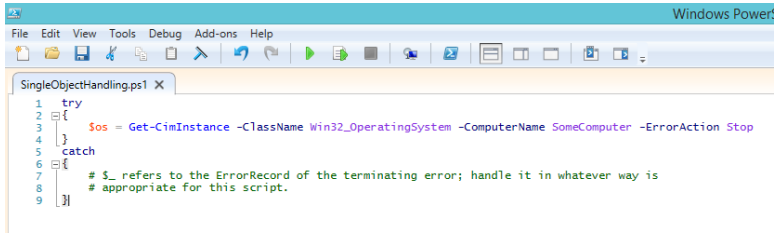

image018.png

Figura 4.4: Utilizar Try/Catch y ErrorAction en Stop cuando se trata de un solo objeto.

Si el comando sólo produce errores Non-Terminating, utilizó ErrorAction, pero esta categoría es más grande de lo que usted pensaría. La mayoría de los errores de un Cmdlet de PowerShell son Non-Terminating:

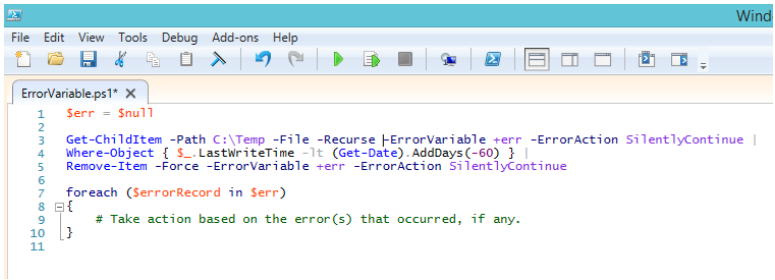

image019.png

Figura 4.5: Uso de ErrorVariable en errores Terminating.

Cuando está examinando el contenido de ErrorVariable, recuerde que normalmente puede obtener información útil acerca de lo que falló al examinar la propiedad CategoryInfo.Activity del objeto ErrorRecord (cuyo Cmdlet produjo el error) y la propiedad TargetObject (cuyo objeto estaba procesando cuando el error ocurrió). Sin embargo, no todos los Cmdlets rellenan el ErrorRecord con un TargetObject, por lo que querrá realizar algunas comprobaciones para determinar cuán útil será esta técnica. Si encuentra una situación en la que un Cmdlet debe estar informándole sobre el TargetObject pero no lo hace, considere cambiar su estructura de código para procesar un objeto a la vez, como se muestra en la figura 4.4. De esa manera, ya sabrá qué objeto se está procesando.

Surge un escenario más complicado si un comando en particular puede producir errores Terminating y Non- Terminating. En esas situaciones, si es práctico, intentó cambiar mi código para llamar al comando en un objeto a la vez. Si se encuentra en una situación en la que esto no es deseable (aunque me parece difícil de encontrar un ejemplo), recomiendo el siguiente enfoque para evitar el comportamiento peculiar de ErrorVariable y evitar llamar a $Error.Clear():

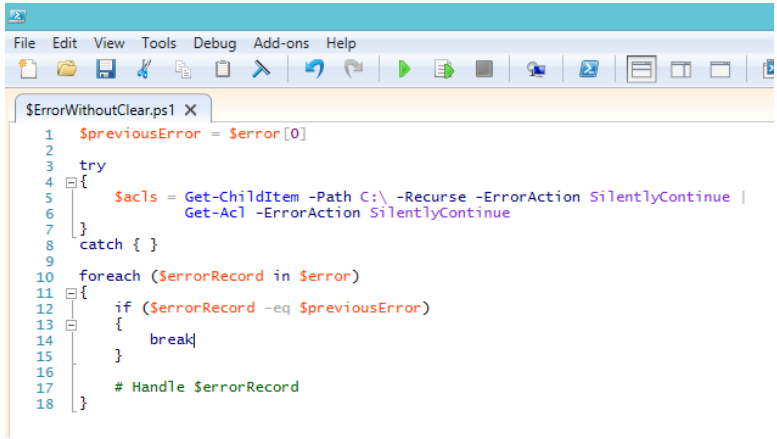

image020.png

Figura 4.6: usando $Error sin llamar a Clear() e ignorando los registros de errores previamente existentes.

Como se puede ver, la estructura de este código es casi igual que cuando se utiliza el parámetro ErrorVariable, con la adición de un bloque Try alrededor del código "problemático" y el uso de la variable $ previousError para asegurarnos de que sólo estamos reaccionando a nuevos errores en la colección $Error. En este caso, tengo un bloque Catch vacío, porque el error Terminating (si se produce) va a ser añadido también a $Error y manejado en el bucle foreach de todos modos. Es posible que prefiera manejar el error Terminating en el bloque Catch y los errores Non- Terminating en el bucle. De cualquier manera funciona.

Llamando a programas externos

Cuando necesite llamar a un ejecutable externo, la mayor parte del tiempo obtendrá los mejores resultados comprobando $LASTEXITCODE, sin embargo, tendrá que asegurarse que el programa externo devuelve información útil a través de su código de salida.

Hay algunos ejecutables "raros" por ahí que siempre devuelven 0, independientemente de si se encontraron o no errores.

Si un ejecutable externo escribe algo en el flujo StdErr, PowerShell a veces se percata de esto y envuelve el texto en un ErrorRecord, pero este comportamiento no parece ser consistente. No estoy seguro aún en qué condiciones se producirán estos errores, por lo que tiendo a utilizar $LASTEXITCODE cuando necesito establecer si un comando externo funcionó o no.

Epílogo

¡Esperamos que haya encontrado útil esta guía! Esto siempre va a ser un trabajo en progreso; así que a medida que la gente aporte materiales y sugerencias, incorporaremos lo mejor que podamos y publicaremos una nueva edición.