

Los errores críticos

Vayamos ahora a la caza de los errores críticos, que se conocen comúnmente como «excepciones». Es así como trataremos de llamarlos para distinguirlos de los errores no críticos.

Gracias a lo que vamos a descubrir en este apartado, podremos tener aún más dominio sobre nuestros scripts. Así en lugar de ver un script detenerse bruscamente a causa de un error crítico, vamos a poder actuar en consecuencia y adoptar las medidas (correctivas o alternativas) necesarias. En efecto, a veces es útil para un script saber si todo se ejecuta con normalidad. Pero para ello tendremos que intentar prever las excepciones antes de que se produzcan...


1. Interceptación de los errores críticos

La caza de las excepciones es un juego especialmente interesante que se práctica mediante la creación de «gestores de interceptación» (el termino americano correspondiente es «trap handler») utilizando la palabra clave «trap».

La sintaxis es la siguiente:

```
trap [Tipo de error a interceptar]
{ ... bloque de código a ejecutar en caso de error ...;
[break | continue] }
```

No está obligado a especificar el tipo de error pero si lo omite, todos los errores se capturarán sin distinción alguna. La indicación del tipo de error permite tener un control más preciso sobre el desarrollo del script.

 Por una vez los creadores de PowerShell no han optado por el modelo verbo-nombre en la definición de este comando. En efecto, «trap» no es un commandlet sino una instrucción.

Antes de tratar un ejemplo concreto, debe recordar que PowerShell trabaja con ámbitos («scopes»). Éstos son muy importantes en el marco de la interceptación de errores. Recuerde, el intérprete PowerShell representa el ámbito global. Cuando ejecuta un script, éste crea un nuevo ámbito en el cual se ejecutará. Del mismo modo, cada función del script se ejecuta en un ámbito propio. Lo mismo para cada bloque de instrucciones, generalmente encuadrado por llaves: {<bloque de instrucciones>}. Hay que tener claro que PowerShell administra varios niveles de ámbitos.

Cuando se crea un gestor de interceptación se nos ofrecen varias opciones para su posicionamiento:


- Se sitúa en un ámbito específico, en cuyo caso su acción se limitará a este ámbito.
- Se sitúa en el ámbito global, así podrá interceptar todas las excepciones, incluidas las generadas en los subámbitos (o ámbitos hijo).

Hay que saber que cuando se genera una excepción, PowerShell procura pasarla al gestor del ámbito en curso; y más concretamente al cuerpo del gestor (la parte entre llaves). Si no encuentra ningún gestor en su ámbito en curso, PowerShell tratará de pasar la excepción al administrador del ámbito padre, y así sucesivamente, hasta llegar al administrador del ámbito global.

Cuando el gestor intercepta una excepción , las instrucciones contenidas en su cuerpo se ejecutan. En esta situación, podemos decirle al script si debe continuar normalmente su ejecución con la instrucción `continue`, o bien si debe detenerse mediante la instrucción `break`. Si no especifica nada, el comportamiento del gestor dependerá del estado de la variable `$ErrorActionPreference`.

Valor de <code>\$ErrorActionPreference</code>	Comportamiento de la instrucción <code>Trap</code>
Stop	Break: muestra los mensajes de error.
Continue	Continue: muestra los mensajes de error.

SilentlyContinue	Continue: no muestra los mensajes de error.
Inquire	Pide confirmación para cada error crítico interceptado.

 Con el fin de eliminar cualquier posible ambigüedad en los diferentes modos de ejecución de los gestores de interceptación, pero también para obtener una mayor flexibilidad, le recomendamos sistemáticamente especificar `continue` o `break`. Puesto que la interceptación de errores no es la cosa más evidente, es mejor ser lo más explícito posible para no tener sorpresas. Por otro lado, cuando trabaje nuevamente con los scripts que escribí varios meses atrás, comprenderá el sentido de estas palabras... Al forzar explícitamente «el modo» `continue` en una instrucción `trap`, los mensajes de errores no se mostrarán. Es una pequeña sutileza que es útil conocer.

Obsérvese que en caso de parada de la ejecución de un bloque de script con `break`, la excepción, además de pasarse al administrador del ámbito en curso, se transmite también a los gestores padres (de nivel superior). Mientras que con `continue`, la excepción no se transmite al gestor del nivel superior y el script continua su curso.

Aquí tiene una pequeña serie de ejemplos para ilustrar todos estos temas.

Ejemplo 1:

Para comprender bien «Continue»...


En este ejemplo, vamos a hacer un bucle sobre un índice que variará de -2 a 2, y posteriormente dividiremos el número 100 por este índice. El objetivo es provocar un error (crítico) de división por cero. Podemos señalar que tenemos dos ámbitos distintos: el del bloque y el del bucle `for`. Continuaremos voluntariamente en la iteración después del valor cero con el objetivo de ver si el bucle continua o no según la forma en que gestionemos las excepciones.

```
PS > &{
>> $ErrorActionPreference = 'continue' #no obligatorio, modo por defecto
>> for ($i=-2; $i -le 2; $i++)
>> {
>>     trap {'¡Error crítico detectado!'}
>>     100/$i
>> }
>> Write-Host 'continua...'
>> }
>>
-50
-100
¡Error crítico detectado !
Intento de dividir por cero.
En línea: 6 Carácter: 5
+         100/ <<<< $i
+ CategoryInfo          : NotSpecified: (:) [], RuntimeException
+ FullyQualifiedErrorId : RuntimeException

100
50
continua...
```

Podemos observar aquí que:

- El gestor ha detectado la excepción y ha mostrado correctamente nuestro mensaje de error personalizado.
- Se ha mostrado el mensaje de error estándar.
- El bucle no se ha detenido después de la excepción y han tenido lugar las demás iteraciones.

 Como no le ha especificado nada, nuestro gestor se ha basado en el valor de `$ErrorActionPreference` para determinar su comportamiento. Como esta variable estaba en modo `continue`, la ejecución del script siguió normalmente pero se visualiza el mensaje de error. Si hubiéramos indicado la instrucción `continue` a nuestro gestor, habríamos obtenido el mismo resultado pero esta vez sin mensaje de error.

Probemos esta vez de forzar la instrucción `continue` en nuestro gestor de interceptación y dar a nuestra variable `$ErrorActionPreference` el valor `stop`.

```
PS > &{
>>     $ErrorActionPreference = 'stop'
>>     for ($i=-2; $i -le 2; $i++)
>>     {
>>         trap {'¡Error crítico detectado!'; continue}
>>         100/$i
>>     }
>>     Write-host 'continua...'
>> }
>>
-50
-100
¡Error crítico detectado!
100
50
continua...
```

Podemos observar aquí que:

- El gestor ha detectado la excepción y ha mostrado correctamente nuestro mensaje de error personalizado.
- No se ha mostrado el mensaje de error estándar.
- El bucle no se ha detenido después de la excepción y han tenido lugar las demás iteraciones

Podemos constatar que, cualquiera que sea el valor de la variable `$ErrorActionPreference`, un gestor de interceptación no lo tiene en cuenta cuando le decimos qué tiene que hacer.

Como ultima declinación de nuestro ejemplo 1, vamos a informar `$ErrorActionPreference` con el valor `continue` y especificaremos la instrucción `Break` a nuestro gestor. Así tendremos contemplado casi todos los casos de la tabla.

```
PS > &{
>>     $ErrorActionPreference = 'continue'
>>     for ($i=-2; $i -le 2; $i++)
>>     {
>>         trap {'¡Error crítico detectado!'; break}
>>         100/$i
>>     }
>>     Write-host 'continua...'
>> }
>>
-50
-100
¡Error crítico detectado!
Intento de dividir por cero.
En línea: 6 Carácter: 5
```

```
+      100/ <<<< $i
+ CategoryInfo          : NotSpecified: (:) [],
ParentContainsErrorRecordException
+ FullyQualifiedErrorId : RuntimeException
```

Esta vez observaremos que:

- El gestor ha detectado la excepción y ha mostrado correctamente nuestro mensaje de error personalizado.
- Se ha mostrado el mensaje de error estándar.
- No se desarrollan las iteraciones siguientes a la excepción.
- El script se detiene (el texto «continua...» no se ha mostrado).

Ejemplo 2:

¡Juguemos ahora con varios gestores de interceptación!

En este ejemplo, vamos a crear dos gestores, cada uno en su propio ámbito. La idea es mostrar cómo se comportan los gestores de interceptación cuando hay uno en cada nivel diferente.

Partiremos del ejemplo anterior al que añadiremos un gestor de interceptación en el ámbito padre.



Para el ejemplo, nos contentaremos con mostrar un simple mensaje de error en pantalla. Sin embargo, en nuestros scripts en producción dirigiremos la última excepción a un archivo. De este modo, un script lanzado por una tarea planificada que se bloquea durante la noche generará un archivo de log indicando exactamente lo que ha pasado. Para recuperar la última excepción del interior de un gestor de interceptación, existe la variable `$_`. Ésta devuelve no sólo la excepción en formato texto, sino sobre todo el objeto de error propiamente (de tipo `ErrorRecord`). De este modo dispondremos de todas sus propiedades y métodos asociados.

```
PS > &{
>> $errorActionPreference = 'continue'
>> trap {"¡Excepción detectada en el ámbito padre!"; continue}
>> for ($i=-2; $i -le 2; $i++)
>> {
>>     trap {"¡Excepción detectada en el ámbito hijo!"; break}
>>     100/$i
>> }
>> Write-host 'continua...'
>> }
>>
-50
-100
¡Excepción detectada en el ámbito hijo!
¡Excepción detectada en el ámbito padre!
continua...
```

Destacaremos en este caso:

- El gestor del bucle `for` ha detectado la excepción y ha mostrado nuestro mensaje.
- Como el `break` se ha especificado en el gestor del bucle, el gestor padre (del bloque) ha detectado también la excepción y ha mostrado nuestro mensaje.

- No se ha mostrado el mensaje de error estándar.
- El bucle se interrumpió después de la excepción y las demás iteraciones no han tenido lugar.
- El script finaliza normalmente mostrando «continua...».

Como se ha producido un error en el ámbito hijo y su gestor estaba en modo **Break**, la excepción se ha propagado al ámbito padre y el intérprete ha abandonado el ámbito en curso. Puesto que el gestor de interceptación de nivel superior estaba en modo **continue**, el script continuó su ejecución sin dar lugar a ninguna visualización de error.

Pero ¿qué pasaría si hubiésemos estado en modo **Break** en el gestor padre?

```
PS > &{
>>     $ErrorActionPreference = 'continue'
>>     trap {"¡Excepción detectada en el ámbito padre!"; break}
>>     for ($i=-2; $i -le 2; $i++)
>>     {
>>         trap {"¡Excepción detectada en el ámbito padre!"; break}
>>         100/$i
>>     }
>>     Write-host 'continua...'
>> }
>>
-50
-100
¡Excepción detectada en el ámbito padre!
¡Excepción detectada en el ámbito padre!
Intento de dividir por cero.
En línea: 7 Carácter: 5
+      100/ <<<< $i
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [],
ParentContainsErrorRecordException
+ FullyQualifiedErrorId : RuntimeException
```

Y como se podía prever, el script se detuvo a la altura de la excepción y el mensaje de error estándar ha tenido lugar.

Finalmente, ¿empieza a comprender el funcionamiento de los gestores de interceptación? Muy bien, vayamos en este caso un poco más rápido.

Ejemplo 3:

¡Juguemos ahora con varios gestores de interceptación de tipos diferentes!

Cuando escribimos «de tipos diferentes» se entiende que los gestores esta vez no van a interceptar la primera excepción que venga, sino más bien interceptarán las excepciones de un tipo determinado.

Seguiremos con el ejemplo anterior, pero con la diferencia de que ahora trataremos de elegir las excepciones que nos interesen. Para ello, hemos definido nuestros gestores de interceptación en el ámbito hijo (el del **bucle for**). No hemos definido el gestor en el ámbito padre porque, de este modo podrá interceptar cualquier excepción que provenga del ámbito hijo.


Por otra parte, hemos introducido en el bucle una nueva instrucción que va a tratar de listar el contenido de un directorio que no existe. Y como por defecto un comando sólo genera errores no críticos, lo hemos forzado a generar errores críticos gracias a **\$ErrorActionPreference = 'stop'**.

En resumen, desde la primera iteración una excepción tendrá lugar debido a **Get-Childitem** pero como ésta estará interceptada por el gestor asociado y éste funciona en «modo **continue**», el script continuará desarrollándose con

normalidad. Después la excepción fatídica de la división por cero llegará a la tercera iteración, provocando la parada del script a causa de los dos gestores en «modo `break`».

```
PS > &{
>> $errorActionPreference = 'stop'
>> trap {"¡Excepción detectada en el ámbito padre!"; break}
>> for ($i=-2 ; $i -le 2; $i++)
>> {
>> trap [System.DivideByZeroException] {
>> '¡Atención, división por cero!'; break}
>> trap [System.Management.Automation.ActionPreferenceStopException]
>> {
>> "¡El directorio indicado no existe!"; continue}
>> Write-Host "--> Iteración No $i <--"
>> 100/$i
>> Get-ChildItem "D:\Scripts\Test"
>> }
>> Write-Host 'continua.'
>> }
>>
--> Iteración No -2 <--
-50
¡El directorio indicado no existe!
--> Iteración No -1 <--
-100
¡El directorio indicado no existe!
--> Iteración No 0 <--
¡Atención, división por cero!
¡Excepción detectada en el ámbito padre!
Intento de dividir por cero.
En línea: 12 Carácter: 5
+ 100/ <<<< $i
      + CategoryInfo          : NotSpecified: (:) [],
ParentContainsErrorRecordException
      + FullyQualifiedErrorId : RuntimeException
```

Ciertamente se preguntará usted cómo conocer de antemano el tipo de excepción que queremos interceptar. Pues bien, esto es lo que vamos a ver en el apartado siguiente.

 Aunque es posible transformar los errores no críticos en errores críticos, asignando el valor `stop` a `$ErrorActionPreference`, e interceptarlos como tales, no le aconsejamos hacerlo. La razón es simple: la excepción que se ha aumentado es de tipo `System.Management.Automation.ActionPreferenceStopException`. Este tipo de excepción nos indica simplemente que, a causa del valor `$ErrorActionPreference='stop'` ha tenido lugar una excepción sin indicarnos cuál es el origen. De este modo no podremos saber en un bloque de script con varios comandos, que potencialmente pueden generar este error, cuál de ellos se ha encontrado con el problema. Es la razón por la que le sugerimos que opte por los mecanismos de gestión de los errores adaptados a cada tipo.

2. Determinar el tipo de errores críticos

Para conocer el tipo de una excepción, lo más sencillo es provocarla y luego ver su tipo en las propiedades de la variable `$Error[0]` (o `$Error[1]` en ciertos casos).

Retomando el ejemplo de la división por cero:

```
PS > &{
>> $cero = 0
```

```

>> 1/$cero
>> $Error[0] | Format-List * -Force
>> }
>>
Intento de dividir por cero.
En línea: 3 Carácter: 3
+ 1/ <<<< $cero
    + CategoryInfo          : NotSpecified: (:) [], RuntimeException
    + FullyQualifiedErrorId : RuntimeException

PSMessageDetails      :
Exception              : System.Management.Automation.RuntimeException:
                        Intento de dividir por cero.
                        ---> System.DivideByZeroException: Intento de
                        dividir por cero.
                        en System.Management.Automation.ParserOps.
                        polyDiv(ExecutionContext context, Token opToken,
                        Object lval, Object rval)
                        --- Fin del seguimiento de la pila
                        de la excepción interna ---
                        en System.Management.Automation.ExpressionNode.
                        ExecuteOp(ExecutionContext context, Object le
                        ft, OperatorToken op, Object right)
                        en System.Management.Automation.ExpressionNode.
                        Execute(Array input, Pipe outputPipe, Executi
                        onContext context)
                        en System.Management.Automation.ParseTreeNode.
                        Execute(Array input, Pipe outputPipe, ArrayList&
                        resultList, ExecutionContext context)
                        en System.Management.Automation.StatementList
                        Node.ExecuteStatement(ParseTreeNode statement,
                        Array input, Pipe outputPipe, ArrayList& resultList,
                        ExecutionContext context)
TargetObject          :
CategoryInfo          : NotSpecified: (:) [], RuntimeException
FullyQualifiedErrorId : RuntimeException
ErrorDetails          :
InvocationInfo         : System.Management.Automation.InvocationInfo
PipelineIterationInfo : {}

```

Acabamos de provocar la excepción que nos interesa, y ahora, gracias al contenido de `$Error[0]` vamos a determinar que su tipo es `System.DivideByZeroException`.

Ahora, podemos por ejemplo interceptar el error de división por cero de la manera siguiente:

```

Trap [System.DivideByZeroException] {'Intento de dividir por cero ha sido
detectado!'; break}

```

3. Generación de excepciones personalizadas

Gracias a la instrucción `throw`, va a poder divertirse creando sus propias excepciones.

La sintaxis es la siguiente:

```
throw ["Texto a mostrar cuando se produce la excepción."]
```

Sepa que no está obligado a especificar una cadena de caracteres tras la instrucción `throw`. Ésta puede utilizarse sólo en su más simple expresión.

Ejemplo:

```
PS > &{
>> $errorActionPreference = 'continue'
>> trap { 'Excepción detectada : ' + $_ }

>> throw '¡Mi primera excepción personalizada!'
>> }
>>
Excepción detectada: ¡Mi primera excepción personalizada!
¡Mi primera excepción personalizada!
En línea: 4 Carácter: 6
+ throw <<<< '¡Mi primera excepción personalizada!'
+ CategoryInfo          : OperationStopped: (¡Mi primera excepción personalizada!
:String) [], RuntimeException
+ FullyQualifiedErrorId : ¡Mi primera excepción personalizada!
```

La instrucción `throw` se utiliza a menudo con funciones o scripts que requieren obligatoriamente parámetros para funcionar.

Ejemplo:

Caso de una función.

```
Function Hola ($nombre = $(throw 'Falta el parámetro -Nombre'))
{
    Write-Host "Hola $nombre" -ForegroundColor green
}
```

Si omite especificar el parámetro al iniciar la función, obtendrá esto:

```
PS > hola
Falta el parámetro -Nombre
En línea: 1 Carácter: 33
+ Function Hola ($nombre = $(throw <<<< 'Falta el parámetro -Nombre'))
+ CategoryInfo          : OperationStopped: (Falta el parámetro
-Nombre:String) [], RuntimeException
+ FullyQualifiedErrorId : Falta el parámetro -Nombre
```

Ejemplo:

Caso de un script.

Cree un script denominado por ejemplo `hola.ps1` e inserte estas líneas en su interior:

```
param([string]$nombre = $(throw 'Falta el parámetro -Nombre'))

Write-Host "Hola $nombre" -ForegroundColor green
```


Después ejecute su script del siguiente modo: `./hola.ps1 -nombre Armando`



Es posible, pero no se recomienda especialmente, sustituir el texto entre comillas después de la instrucción `throw` por un bloque de script.

Ejemplo:

```
Function Hola ($nombre = ${throw `
    &{ write-host 'Falta el parámetro -Nombre' -ForegroundColor green
        Get-Childitem c:\
    } })
{
    Write-Host "Hola $nombre" -ForegroundColor red
}
```

Los creadores de PowerShell han simplificado al máximo la forma de establecer excepciones, tenga en cuenta sin embargo que es posible pasar directamente objetos de tipo `ErrorRecord` o excepciones .NET en lugar de texto.

4. Gestionar los errores críticos con Try-Catch-Finally

Como dijimos anteriormente, cuando ocurre un error crítico, la ejecución del comando, o del script en algunos casos, se interrumpe. Para evitar tener que gestionar los errores de forma manual, PowerShell v2 añade la posibilidad de utilizar los bloques de ejecución Try, Catch y Finally para definir las secciones en las que PowerShell va a supervisar los errores. Try, Catch y Finally presentan un modo de funcionamiento probado muy bien en muchos otros lenguajes de desarrollo.

El bloque **try** contiene el código protegido susceptible de provocar la excepción. El bloque se ejecutará hasta que se produzca la excepción (ejemplo: división por cero) o hasta su finalización total. Si un error con final de ejecución se produce durante la ejecución de las instrucciones, PowerShell pasa el objeto error del bloque Try a un bloque Catch apropiado.

La sintaxis del bloque Try es la siguiente:

```
try {<bloque de instrucciones>}
```

La clausula **catch** contiene el bloque de ejecución asociado a un tipo de error interceptado. Catch puede utilizarse sin argumento. En este caso interceptará todo tipo de excepciones y se denomina la cláusula catch general.

La sintaxis del bloque Catch es la siguiente:

```
catch [<tipo de error>] {<bloque de instrucciones>}
```



Véase la sección Determinar el tipo de errores críticos en este capítulo para saber cómo conocer los tipos de errores críticos.

Opcionalmente, **Finally** se utiliza seguido de un bloque de script que se ejecuta cada vez que se ejecuta el script. Tanto si se ha interceptado un error como si no.

La sintaxis del bloque Finally es la siguiente:

```
finally {<bloque de instrucciones>}
```

Ilustraremos todo ello con un intento de división por Cero:

```
Function Bucle
```

```
{
For ($i=-2 ;$i -le 2;$i++)
{
Try { 100/$i }
Catch {Write-Host '¡Error en el script!'}
}
}
```

Resultado:

```
PS > bucle
-50
-100
¡Error en el script!
100
50
```

La operación 100/\$i se comprueba a cada paso en el bucle a fin de determinar si se produce una indicación de error. Si se produce un error, entonces se ejecuta el comando Write-Host. Sin embargo, en el ejemplo inferior, el bloque de instrucciones no tiene en cuenta el tipo de error encontrado. Para filtrar uno o varios tipos de errores, basta con precisarlo con el bloque de instrucciones, como se muestra a continuación:

```
Function Bucle
{
For ($i=-2 ;$i -le 2;$i++)
{
Try {100/$i}
Catch [System.DivideByZeroException] {
Write-Host '¡Error en el script!'}
}
}
```

Resultado:

```
PS > bucle
-50
-100
¡Error en el script!
100
50
```

De este modo, la ejecución del bloque Catch está vinculada exclusivamente a un error cometido por una división por cero.

Como sin duda habrá observado, la finalidad de la interceptación de un error crítico con Trap y la utilización de Try-Catch es similar. Sin embargo, existen algunas diferencias que detallamos a continuación:

Trap	Try/Catch
Disponible en PowerShell v1 y v2.	Únicamente disponible en PowerShell v2.
Diseñado para una utilización simple por los administradores de sistemas.	Diseñado para una utilización orientada al desarrollo.
Puede interceptar un error generado en el ámbito global del script/función.	Puede interceptar únicamente un error generado en el ámbito del bloque de instrucción TRY.