Capítulo 3
Creación de módulos

1. Introducción

La palabra "módulo" generalmente evoca la idea de "modular". Se concibe como un elemento básico central, alrededor del cual se agrupan otros elementos más pequeños que completan su funcionalidad. Estos elementos pueden eliminarse, mejorarse, añadirse o incluso transferirse.

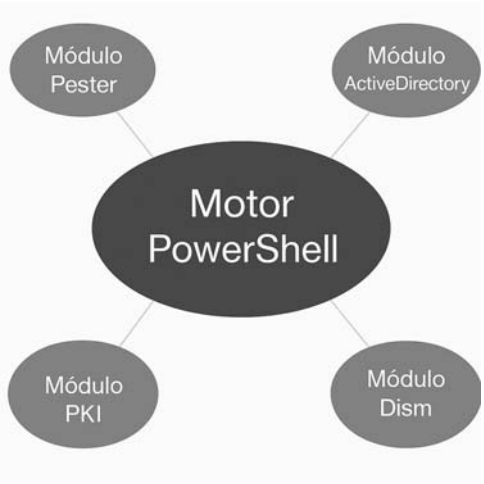

Ilustración de la modularidad en PowerShell

Lo mismo ocurre con los módulos PowerShell, aunque son algo más que simples complementos. De hecho, todos los comandos de PowerShell, especialmente los comandos básicos, están contenidos en módulos. Los módulos se introdujeron con PowerShell 2.0. Sustituyeron a los *snapins*, que aparecieron en la primera versión de PowerShell. Los snapins se siguen utilizando, pero Microsoft recomienda el uso de módulos, que son mucho más sencillos de configurar.

Los módulos permiten organizar adecuadamente el código. Sobre todo, facilitan el intercambio de código con colegas e incluso con toda la comunidad.

Observación

Cuando se escribió este libro, Microsoft tenía una lista de 11.554 módulos publicados en su plataforma de descargas PowerShell Gallery (véase el capítulo Gestión de módulos y paquetes). Esto da una idea de su verdadera relevancia y del impacto que pueden tener.

¿Qué tipos de archivos son? ¿Y qué pueden contener?

- PSM1: archivo de script. Es el archivo principal y basta con incluirlo para definir un módulo. Se ejecuta para cargar todas las funciones.
- DLL: archivo binario, normalmente compilado en C#. Puede contener cmdlets, proveedores, recursos DSC, variables o incluso clases.
- PS1: archivo de script. Este archivo generalmente declara funciones, variables o clases.
- PSD1: manifiesto del módulo. Se trata de un archivo utilizado para rellenar un gran número de elementos relativos al módulo. Se describirá con más detalle más adelante en este capítulo.
- PSRC: archivo de configuración de roles JEA (*Just Enough Administration*). Este tipo de archivo se aborda en el capítulo Gestión remota avanzada.

Existen varios tipos de módulos: módulo script, módulo binario, módulo dinámico. Todos ellos se tratarán en las secciones siguientes.

2. Módulo dinámico

La segunda opción para crear un módulo es el comando `New-Module`. Esto crea un módulo dinámico. Se carga en memoria y existe mientras dura la sesión de PowerShell. Esto significa que no se escribe en el disco y, por lo tanto, es invisible para el comando `Get-Module` (más adelante veremos cómo hacerlo visible). Sin embargo, las funciones creadas permanecen visibles a través del comando `Get-Command`.

Argumentos para el comando `New-Module`:

Parámetro	Descripción
`ArgumentList`	Especifica un arreglo de valores para pasar al parámetro `-ScriptBlock`.
`AsCustomObject`	Devuelve el módulo como un `CustomObject`cuando se crea. A continuación, se puede asignar a una variable. Las funciones o cmdlets exportados se presentan como métodos de ese objeto.
`Cmdlet`	Filtra los cmdlets exportados desde el módulo. Para ello, es necesario especificar una lista con el nombre de los cmdlets a exportar. Por defecto, se exportan todos los cmdlets. Se admiten *wildcards*.
`Function`	Filtra las funciones exportadas desde `-ScriptBlock`. Misma sintaxis que el parámetro `-Cmdlet`.
`Name`	Especifica un nombre de módulo. Como es obligatorio, si el valor es nulo o está vacío, PowerShell genera un nombre de configuración aleatorio. Su nombre empieza por "`_DynamicModule`". Irá seguido de un GUID.
`ReturnResult`	Devuelve la salida del bloque de script, así como la creación del módulo.

Parámetro	Descripción
`ScriptBlock`	Indica el contenido del módulo dinámico. Este bloque está delimitado por llaves. Las funciones deben ir separadas por un salto de línea o un punto y coma. Se podría decir que equivale al archivo PSM1 de un módulo script.

Ahora se crearán dos funciones que se integrarán en el `-ScriptBlock` del módulo dinámico. La primera, `Get-PSInstalledHotFix`, enumera las actualizaciones instaladas en una estación de trabajo. La segunda, `Rename-PSWindowUITitle`, se utiliza para cambiar el nombre de la ventana de PowerShell. Esto puede resultar útil cuando hay muchas ventanas abiertas y resulta complicado identificarlas. Por último, también se crea un alias, `gpih`. Sirve como alias de la función `Get-PSInstalledHotFix`.

Creación del módulo dinámico

```
PS > New-Module -ScriptBlock {
   Function Get-PSInstalledHotFix {
       Get-WmiObject -Class Win32_QuickFixEngineering |
Select HotFixID,InstalledOn,Description
   }
   Function Rename-PSWindowUITitle ($NewName) {
       $host.UI.RawUI.WindowTitle = $NewName
   }
   Set-Alias -Name gpih -Value Get-PSInstalledHotFix
}

ModuleType Version  Name                        ExportedCommands
---------- -------  ----                        ----------------
Script     0 0       __DynamicModule_9a4c... {Get-PSInstalledHo...
```

No se ha especificado el parámetro `-Name`. Como resultado, PowerShell generó un nombre aleatorio.

Para listar los comandos

```
PS > Get-Command -Module "__DynamicModule_*"

CommandType  Name                   Version  Source
-----------  ----                   -------  ------
Function     Get-PSInstalledHotFix  0.0      __DynamicModule_...
```

```
Function     Rename-PSWindowUITitle 0.0      __DynamicModule_...
```

Los dos comandos del módulo están presentes. ¿Puede decirse lo mismo del alias?

```
PS >  get-alias gpih
get-alias : Cette commande ne trouve pas d'alias correspondant,
car l'alias avec name « gpih » n'existe pas.
Au caractère Ligne:1 : 1
+ get-alias gpih
+ ~~~~~~~~~~~~~~
   + CategoryInfo          : ObjectNotFound: (gpih:String)
[Get-Alias], ItemNotFoundException
   + FullyQualifiedErrorId : ItemNotFoundException,Microsoft.
PowerShell.Commands.GetAliasCommand
```

Como puede verse, se genera una excepción, lo que indica que el alias no se ha exportado junto con las funciones. Los módulos dinámicos no exportan variables ni alias. Sin embargo, es posible forzar este comportamiento utilizando el comando `Export-ModuleMember`:

```
PS > New-Module -ScriptBlock {
   Function Get-PSInstalledHotFix {
       Get-WmiObject -Class Win32_QuickFixEngineering |
Select HotFixID,InstalledOn,Description
   }
   Function Rename-PSWindowUITitle ($NewName) {
       $host.UI.RawUI.WindowTitle = $NewName
   }
   Set-Alias -Name gpih -Value Get-PSInstalledHotFix
   Export-ModuleMember -Alias gpih `
     -Function Get-PSInstalledHotFix,Rename-PSWindowUITitle
}
PS > get-alias gpih

CommandType  Name                                Source
-----------  ----                                ------
Alias        gpih -> Get-PSInstalledHotFix       __DynamicModule_9...
```

Volveremos al comando `Export-ModuleMember` más adelante en este capítulo.

Ya se ha dicho que el comando `Get-Module` no es capaz de ver un módulo dinámico. El truco consiste en añadir el cmdlet `Import-Module` a la izquierda de la creación del módulo, separado por un pipeline:

```
PS > New-Module -ScriptBlock {…} | Import-Module
PS C:\Users\Nicol> Get-Module

ModuleType Version     Name                  ExportedCommands
---------- -------     ----                  ----------------
Script     0 0         __DynamicModule_...   {Get-PSInstalledHot...
```

3. Módulo binario

Los módulos binarios son una categoría particular dentro de los módulos de PowerShell. Contienen código compilado en un lenguaje .NET. Pueden citarse ejemplos en C#, en VB.NET e incluso en ASP.NET. Aunque el diseño de un módulo binario es similar al de un snapin, existen diferencias importantes entre ambos. La instalación de un snapin requiere derechos de administrador, lo que no ocurre con un módulo binario. Basta con copiar los archivos, como sucede con los módulos de script (salvo que se copien en una carpeta que requiera privilegios elevados, como Program Files).

Diseñar cmdlets en un lenguaje .NET resulta especialmente útil. Proporciona un tiempo de ejecución más rápido que las funciones PowerShell. Sin embargo, descubrir estas funciones está fuera del alcance de este libro. Está dirigido a desarrolladores. No obstante, en términos generales, se recomienda encarecidamente interesarse por el framework .NET, especialmente si se busca constantemente el rendimiento en los scripts y funciones de PowerShell. .NET ofrece una gran variedad de clases que permiten ahorrar una cantidad considerable de tiempo.

4. Módulo script

Los módulos script son los más conocidos porque son rápidos y fáciles de crear. También son sencillos de desplegar y de utilizar.

Crear un módulo es relativamente sencillo. Se puede almacenar en diferentes lugares. Para ello, conviene comprobar el contenido de la variable de entorno `$Env:PSModulePath`. Si el módulo creado no está contenido en uno de los directorios de esta variable, entonces no será descubierto automáticamente por PowerShell. También es posible modificar esta variable para añadir sus propias rutas de acceso.

Observación

Nota: si modifica esta variable, asegúrese de conservar los valores predeterminados ya existentes.

Si se decide prescindir de`$Env:PSModulePath`, siempre es posible importar el módulo manualmente, especificando la ruta completa al applet en el comando `Import-Module`. Puede consultar el contenido de la variable `$Env:PSModulePath` con el siguiente comando:

```
PS > $Env:PSModulePath.Split(';')
C:\Users\Nicol\Documents\WindowsPowerShell\Modules
C:\Program Files\WindowsPowerShell\Modules
C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules
```

Una vez encontrada la ubicación, basta con crear una carpeta con el nombre del módulo y, a continuación, un archivo PSM1. Debe tener el mismo nombre que la carpeta. Este archivo contendrá el código PowerShell, e incluso puede llamar a scripts PS1.

Ejemplo de creación de un módulo

```
PS > New-Item -Path $Home\Documents\WindowsPowerShell\Modules `
             -Name MyModuleTools -Type directory


PS > New-Item -Name MyModuleTools.psm1 -Type File `
-Path $Home\Documents\WindowsPowerShell\Modules\MyModuleTools
```

Por el momento, el módulo está vacío. A continuación, añadiremos algunas funciones útiles. También es una buena idea incluir una función que llame a una segunda función bastante larga que se utiliza regularmente.

Observación

Los usuarios de Bash en Linux suelen crear alias para definir comportamientos preestablecidos de ciertos comandos. En PowerShell, los alias no se utilizan de esta manera. Es necesario declarar una nueva función que actuará como alias y llamará a la función o comando que quieres acortar. Estas nuevas funciones se denominan generalmente funciones proxy.

En este caso, simplemente añadiremos dos funciones al archivo PSM1:

– La función `Get-PSSerialNumber`: recupera el número de serie del ordenador.

– La función `Get-PSFolderSiz`: determina el tamaño de una carpeta con todos los archivos que contiene. El valor devuelto está en megabytes.

Código de funciones

```
Function Get-PSSerialNumber {
    (Get-CIMInstance -ClassName Win32_Bios).SerialNumber
}

Function Get-PSFolderSize {
    Param(
        [ValidateScript({Test-Path $_})]
        [String]$Path
    )
    (Get-ChildItem -Path $Path -Recurse -File | `
    Measure-Object -Property Length -Sum ).Sum /1MB
}
```

Ahora es posible constatar que el módulo se ha tenido en cuenta en el descubrimiento automático de módulos por parte de PowerShell:

```
PS > Get-Module -ListAvailable

    Directorio :
C:\Users\Nicolas\Documents\WindowsPowerShell\Modules
ModuleType Version Name                    ExportedCommands
---------- ------- ----                    ----------------
Script     0.0     MyModuleTools           {Get-PSFolder...

    Directorio  :
C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules
```

```
ModuleType Version   Name                 ExportedCommands
---------- -------   ----                 ----------------
Manifest   1.0.0.0   AppBackgroundTask    {Disable-AppBackgro...
Manifest   2.0.0.0   AppLocker            {Get-AppLockerFileI...
Manifest   1.0.0.0   AppvClient           {Add-AppvClientConn...
...
```

Como se mencionó anteriormente, no es necesario importar el módulo para utilizar las funciones que acabamos de crear:

```
 PS > Get-PSFolderSize -Path .
10682,433052063
```

4.1 Conversión de un script en un módulo

Convertir un script en un módulo PowerShell es un proceso sencillo. En muchos casos, basta con cambiar la extensión del archivo a .psm1. No obstante, asegúrese de que el código esté organizado en funciones PowerShell. Veremos más adelante que es posible exportar solo unos pocos elementos de todo el código.

Ejemplo de renombramiento

```
PS > Rename-Item -Path MyFunction.ps1 -NewName MyFunction.psm1
```

Para el resto, repetiremos las instrucciones dadas en la sección anterior. En función de la política de importación de módulos configurada, es posible que deba importar el nuevo módulo manualmente, utilizando el applet de comandos `Import-Module`.

```
PS > Import-Module ./MyFunction.psm1
```

4.2 Convenciones de nomenclatura

La convención de nomenclatura recomendada por Microsoft para las funciones es `Verbo-Nombre`.

Si el nombre de la función no sigue esta convención, aparecerá un mensaje de advertencia al cargar el módulo.