Microsoft Windows Server System

Guía básica de Windows PowerShell™

Microsoft Corporation

Publicación: septiembre de 2006

Resumen

Windows PowerShell™ es un nuevo shell de línea de comandos de Windows que se ha diseñado expresamente para los administradores de sistemas. El shell incluye un símbolo del sistema interactivo y un entorno de scripting que se pueden utilizar de forma independiente o conjunta.

En este documento se describen los conceptos básicos y las funciones de Windows PowerShell, y se sugieren formas en que se puede usar Windows PowerShell para la administración de sistemas.

Microsoft

Contenido

Aviso de copyright de la Guía básica de Windows PowerShell™ 9

Introducción a Windows PowerShell 10
Público al que se dirige 10

Acerca de Windows PowerShell 10
Aprendizaje sencillo 10
Coherencia 11
Entorno interactivo y de scripting 11
Orientación a objetos 11
Transición sencilla a la creación de scripts 12

Instalar y ejecutar Windows PowerShell 12
Requisitos de instalación 12
Instalar Windows PowerShell 12
Ejecutar Windows PowerShell 13

Conceptos básicos de Windows PowerShell 13

Conceptos importantes de Windows PowerShell 14
Los comandos no están basados en texto 14
El conjunto de comandos es ampliable 15
Windows PowerShell controla la entrada y la presentación de la consola 15
Windows PowerShell utiliza sintaxis del lenguaje C# 16

Aprender los nombres de comandos y parámetros de Windows PowerShell 16
Los cmdlets utilizan nombres con verbo y sustantivo para reducir la memorización de comandos 17
Los cmdlets utilizan parámetros estándar 19
Parámetro Help (?) 19
Parámetros comunes 20
Parámetros recomendados 20

Obtener información resumida de comandos 20
Mostrar los tipos de comandos disponibles 21

Obtener información de ayuda detallada 22

Usar nombres de comandos familiares 23

Interpretación de los alias estándar 24
Crear nuevos alias 25

Completar nombres automáticamente con el tabulador 25

Canalización de objetos 27

Canalización de Windows PowerShell 27

Ver la estructura de objetos (Get-Member) 29

Usar comandos de formato para cambiar la presentación de los resultados 31
Usar Format-Wide para resultados con un único elemento 32
Controlar la presentación con columnas de Format-Wide 32
Usar Format-List para una vista de lista 32
Obtener información detallada utilizando Format-List con caracteres comodín 33
Usar Format-Table para mostrar resultados con formato de tabla 33
Mejorar el resultado obtenido con Format-Table (AutoSize) 34
Ajustar en columnas el resultado obtenido con Format-Table (Wrap) 35
Organizar los resultados con formato de tabla (-GroupBy) 36

Redirigir datos con los cmdlets Out-* 37
Dividir en páginas el resultado mostrado en la consola (Out-Host) 38
Descartar el resultado (Out-Null) 39
Imprimir datos (Out-Printer) 39
Almacenar datos (Out-File) 39

Desplazamiento por Windows PowerShell 40

Administrar la ubicación actual en Windows PowerShell 41
Obtener la ubicación actual (Get-Location) 41
Establecer la ubicación actual (Set-Location) 42
Almacenar y recuperar ubicaciones recientes (Push-Location y Pop-Location) 43

Administrar las unidades de Windows PowerShell 45
Agregar nuevas unidades de Windows PowerShell (New-PSDrive) 47
Eliminar unidades de Windows PowerShell (Remove-PSDrive) 49
Agregar y eliminar unidades fuera de Windows PowerShell 49

Trabajar con archivos, carpetas y claves del Registro 49
Enumerar archivos, carpetas y claves del Registro (Get-ChildItem) 50
Crear una lista de todos los elementos contenidos (-Recurse) 50
Filtrar elementos por nombre (-Name) 51
Forzar la presentación de los elementos ocultos (-Force) 51

Usar caracteres comodín para buscar nombres de elementos 51
Excluir elementos (-Exclude) 52
Combinar parámetros de Get-ChildItem 53

Manipular elementos directamente 54
Crear nuevos elementos (New-Item) 54
Por qué los valores del Registro no son elementos 55
Cambiar nombres de elementos existentes (Rename-Item) 56
Desplazar elementos (Move-Item) 56
Copiar elementos (Copy-Item) 57
Eliminar elementos (Remove-Item) 58
Ejecutar elementos (Invoke-Item) 58

Trabajar con objetos 59

Obtener objetos de WMI (Get-WmiObject) 59
Obtener objetos de WMI (Get-WmiObject) 59
Enumerar las clases de WMI 59
Obtener información detallada sobre las clases de WMI 61
Mostrar propiedades no predeterminadas con los cmdlets Format 62

Crear objetos .NET y COM (New-Object) 62
Usar New-Object para el acceso a registros de eventos 63
Usar constructores con New-Object 63
Almacenar objetos en variables 64
Acceso a un registro de eventos remoto con New-Object 64
Borrar un registro de eventos con métodos de objetos 65
Crear objetos COM con New-Object 65
Crear accesos directos de escritorio con WScript.Shell 66
Usar Internet Explorer desde Windows PowerShell 68
Obtener advertencias acerca de objetos COM contenidos en .NET 70

Usar clases y métodos estáticos 70
Obtener datos de entorno con System.Environment 71
Hacer referencia a la clase estática System.Environment 71
Mostrar las propiedades estáticas de System.Environment 72
Operaciones matemáticas con System.Math 73

Eliminar objetos de la canalización (Where-Object) 74
Realizar pruebas sencillas con Where-Object 75
Filtrado basado en propiedades de objetos 76

Repetir una tarea para varios objetos (ForEach-Object) 78

Seleccionar partes de objetos (Select-Object) 79

Ordenar objetos 80

Usar variables para almacenar objetos 81
Crear una variable 81
Manipular variables 82
Usar variables de Cmd.exe 82

Usar Windows PowerShell para tareas de administración 83

Administrar procesos locales 83
Mostrar la lista de procesos (Get-Process) 84
Detener procesos (Stop-Process) 85
Detener todas las demás sesiones de Windows PowerShell 86

Administrar servicios locales 87
Mostrar la lista de servicios 87
Detener, iniciar, suspender y reiniciar servicios 88

Recopilar información acerca de equipos 89
Mostrar la lista de configuraciones de escritorio 89
Mostrar información del BIOS 90
Mostrar información de procesadores 90
Mostrar el fabricante y el modelo del equipo 90
Mostrar las revisiones instaladas 91
Mostrar información de versión del sistema operativo 92
Mostrar los usuarios y propietarios locales 92
Obtener el espacio en disco disponible 92
Obtener información de sesiones iniciadas 93
Obtener el usuario que ha iniciado una sesión en un equipo 93
Obtener la hora local de un equipo 93
Mostrar el estado de un servicio 94

Trabajar con instalaciones de software 94
Mostrar las aplicaciones instaladas con Windows Installer 95
Mostrar todas las aplicaciones que se pueden desinstalar 96
Instalar aplicaciones 98
Eliminar aplicaciones 99
Actualizar aplicaciones instaladas con Windows Installer 99

Cambiar el estado del equipo: bloquear, cerrar la sesión, apagar y reiniciar 100
Bloquear un equipo 100

Cerrar la sesión actual 100
Apagar o reiniciar un equipo 101

Trabajar con impresoras 101
Crear una lista de conexiones de impresora 101
Agregar una impresora de red 101
Configurar una impresora predeterminada 102
Quitar una conexión de impresora 102

Realizar tareas de red 102
Crear una lista de direcciones IP utilizadas en un equipo 102
Mostrar los datos de configuración de IP 103
Hacer ping en equipos 104
Recuperar propiedades de adaptadores de red 105
Asignar el dominio DNS para un adaptador de red 105
Realizar tareas de configuración de DHCP 106
Determinar los adaptadores con DHCP habilitado 106
Recuperar propiedades de DHCP 106
Habilitar DHCP en cada adaptador 106
Liberar y renovar concesiones DHCP en adaptadores específicos 107
Liberar y renovar concesiones DHCP en todos los adaptadores 107
Crear un recurso compartido de red 108
Eliminar un recurso compartido de red 108
Conectar una unidad de red accesible desde Windows 109

Trabajar con archivos y carpetas 109
Mostrar todos los archivos y carpetas que contiene una carpeta 109
Copiar archivos y carpetas 110
Crear archivos y carpetas 111
Eliminar todos los archivos y carpetas que contiene una carpeta 111
Asignar una carpeta local como una unidad accesible desde Windows 111
Leer un archivo de texto en una matriz 112

Trabajar con claves del Registro 112
Mostrar todas las subclaves de una clave del Registro 113
Copiar claves 114
Crear claves 114
Eliminar claves 115
Eliminar todas las claves contenidas en una clave específica 115

Trabajar con entradas del Registro 115
Mostrar las entradas del Registro 116

Obtener una sola entrada del Registro 117
Crear nuevas entradas del Registro 118
Cambiar el nombre de entradas del Registro 119
Eliminar entradas del Registro 120

Apéndice 1: Alias de compatibilidad 120

Apéndice 2: Crear accesos directos personalizados de PowerShell 121

Aviso de copyright de la Guía básica de Windows PowerShell™

Introducción a Windows PowerShell

Windows PowerShell es un shell de línea de comandos y un entorno de scripting que ofrece la eficacia de .NET Framework a los usuarios de la línea de comandos y a los creadores de scripts. Windows PowerShell presenta diversos conceptos nuevos y eficaces que permiten ampliar los conocimientos adquiridos y los scripts creados en los entornos del símbolo del sistema de Windows y Windows Script Host.

Público al que se dirige

La Guía básica de Windows PowerShell está destinada a los profesionales de tecnologías de la información (TI), programadores y usuarios avanzados que carezcan de conocimientos sobre Windows PowerShell. Aunque los conocimientos sobre creación de scripts y WMI ayudan, no se presuponen ni son necesarios para entender este documento.

Acerca de Windows PowerShell

Windows PowerShell se ha diseñado para mejorar el entorno de línea de comandos y scripting mediante la eliminación de antiguos problemas y la incorporación de nuevas funciones.

Aprendizaje sencillo

Windows PowerShell permite descubrir fácilmente sus funciones. Por ejemplo, para obtener una lista de los cmdlets que permiten ver y cambiar servicios de Windows, escriba:

```
get-command *-service
```

Después de descubrir qué cmdlet realiza una tarea, puede obtener más información acerca del cmdlet en cuestión mediante el cmdlet Get-Help. Por ejemplo, para mostrar la Ayuda acerca del cmdlet Get-Service, escriba:

```
get-help get-service
```

Para entender totalmente el resultado de este cmdlet, canalice dicho resultado al cmdlet Get-Member. Por ejemplo, el siguiente comando muestra información acerca de los miembros del objeto que da como resultado el cmdlet Get-Service:

```
get-service | get-member
```

Coherencia

La administración de sistemas puede ser una tarea compleja, y herramientas con una interfaz común ayudan a controlar esta complejidad inherente. Por desgracia, ni las herramientas de línea de comandos ni los objetos COM que se pueden utilizar en scripts destacan por su coherencia.

La coherencia de Windows PowerShell constituye uno de sus valores principales. Por ejemplo, si aprende a utilizar el cmdlet Sort-Object, podrá usar estos conocimientos para ordenar el resultado de cualquier cmdlet. No es necesario que aprenda las distintas rutinas de ordenación de cada cmdlet.

Además, los programadores de cmdlets no tienen que diseñar funciones de ordenación para sus cmdlets. Windows PowerShell les proporciona un marco de trabajo con las funciones básicas y les obliga a ser coherentes en muchos aspectos de la interfaz. Este marco de trabajo elimina algunas de las opciones que suelen dejarse en manos del programador pero, a cambio, la programación de cmdlets eficaces y fáciles de usar se vuelve una tarea mucho más sencilla.

Entorno interactivo y de scripting

Windows PowerShell combina un entorno interactivo con un entorno de scripting que ofrece acceso a herramientas de línea de comandos y objetos COM, y permite aprovechar la eficacia de la biblioteca de clases de .NET Framework (FCL).

Este entorno combinado mejora el símbolo del sistema de Windows, que proporciona un entorno interactivo con varias herramientas de línea de comandos. También mejora los scripts de Windows Script Host (WSH), que permiten utilizar varias herramientas de línea de comandos y objetos de automatización COM, pero que no proporcionan un entorno interactivo.

Al combinar el acceso a todas estas características, Windows PowerShell amplía la capacidad del usuario interactivo y del creador de scripts, además de facilitar la administración de sistemas.

Orientación a objetos

Aunque la interacción con Windows PowerShell se realiza mediante comandos de texto, Windows PowerShell está basado en objetos, no en texto. El resultado de un comando es un objeto. Puede enviar el objeto de salida como entrada a otro comando. En consecuencia, Windows PowerShell proporciona una interfaz familiar a usuarios con experiencia en otros shells, al tiempo que introduce un nuevo y eficaz paradigma

de línea de comandos. Asimismo, extiende el concepto de envío de datos entre comandos al permitir enviar objetos en lugar de texto.

Transición sencilla a la creación de scripts

Windows PowerShell facilita la transición desde escribir comandos de forma interactiva a crear y ejecutar scripts. Puede escribir comandos en el símbolo del sistema de Windows PowerShell para descubrir los comandos que realizan una tarea. A continuación, puede guardar estos comandos en una transcripción o un historial y después copiarlos en un archivo para utilizarlos como un script.

Instalar y ejecutar Windows PowerShell

Requisitos de instalación

Antes de instalar Windows PowerShell, asegúrese de que el sistema dispone de los programas de software que necesita Windows PowerShell. Estos programas son:

- Windows XP Service Pack 2, Windows 2003 Service Pack 1 o versiones posteriores de Windows
- Microsoft .NET Framework 2.0

Si ya hay instalada una versión de Windows PowerShell en el equipo, utilice **Agregar o quitar programas** del Panel de control para desinstalarla antes de instalar una nueva versión.

Instalar Windows PowerShell

Para instalar Windows PowerShell:

1. Descargue el archivo de instalación de Windows PowerShell. (El nombre del archivo es distinto según la plataforma, el sistema operativo y el paquete de idioma utilizados.)
2. Para iniciar la instalación, haga clic en Abrir.
3. Siga las instrucciones que se muestran en las páginas del Asistente para instalación.

También puede guardar los archivos de Windows PowerShell en un recurso compartido de red para realizar la instalación en varios equipos.

Para realizar una instalación silenciosa, escriba:

```
<nombre-ejecutable-PowerShell> /quiet
```

Por ejemplo:

```
PowerShellSetup_x86_fre.exe /quiet
```

En versiones de 32 bits de Windows, Windows PowerShell se instala, de manera predeterminada, en el directorio %SystemRoot%\System32\WindowsPowerShell\v1.0. En versiones de 64 bits de Windows, se instala una versión de 32 bits de Windows PowerShell en el directorio %SystemRoot%\SystemWow64\WindowsPowerShell\v1.0 y una versión de 64 bits de Windows PowerShell en el directorio %SystemRoot%\System32\WindowsPowerShell\v1.0.

Ejecutar Windows PowerShell

Para iniciar Windows PowerShell desde el menú Inicio, haga clic en **Inicio**, **Todos los programas**, **Windows PowerShell 1.0** y, a continuación, en el icono **Windows PowerShell**.

Para iniciar Windows PowerShell desde el cuadro Ejecutar, haga clic en **Inicio**, **Ejecutar**, escriba **powershell** y, a continuación, haga clic en **Aceptar**.

Para iniciar Windows PowerShell desde una ventana del símbolo del sistema (cmd.exe), escriba **powershell**. Dado que Windows PowerShell se ejecuta en una sesión de consola, puede emplear esta misma técnica para ejecutarlo en una sesión de SSH o Telnet remota. Para volver a la sesión del símbolo del sistema, escriba **exit**.

Conceptos básicos de Windows PowerShell

Las interfaces gráficas emplean algunos conceptos básicos que conocen bien la mayoría de los usuarios, y que les ayudan a realizar las tareas. Los sistemas operativos ofrecen a los usuarios una representación gráfica de los elementos que se pueden explorar, normalmente con menús desplegables para el acceso a funciones específicas y menús contextuales para el acceso a funciones que dependen del contexto.

Una interfaz de línea de comandos (CLI), como Windows PowerShell, debe aplicar un enfoque distinto para exponer la información, ya que carece de menús o sistemas gráficos que sirvan de ayuda al usuario. Es necesario que el usuario conozca los nombres de los comandos para poder utilizarlos. Aunque puede escribir comandos complejos equivalentes a las funciones de un entorno GUI, es necesario que se familiarice con los comandos y los parámetros más usados.

La mayoría de las CLI carecen de patrones que puedan ayudar al usuario en el aprendizaje de la interfaz. Dado que las CLI fueron los primeros shells de sistemas

operativos, muchos nombres de comandos y de parámetros se seleccionaron de forma arbitraria. Generalmente se eligieron nombres concisos pero poco intuitivos. Aunque la mayoría de las CLI integran los estándares de diseño de comandos y sistemas de Ayuda, normalmente se han diseñado para ser compatibles con los comandos anteriores, por lo que los comandos siguen estando basados en decisiones que se tomaron hace décadas.

Windows PowerShell se ha diseñado para aprovechar los conocimientos históricos sobre CLI de los usuarios. En este capítulo, hablaremos sobre algunas herramientas y conceptos básicos que le ayudarán a aprender a utilizar Windows PowerShell rápidamente. Entre ellos, se incluyen los siguientes:

- Uso de Get-Command
- Uso de Cmd.exe y comandos UNIX
- Uso de comandos externos
- Completar con tabulaciones
- Uso de Get-Help

Conceptos importantes de Windows PowerShell

El diseño de Windows PowerShell integra conceptos de muchos entornos distintos. Algunos de ellos son familiares para los usuarios con experiencia en shells o entornos de programación específicos, pero muy pocos los conocen todos. Examinar detenidamente estos conceptos ofrece una útil descripción general del shell.

Los comandos no están basados en texto

A diferencia de los comandos tradicionales de una interfaz de línea de comandos, los cmdlets de Windows PowerShell están diseñados para usar objetos: información estructurada que es más que una simple cadena de caracteres que se muestra en pantalla. El resultado de los comandos contiene siempre información adicional que puede usar si es necesario. Trataremos este tema en profundidad en este documento.

Si en el pasado utilizó herramientas de procesamiento de textos para procesar los datos de línea de comandos e intenta usarlas en Windows PowerShell, observará que se comportan de forma distinta. En la mayoría de los casos, no es necesario utilizar herramientas de procesamiento de textos para extraer información específica. Puede tener acceso directamente a partes de los datos mediante comandos estándar para la manipulación de objetos de Windows PowerShell.

El conjunto de comandos es ampliable

Las interfaces como Cmd.exe no proporcionan al usuario una manera de ampliar directamente el conjunto de comandos integrados. Puede crear herramientas externas de línea de comandos que se ejecuten en Cmd.exe, pero estas herramientas externas carecen de servicios, como la integración de la Ayuda, y Cmd.exe no reconoce automáticamente que son comandos válidos.

Los comandos binarios nativos de Windows PowerShell, denominados cmdlets, se pueden ampliar con cmdlets que cree y que agregue a Windows PowerShell mediante complementos. Los complementos de Windows PowerShell se compilan, al igual que las herramientas binarias de cualquier otra interfaz. Puede utilizarlos para agregar proveedores de Windows PowerShell al shell, así como nuevos cmdlets.

Debido a la naturaleza especial de los comandos internos de Windows PowerShell, los llamaremos cmdlets.

Nota:

Windows PowerShell puede ejecutar comandos que no sean cmdlets. No los vamos a tratar detenidamente en la **Guía básica de Windows PowerShell**, pero resulta útil conocerlos como categorías de tipos de comandos. Windows PowerShell admite scripts análogos a los scripts del shell de UNIX y los archivos por lotes de Cmd.exe, pero tienen la extensión de nombre de archivo .ps1. Windows PowerShell también permite crear funciones internas que pueden utilizarse directamente en la interfaz o en scripts.

Windows PowerShell controla la entrada y la presentación de la consola

Cuando se escribe un comando, Windows PowerShell procesa siempre la entrada de la línea de comandos directamente. También aplica formato a los resultados que se muestran en pantalla. Esto es importante, ya que reduce el trabajo necesario de cada cmdlet y asegura que el usuario puede hacer siempre las cosas de la misma manera, independientemente del cmdlet que utilice. Un ejemplo de cómo esto hace la vida más fácil a los programadores de herramientas y a los usuarios es la Ayuda de la línea de comandos.

Las herramientas tradicionales de línea de comandos cuentan con sus propios esquemas para solicitar y mostrar la Ayuda. Algunas herramientas utilizan **/?** para activar la presentación de la Ayuda, mientras que otras utilizan **-?**, **/H** o incluso **//**. Algunas muestran la Ayuda en una ventana de la GUI y no en la consola. Algunas herramientas complejas, como las de actualización de aplicaciones, descomprimen archivos internos antes de mostrar la Ayuda correspondiente. Si usa un parámetro equivocado, es posible

que la herramienta pase por alto lo que ha escrito y comience a realizar una tarea automáticamente.

Cuando escriba un comando en Windows PowerShell, todo lo que escriba lo analizará y preprocesará Windows PowerShell automáticamente. Si usa el parámetro **-?** con un cmdlet de Windows PowerShell, siempre significará "muéstrame la Ayuda de este comando". Los programadores de cmdlets no tienen que analizar el comando; sólo tienen que proporcionar el texto de la Ayuda.

Es importante entender que las características de Ayuda de Windows PowerShell están disponibles incluso cuando se ejecutan herramientas tradicionales de línea de comandos en Windows PowerShell. Windows PowerShell procesa los parámetros y analiza los resultados para las herramientas externas.

Nota:

Si ejecuta una aplicación gráfica en Windows PowerShell, se abrirá la ventana de la aplicación. Windows PowerShell interviene únicamente al procesar la entrada de línea de comandos proporcionada o la salida de la aplicación que se devuelve a la ventana de la consola; no interviene en el funcionamiento interno de la aplicación.

Windows PowerShell utiliza sintaxis del lenguaje C#

Windows PowerShell incluye palabras clave y funciones de sintaxis muy parecidas a las que se usan en el lenguaje de programación C#, ya que también se basa en .NET Framework. Aprender a utilizar Windows PowerShell facilita mucho el aprendizaje de C#, si está interesado en este lenguaje.

Si no es un programador de C#, esta similitud no es importante. No obstante, si ya está familiarizado con C#, las similitudes pueden facilitar enormemente el aprendizaje de Windows PowerShell.

Aprender los nombres de comandos y parámetros de Windows PowerShell

Para la mayoría de las interfaces de línea de comandos, hay que dedicar mucho tiempo a aprender los nombres de comandos y parámetros. El problema es que hay muy pocos patrones que seguir, por lo que la única manera es memorizar cada comando y cada parámetro que se vaya a utilizar con frecuencia.

Cuando se trabaja con un comando o parámetro nuevo, no se puede usar normalmente lo que ya se sabe; es necesario buscar y aprender un nombre nuevo. Si observa cómo

aumentan de tamaño las interfaces, desde un pequeño conjunto de herramientas hasta funciones, con adiciones progresivas, es fácil entender por qué la estructura no está normalizada. En lo que respecta a los nombres de comandos en concreto, esto puede parecer lógico puesto que cada comando es una herramienta independiente, pero hay una manera mejor de tratar los nombres de comandos.

La mayoría de los comandos se crean para administrar elementos del sistema operativo o aplicaciones, como servicios o procesos. Los comandos tienen nombres diversos, que pueden ajustarse o no a un grupo. Por ejemplo, en sistemas Windows, se pueden utilizar los comandos **net start** y **net stop** para iniciar o detener un servicio. También hay otra herramienta de control de servicios más generalizada para Windows con un nombre totalmente distinto, **sc**, que no encaja en el patrón de nomenclatura de los comandos de servicio **net**. Para la administración de procesos, Windows cuenta con el comando **tasklist** para enumerar procesos y con el comando **taskkill** para eliminar procesos.

Los comandos con parámetros tienen especificaciones irregulares para estos últimos. No se puede usar el comando **net start** para iniciar un servicio en un equipo remoto. El comando **sc** inicia un servicio en un equipo remoto pero, para especificar este último, es necesario escribir dos barras diagonales inversas como prefijo del nombre. Por ejemplo, para iniciar el servicio de cola de impresión en un equipo remoto llamado DC01, debe escribir **sc \\DC01 start spooler**. Para obtener una lista de las tareas que se están ejecutando en DC01, deberá utilizar el parámetro **/S** (de "sistema") y proporcionar el nombre DC01 sin las barras diagonales inversas: **tasklist /S DC01**.

Aunque hay importantes diferencias técnicas entre un servicio y un proceso, ambos son ejemplos de elementos fáciles de administrar en un equipo con un ciclo de vida bien definido. Quizá desee iniciar o detener un servicio o proceso, u obtener una lista de todos los servicios o procesos en ejecución actualmente. En otras palabras, aunque un servicio y un proceso son cosas distintas, las acciones que realizamos en un servicio o proceso son a menudo las mismas, conceptualmente hablando. Además, las elecciones que realicemos para personalizar una acción mediante parámetros pueden ser también conceptualmente parecidas.

Windows PowerShell aprovecha estas similitudes para reducir el número de nombres distintos que el usuario necesita conocer para entender y usar los cmdlets.

Los cmdlets utilizan nombres con verbo y sustantivo para reducir la memorización de comandos

Windows PowerShell utiliza un sistema de nombres con la estructura "verbo-sustantivo": el nombre de cada cmdlet consta de un verbo estándar y un sustantivo concreto unidos por un guión. Los verbos de Windows PowerShell no siempre están en inglés, pero expresan acciones concretas en Windows PowerShell. Los sustantivos son muy parecidos a los de cualquier idioma, ya que describen tipos de objetos concretos

que son importantes para la administración del sistema. Resulta muy fácil entender cómo estos nombres que constan de dos partes reducen el esfuerzo de aprendizaje si observamos varios ejemplos de verbos y sustantivos.

Los sustantivos están menos limitados, pero deben describir siempre a qué se aplica un comando. Windows PowerShell incluye comandos como **Get-Process**, **Stop-Process**, **Get-Service** y **Stop-Service**.

En el caso de dos sustantivos y dos verbos, la coherencia no simplifica tanto el aprendizaje. No obstante, en el caso de un conjunto estándar de 10 verbos y 10 sustantivos, tendría solamente 20 palabras que aprender, pero éstas se pueden usar para formar 100 nombres de comandos distintos.

A menudo se reconoce la función de un comando con sólo leer su nombre, y suele ser evidente el nombre que debe utilizarse para un comando nuevo. Por ejemplo, un comando que apaga el equipo podría ser **Stop-Computer**. Un comando que enumera todos los equipos de una red podría ser **Get-Computer**. El comando que obtiene la fecha del sistema es **Get-Date**.

Puede obtener una lista de todos los comandos que incluyen un verbo concreto con el parámetro **-Verb** de **Get-Command** (trataremos **Get-Command** en profundidad en la siguiente sección). Por ejemplo, para ver todos los cmdlets que utilizan el verbo **Get**, escriba:

```
PS> Get-Command -Verb Get
CommandType     Name                              Definition
-----------     ----                              ----------
Cmdlet          Get-Acl                           Get-Acl [[-Path] <String[]>]...
Cmdlet          Get-Alias                         Get-Alias [[-Name] <String[]...
Cmdlet          Get-AuthenticodeSignature         Get-AuthenticodeSignature [-...
Cmdlet          Get-ChildItem                     Get-ChildItem [[-Path] <Stri...
...
```

El parámetro **-Noun** es incluso más útil porque permite ver una familia de comandos que se aplican al mismo tipo de objeto. Por ejemplo, si desea ver qué comandos están disponibles para administrar servicios, escriba el siguiente comando:

```
PS> Get-Command -Noun Service
CommandType     Name                              Definition
-----------     ----                              ----------
Cmdlet          Get-Service                       Get-Service [[-Name] <String...
Cmdlet          New-Service                       New-Service [-Name] <String>...
Cmdlet          Restart-Service                   Restart-Service [-Name] <Str...
Cmdlet          Resume-Service                    Resume-Service [-Name] <Stri...
Cmdlet          Set-Service                       Set-Service [-Name] <String>...
Cmdlet          Start-Service                     Start-Service [-Name] <Strin...
Cmdlet          Stop-Service                      Stop-Service [-Name] <String...
Cmdlet          Suspend-Service                   Suspend-Service [-Name] <Str...
...
```

Un comando no es necesariamente un cmdlet simplemente porque tenga un esquema de nomenclatura “verbo-sustantivo”. Un ejemplo de un comando nativo de Windows PowerShell que no es un cmdlet, pero tiene un nombre con verbo y sustantivo, es el comando para borrar el contenido de una ventana de consola, Clear-Host. El comando Clear-Host es en realidad una función interna, como puede observar si ejecuta Get-Command respecto a este comando:

```
PS> Get-Command -Name Clear-Host

CommandType     Name                              Definition
-----------     ----                              ----------
Function        Clear-Host                        $spaceType = [System.Managem...
```

Los cmdlets utilizan parámetros estándar

Como se ha indicado anteriormente, los nombres de los parámetros de los comandos utilizados en las interfaces tradicionales de línea de comandos no suelen ser coherentes. Algunos parámetros no tienen nombre. Si tienen nombre, suele ser una palabra abreviada o de un solo carácter que se pueden escribir rápidamente, pero que los usuarios nuevos no entienden fácilmente.

A diferencia de la mayoría de las interfaces tradicionales de línea de comandos, Windows PowerShell procesa los parámetros directamente y usa este acceso directo a los parámetros, junto con las directrices del programador, para normalizar los nombres de parámetros. Aunque esto no garantiza que todos los cmdlets se ajusten siempre a los estándares, sí lo fomenta.

Nota:

Los nombres de parámetros se utilizan siempre con un guión (-) como prefijo, para que Windows PowerShell los identifique claramente como parámetros. En el ejemplo de **Get-Command -Name Clear-Host**, el nombre del parámetro es **Name**, pero se escribe como -**Name**.

A continuación se describen algunas de las características generales de los usos y nombres de parámetros estándar.

Parámetro Help (?)

Cuando se especifica el parámetro **-?** en cualquier cmdlet, no se ejecuta el cmdlet, sino que se muestra la Ayuda correspondiente.

Parámetros comunes

Windows PowerShell incluye varios parámetros conocidos como los parámetros comunes. Dado que se controlan mediante el motor de Windows PowerShell, estos parámetros se comportan de la misma manera siempre que un cmdlet los implementa. Los parámetros comunes son **WhatIf**, **Confirm**, **Verbose**, **Debug**, **Warn**, **ErrorAction**, **ErrorVariable**, **OutVariable** y **OutBuffer**.

Parámetros recomendados

Los cmdlets principales de Windows PowerShell utilizan nombres estándar para parámetros similares. Aunque el uso de nombres de parámetros no es obligatorio, existen unas directrices de uso explícitas a fin de fomentar la normalización.

Por ejemplo, estas directrices recomiendan llamar **ComputerName** a un parámetro que haga referencia a un equipo por su nombre, en lugar de Server, Host, System, Node u otras palabras alternativas comunes. Algunos nombres de parámetros recomendados importantes son **Force**, **Exclude**, **Include**, **PassThru**, **Path** y **CaseSensitive**.

Obtener información resumida de comandos

El cmdlet **Get-Command** de Windows PowerShell recupera los nombres de todos los comandos disponibles. Si escribe **Get-Command** en el símbolo del sistema de Windows PowerShell, obtendrá un resultado similar al siguiente:

```
PS> Get-Command
CommandType     Name                              Definition
-----------     ----                              ----------
Cmdlet          Add-Content                       Add-Content [-Path] <String[...
Cmdlet          Add-History                       Add-History [[-InputObject] ...
Cmdlet          Add-Member                        Add-Member [-MemberType] <PS...
...
```

Este resultado es muy parecido al de la Ayuda de Cmd.exe: un resumen de los comandos internos con formato de tabla. En el extracto del resultado del comando **Get-Command** antes mostrado se especifica Cmdlet como valor de CommandType para cada comando mostrado. Un cmdlet es el tipo de comando intrínseco de Windows PowerShell, similar a los comandos **dir** y **cd** de Cmd.exe y a elementos integrados en shells de UNIX, como BASH.

En el resultado del comando **Get-Command**, todas las definiciones finalizan con puntos suspensivos (...) para indicar que PowerShell no puede mostrar todo el contenido en el

espacio disponible. Cuando Windows PowerShell muestra el resultado, le aplica formato de texto y después lo organiza de forma que los datos quepan perfectamente en una ventana de consola. Trataremos esta cuestión más adelante en la sección dedicada a los formateadores.

El cmdlet **Get-Command** tiene un parámetro **Syntax** que permite recuperar sólo la sintaxis de cada cmdlet. Escriba el comando **Get-Command -Syntax** para mostrar el resultado completo:

```
PS> Get-Command -Syntax
Add-Content [-Path] <String[]> [-Value] <Object[]> [-PassThru] [-Filter <String>]
[-Include <String[]>] [-Exclude <String[]>] [-Force] [Credential <PSCredential>]
[-Verbose] [-Debug] [-ErrorAction <ActionPreference>] [-ErrorVariable <String>] [-
OutVariable <String>] [-OutBuffer <Int32>] [-WhatIf] [-Confirm] [-Encoding
<FileSystemCmdletProviderEncoding>]

Add-History [[-InputObject] <PSObject[]>] [-Passthru] [-Verbose] [-Debug] [-
ErrorAction <ActionPreference>] [-ErrorVariable <String>] [-OutVariable
<String>][-OutBuffer <Int32>]...
```

Mostrar los tipos de comandos disponibles

El comando **Get-Command** no muestra todos los comandos disponibles en Windows PowerShell. Sólo muestra los cmdlets incluidos en la sesión de Windows PowerShell. En realidad, Windows PowerShell admite otros tipos de comandos. Los alias, funciones y scripts son también comandos de Windows PowerShell, aunque no se tratan detenidamente en la Guía básica de Windows PowerShell. Los archivos externos que son archivos ejecutables o tienen un controlador de tipo de archivo registrado también se clasifican como comandos.

Para obtener una lista de todos los elementos que se pueden invocar, escriba el siguiente comando:

```
PS> Get-Command *
```

Dado que esta lista incluye archivos externos en la ruta de búsqueda, puede contener miles de elementos. Resulta más útil examinar un conjunto reducido de comandos. Para buscar comandos nativos de otros tipos, puede usar el parámetro **CommandType** del cmdlet **Get-Command**. Aunque todavía no hemos hablado de estos tipos de comandos, puede mostrarlos si conoce el nombre de CommandType correspondiente a una clase de comandos.

Nota:

Aunque todavía no hemos tratado esta cuestión, el asterisco (*) es un carácter comodín para buscar argumentos de comandos de Windows PowerShell.

Equivale a "uno o más caracteres". Puede escribir **Get-Command a*** para buscar todos los comandos que comiencen por la letra "a". A diferencia de los caracteres comodín de Cmd.exe, el carácter comodín de Windows PowerShell también permite detectar un punto.

Para mostrar los alias especiales de las categorías de comandos (los alias son sobrenombres utilizados como alternativa a los nombres estándar de los comandos), escriba el siguiente comando:

```
PS> Get-Command -CommandType Alias
```

Para mostrar todas las funciones de Windows PowerShell, escriba el siguiente comando:

```
PS> Get-Command -CommandType Function
```

Para mostrar los scripts externos que estén en la ruta de búsqueda de Windows PowerShell, escriba el siguiente comando:

```
PS> Get-Command -CommandType ExternalScript
```

Obtener información de ayuda detallada

Windows PowerShell incluye información de ayuda detallada para todos los cmdlets. Si desea ver los temas de ayuda, utilice el cmdlet Get-Help. Por ejemplo, para ver la ayuda del cmdlet Get-Childitem, escriba:

```
get-help get-childitem
```

O bien

```
get-childitem -?
```

También puede ver los temas de ayuda página a página mediante las funciones **man** y **help**. Para utilizarlas, escriba **man** o **help** seguido del nombre del cmdlet. Por ejemplo, para mostrar la ayuda del cmdlet Get-Childitem, escriba:

```
man get-childitem
```

O bien

```
help get-childitem
```

El cmdlet Get-Help muestra también información acerca de los temas conceptuales de Windows PowerShell. Los nombres de los temas de ayuda conceptuales tienen el prefijo

"about_" (p. ej., about_line_editing). Para mostrar la lista de temas conceptuales, escriba:

```
get-help about_*
```

Para mostrar un tema de ayuda específico, escriba el nombre del tema. Por ejemplo:

```
get-help about_line_editing
```

Usar nombres de comandos familiares

El mecanismo de alias de Windows PowerShell permite a los usuarios hacer referencia a comandos mediante nombres alternativos. Los alias permiten a los usuarios con experiencia en otros shells volver a usar nombres de comandos comunes que ya conocen para realizar operaciones similares en Windows PowerShell. Aunque no vamos a tratar detenidamente los alias de Windows PowerShell, puede usarlos mientras aprende a usar Windows PowerShell.

Un alias asocia un nombre de comando con otro comando. Por ejemplo, Windows PowerShell cuenta con una función interna llamada **Clear-Host** que borra el contenido de la ventana de resultados. Si escribe el comando **cls** o **clear** en un símbolo del sistema, Windows PowerShell interpretará que se trata de un alias de la función **Clear-Host** y ejecutará la función **Clear-Host**.

Esta característica facilita a los usuarios el aprendizaje de Windows PowerShell. En primer lugar, la mayoría de los usuarios de Cmd.exe y UNIX cuentan con un amplio repertorio de comandos que ya conocen por su nombre y, aunque los comandos equivalentes en Windows PowerShell pueden no generar exactamente los mismos resultados, se parecen lo bastante como para que los usuarios puedan utilizarlos para realizar tareas sin tener que memorizar primero los nombres de comandos de Windows PowerShell. En segundo lugar, la frustración principal a la hora de aprender a utilizar un nuevo shell cuando el usuario ya está familiarizado con otro radica en los errores debidos a la "memoria táctil". Si lleva años usando Cmd.exe, cuando tenga una pantalla llena de resultados y quiera borrarla, de manera refleja escribirá el comando **cls** y presionará la tecla ENTRAR. Sin el alias de la función **Clear-Host** en Windows PowerShell, simplemente aparecería el mensaje de error "**El término 'cls' no se reconoce como un cmdlet, función, programa ejecutable ni archivo de script.**" Y se quedaría sin saber qué hacer para borrar el contenido de la pantalla.

A continuación se muestra una pequeña lista de los comandos comunes de Cmd.exe y UNIX que puede utilizar en Windows PowerShell:

cat	dir	mount	rm
cd	echo	move	rmdir
chdir	erase	popd	sleep
clear	h	ps	sort
cls	history	pushd	tee
copy	kill	pwd	type
del	lp	r	write
diff	ls	ren	

Si observa que utiliza uno de estos comandos de manera refleja y desea conocer el nombre real del comando nativo de Windows PowerShell, puede usar el comando **Get-Alias**:

```
PS> Get-Alias cls

CommandType     Name                            Definition
-----------     ----                            ----------
Alias           cls                             Clear-Host
```

Para facilitar la lectura de los ejemplos, en la Guía básica de Windows PowerShell se evita normalmente el uso de alias. No obstante, aprender más acerca de los alias en esta temprana etapa puede resultar útil si trabaja con fragmentos arbitrarios de código de Windows PowerShell procedentes de otro origen o si desea definir sus propios alias. El resto de esta sección trata de los alias estándar y de cómo definir sus propios alias.

Interpretación de los alias estándar

A diferencia de los alias descritos anteriormente, diseñados para la compatibilidad con nombres de comandos de otras interfaces, los alias integrados en Windows PowerShell suelen estar diseñados con fines de concisión en la escritura. Estos nombres más cortos se pueden escribir rápidamente, pero no se entienden si no se sabe a qué hacen referencia.

Windows PowerShell intenta encontrar el equilibrio entre claridad y concisión al proporcionar un conjunto de alias estándar basados en nombres abreviados de verbos y sustantivos comunes. Esto permite usar un conjunto principal de alias de cmdlets comunes que se pueden leer una vez se conocen los nombres abreviados. Por ejemplo, con alias estándar, el verbo **Get** se abrevia como **g** y **Set** como **s**, mientras que el sustantivo **Item** se abrevia como **i**, **Location** como **l** y Command como cm.

A continuación se muestra un breve ejemplo para ilustrar cómo funciona esto: el alias estándar de Get-Item procede de combinar la g de Get y la i de Item: gi. El alias estándar de Set-Item procede de combinar la s de Set y la i de Item: si. El alias estándar de Get-Location procede de combinar la g de Get y la l de Location: gl. El alias estándar de Set-Location procede de combinar la s de Set y la l de Location: sl. El alias estándar de Get-Command procede de combinar la g de Get y cm de Command: gcm. No hay un cmdlet llamado Set-Command, pero si lo hubiera, podríamos deducir que el alias estándar procede de combinar la s de Set y cm de Command: scm. Además, los usuarios familiarizados con los alias de Windows PowerShell que se encuentren con scm podrían deducir que el alias corresponde a Set-Command.

Crear nuevos alias

Puede crear sus propios alias con el cmdlet Set-Alias. Por ejemplo, las siguientes instrucciones crean los alias estándar de cmdlets descritos en la sección Interpretación de los alias estándar:

```
Set-Alias -Name gi -Value Get-Item
Set-Alias -Name si -Value Set-Item
Set-Alias -Name gl -Value Get-Location
Set-Alias -Name sl -Value Set-Location
Set-Alias -Name gcm -Value Get-Command
```

Internamente, Windows PowerShell utiliza comandos como éstos durante el inicio, pero estos alias no se pueden modificar. Si intenta ejecutar realmente uno de estos comandos, aparecerá un error que indica que el alias no se puede modificar.
Por ejemplo:

```
PS> Set-Alias -Name gi -Value Get-Item

Set-Alias : No se puede escribir en el alias gi porque es de sólo lectura o
constante y no se puede escribir en él.

En línea:1 carácter:10

+ Set-Alias  <<<< -Name gi -Value Get-Item
```

Completar nombres automáticamente con el tabulador

Los shells de línea de comandos a menudo proporcionan una manera de completar automáticamente los nombres de archivos largos o comandos, lo que agiliza la escritura

de comandos y proporciona sugerencias. Windows PowerShell permite completar nombres de archivos y cmdlets con la tecla **Tabulador**.

Nota:

Esta funcionalidad está controlada por la función interna TabExpansion. Esta función se puede modificar o reemplazar, por lo que esta explicación sólo es una guía del comportamiento de la configuración predeterminada de Windows PowerShell.

Para completar automáticamente un nombre de archivo o una ruta de acceso con las opciones disponibles, escriba parte del nombre o la ruta, y presione la tecla **Tabulador**. Windows PowerShell completará automáticamente el nombre con la primera coincidencia que encuentre. Para recorrer todas las opciones disponibles, presione repetidamente la tecla **Tabulador**.

En el caso de los nombres de cmdlets, esto funciona de forma ligeramente distinta. Para completar el nombre de un cmdlet con el tabulador, escriba la primera parte completa del nombre (el verbo) y el guión que le sigue. Puede escribir un poco más del nombre para buscar una coincidencia parcial. Por ejemplo, si escribe **get-co** y presiona la tecla **Tabulador**, Windows PowerShell lo completará automáticamente como **Get-Command** (observe que también se cambian las letras mayúsculas o minúsculas del nombre a su forma estándar). Si vuelve a presionar la tecla **Tabulador**, Windows PowerShell lo reemplazará por el otro nombre de cmdlet coincidente, **Get-Content**.

Puede completar varias veces con el tabulador en una misma línea. Por ejemplo, para completar el nombre del cmdlet **Get-Content** con el tabulador, escriba:

```
PS> Get-Con<Tabulador>
```

Cuando presione la tecla **Tabulador**, el nombre del comando se ampliará hasta:

```
PS> Get-Content
```

Posteriormente, puede especificar parte de la ruta de acceso al archivo de registro de Active Setup y volver a ampliar con el tabulador:

```
PS> Get-Content c:\windows\acts<Tab>
```

Cuando presione la tecla **Tabulador**, el nombre del comando se ampliará hasta:

```
PS> Get-Content C:\windows\actsetup.log
```

Nota:

Una limitación de este procedimiento es que las tabulaciones se interpretan siempre como intentos de completar una palabra. Si copia y pega ejemplos de comandos en una consola de Windows PowerShell, asegúrese de que

la muestra no contiene tabulaciones pues, en caso contrario, los resultados serán imprevisibles y casi seguro que no serán los que pretendía obtener.

Canalización de objetos

Las canalizaciones actúan como una serie de tubos conectados. Los elementos que se desplazan por la canalización pasan a través de cada tubo. Para crear una canalización en Windows PowerShell, se conectan comandos con el operador de canalización "|" y el resultado de cada comando se utiliza como entrada del siguiente.

Las canalizaciones son probablemente el concepto más valioso de las interfaces de línea de comandos. Utilizadas correctamente, las canalizaciones no sólo reducen el esfuerzo necesario para escribir comandos complejos, sino que también permiten ver más fácilmente el flujo de trabajo de los comandos. Una útil característica relacionada con las canalizaciones es que, como se aplican a cada elemento por separado, no es necesario modificarlas en función de si la canalización va a contener cero, uno o muchos elementos. Además, cada comando de una canalización (llamado elemento de canalización) suele pasar su resultado al siguiente comando de la misma, elemento a elemento. Normalmente esto reduce la demanda de recursos de los comandos complejos y permite obtener el resultado inmediatamente.

En este capítulo, vamos a explicar en qué se diferencia la canalización de Windows PowerShell de las canalizaciones de los shells más conocidos y, a continuación, mostraremos algunas herramientas básicas que sirven para controlar los resultados de la canalización y para ver cómo funcionan las canalizaciones.

Canalización de Windows PowerShell

La canalización se puede usar prácticamente en cualquier parte de Windows PowerShell. Aunque se muestre texto en pantalla, Windows PowerShell no canaliza texto entre los comandos, sino objetos.

La notación utilizada para las canalizaciones es parecida a la que se utiliza en otros shells por lo que, a primera vista, puede no ser evidente que Windows PowerShell introduce novedades. Por ejemplo, si usa el cmdlet **Out-Host** para forzar una presentación página a página del resultado de otro comando, dicho resultado se parecerá al texto normal mostrado en pantalla, dividido en páginas:

```
PS> Get-ChildItem -Path C:\WINDOWS\System32 | Out-Host -Paging
    Directorio: Microsoft.Windows PowerShell.Core\FileSystem::C:\WINDOWS\system32
```

```
Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         2005-10-22  11:04 PM        315 $winnt$.inf
-a---         2004-08-04   8:00 AM      68608 access.cpl
-a---         2004-08-04   8:00 AM      64512 acctres.dll
-a---         2004-08-04   8:00 AM     183808 accwiz.exe
-a---         2004-08-04   8:00 AM      61952 acelpdec.ax
-a---         2004-08-04   8:00 AM     129536 acledit.dll
-a---         2004-08-04   8:00 AM     114688 aclui.dll
-a---         2004-08-04   8:00 AM     194048 activeds.dll
-a---         2004-08-04   8:00 AM     111104 activeds.tlb
-a---         2004-08-04   8:00 AM       4096 actmovie.exe
-a---         2004-08-04   8:00 AM     101888 actxprxy.dll
-a---         2003-02-21   6:50 PM     143150 admgmt.msc
-a---         2006-01-25   3:35 PM      53760 admparse.dll
<ESPACIO> página siguiente; <RETORNO> línea siguiente; Q salir
...
```

El comando Out-Host -Paging es un elemento de canalización que le resultará útil siempre que disponga de un resultado largo que desee ver poco a poco. Resulta especialmente útil si la operación hace un uso intensivo de la CPU. Dado que el procesamiento se transfiere al cmdlet Out-Host cuando hay una página completa ya lista para mostrarse, los cmdlets que le siguen en la canalización se detienen hasta que la siguiente página de resultados esté disponible. Esto se puede observar si se utiliza el Administrador de tareas de Windows para supervisar el uso que hace Windows PowerShell de la CPU y de la memoria.

Ejecute el siguiente comando: **Get-ChildItem C:\Windows -Recurse**. Compare el uso de la CPU y de la memoria con este comando: **Get-ChildItem C:\Windows -Recurse | Out-Host -Paging**. Lo que se muestra en pantalla es texto, pero es así porque es necesario representar los objetos como texto en una ventana de consola. En realidad se trata únicamente de una representación de lo que está ocurriendo dentro de Windows PowerShell. Por ejemplo, considere el cmdlet Get-Location. Si escribe **Get-Location** mientras la ubicación actual es la raíz de la unidad C, se mostrará el siguiente resultado:

```
PS> Get-Location

Path
----
C:\
```

Si Windows PowerShell canalizara el texto, al emitir un comando como **Get-Location | Out-Host**, se pasaría de **Get-Location** a **Out-Host** un conjunto de caracteres en el orden en que se muestran en pantalla. En otras palabras, si pasara por alto la información de encabezado, **Out-Host** recibiría primero el carácter '**C**', después

el carácter '**:**' y, por último, el carácter '****'. El cmdlet **Out-Host** no podría determinar el significado de los caracteres del resultado del cmdlet **Get-Location**.

En lugar de usar texto para la comunicación entre comandos de una canalización, Windows PowerShell usa objetos. Desde el punto de vista de un usuario, los objetos empaquetan la información relacionada en un formato que permite manipularla fácilmente como una unidad, así como extraer los elementos concretos que se necesiten.

El comando **Get-Location** no devuelve texto con la ruta de acceso actual. Lo que devuelve es un paquete de información, un objeto **PathInfo**, que contiene la ruta de acceso actual junto con otra información. A continuación, el cmdlet **Out-Host** envía este objeto **PathInfo** a la pantalla y Windows PowerShell decide qué información va a mostrar y cómo la va a mostrar en función de sus reglas de formato.

De hecho, la información de encabezado que da como resultado el cmdlet **Get-Location** se agrega sólo al final, como parte del proceso de aplicar formato a los datos para su presentación en pantalla. Lo que se muestra en pantalla es un resumen de la información, y no una representación completa del objeto de salida.

Dado que en el resultado de un comando de Windows PowerShell puede haber más información que la que se muestra en la ventana de la consola, ¿cómo se pueden recuperar los elementos no visibles? ¿Cómo se pueden ver los datos adicionales? ¿Cómo se pueden ver los datos en un formato distinto al que utiliza normalmente Windows PowerShell?

En el resto de este capítulo se explica cómo puede descubrir la estructura de objetos específicos de Windows PowerShell (mediante la selección de elementos concretos y la aplicación de formato a estos elementos para facilitar su presentación) y cómo puede enviar esta información a ubicaciones de salida alternativas, como archivos e impresoras.

Ver la estructura de objetos (Get-Member)

Dado que los objetos desempeñan un papel central en Windows PowerShell, hay varios comandos nativos diseñados para trabajar con tipos de objetos arbitrarios. El más importante es el comando **Get-Member**.

La técnica más sencilla para analizar los objetos que devuelve un comando es canalizar el resultado de este comando al cmdlet **Get-Member**. Este cmdlet muestra el nombre formal del tipo de objeto y una lista completa de sus miembros. A veces el número de

elementos devueltos puede ser enorme. Por ejemplo, un objeto de proceso puede tener más de 100 miembros.

Para ver todos los miembros de un objeto Process y dividir en páginas el resultado para así poder verlo todo, escriba:

```
PS> Get-Process | Get-Member | Out-Host -Paging
```

El resultado de este comando será similar al siguiente:

```
   TypeName: System.Diagnostics.Process

Name                           MemberType     Definition
----                           ----------     ----------
Handles                        AliasProperty  Handles = Handlecount
Name                           AliasProperty  Name = ProcessName
NPM                            AliasProperty  NPM = NonpagedSystemMemorySize
PM                             AliasProperty  PM = PagedMemorySize
VM                             AliasProperty  VM = VirtualMemorySize
WS                             AliasProperty  WS = WorkingSet
add_Disposed                   Method         System.Void add_Disposed(Event...
...
```

Puede hacer que esta larga lista de información sea más útil si la filtra por los elementos que desea ver. El comando **Get-Member** permite crear una lista que sólo contenga los miembros que son propiedades. Hay varios tipos de propiedades. El cmdlet muestra las propiedades de cualquier tipo si establecemos **Properties** como valor del parámetro **Get-MemberMemberType**. La lista resultante sigue siendo muy larga, pero un poco más manejable:

```
PS> Get-Process | Get-Member -MemberType Properties


   TypeName: System.Diagnostics.Process

Name                        MemberType     Definition
----                        ----------     ----------
Handles                     AliasProperty  Handles = Handlecount
Name                        AliasProperty  Name = ProcessName
...
ExitCode                    Property       System.Int32 ExitCode {get;}
...
Handle                      Property       System.IntPtr Handle {get;}
...
CPU                         ScriptProperty System.Object CPU {get=$this.Total...
...
Path                        ScriptProperty System.Object Path {get=$this.Main...
...
```

Nota:

Los valores permitidos de MemberType son AliasProperty, CodeProperty, Property, NoteProperty, ScriptProperty, Properties, PropertySet, Method, CodeMethod, ScriptMethod, Methods, ParameterizedProperty, MemberSet y All.

Hay más de 60 propiedades para un proceso. El motivo por el que Windows PowerShell suele mostrar únicamente un subconjunto de las propiedades de cualquier objeto conocido es que, si los mostrara todos, el resultado sería una cantidad de información muy difícil de manejar.

Nota:

Windows PowerShell determina cómo se va a mostrar un tipo de objeto utilizando la información almacenada en archivos XML con nombres que finalizan en .format.ps1xml. Los datos de formato correspondientes a objetos de proceso, que son objetos .NET de la clase System.Diagnostics.Process, se almacenan en PowerShellCore.format.ps1xml.

Si necesita examinar propiedades distintas de las que Windows PowerShell muestra de manera predeterminada, deberá dar formato a los datos de salida usted mismo. Para ello, puede utilizar los cmdlets de formato.

Usar comandos de formato para cambiar la presentación de los resultados

Windows PowerShell incluye un conjunto de cmdlets que permiten controlar qué propiedades se muestran de objetos específicos. Los nombres de todos los cmdlets comienzan por el verbo **Format** y permiten seleccionar una o más propiedades para mostrarlas.

Los cmdlets **Format** son **Format-Wide**, **Format-List**, **Format-Table** y **Format-Custom**. En esta Guía básica únicamente describiremos los cmdlets **Format-Wide**, **Format-List** y **Format-Table**.

Cada cmdlet de formato tiene propiedades predeterminadas que se utilizarán si el usuario no especifica las propiedades concretas que desea que se muestren. También se usa en cada cmdlet el mismo nombre de parámetro, **Property**, para especificar las propiedades que se desea mostrar. Dado que **Format-Wide** sólo muestra una propiedad, su parámetro **Property** toma sólo un valor, pero los parámetros de propiedad de **Format-List** y **Format-Table** aceptan una lista de nombres de propiedades.

Si usa el comando **Get-Process -Name powershell** con dos instancias de Windows PowerShell en ejecución, obtendrá un resultado similar al siguiente:

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    995       9    30308      27996   152     2.73   2760 powershell
    331       9    23284      29084   143     1.06   3448 powershell
```

En el resto de esta sección, exploraremos cómo utilizar los cmdlets **Format** para cambiar la manera en que se muestra el resultado de este comando.

Usar Format-Wide para resultados con un único elemento

De manera predeterminada, el cmdlet **Format-Wide** muestra sólo la propiedad predeterminada de un objeto. La información asociada a cada objeto se muestra en una sola columna:

```
PS> Get-Process -Name powershell | Format-Wide

powershell                              powershell
```

También puede especificar una propiedad que no sea predeterminada:

```
PS> Get-Process -Name powershell | Format-Wide -Property Id

2760                                    3448
```

Controlar la presentación con columnas de Format-Wide

Con el cmdlet **Format-Wide**, puede mostrar una sola propiedad cada vez. Esto resulta útil para mostrar listas sencillas que muestren sólo un elemento por línea. Para obtener una lista sencilla, establezca el valor del parámetro **Column** en 1. Para ello, escriba:

```
Get-Command Format-Wide -Property Name -Column 1
```

Usar Format-List para una vista de lista

El cmdlet **Format-List** muestra un objeto en forma de lista, con cada propiedad etiquetada y mostrada en una línea independiente:

```
PS> Get-Process -Name powershell | Format-List

Id      : 2760
Handles : 1242
CPU     : 3.03125
Name    : powershell

Id      : 3448
Handles : 328
```

```
CPU     : 1.0625
Name    : powershell
```

Puede especificar tantas propiedades como desee:

```
PS> Get-Process -Name powershell | Format-List -Property ProcessName,FileVersion
,StartTime,Id


ProcessName : powershell
FileVersion : 1.0.9567.1
StartTime   : 2006-05-24 13:42:00
Id          : 2760

ProcessName : powershell
FileVersion : 1.0.9567.1
StartTime   : 2006-05-24 13:54:28
Id          : 3448
```

Obtener información detallada utilizando Format-List con caracteres comodín

El cmdlet **Format-List** permite utilizar un carácter comodín como valor del parámetro **Property** correspondiente. De esta manera se puede mostrar información detallada. A menudo, los objetos contienen más información de la necesaria, que es el motivo por el que Windows PowerShell no muestra todos los valores de propiedades de manera predeterminada. Para que se muestren todas las propiedades de un objeto, utilice el comando **Format-List -Property ***. El siguiente comando genera más de 60 líneas de resultados de un solo proceso:

```
Get-Process -Name powershell | Format-List -Property *
```

Aunque el comando **Format-List** resulta útil para mostrar información detallada, si desea obtener información general de resultados que contienen muchos elementos, una vista de tabla más sencilla suele ser más conveniente.

Usar Format-Table para mostrar resultados con formato de tabla

Si usa el cmdlet **Format-Table** sin especificar nombres de propiedades para dar formato al resultado del comando **Get-Process**, obtendrá exactamente lo mismo que si no aplica ningún formato. El motivo es que los procesos se muestran normalmente en formato de tabla, como la mayoría de los objetos de Windows PowerShell.

```
PS> Get-Process -Name powershell | Format-Table
```

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
   1488       9    31568      29460   152     3.53   2760 powershell
    332       9    23140        632   141     1.06   3448 powershell
```

Mejorar el resultado obtenido con Format-Table (AutoSize)

Aunque la vista de tabla resulta útil para mostrar mucha información comparable, puede ser difícil de interpretar si la presentación es demasiado estrecha para los datos. Por ejemplo, si intenta mostrar la ruta de acceso del proceso, el Id., el nombre y la compañía, las columnas de ruta de acceso del proceso y de compañía aparecerán truncadas:

```
PS> Get-Process -Name powershell | Format-Table -Property Path,Name,Id,Company

Path                Name                                Id Company
----                ----                                -- -------
C:\Archivos de programa... powershell                         2836 Microsoft
Corpor...
```

Si especifica el parámetro **AutoSize** al ejecutar el comando **Format-Table**, Windows PowerShell calculará los anchos de columna en función de los datos que va a mostrar. De esta manera podrá leerse la columna **Path**, pero los valores de la columna Company se seguirán mostrando truncados:

```
PS> Get-Process -Name powershell | Format-Table -Property Path,Name,Id,Company -
AutoSize

Path                                                  Name         Id Company
----                                                  ----         -- -------
C:\Archivos de programa\Windows PowerShell\v1.0\powershell.exe powershell 2836
Micr...
```

El cmdlet **Format-Table** sigue truncando los datos, pero sólo al final de la pantalla. Se proporciona a las propiedades (salvo a la última mostrada) el espacio necesario para que el elemento de datos más largo se muestre correctamente. Si se intercambia la ubicación de **Path** y **Company** en la lista de valores de **Property**, se muestra el nombre de la compañía, pero la ruta de acceso aparece truncada:

```
PS> Get-Process -Name powershell | Format-Table -Property Company,Name,Id,Path -
AutoSize

Company               Name         Id Path
-------               ----         -- ----
Microsoft Corporation powershell 2836 C:\Archivos de programa\Windows
PowerShell\v1...
```

El comando **Format-Table** supone que cuanto más próxima esté una propiedad al principio de la lista de propiedades, más importante es. Por tanto, intenta mostrar en su totalidad las propiedades que estén más cerca del principio. Si el comando **Format-Table** no puede mostrar todas las propiedades, quitará algunas de las columnas de la presentación y mostrará una advertencia. Puede observar este comportamiento si establece **Name** como la última propiedad de la lista:

```
PS> Get-Process -Name powershell | Format-Table -Property Company,Path,Id,Name -
AutoSize

ADVERTENCIA: en la presentación no cabe la columna "Name" y se ha quitado.

Company               Path                                                    I
                                                                              d
-------               ----                                                    -
Microsoft Corporation C:\Archivos de programa\Windows
PowerShell\v1.0\powershell.exe 6
```

En el resultado anterior, la columna de Id. se muestra truncada, de forma que quepa en la lista, y los encabezados de columna se muestran apilados. El ajuste automático del tamaño de las columnas no siempre proporciona los resultados deseados.

Ajustar en columnas el resultado obtenido con Format-Table (Wrap)

Puede forzar que los datos largos de **Format-Table** se ajusten a su columna correspondiente con el parámetro **Wrap**. Es posible que los resultados no sean los esperados si se utiliza únicamente el parámetro **Wrap**, ya que éste aplica valores predeterminados si no se especifica también **AutoSize**:

```
PS> Get-Process -Name powershell | Format-Table -Wrap -Property Name,Id,Company,
Path

Name                                Id Company             Path
----                                -- -------             ----
powershell                        2836 Microsoft Corporati C:\Archivos de
programa\Wi
                                       on                  ndows PowerShell\v1
                                                           .0\powershell.exe
```

Una ventaja de usar sólo el parámetro **Wrap** es que no ralentiza demasiado el procesamiento. Si usa **AutoSize** y crea una lista recursiva de archivos de un gran sistema de directorios, es posible que este proceso tarde mucho tiempo en completarse y que consuma mucha memoria antes de que se muestren los primeros elementos de salida.

Si la carga del sistema no es un problema, entonces **AutoSize** funciona bien con el parámetro **Wrap**. A las columnas iniciales se les asigna siempre el ancho necesario

para mostrar los elementos en una línea, como cuando se especifica **AutoSize** sin el parámetro **Wrap**. La única diferencia es que se ajusta la última columna, si es necesario:

```
PS> Get-Process -Name powershell | Format-Table -Wrap -AutoSize -Property Name,I
d,Company,Path

Name         Id Company               Path
----         -- -------               ----
powershell 2836 Microsoft Corporation C:\Archivos de programa\Windows
PowerShell\v1.0\
                                      powershell.exe
```

Es posible que algunas columnas no se muestren si se especifican primero las columnas más anchas, por lo que lo más seguro es especificar primero los elementos de datos más cortos. En el siguiente ejemplo hemos especificado primero el elemento de ruta de acceso, que es bastante ancho, por lo que perdemos la última columna (**Name**) aunque apliquemos el ajuste de línea:

```
 PS> Get-Process -Name powershell | Format-Table -Wrap -AutoSize -Property Path,I
d,Company,Name

ADVERTENCIA: en la presentación no cabe la columna "Name" y se ha quitado.

Path                                                        Id Company
----                                                        -- -------
C:\Archivos de programa\Windows PowerShell\v1.0\powershell.exe 2836 Microsoft
Corporat
                                                               ion
```

Organizar los resultados con formato de tabla (-GroupBy)

Otro parámetro útil para el control de los resultados con formato de tabla es **GroupBy**. Las listas largas con formato de tabla pueden ser difíciles de comparar. El parámetro **GroupBy** agrupa el resultado en función del valor de una propiedad. Por ejemplo, podemos agrupar los procesos por compañía para facilitar la inspección, omitiendo el valor de compañía de la lista de propiedades:

```
PS> Get-Process -Name powershell | Format-Table -Wrap -AutoSize -Property Name,I
d,Path -GroupBy Company


   Company: Microsoft Corporation

Name         Id Path
----         -- ----
powershell 1956 C:\Archivos de programa\Windows PowerShell\v1.0\powershell.exe
powershell 2656 C:\Archivos de programa\Windows PowerShell\v1.0\powershell.exe
```

Redirigir datos con los cmdlets Out-*

Windows PowerShell incluye varios cmdlets que permiten controlar directamente la salida de datos. Estos cmdlets comparten dos características importantes.

En primer lugar, suelen transformar los datos en algún tipo de texto. El motivo es que envían los datos a componentes del sistema que requieren texto como entrada. Esto significa que necesitan representar los objetos como texto. Por tanto, el texto con formato aplicado es el que se muestra en la ventana de la consola de Windows PowerShell.

En segundo lugar, estos cmdlets tienen el verbo **Out** de Windows PowerShell porque envían información desde Windows PowerShell a cualquier otra ubicación. El cmdlet **Out-Host** no es una excepción: la presentación de la ventana del host está fuera de Windows PowerShell. Esto es importante, ya que cuando se envían datos fuera de Windows PowerShell, en realidad dichos datos se quitan. Para comprobarlo, cree una canalización que envíe datos divididos en páginas a la ventana del host y, a continuación, intente darles formato de lista, como se muestra a continuación:

```
PS> Get-Process | Out-Host -Paging | Format-List
```

Quizá espere que el comando muestre páginas de información de procesos en formato de lista, pero lo que muestra es la lista con formato de tabla predeterminada:

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    101       5     1076       3316    32     0.05   2888 alg
...
    618      18    39348      51108   143   211.20    740 explorer
    257       8     9752      16828    79     3.02   2560 explorer
...
<ESPACIO> página siguiente; <RETORNO> línea siguiente; Q salir
...
```

El cmdlet **Out-Host** envía los datos directamente a la consola, por lo que el comando **Format-List** no recibe datos a los que aplicar formato.

La forma correcta de estructurar este comando es colocar el cmdlet **Out-Host** al final de la canalización, como se muestra a continuación. Esto hace que se aplique formato de lista a los datos de procesos antes de dividirlos en páginas y mostrarlos.

```
PS> Get-Process | Format-List | Out-Host -Paging


Id      : 2888
Handles : 101
CPU     : 0.046875
```

```
Name    : alg
...

Id      : 740
Handles : 612
CPU     : 211.703125
Name    : explorer

Id      : 2560
Handles : 257
CPU     : 3.015625
Name    : explorer
...
<ESPACIO> página siguiente; <RETORNO> línea siguiente; Q salir
...
```

Esto es aplicable a todos los cmdlets **Out**. Un cmdlet **Out** debe aparecer siempre al final de la canalización.

Nota:

Todos los cmdlets **Out** representan la salida como texto, utilizando el formato vigente para la ventana de la consola, incluidas las limitaciones de longitud de línea.

Dividir en páginas el resultado mostrado en la consola (Out-Host)

De manera predeterminada, Windows PowerShell envía datos a la ventana del host, que es exactamente lo que hace el cmdlet Out-Host. El uso principal del cmdlet Out-Host es dividir los datos en páginas, como se ha explicado anteriormente. Por ejemplo, el siguiente comando utiliza Out-Host para dividir el resultado del cmdlet Get-Command en páginas:

```
PS> Get-Command | Out-Host -Paging
```

También puede usar la función **more** para dividir datos en páginas. En Windows PowerShell, **more** es una función que llama a **Out-Host -Paging**. El siguiente comando muestra el uso de la función **more** para dividir el resultado de Get-Command en páginas:

```
PS> Get-Command | more
```

Si incluye uno o más nombres de archivos como argumentos de la función more, esta función leerá los archivos especificados y mostrará su contenido dividido en páginas en el host:

```
PS> more c:\boot.ini
[boot loader]
timeout=5
```

```
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
...
```

Descartar el resultado (Out-Null)

El cmdlet **Out-Null** se ha diseñado para descartar inmediatamente cualquier entrada que reciba. Esto resulta útil para descartar datos innecesarios que se obtengan como efecto secundario de ejecutar un comando. Si escribe el siguiente comando, no recibirá nada del comando:

```
PS> Get-Command | Out-Null
```

El cmdlet **Out-Null** no evita que se muestren mensajes de error. Por ejemplo, si escribe el siguiente comando, aparecerá un mensaje para informarle de que Windows PowerShell no reconoce 'Is-NotACommand':

```
PS> Get-Command Is-NotACommand | Out-Null
Get-Command : El término 'Is-NotACommand' no se reconoce como un cmdlet, función,
programa
ejecutable ni archivo de script.
En línea:1 carácter:12
+ Get-Command  <<<< Is-NotACommand | Out-Null
```

Imprimir datos (Out-Printer)

Puede imprimir datos con el cmdlet **Out-Printer**. El cmdlet **Out-Printer** usa la impresora predeterminada si no se proporciona un nombre de impresora. Para utilizar cualquier otra impresora basada en Windows, debe especificar su nombre para mostrar. No es necesario asignar ningún tipo de puerto de impresora ni una impresora física real. Por ejemplo, si tiene instaladas las herramientas de creación de imágenes de documentos de Microsoft Office, puede enviar los datos a un archivo de imagen con el siguiente comando:

```
PS> Get-Command Get-Command | Out-Printer -Name "Microsoft Office Document Image
Writer"
```

Almacenar datos (Out-File)

Puede usar el cmdlet **Out-File** para enviar el resultado a un archivo, en lugar de a la ventana de la consola. La siguiente línea de comandos envía una lista de procesos al archivo **C:\temp\processlist.txt**:

```
PS> Get-Process | Out-File -FilePath C:\temp\processlist.txt
```

Los resultados de usar el cmdlet **Out-File** pueden no ser los esperados si está acostumbrado a la redirección tradicional de la salida. Para entender este comportamiento, debe tener en cuenta el contexto en el que se utiliza el cmdlet **Out-File**.

De manera predeterminada, el cmdlet **Out-File** crea un archivo Unicode. Esta configuración predeterminada es la mejor a la larga, pero significa que las herramientas que esperan archivos ASCII no funcionarán correctamente con el formato predeterminado para los resultados. Puede cambiar este formato predeterminado a ASCII con el parámetro **Encoding**:

```
PS> Get-Process | Out-File -FilePath C:\temp\processlist.txt -Encoding ASCII
```

El cmdlet **Out-File** aplica formato al contenido del archivo para que se parezca al resultado de la consola. En la mayoría de los casos, esto hace que el resultado aparezca truncado como en una ventana de consola. Por ejemplo, si ejecuta el siguiente comando:

```
PS> Get-Command | Out-File -FilePath c:\temp\output.txt
```

El resultado se parecerá al siguiente:

```
CommandType     Name                               Definition
-----------     ----                               ----------
Cmdlet          Add-Content                        Add-Content [-Path] <String[...
Cmdlet          Add-History                        Add-History [[-InputObject] ...
...
```

Si desea no se realicen ajustes de línea el resultado obtenido para adaptar los datos al ancho de pantalla, puede utilizar el parámetro **Width** para especificar el ancho de línea. Dado que **Width** es un parámetro cuyo valor es un entero de 32 bits, el valor máximo que puede contener es 2147483647. Para establecer el ancho de línea en este valor máximo, escriba lo siguiente:

```
Get-Command | Out-File -FilePath c:\temp\output.txt -Width 2147483647
```

El cmdlet **Out-File** es muy útil si se desea guardar el resultado tal como se habría mostrado en la consola. Para tener un mayor control sobre el formato de los resultados se necesitan herramientas más avanzadas. Examinaremos estas herramientas en el siguiente capítulo, junto con información detallada acerca de la manipulación de objetos.

Desplazamiento por Windows PowerShell

Las carpetas constituyen una característica organizativa muy conocida de la interfaz del Explorador de Windows, Cmd.exe y herramientas de UNIX como BASH. Las carpetas (o directorios, como se las suele llamar) son un concepto muy útil para la organización

de archivos y otros directorios. Los sistemas operativos de la familia UNIX amplían este concepto al tratar todo lo posible como si fueran archivos; las conexiones específicas de hardware y de red aparecen como archivos en carpetas concretas. Este enfoque no asegura que el contenido lo puedan leer o utilizar aplicaciones concretas, pero facilita la búsqueda de elementos específicos. Las herramientas que permiten enumerar o buscar en archivos y carpetas también se pueden usar con estos dispositivos. Asimismo, se puede dirigir un elemento concreto utilizando la ruta de acceso al archivo que lo representa.

Análogamente, la infraestructura de Windows PowerShell permite exponer como una unidad de Windows PowerShell prácticamente todo lo que se pueda explorar como una unidad de disco estándar de Microsoft Windows o un sistema de archivos de UNIX. Una unidad de Windows PowerShell no representa necesariamente una unidad real localmente o en la red. Este capítulo se centra principalmente en el desplazamiento por sistemas de archivos, pero los conceptos son aplicables a unidades de Windows PowerShell que no estén asociadas a sistemas de archivos.

Administrar la ubicación actual en Windows PowerShell

Al desplazarse por sistemas de carpetas en el Explorador de Windows, se suele tener una ubicación de trabajo concreta, es decir, la carpeta abierta actualmente. Los elementos de la carpeta actual se pueden manipular fácilmente con sólo hacer clic en ellos. En interfaces de línea de comandos como Cmd.exe, si se encuentra en la misma carpeta que un archivo específico, puede tener acceso al archivo mediante la especificación de un nombre relativamente corto, en lugar de tener que especificar la ruta de acceso al archivo completa. El directorio actual se denomina directorio de trabajo.

Windows PowerShell utiliza el sustantivo **Location** para hacer referencia al directorio de trabajo e incluye un conjunto de cmdlets para examinar y manipular la ubicación.

Obtener la ubicación actual (Get-Location)

Para determinar la ruta de acceso a la ubicación actual, escriba el comando **Get-Location**:

```
PS> Get-Location
Path
----
C:\Documents and Settings\PowerUser
```

Nota:

El cmdlet Get-Location es similar al comando pwd del shell BASH. El cmdlet Set-Location es similar al comando cd de Cmd.exe.

Establecer la ubicación actual (Set-Location)

El comando **Get-Location** se utiliza junto con el comando **Set-Location**. El comando **Set-Location** permite especificar la ubicación actual.

```
PS> Set-Location -Path C:\Windows
```

Después de escribir el comando, observará que no recibe ninguna información directa sobre el efecto del comando. La mayoría de los comandos de Windows PowerShell que realizan una acción generan pocos resultados o ninguno, ya que éstos no siempre son útiles. Para comprobar que se ha producido el cambio correcto de directorio al escribir el comando **Set-Location**, incluya el parámetro **-PassThru**:

```
PS> Set-Location -Path C:\Windows -PassThru
Path
----
C:\WINDOWS
```

El parámetro **-PassThru** se puede utilizar con muchos comandos Set de Windows PowerShell para obtener información sobre el resultado en casos en los que no se muestran resultados de manera predeterminada.

Puede especificar rutas de acceso relativas a la ubicación actual de la misma manera que lo haría en la mayoría de los shells de comandos de UNIX y Windows. En la notación estándar para rutas de acceso relativas, la carpeta actual se representa mediante un punto (**.**) y el directorio principal de la ubicación actual se representa mediante dos puntos (**..**).

Por ejemplo, si se encuentra en la carpeta **C:\Windows**, un punto (**.**) representa **C:\Windows** y dos puntos (**..**) representan **C:**. Para cambiar de la ubicación actual a la raíz de la unidad C:, escriba:

```
PS> Set-Location -Path .. -PassThru

Path

----

C:\
```

La misma técnica se puede aplicar a unidades de Windows PowerShell que no son unidades del sistema de archivos, como **HKLM:**. Para establecer la ubicación en la clave HKLM\Software del Registro, escriba:

```
 PS> Set-Location -Path HKLM:\SOFTWARE -PassThru
```

```
Path
----
HKLM:\SOFTWARE
```

Posteriormente, puede cambiar la ubicación al directorio principal, que es la raíz de la unidad HKLM: de Windows PowerShell, utilizando una ruta de acceso relativa:

```
PS> Set-Location -Path .. -PassThru

Path
----
HKLM:\
```

Puede escribir Set-Location o usar cualquiera de los alias integrados de Windows PowerShell para Set-Location (cd, chdir, sl). Por ejemplo:

```
cd -Path C:\Windows
```

```
chdir -Path .. -PassThru
```

```
sl -Path HKLM:\SOFTWARE -PassThru
```

Almacenar y recuperar ubicaciones recientes (Push-Location y Pop-Location)

Al cambiar de ubicación, resulta útil registrar la ubicación anterior y poder volver a ella. El cmdlet **Push-Location** de Windows PowerShell crea un historial ordenado (una "pila") de rutas de directorios en los que ha estado y el cmdlet **Pop-Location** complementario permite retroceder por este historial.

Por ejemplo, una sesión de Windows PowerShell se inicia normalmente en el directorio principal del usuario.

```
 PS> Get-Location

Path
----
C:\Documents and Settings\PowerUser
```

Nota:

La pila de palabras tiene un significado especial en muchos entornos de programación, incluido .NET. Al igual que en una pila física de elementos,

el último que se coloca en la pila es el primero que se puede extraer. El proceso de agregar un elemento a una pila también se denomina "insertar" el elemento en la pila. El proceso de sacar un elemento de una pila también se denomina "extraer" el elemento de la pila.

Para insertar la ubicación actual en la pila y, a continuación, desplazarse a la carpeta Configuración local, escriba:

```
 PS> Push-Location -Path "Configuración local"
```

Después puede insertar la ubicación Configuración local en la pila y desplazarse a la carpeta Temp. Para ello, escriba:

```
PS> Push-Location -Path Temp
```

Para comprobar que ha cambiado de directorio, escriba el comando **Get-Location**:

```
PS> Get-Location

Path
----
C:\Documents and Settings\PowerUser\Configuración local\Temp
```

A continuación, puede retroceder al último directorio visitado mediante el comando **Pop-Location**, así como comprobar el cambio con el comando **Get-Location**:

```
PS> Pop-Location
PS> Get-Location

Path
----
C:\Documents and Settings\yo\Configuración local
```

Al igual que con el cmdlet **Set-Location**, puede incluir el parámetro **-PassThru** cuando escriba el cmdlet **Pop-Location** para que se muestre el directorio que especificó:

```
PS> Pop-Location -PassThru

Path
----
C:\Documents and Settings\PowerUser
```

También puede utilizar los cmdlets Location con rutas de red. Si tiene un servidor llamado FS01 con el recurso compartido Public, puede cambiar la ubicación con el siguiente comando:

```
Set-Location \\FS01\Public
```

O bien

```
Push-Location \\FS01\Public
```

Puede usar los comandos **Push-Location** y **Set-Location** para cambiar la ubicación a cualquier unidad disponible. Por ejemplo, si tiene una unidad local de CD-ROM con la letra de unidad D que contiene un CD de datos, puede cambiar la ubicación a la unidad de CD con el comando **Set-Location D:**.

Si la unidad está vacía, aparecerá el siguiente mensaje de error:

```
PS> Set-Location D:
Set-Location : No se encuentra la ruta de acceso 'D:\' porque no existe.
```

Si utiliza una interfaz de línea de comandos, no es conveniente examinar las unidades físicas disponibles con el Explorador de Windows. Además, el Explorador de Windows no mostrará todas las unidades de Windows PowerShell. Windows PowerShell proporciona un conjunto de comandos para manipular las unidades de Windows PowerShell, que es de lo que nos ocuparemos a continuación.

Administrar las unidades de Windows PowerShell

Una unidad de Windows PowerShell es una ubicación de almacén de datos a la que se puede tener acceso como si fuera una unidad del sistema de archivos en Windows PowerShell. Los proveedores de Windows PowerShell crean automáticamente algunas unidades, como las unidades del sistema de archivos (incluidas C: y D:), las unidades del Registro (HKCU: y HKLM:) y la unidad de certificados (Cert:), pero puede crear sus propias unidades de Windows PowerShell. Estas unidades son muy útiles, pero están disponibles únicamente desde Windows PowerShell. No se puede tener acceso a ellas mediante otras herramientas de Windows, como el Explorador de Windows o Cmd.exe.

Windows PowerShell utiliza el sustantivo **PSDrive** para los comandos que se pueden usar con unidades de Windows PowerShell. Para obtener una lista de las unidades de Windows PowerShell de la sesión, utilice el cmdlet **Get-PSDrive**:

```
PS> Get-PSDrive

Name       Provider      Root                                  CurrentLocation
----       --------      ----                                  ---------------
A          FileSystem    A:\
Alias      Alias
```

```
C           FileSystem    C:\                                  ...And Settings\me
cert        Certificate   \
D           FileSystem    D:\
Env         Environment
Function    Function
HKCU        Registry      HKEY_CURRENT_USER
HKLM        Registry      HKEY_LOCAL_MACHINE
Variable    Variable
```

Aunque las unidades mostradas varían respecto a las unidades del sistema, la lista será similar al resultado del comando **Get-PSDrive** mostrado anteriormente.

Las unidades del sistema de archivos son un subconjunto de las unidades de Windows PowerShell. Puede identificar las unidades del sistema de archivos por la entrada FileSystem de la columna Provider. (El proveedor FileSystem de Windows PowerShell admite las unidades del sistema de archivos de Windows PowerShell.)

Para ver la sintaxis del cmdlet **Get-PSDrive**, escriba el comando **Get-Command** con el parámetro **Syntax**:

```
PS> Get-Command -Name Get-PSDrive -Syntax
Get-PSDrive [[-Name] <String[]>] [-Scope <String>] [-PSProvider <String[]>] [-V
erbose] [-Debug] [-ErrorAction <ActionPreference>] [-ErrorVariable <String>] [-
OutVariable <String>] [-OutBuffer <Int32>]
```

El parámetro **PSProvider** permite mostrar únicamente las unidades de Windows PowerShell admitidas por un proveedor determinado. Por ejemplo, para mostrar sólo las unidades de Windows PowerShell admitidas por el proveedor FileSystem de Windows PowerShell, escriba el comando **Get-PSDrive** con el parámetro **PSProvider** y el valor **FileSystem**:

```
PS> Get-PSDrive -PSProvider FileSystem

Name        Provider      Root                                    CurrentLocation
----        --------      ----                                    ---------------
A           FileSystem    A:\
C           FileSystem    C:\                              ...nd Settings\PowerUser
D           FileSystem    D:\
```

Para ver las unidades de Windows PowerShell que representan subárboles del Registro, utilice el parámetro **PSProvider** para que se muestren únicamente las unidades de Windows PowerShell admitidas por el proveedor del Registro de Windows PowerShell:

```
PS> Get-PSDrive -PSProvider Registry


Name        Provider      Root                                    CurrentLocation
----        --------      ----                                    ---------------
```

```
HKCU        Registry      HKEY_CURRENT_USER

HKLM        Registry      HKEY_LOCAL_MACHINE
```

También puede utilizar los cmdlets Location estándar con las unidades de Windows PowerShell:

```
PS> Set-Location HKLM:\SOFTWARE

PS> Push-Location .\Microsoft

PS> Get-Location


Path

----

HKLM:\SOFTWARE\Microsoft
```

Agregar nuevas unidades de Windows PowerShell (New-PSDrive)

Puede agregar sus propias unidades de Windows PowerShell con el comando **New-PSDrive**. Para obtener la sintaxis del comando **New-PSDrive**, escriba el comando **Get-Command** con el parámetro **Syntax**:

```
PS> Get-Command -Name New-PSDrive -Syntax
New-PSDrive [-Name] <String> [-PSProvider] <String> [-Root] <String> [-Descript
ion <String>] [-Scope <String>] [-Credential <PSCredential>] [-Verbose] [-Debug
] [-ErrorAction <ActionPreference>] [-ErrorVariable <String>] [-OutVariable <St
ring>] [-OutBuffer <Int32>] [-WhatIf] [-Confirm]
```

Para crear una nueva unidad de Windows PowerShell, debe proporcionar tres parámetros:

- Un nombre para la unidad (puede utilizar cualquier nombre válido de Windows PowerShell)
- El valor de PSProvider (utilice "FileSystem" para ubicaciones del sistema de archivos y "Registry" para ubicaciones del Registro)
- El directorio raíz (es decir, la ruta de acceso al directorio raíz de la nueva unidad)

Por ejemplo, puede crear una unidad llamada "Office" que esté asignada a la carpeta que contiene las aplicaciones de Microsoft Office en el equipo, como **C:\Archivos de programa\Microsoft Office\OFFICE11**. Para crear la unidad, escriba el siguiente comando:

```
PS> New-PSDrive -Name Office -PSProvider FileSystem -Root "C:\Archivos de programa
\Microsoft Office\OFFICE11"

Name        Provider      Root                                    CurrentLocation
```

```
----       --------      ----                                    ---------------
Office     FileSystem    C:\Archivos de programa\Microsoft Offic...
```

Nota:

En general, las rutas de acceso no distinguen entre mayúsculas y minúsculas.

Se hará referencia a la nueva unidad de Windows PowerShell de la misma manera que a todas las unidades de Windows PowerShell: por su nombre seguido de dos puntos (:).

Una unidad de Windows PowerShell puede simplificar muchas tareas. Por ejemplo, algunas de las claves más importantes del Registro de Windows tienen rutas de acceso muy largas, lo cual dificulta su acceso y su memorización. Los datos importantes de configuración se encuentran en **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion**. Para ver y cambiar elementos de la clave CurrentVersion del Registro, puede crear una unidad de Windows PowerShell cuya raíz sea esta clave. Para ello, escriba:

```
PS> New-PSDrive -Name cvkey -PSProvider Registry -Root HKLM\Software\Microsoft\W
indows\CurrentVersion


Name       Provider      Root                                    CurrentLocation
----       --------      ----                                    ---------------
cvkey      Registry      HKLM\Software\Microsoft\Windows\...
```

A continuación, puede cambiar la ubicación a la unidad **cvkey:** del mismo modo que si se tratara de cualquier otra unidad:

```
PS> cd cvkey:
```

O bien:

```
PS> Set-Location cvkey: -PassThru


Path
----
cvkey:\
```

El cmdlet New-PsDrive agrega la nueva unidad sólo a la sesión de consola actual. Si cierra la consola o la ventana de Windows PowerShell, perderá la nueva unidad. Para guardar una unidad de Windows PowerShell, utilice el cmdlet Export-Console para exportar la consola actual y use posteriormente el parámetro PSConsoleFile de PowerShell.exe para importarla a una nueva sesión. O bien, agregue la nueva unidad a su perfil de Windows PowerShell.

Eliminar unidades de Windows PowerShell (Remove-PSDrive)

Puede eliminar unidades de Windows PowerShell con el cmdlet **Remove-PSDrive**. Este cmdlet es fácil de usar; para eliminar una unidad específica de Windows PowerShell, basta con que proporcione el nombre de dicha unidad.

Por ejemplo, si agregó la unidad Office: de Windows PowerShell de la manera indicada en el tema dedicado a **New-PSDrive**, para eliminarla debe escribir:

```
PS> Remove-PSDrive -Name Office
```

Para eliminar la unidad **cvkey:** de Windows PowerShell, mostrada también en el tema dedicado a **New-PSDrive**, utilice el siguiente comando:

```
PS> Remove-PSDrive -Name cvkey
```

Eliminar una unidad de Windows PowerShell es fácil, pero no podrá hacerlo si se encuentra en esa unidad. Por ejemplo:

```
PS> cd office:
PS Office:\> remove-psdrive -name office
Remove-PSDrive : No se puede quitar la unidad 'Office' porque está en uso.
En línea:1 carácter:15
+ remove-psdrive  <<<< -name office
```

Agregar y eliminar unidades fuera de Windows PowerShell

Windows PowerShell detecta las unidades del sistema de archivos que se agregan o quitan en Windows (incluidas unidades de red asignadas, unidades USB conectadas y unidades eliminadas) con el comando **net use** o los métodos **WScript.NetworkMapNetworkDrive** y **RemoveNetworkDrive** de un script de Windows Script Host (WSH).

Trabajar con archivos, carpetas y claves del Registro

Windows PowerShell utiliza el sustantivo **Item** para referirse a elementos que se encuentran en una unidad de Windows PowerShell. Cuando utilice el proveedor FileSystem de Windows PowerShell, un **Item** puede ser un archivo, una carpeta o la unidad de Windows PowerShell. La enumeración y el uso de estos elementos

constituyen una importante tarea básica en la mayoría de las configuraciones administrativas y, por este motivo, queremos tratar detenidamente estas tareas.

Enumerar archivos, carpetas y claves del Registro (Get-ChildItem)

Dado que obtener un conjunto de elementos de una ubicación concreta es una tarea muy habitual, el cmdlet **Get-ChildItem** se ha diseñado específicamente para devolver todos los elementos incluidos en un contenedor (por ejemplo, una carpeta).

Si desea devolver todos los archivos y carpetas que contiene directamente la carpeta C:\Windows, escriba:

```
PS> Get-ChildItem -Path C:\Windows
    Directorio: Microsoft.Windows PowerShell.Core\FileSystem::C:\Windows
Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         2006-05-16   8:10 AM          0 0.log
-a---         2005-11-29   3:16 PM         97 acc1.txt
-a---         2005-10-23  11:21 PM       3848 actsetup.log
...
```

La lista es similar a la que se mostraría si escribe el comando **dir** en **Cmd.exe** o el comando **ls** en un shell de comandos de UNIX.

Puede usar parámetros del cmdlet **Get-ChildItem** para crear listas muy complejas. Analizaremos varios escenarios a continuación. Para ver la sintaxis del cmdlet **Get-ChildItem**, escriba:

```
PS> Get-Command -Name Get-ChildItem -Syntax
```

Estos parámetros se pueden combinar y comparar para obtener resultados muy personalizados.

Crear una lista de todos los elementos contenidos (-Recurse)

Para ver tanto los elementos incluidos en una carpeta de Windows como los elementos incluidos en las subcarpetas, utilice el parámetro **Recurse** de **Get-ChildItem**. La lista mostrará todo el contenido de la carpeta de Windows, así como los elementos incluidos en las subcarpetas. Por ejemplo:

```
PS> Get-ChildItem -Path C:\WINDOWS -Recurse

    Directorio: Microsoft.Windows PowerShell.Core\FileSystem::C:\WINDOWS
    Directorio: Microsoft.Windows PowerShell.Core\FileSystem::C:\WINDOWS\AppPatch
Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         2004-08-04   8:00 AM    1852416 AcGenral.dll
```

```
...
```

Filtrar elementos por nombre (-Name)

Para mostrar únicamente los nombres de los elementos, utilice el parámetro **Name** de **Get-Childitem**:

```
PS> Get-ChildItem -Path C:\WINDOWS -Name
addins
AppPatch
assembly
...
```

Forzar la presentación de los elementos ocultos (-Force)

Los elementos que no se muestran normalmente en el Explorador de Windows o en Cmd.exe tampoco se muestran en el resultado del comando **Get-ChildItem**. Para mostrar los elementos ocultos, utilice el parámetro **Force** de **Get-ChildItem**. Por ejemplo:

```
Get-ChildItem -Path C:\Windows -Force
```

Este parámetro recibe el nombre de Force porque cambia a la fuerza el comportamiento normal del comando **Get-ChildItem**. El parámetro Force se usa a menudo y fuerza una acción que un cmdlet no realizaría normalmente, aunque no ejecutará ninguna acción que pueda poner en peligro la seguridad del sistema.

Usar caracteres comodín para buscar nombres de elementos

El comando **Get-ChildItem** acepta caracteres comodín en la ruta de acceso a los elementos que se van a enumerar.

Dado que el motor de Windows PowerShell controla la búsqueda con caracteres comodín, todos los cmdlets que aceptan caracteres comodín utilizan la misma notación y tienen el mismo comportamiento en la búsqueda de coincidencias. La notación de caracteres comodín de Windows PowerShell es la siguiente:

- El asterisco (*) busca cero o más instancias de cualquier carácter.
- El signo de interrogación (?) busca exactamente un carácter.
- Los caracteres de corchete izquierdo ([) y corchete derecho (]) rodean un conjunto de caracteres de los que se van a buscar coincidencias.

A continuación se muestran algunos ejemplos.

Para buscar todos los archivos del directorio Windows que tengan la extensión **.log** y exactamente cinco caracteres en el nombre base, escriba el siguiente comando:

```
PS> Get-ChildItem -Path C:\Windows\?????.log
    Directorio: Microsoft.Windows PowerShell.Core\FileSystem::C:\Windows
Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
...
-a---         2006-05-11   6:31 PM     204276 ocgen.log
-a---         2006-05-11   6:31 PM      22365 ocmsn.log
...
-a---         2005-11-11   4:55 AM         64 setup.log
-a---         2005-12-15   2:24 PM      17719 VxSDM.log
...
```

Para buscar todos los archivos del directorio Windows que comiencen por la letra **x**, escriba:

```
 Get-ChildItem -Path C:\Windows\x*
```

Para buscar todos los archivos cuyo nombre comience por la letra **x** o **z**, escriba:

```
Get-ChildItem -Path C:\Windows\[xz]*
```

Excluir elementos (-Exclude)

Puede excluir elementos específicos con el parámetro Exclude de Get-ChildItem. Este parámetro permite aplicar filtros complejos en una sola instrucción.

Por ejemplo, suponga que desea buscar el archivo DLL del servicio de hora de Windows en la carpeta System32 y todo lo que recuerda del nombre del archivo DLL es que comienza por "W" y que contiene "32".

Una expresión como **w*32*.dll** encontrará todos los archivos DLL que cumplan las condiciones indicadas, pero también podría devolver los archivos DLL de compatibilidad con las versiones Windows 95 y Windows de 16 bits que contienen "95" o "16" en sus nombres. Para omitir los archivos cuyo nombre contenga estos números, use el parámetro Exclude con el patrón *[9516]*:

```
 PS> Get-ChildItem -Path C:\WINDOWS\System32\w*32*.dll -Exclude *[9516]*


    Directorio: Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS\System32


Mode                 LastWriteTime     Length Name
```

```
----                 -------------     ------ ----
-a---         2004-08-04   8:00 AM     174592 w32time.dll
-a---         2004-08-04   8:00 AM      22016 w32topl.dll
-a---         2004-08-04   8:00 AM     101888 win32spl.dll
-a---         2004-08-04   8:00 AM     172032 wldap32.dll
-a---         2004-08-04   8:00 AM     264192 wow32.dll
-a---         2004-08-04   8:00 AM      82944 ws2_32.dll
-a---         2004-08-04   8:00 AM      42496 wsnmp32.dll
-a---         2004-08-04   8:00 AM      22528 wsock32.dll
-a---         2004-08-04   8:00 AM      18432 wtsapi32.dll
```

Combinar parámetros de Get-ChildItem

Puede utilizar varios de los parámetros del cmdlet **Get-ChildItem** en un mismo comando. Antes de combinar parámetros, asegúrese de que entiende cómo se pueden realizar búsquedas de coincidencias con caracteres comodín. Por ejemplo, el siguiente comando no devuelve ningún resultado:

```
PS> Get-ChildItem -Path C:\Windows\*.dll -Recurse -Exclude [a-y]*.dll
```

No se obtienen resultados, aunque hay dos archivos DLL que comienzan por la letra "z" en la carpeta Windows.

El motivo por el que no se devuelven resultados es que hemos especificado el carácter comodín como parte de la ruta de acceso. Aunque el comando era recursivo, el cmdlet **Get-ChildItem** ha limitado los elementos a los incluidos en la carpeta de Windows cuyo nombre termina en ".dll".

Para especificar una búsqueda recursiva de archivos cuyo nombre coincide con un patrón especial, utilice el parámetro **–Include**:

```
PS> Get-ChildItem -Path C:\Windows -Include *.dll -Recurse -Exclude [a-y]*.dll

    Directorio: Microsoft.Windows
PowerShell.Core\FileSystem::C:\Windows\System32\Setup


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         2004-08-04   8:00 AM       8261 zoneoc.dll


    Directorio: Microsoft.Windows PowerShell.Core\FileSystem::C:\Windows\System32
```

```
Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         2004-08-04   8:00 AM     337920 zipfldr.dll
```

Manipular elementos directamente

Los elementos que se muestran en las unidades de Windows PowerShell (como los archivos y carpetas de las unidades del sistema de archivos), así como las claves del Registro incluidas en las unidades del Registro de Windows PowerShell, reciben el nombre de elementos en Windows PowerShell. Los cmdlets que permiten trabajar con estos elementos tienen el sustantivo **Item** en sus nombres.

El resultado del comando **Get-Command -Noun Item** muestra que hay nueve cmdlets Item en Windows PowerShell:

```
PS> Get-Command -Noun Item

CommandType     Name                               Definition
-----------     ----                               ----------
Cmdlet          Clear-Item                         Clear-Item [-Path] <String[]...
Cmdlet          Copy-Item                          Copy-Item [-Path] <String[]>...
Cmdlet          Get-Item                           Get-Item [-Path] <String[]> ...
Cmdlet          Invoke-Item                        Invoke-Item [-Path] <String[...
Cmdlet          Move-Item                          Move-Item [-Path] <String[]>...
Cmdlet          New-Item                           New-Item [-Path] <String[]> ...
Cmdlet          Remove-Item                        Remove-Item [-Path] <String[...
Cmdlet          Rename-Item                        Rename-Item [-Path] <String>...
Cmdlet          Set-Item                           Set-Item [-Path] <String[]> ...
```

Crear nuevos elementos (New-Item)

Para crear un nuevo elemento en el sistema de archivos, utilice el cmdlet **New-Item**. Asimismo, incluya el parámetro Path con la ruta de acceso al elemento y el parámetro ItemType con el valor "file" o "directory".

Por ejemplo, para crear un directorio llamado "New.Directory" en el directorio C:\Temp, escriba:

```
PS> New-Item -Path c:\temp\New.Directory -ItemType Directory

    Directorio: Microsoft.Windows PowerShell.Core\FileSystem::C:\temp

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
```

```
d----         2006-05-18  11:29 AM            New.Directory
```

Para crear un archivo, cambie el valor del parámetro ItemType a "file". Por ejemplo, para crear un archivo llamado "file1.txt" en el directorio New.Directory, escriba:

```
PS> New-Item -Path C:\temp\New.Directory\file1.txt -ItemType file

    Directorio: Microsoft.Windows
PowerShell.Core\FileSystem::C:\temp\New.Directory

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         2006-05-18  11:44 AM          0 file1
```

Puede aplicar la misma técnica para crear una clave del Registro. De hecho, resulta más fácil crear una clave del Registro porque el único tipo de elemento que puede encontrarse en el Registro de Windows es una clave (las entradas del Registro son propiedades de elementos). Por ejemplo, para crear una clave llamada "_Test" en la subclave CurrentVersion, escriba:

```
PS> New-Item -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\_Test

   Hive: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Micros
oft\Windows\CurrentVersion

SKC  VC Name                           Property
---  -- ----                           --------
  0   0 _Test                         {}
```

Cuando escriba una ruta de acceso al Registro, asegúrese de incluir los dos puntos (:) en los nombres de las unidades HKLM: y HKCU: de Windows PowerShell. Sin los dos puntos, Windows PowerShell no reconocerá el nombre de la unidad en la ruta de acceso.

Por qué los valores del Registro no son elementos

Cuando se utiliza el cmdlet **Get-ChildItem** para buscar los elementos de una clave del Registro, nunca se muestran las entradas reales del Registro ni sus valores.

Por ejemplo, la clave del Registro **HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run** contiene normalmente varias entradas del Registro que representan aplicaciones que se ejecutan cuando se inicia el sistema.

No obstante, cuando se utiliza **Get-ChildItem** para buscar los elementos secundarios de la clave, todo lo que se muestra es la subclave **OptionalComponents** de la clave:

```
PS> Get-ChildItem HKLM:\Software\Microsoft\Windows\CurrentVersion\Run
   Hive: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software\Micros
oft\Windows\CurrentVersion\Run
SKC  VC Name                           Property
---  -- ----                           --------
  3   0 OptionalComponents             {}
```

Aunque resultaría cómodo tratar las entradas del Registro como elementos, no es posible especificar una ruta de acceso a una entrada del Registro de una manera que garantice que es única. La notación de rutas de acceso no distingue entre la subclave del Registro **Run** y la entrada del Registro **(Default)** de la subclave **Run**. Además, dado que los nombres de entradas del Registro pueden contener la barra diagonal inversa (****), si se trataran las entradas del Registro como elementos, no se podría utilizar la notación de rutas de acceso para distinguir una entrada del Registro llamada **Windows\CurrentVersion\Run** de la subclave ubicada en esa ruta de acceso.

Cambiar nombres de elementos existentes (Rename-Item)

El cmdlet **Rename-Item** permite cambiar el nombre de un archivo o una carpeta.
El siguiente comando cambia el nombre del archivo file1.txt a fileOne.txt:

```
PS> Rename-Item -Path C:\temp\New.Directory\file1.txt fileOne.txt
```

El cmdlet **Rename-Item** puede cambiar el nombre de un archivo o una carpeta, pero no puede mover un elemento. Al intentar mover el archivo desde el directorio New.Directory al directorio Temp, se produce un error al ejecutar el siguiente comando:

```
PS> Rename-Item -Path C:\temp\New.Directory\fileOne.txt c:\temp\fileOne.txt
Rename-Item : No se puede cambiar el nombre porque el destino especificado no es
una ruta de acceso.
En línea:1 carácter:12
+ Rename-Item  <<<< -Path C:\temp\New.Directory\fileOne c:\temp\fileOne.txt
```

Desplazar elementos (Move-Item)

El cmdlet **Move-Item** permite mover un archivo o una carpeta.

Por ejemplo, el siguiente comando mueve el directorio New.Directory desde el directorio C:\temp a la raíz de la unidad C:. Para comprobar que se ha movido el elemento, incluya el parámetro **PassThru** del cmdlet **Move-Item**. Sin Passthru, el cmdlet **Move-Item** no muestra ningún resultado.

```
PS> Move-Item -Path C:\temp\New.Directory -Destination C:\ -PassThru

    Directorio: Microsoft.Windows PowerShell.Core\FileSystem::C:\
```

```
Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         2006-05-18  12:14 PM            New.Directory
```

Copiar elementos (Copy-Item)

Si está familiarizado con las operaciones de copia de otros shells, puede que el comportamiento del cmdlet **Copy-Item** de Windows PowerShell le resulte inusual. Cuando se copia un elemento desde una ubicación a otra, Copy-Item no copia su contenido de manera predeterminada.

Por ejemplo, si copia el directorio **New.Directory** desde la unidad C: al directorio C:\temp, el comando se ejecutará correctamente, pero los archivos del directorio New.Directory no se copiarán.

```
PS> Copy-Item -Path C:\New.Directory -Destination C:\temp
```

Si muestra el contenido del directorio **C:\temp\New.Directory**, observará que no contiene ningún archivo:

```
PS> Get-ChildItem -Path C:\temp\New.Directory
PS>
```

¿Por qué el cmdlet **Copy-Item** no copia el contenido en la nueva ubicación?

El cmdlet **Copy-Item** ha sido diseñado de manera que sea genérico; no sirve únicamente para copiar archivos y carpetas. Así, incluso al copiar archivos y carpetas, es posible que desee copiar únicamente el contenedor y no los elementos que contiene.

Para copiar todo el contenido de una carpeta, incluya el parámetro **Recurse** del cmdlet **Copy-Item**. Si ya ha copiado el directorio sin el contenido, agregue el parámetro **Force**, que permite sobrescribir la carpeta vacía.

```
PS> Copy-Item -Path C:\New.Directory -Destination C:\temp -Recurse -Force -
Passthru
    Directorio: Microsoft.Windows PowerShell.Core\FileSystem::C:\temp

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         2006-05-18   1:53 PM            New.Directory

    Directorio: Microsoft.Windows
PowerShell.Core\FileSystem::C:\temp\New.Directory

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         2006-05-18  11:44 AM          0 file1
```

Eliminar elementos (Remove-Item)

El cmdlet **Remove-Item** permite eliminar archivos y carpetas. Los cmdlets de Windows PowerShell que pueden realizar cambios irreversibles importantes, como es el caso de **Remove-Item**, suelen solicitar confirmación. Por ejemplo, si intenta quitar la carpeta **New.Directory**, se le pedirá que confirme el comando, ya que esta carpeta contiene archivos:

```
PS> Remove-Item C:\New.Directory

Confirmar
El elemento situado en C:\temp\New.Directory tiene elementos secundarios y no se
especificó el parámetro -recurse. Si continúa, se quitarán todos los secundarios
junto con el elemento. ¿Está seguro de que desea continuar?
[S] Sí  [O] Sí a todo  [N] No  [T] No a todo  [S] Suspender  [?] Ayuda
(el valor predeterminado es "S"):
```

Dado que la respuesta predeterminada es Sí, para eliminar la carpeta y los archivos que contiene, deberá presionar la tecla **Entrar**. Para quitar la carpeta sin que se solicite confirmación, utilice el parámetro **-Recurse**.

```
PS> Remove-Item C:\temp\New.Directory -Recurse
```

Ejecutar elementos (Invoke-Item)

Windows PowerShell utiliza el cmdlet **Invoke-Item** para realizar una acción predeterminada relativa a un archivo o una carpeta. Esta acción predeterminada está determinada por el controlador de aplicaciones predeterminado en el Registro; el efecto es el mismo que si se hace doble clic en el elemento en el Explorador de Windows.

Por ejemplo, suponga que ejecuta el siguiente comando:

```
PS> Invoke-Item C:\WINDOWS
```

Aparecerá una ventana del Explorador que se encuentra en C:\Windows, como si hubiera hecho doble clic en la carpeta C:\Windows.

Si invoca el archivo **Boot.ini** en un sistema anterior a Windows Vista:

```
PS> Invoke-Item C:\boot.ini
```

Si el tipo de archivo .ini está asociado al Bloc de notas, el archivo boot.ini se abrirá en esta aplicación.

Trabajar con objetos

Hemos explicado cómo Windows PowerShell utiliza objetos para transferir datos entre cmdlets y hemos mostrado varias maneras de mostrar información detallada acerca de los objetos con los cmdlets Get-Member y Format, que permiten ver propiedades específicas de los objetos.

La eficacia de los objetos reside en que proporcionan acceso a muchos datos complejos que ya están correlacionados. Mediante algunas técnicas sencillas es posible manipular los objetos para realizar más tareas. En este capítulo, veremos algunos tipos de objetos específicos y maneras en las que se pueden manipular.

Obtener objetos de WMI (Get-WmiObject)

Obtener objetos de WMI (Get-WmiObject)

Instrumental de administración de Windows (WMI) es una tecnología básica para la administración de sistemas Windows, ya que expone una amplia variedad de información de manera uniforme. Debido a todo lo que se puede hacer con WMI, el cmdlet de Windows PowerShell para el acceso a objetos WMI, **Get-WmiObject**, es uno de los más útiles para el trabajo real. Vamos a explicar cómo se puede utilizar Get-WmiObject para tener acceso a objetos WMI y, a continuación, cómo se pueden usar los objetos WMI para realizar tareas específicas.

Enumerar las clases de WMI

El primer problema al que se enfrentan la mayoría de los usuarios de WMI es intentar averiguar qué se puede hacer con WMI. Las clases de WMI describen los recursos que se pueden administrar. Hay cientos de clases de WMI, y algunas de ellas contienen docenas de propiedades.

Este problema se resuelve con **Get-WmiObject**, que permite obtener información de WMI. Para obtener una lista de las clases de WMI disponibles en el equipo local, escriba:

```
PS> Get-WmiObject -List


__SecurityRelatedClass                  __NTLMUser9X
__PARAMETERS                            __SystemSecurity
__NotifyStatus                          __ExtendedStatus
Win32_PrivilegesStatus                  Win32_TSNetworkAdapterSettingError
Win32_TSRemoteControlSettingError       Win32_TSEnvironmentSettingError
```

```
...
```

Puede recuperar la misma información de un equipo remoto mediante el parámetro ComputerName, especificando un nombre de equipo o una dirección IP:

```
PS> Get-WmiObject -List -ComputerName 192.168.1.29


__SystemClass                           __NAMESPACE
__Provider                              __Win32Provider
__ProviderRegistration                  __ObjectProviderRegistration
...
```

La lista de clases devuelta por los equipos remotos puede variar en función del sistema operativo que se ejecute en el equipo y de las extensiones de WMI agregadas por las aplicaciones instaladas.

Nota:

Cuando se utiliza Get-WmiObject para conectar con un equipo remoto, en dicho equipo debe estar ejecutándose WMI y, con la configuración predeterminada, la cuenta que se esté utilizando debe pertenecer al grupo de administradores locales. No es necesario que el sistema remoto tenga instalado Windows PowerShell. Esto permite administrar sistemas operativos que no ejecutan Windows PowerShell, pero que disponen de WMI.

Incluso puede especificar el parámetro ComputerName al conectarse al sistema local. Como nombre de equipo puede usar el nombre del equipo local, su dirección IP (o la dirección de bucle invertido 127.0.0.1) o el punto ('.') de WMI. Si ejecuta Windows PowerShell en un equipo llamado Admin01 con dirección IP 192.168.1.90, los comandos siguientes devolverán la lista de clases de WMI del equipo:

```
Get-WmiObject -List
Get-WmiObject -List -ComputerName .
Get-WmiObject -List -ComputerName Admin01
Get-WmiObject -List -ComputerName 192.168.1.90
Get-WmiObject -List -ComputerName 127.0.0.1
Get-WmiObject -List -ComputerName localhost
```

De manera predeterminada, Get-WmiObject usa el espacio de nombres root/cimv2. Si desea especificar otro espacio de nombres de WMI, utilice el parámetro **Namespace** y especifique la ruta de acceso al espacio de nombres correspondiente:

```
PS> Get-WmiObject -List -ComputerName 192.168.1.29 -Namespace root


__SystemClass                           __NAMESPACE
__Provider                              __Win32Provider
```

```
...
```

Obtener información detallada sobre las clases de WMI

Si ya sabe el nombre de una clase de WMI, puede utilizarlo para obtener información de forma inmediata. Por ejemplo, una de las clases de WMI que se usan habitualmente para recuperar información acerca de un equipo es **Win32_OperatingSystem**.

```
PS> Get-WmiObject -Class Win32_OperatingSystem -Namespace root/cimv2 -ComputerName
.


SystemDirectory : C:\WINDOWS\system32
Organization    : Global Network Solutions
BuildNumber     : 2600
RegisteredUser  : Oliver W. Jones
SerialNumber    : 12345-678-9012345-67890
Version         : 5.1.2600
```

Aunque se muestran todos los parámetros, el comando se puede expresar de manera más concisa. El parámetro **ComputerName** no es necesario al conectarse al sistema local. Lo mostramos como demostración del caso más general y para recordarle que existe. El valor predeterminado de **Namespace**, que también se puede omitir, es root/cimv2. Por último, la mayoría de los cmdlets permiten omitir el nombre de parámetros comunes. En el caso de Get-WmiObject, si no se especifica ningún nombre para el primer parámetro, Windows PowerShell lo procesará como el parámetro **Class**. Esto significa que el comando anterior se podría haber escrito así:

```
Get-WmiObject Win32_OperatingSystem
```

La clase **Win32_OperatingSystem** tiene muchas más propiedades que las que se muestran aquí. Puede usar Get-Member para ver todas las propiedades. Las propiedades de una clase de WMI están disponibles automáticamente como otras propiedades de un objeto:

```
PS> Get-WmiObject -Class Win32_OperatingSystem -Namespace root/cimv2 -ComputerName
. | Get-Member -MemberType Property


   TypeName: System.Management.ManagementObject#root\cimv2\Win32_OperatingSyste
m

Name                                       MemberType Definition
----                                       ---------- ----------
__CLASS                                    Property   System.String __CLASS {...
...
```

```
BootDevice                                  Property   System.String BootDevic...
BuildNumber                                 Property   System.String BuildNumb...
...
```

Mostrar propiedades no predeterminadas con los cmdlets Format

Si desea ver información contenida en la clase **Win32_OperatingSystem** que no se muestra de manera predeterminada, puede usar los cmdlets **Format**. Por ejemplo, si desea mostrar los datos de memoria disponible, escriba:

```
PS> Get-WmiObject -Class Win32_OperatingSystem -Namespace root/cimv2 -ComputerName . | Format-Table -
Property
TotalVirtualMemorySize,TotalVisibleMemorySize,FreePhysicalMemory,FreeVirtualMemory,FreeSpaceInPagingFiles

TotalVirtualMemorySize TotalVisibleMem FreePhysicalMem FreeVirtualMemo FreeSpaceInPagi
                            ory              ry         ngFiles
---------------------- ---------------- ---------------- ---------------- ----------------
               2097024           785904           305808          2056724          1558232
```

Nota:

Se pueden utilizar caracteres comodín con nombres de propiedades en **Format-Table**, de manera que el último elemento de la canalización se puede reducir a **Format-Table -Property TotalV*,Free***.

Puede facilitar la lectura de los datos de memoria si les aplica formato de lista mediante el siguiente comando:

```
PS> Get-WmiObject -Class Win32_OperatingSystem -Namespace root/cimv2 -ComputerName . | Format-List
TotalVirtualMemorySize,TotalVisibleMemorySize,FreePhysicalMemory,FreeVirtualMemory,FreeSpaceInPagingFiles


TotalVirtualMemorySize : 2097024
TotalVisibleMemorySize : 785904
FreePhysicalMemory     : 301876
FreeVirtualMemory      : 2056724
FreeSpaceInPagingFiles : 1556644
```

Crear objetos .NET y COM (New-Object)

Hay componentes de software con interfaces de .NET Framework y COM que permiten realizar muchas tareas de administración de sistemas. Windows PowerShell permite usar estos componentes, así que no está limitado a las tareas que se pueden realizar con

cmdlets. Muchos de los cmdlets de la primera versión de Windows PowerShell no se pueden utilizar con equipos remotos. Vamos a mostrar cómo se puede superar esta limitación al administrar registros de eventos utilizando la clase **System.Diagnostics.EventLog** de .NET directamente en Windows PowerShell.

Usar New-Object para el acceso a registros de eventos

La biblioteca de clases de .NET Framework contiene una clase llamada **System.Diagnostics.EventLog** que permite administrar registros de eventos. Puede crear una nueva instancia de una clase .NET mediante el cmdlet **New-Object** con el parámetro **TypeName**. Por ejemplo, el siguiente comando crea una referencia de registro de eventos:

```
PS> New-Object -TypeName System.Diagnostics.EventLog

  Max(K) Retain OverflowAction        Entries Name
  ------ ------ --------------        ------- ----
```

Aunque el comando ha creado una instancia de la clase EventLog, la instancia no contiene datos. Esto es debido a que no hemos especificado un registro de eventos concreto. ¿Cómo podemos obtener un registro de eventos real?

Usar constructores con New-Object

Para hacer referencia a un registro de eventos específico, es necesario especificar el nombre del registro. **New-Object** cuenta con el parámetro **ArgumentList**. los argumentos que se pasan como valores a este parámetro se usan en un método especial de inicio del objeto. Este método recibe el nombre de constructor porque se usa para construir el objeto. Por ejemplo, para obtener una referencia al registro de aplicación, debe especificar la cadena 'Application' como argumento:

```
PS> New-Object -TypeName System.Diagnostics.EventLog -ArgumentList Application

Max(K) Retain OverflowAction        Entries Name
------ ------ --------------        ------- ----
16,384      7 OverwriteOlder          2,160 Aplicación
```

Nota:

Puesto que la mayoría de las clases .NET principales están incluidas en el espacio de nombres System, si Windows PowerShell no encuentra el nombre de tipo que especifique, automáticamente intentará buscarlo en este espacio de nombres. Esto significa que puede especificar Diagnostics.EventLog en lugar de System.Diagnostics.EventLog.

Almacenar objetos en variables

Quizá desee almacenar una referencia a un objeto, para así poder utilizarlo durante una sesión de Windows PowerShell. Aunque Windows PowerShell permite realizar muchas tareas con canalizaciones, lo que reduce la necesidad de usar variables, a veces almacenar las referencias a objetos en variables permite manipular más fácilmente dichos objetos.

Windows PowerShell permite crear variables que son básicamente objetos con nombre. El resultado de cualquier comando válido de Windows PowerShell se puede almacenar en una variable. Los nombres de variables siempre comienzan por $. Si desea almacenar la referencia al registro de aplicación en una variable llamada $AppLog, escriba el nombre de la variable seguida de un signo de igual y, a continuación, escriba el comando que se utiliza para crear el objeto del registro de aplicación:

```
PS> $AppLog = New-Object -TypeName System.Diagnostics.EventLog -ArgumentList
Application
```

Si escribe a continuación $AppLog, podrá ver que contiene el registro de aplicación:

```
PS> $AppLog

  Max(K) Retain OverflowAction        Entries Name
  ------ ------ --------------        ------- ----
  16,384      7 OverwriteOlder          2,160 Aplicación
```

Acceso a un registro de eventos remoto con New-Object

Los comandos utilizados en la sección anterior van dirigidos al equipo local; el cmdlet **Get-EventLog** sirve para esta tarea. Para tener acceso al registro de aplicación de un equipo remoto, debe proporcionar el nombre del registro y el nombre del equipo (o dirección IP) como argumentos:

```
PS> $RemoteAppLog = New-Object -TypeName System.Diagnostics.EventLog
Application,192.168.1.81
PS> $RemoteAppLog

  Max(K) Retain OverflowAction        Entries Name
  ------ ------ --------------        ------- ----
     512      7 OverwriteOlder             262 Aplicación
```

Ahora que tenemos una referencia a un registro de eventos almacenado en la variable $RemoteAppLog, ¿qué tareas podemos realizar con él?

Borrar un registro de eventos con métodos de objetos

Los objetos suelen tener métodos que se pueden llamar para realizar tareas. Puede utilizar **Get-Member** para mostrar los métodos asociados a un objeto. El siguiente comando y el resultado seleccionado muestran algunos de los métodos de la clase EventLog:

```
PS> $RemoteAppLog | Get-Member -MemberType Method


   TypeName: System.Diagnostics.EventLog

Name                      MemberType Definition
----                      ---------- ----------
...
Clear                     Method     System.Void Clear()
Close                     Method     System.Void Close()
...
GetType                   Method     System.Type GetType()
...
ModifyOverflowPolicy      Method     System.Void ModifyOverflowPolicy(Overfl...
RegisterDisplayName       Method     System.Void RegisterDisplayName(String ...
...
ToString                  Method     System.String ToString()
WriteEntry                Method     System.Void WriteEntry(String message),...
WriteEvent                Method     System.Void WriteEvent(EventInstance in...
```

El método **Clear()** permite borrar el contenido del registro de eventos. Al llamar a un método, debe especificar siempre el nombre del método entre paréntesis, aunque el método no requiera argumentos. Esto permite a Windows PowerShell distinguir entre el método y una posible propiedad con el mismo nombre. Para llamar al método **Clear**, escriba lo siguiente:

```
PS> $RemoteAppLog.Clear()
```

Escriba lo siguiente para mostrar el registro. Observe que se ha borrado el contenido del registro de eventos y ahora contiene 0 entradas en lugar de 262:

```
PS> $RemoteAppLog

  Max(K) Retain OverflowAction        Entries Name
  ------ ------ --------------        ------- ----
     512      7 OverwriteOlder              0 Application
```

Crear objetos COM con New-Object

Puede utilizar **New-Object** para trabajar con componentes del Modelo de objetos componentes (COM). Estos componentes abarcan desde las diversas bibliotecas que se

incluyen con Windows Script Host (WSH) hasta aplicaciones ActiveX, como Internet Explorer, instaladas en la mayoría de los sistemas.

New-Object usa contenedores a los que se puede llamar en tiempo de ejecución de .NET Framework para crear objetos COM, por lo que cuenta con las mismas limitaciones que .NET al llamar a objetos COM. Para crear un objeto COM, debe especificar el parámetro **ComObject** con el identificador de programación, o ProgID, de la clase COM que desee utilizar. Una descripción completa de las limitaciones de uso de COM y de cómo determinar los ProgID que están disponibles en un sistema está fuera del ámbito de esta Guía básica, pero la mayoría de los objetos conocidos de entornos como WSH pueden usarse en Windows PowerShell.

Puede crear los objetos de WSH especificando estos ProgID: **WScript.Shell**, **WScript.Network**, **Scripting.Dictionary** y **Scripting.FileSystemObject**. Los siguientes comandos crean estos objetos:

```
New-Object -ComObject WScript.Shell
New-Object -ComObject WScript.Network
New-Object -ComObject Scripting.Dictionary
New-Object -ComObject Scripting.FileSystemObject
```

Aunque la mayoría de las funciones de estas clases están disponibles de otras maneras en Windows PowerShell, algunas tareas, como la creación de accesos directos, son más fáciles con clases de WSH.

Crear accesos directos de escritorio con WScript.Shell

Una tarea que se puede realizar fácilmente con un objeto COM es crear un acceso directo. Supongamos que desea crear un acceso directo en el escritorio que vincule a la carpeta principal de Windows PowerShell. Primero debe crear una referencia a **WScript.Shell** y almacenarla en una variable llamada **$WshShell**:

```
$WshShell = New-Object -ComObject WScript.Shell
```

Get-Member se puede usar con objetos COM, por lo que puede explorar los miembros del objeto si escribe:

```
PS> $WshShell | Get-Member


   TypeName: System.__ComObject#{41904400-be18-11d3-a28b-00104bd35090}

Name                      MemberType            Definition
----                      ----------            ----------
AppActivate               Method                bool AppActivate (Variant, Va...
CreateShortcut            Method                IDispatch CreateShortcut (str...
...
```

Nota:

Get-Member cuenta con el parámetro opcional **InputObject**, que puede utilizar en lugar de la canalización para proporcionar la entrada de **Get-Member**. Puede obtener el mismo resultado con el comando **Get-Member -InputObject $WshShell**. Si usa **InputObject**, tratará su argumento como un único elemento. Por tanto, si una variable contiene varios objetos, **Get-Member** los tratará como una matriz de objetos. Por ejemplo:

```
PS> $a = 1,2,"tres"
PS> Get-Member -InputObject $a


   TypeName: System.Object[]


Name                MemberType    Definition
----                ----------    ----------
Count               AliasProperty Count = Length


...
```

El método **WScript.Shell CreateShortcut** acepta un único argumento, la ruta del archivo de acceso directo que se va a crear. Podríamos escribir la ruta completa al escritorio, pero hay una manera más fácil de hacerlo. El escritorio se suele representar mediante una carpeta llamada Escritorio, incluida en la carpeta principal del usuario actual. Windows PowerShell tiene un variable **$Home** que contiene la ruta de acceso a esta carpeta. Podemos especificar la ruta de acceso a la carpeta principal mediante esta variable y, a continuación, agregar el nombre de la carpeta Escritorio y el nombre del acceso directo que se va crear de la siguiente manera:

```
$lnk = $WshShell.CreateShortcut("$Home\Escritorio\PSHome.lnk")
```

Nota:

Si especifica algo parecido al nombre de una variable entre comillas dobles, Windows PowerShell intentará reemplazar un valor que coincida. Si lo especifica entre comillas simples, Windows PowerShell no intentará reemplazar el valor de la variable. Por ejemplo, pruebe a escribir los siguientes comandos:

```
PS> "$Home\Escritorio\PSHome.lnk"
C:\Documents and Settings\aka\Escritorio\PSHome.lnk
PS> '$Home\Escritorio\PSHome.lnk'
```

```
    $Home\Escritorio\PSHome.lnk
```

Ahora tenemos una variable llamada **$lnk** que contiene una nueva referencia de acceso directo. Para ver sus miembros, puede canalizarla a **Get-Member**. El resultado siguiente muestra los miembros que debemos utilizar para terminar de crear el acceso directo:

```
PS> $lnk | Get-Member


   TypeName: System.__ComObject#{f935dc23-1cf0-11d0-adb9-00c04fd58a0b}


Name             MemberType   Definition
----             ----------   ----------
...
Save             Method       void Save ()
...
TargetPath       Property     string TargetPath () {get} {set}
...
```

Hay que especificar **TargetPath**, que es la carpeta de aplicación de Windows PowerShell y, a continuación, guardar el acceso directo **$lnk** llamando al método **Save**. La ruta de acceso a la carpeta de aplicación de Windows PowerShell se almacena en la variable **$PSHome**, por lo que para hacerlo podemos escribir:

```
$lnk.TargetPath = $PSHome
$lnk.Save()
```

Usar Internet Explorer desde Windows PowerShell

Muchas aplicaciones (incluida la familia de aplicaciones Microsoft Office e Internet Explorer) se pueden automatizar mediante COM. Internet Explorer ilustra algunas de las técnicas para trabajar con aplicaciones basadas en COM y los problemas que suelen surgir.

Para crear una instancia de Internet Explorer, debe especificar su ProgID, **InternetExplorer.Application**:

```
$ie = New-Object -ComObject InternetExplorer.Application
```

Este comando inicia Internet Explorer, pero no lo muestra. Si escribe Get-Process, podrá observar que se está ejecutando un proceso llamado iexplore. De hecho, si cierra Windows PowerShell, el proceso seguirá ejecutándose. Deberá reiniciar el equipo o

utilizar una herramienta como el Administrador de tareas para finalizar el proceso iexplore.

Nota:

Los objetos COM que se inician como procesos independientes, llamados habitualmente ejecutables ActiveX, pueden mostrar o no una ventana de interfaz de usuario cuando se inician. Si crean una ventana pero no la muestran, como Internet Explorer, el foco se desplaza generalmente al escritorio de Windows y será necesario que haga visible la ventana para poder interactuar con ella.

Si escribe **$ie | Get-Member**, puede ver las propiedades y los métodos para Internet Explorer. Para que se muestre la ventana de Internet Explorer, establezca la propiedad Visible en $true; para ello, escriba:

```
$ie.Visible = $true
```

Después puede desplazarse a una dirección Web específica con el método Navigate:

```
$ie.Navigate("http://www.microsoft.com/technet/scriptcenter/default.mspx")
```

Otros miembros del modelo de objetos de Internet Explorer permiten recuperar contenido de texto de la página Web. El siguiente comando muestra el texto HTML de la página web actual:

```
$ie.Document.Body.InnerText
```

Para cerrar Internet Explorer desde PowerShell, llame al método Quit() correspondiente:

```
$ie.Quit()
```

Esto hará que se cierre. La variable $ie ya no contiene una referencia válida, aunque sigue pareciendo un objeto COM. Si intenta utilizarla, aparecerá un error de automatización:

```
PS> $ie | Get-Member
Get-Member : Excepción al recuperar la representación de cadena de la propiedad
"Application" : "El objeto invocado ha desconectado de sus clientes. (Exception
from HRESULT: 0x80010108 (RPC_E_DISCONNECTED))"
En línea:1 carácter:16
+ $ie | Get-Member <<<<
```

Puede quitar la referencia que queda con un comando como $ie = $null, o quitar totalmente la variable mediante el comando:

```
Remove-Variable ie
```

Nota:

No hay un estándar común en lo que respecta a que los ejecutables ActiveX se cierren o sigan ejecutándose cuando se quita una referencia a uno de ellos. Según sean las circunstancias (por ejemplo, si la aplicación está visible, si hay abierto un documento modificado en la aplicación e incluso si se está ejecutando Windows PowerShell), la aplicación puede cerrarse o no. Por este motivo, debe comprobar el comportamiento de finalización de cada ejecutable ActiveX que desee utilizar en Windows PowerShell.

Obtener advertencias acerca de objetos COM contenidos en .NET

En algunos casos, un objeto COM puede tener asociado un contenedor al que se puede llamar en tiempo de ejecución de .NET. Este contenedor se puede utilizar en **New-Object**. Dado que el comportamiento del contenedor puede ser distinto al del objeto COM normal, **New-Object** cuenta con un parámetro **Strict** que le avisa del acceso al contenedor. Si especifica el parámetro **Strict** y, a continuación, crea un objeto COM que usa un contenedor, aparecerá un mensaje de advertencia:

```
PS> $xl = New-Object -ComObject Excel.Application -Strict
New-Object : El objeto escrito en la canalización es una instancia del tipo "Mic
rosoft.Office.Interop.Excel.ApplicationClass" del ensamblado de interoperabilidad
primario del componente. Si este tipo expone miembros distintos a los de IDispatch
, los scripts escritos para trabajar con este objeto podrían no funcionar si no
está instalado el ensamblado.
En línea:1 carácter:17
+ $xl = New-Object  <<<< -ComObject Excel.Application -Strict
```

Aunque se crea el objeto, se le avisará de que no es un objeto COM estándar.

Usar clases y métodos estáticos

No todas las clases de .NET Framework se pueden crear con **New-Object**. Por ejemplo, si intenta crear un objeto **System.Environment** o **System.Math** con **New-Object**, aparecerán estos mensajes de error:

```
PS> New-Object System.Environment
New-Object : No se encontró el constructor. No se encuentra ningún constructor
adecuado para
el tipo System.Environment.
En línea:1 carácter:11
+ New-Object  <<<< System.Environment
PS> New-Object System.Math
```

```
New-Object : No se encontró el constructor. No se encuentra ningún constructor
adecuado para
el tipo System.Math.
En línea:1 carácter:11
+ New-Object  <<<< System.Math
```

Estos errores se producen porque no hay manera de crear un objeto a partir de estas clases. Estas clases son bibliotecas de referencia de métodos y propiedades que no cambian el estado. No es necesario crearlas, basta con utilizarlas. Las clases y métodos de este tipo reciben el nombre de clases y métodos estáticos porque no se crean, destruyen ni modifican. Para aclarar esta cuestión, proporcionaremos algunos ejemplos en los que se usan clases estáticas.

Obtener datos de entorno con System.Environment

Normalmente, el primer paso cuando se trabaja con un objeto en Windows PowerShell es utilizar Get-Member para averiguar qué miembros contiene. Con las clases estáticas, el proceso es ligeramente distinto porque la clase no es un objeto.

Hacer referencia a la clase estática System.Environment

Para hacer referencia a una clase estática debe especificar el nombre de la clase entre corchetes. Por ejemplo, puede hacer referencia a **System.Environment** si escribe el nombre entre corchetes. Al hacerlo, se mostrará información general sobre el tipo:

```
PS> [System.Environment]

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    Environment                              System.Object
```

Nota:

Como hemos mencionado anteriormente, Windows PowerShell antepone automáticamente '**System.**' a los nombres de tipos cuando se utiliza **New-Object**. Lo mismo ocurre cuando se usa un nombre de tipo entre corchetes; por tanto, puede especificar **[System.Environment]** como **[Environment]**.

La clase **System.Environment** contiene información general acerca del entorno de trabajo del proceso actual, que es powershell.exe cuando se trabaja en Windows PowerShell.

Si intenta ver información detallada de esta clase con **[System.Environment] | Get-Member**, el tipo de objeto mostrado será **System.RuntimeType**, no **System.Environment**:

```
PS> [System.Environment] | Get-Member

   TypeName: System.RuntimeType
```

Para ver los miembros estáticos con Get-Member, especifique el parámetro **Static**:

```
PS> [System.Environment] | Get-Member -Static


   TypeName: System.Environment

Name                        MemberType Definition
----                        ---------- ----------
Equals                      Method     static System.Boolean Equals(Object ob...
Exit                        Method     static System.Void Exit(Int32 exitCode)
...
CommandLine                 Property   static System.String CommandLine {get;}
CurrentDirectory            Property   static System.String CurrentDirectory ...
ExitCode                    Property   static System.Int32 ExitCode {get;set;}
HasShutdownStarted          Property   static System.Boolean HasShutdownStart...
MachineName                 Property   static System.String MachineName {get;}
NewLine                     Property   static System.String NewLine {get;}
OSVersion                   Property   static System.OperatingSystem OSVersio...
ProcessorCount              Property   static System.Int32 ProcessorCount {get;}
StackTrace                  Property   static System.String StackTrace {get;}
SystemDirectory             Property   static System.String SystemDirectory {...
TickCount                   Property   static System.Int32 TickCount {get;}
UserDomainName              Property   static System.String UserDomainName {g...
UserInteractive             Property   static System.Boolean UserInteractive ...
UserName                    Property   static System.String UserName {get;}
Version                     Property   static System.Version Version {get;}
WorkingSet                  Property   static System.Int64 WorkingSet {get;}
TickCount                                ExitCode
```

Ahora puede seleccionar las propiedades de System.Environment que desea ver.

Mostrar las propiedades estáticas de System.Environment

Las propiedades de System.Environment también son estáticas y deben especificarse de forma distinta que las propiedades normales. Utilizamos **::** para indicar a Windows PowerShell que queremos usar un método o propiedad estáticos. Para ver el comando que se utilizó para iniciar Windows PowerShell, podemos comprobar el valor de la propiedad **CommandLine**; para ello, escriba:

```
PS> [System.Environment]::Commandline
"C:\Archivos de programa\Windows PowerShell\v1.0\powershell.exe"
```

Para comprobar la versión del sistema operativo debe mostrar el valor de la propiedad OSVersion; para ello, escriba:

```
PS> [System.Environment]::OSVersion

          Platform ServicePack         Version             VersionString
          -------- -----------         -------             -------------
           Win32NT Service Pack 2      5.1.2600.131072     Microsoft Window...
```

Para comprobar si el equipo se está apagando, puede mostrar el valor de la propiedad **HasShutdownStarted**:

```
PS> [System.Environment]::HasShutdownStarted
False
```

Operaciones matemáticas con System.Math

La clase estática System.Math resulta útil para realizar algunas operaciones matemáticas. Los miembros importantes de **System.Math** son principalmente métodos, que podemos mostrar con **Get-Member**.

Nota:

System.Math tiene varios métodos con el mismo nombre, pero se distinguen por el tipo de sus parámetros.

Para obtener una lista de los métodos de la clase **System.Math**, escriba el siguiente comando:

```
PS> [System.Math] | Get-Member -Static -MemberType Methods


   TypeName: System.Math

Name            MemberType Definition
----            ---------- ----------
Abs             Method     static System.Single Abs(Single value), static Sy...
Acos            Method     static System.Double Acos(Double d)
Asin            Method     static System.Double Asin(Double d)
Atan            Method     static System.Double Atan(Double d)
Atan2           Method     static System.Double Atan2(Double y, Double x)
BigMul          Method     static System.Int64 BigMul(Int32 a, Int32 b)
Ceiling         Method     static System.Double Ceiling(Double a), static Sy...
Cos             Method     static System.Double Cos(Double d)
Cosh            Method     static System.Double Cosh(Double value)
DivRem          Method     static System.Int32 DivRem(Int32 a, Int32 b, Int3...
Equals          Method     static System.Boolean Equals(Object objA, Object ...
Exp             Method     static System.Double Exp(Double d)
Floor           Method     static System.Double Floor(Double d), static Syst...
IEEERemainder   Method     static System.Double IEEERemainder(Double x, Doub...
Log             Method     static System.Double Log(Double d), static System...
Log10           Method     static System.Double Log10(Double d)
```

```
Max             Method     static System.SByte Max(SByte val1, SByte val2), ...
Min             Method     static System.SByte Min(SByte val1, SByte val2), ...
Pow             Method     static System.Double Pow(Double x, Double y)
ReferenceEquals Method     static System.Boolean ReferenceEquals(Object objA...
Round           Method     static System.Double Round(Double a), static Syst...
Sign            Method     static System.Int32 Sign(SByte value), static Sys...
Sin             Method     static System.Double Sin(Double a)
Sinh            Method     static System.Double Sinh(Double value)
Sqrt            Method     static System.Double Sqrt(Double d)
Tan             Method     static System.Double Tan(Double a)
Tanh            Method     static System.Double Tanh(Double value)
Truncate        Method     static System.Decimal Truncate(Decimal d), static...
```

Se muestran varios métodos matemáticos. A continuación se muestra una lista de comandos que demuestran cómo se usan algunos métodos comunes:

```
PS> [System.Math]::Sqrt(9)
3
PS> [System.Math]::Pow(2,3)
8
PS> [System.Math]::Floor(3.3)
3
PS> [System.Math]::Floor(-3.3)
-4
PS> [System.Math]::Ceiling(3.3)
4
PS> [System.Math]::Ceiling(-3.3)
-3
PS> [System.Math]::Max(2,7)
7
PS> [System.Math]::Min(2,7)
2
PS> [System.Math]::Truncate(9.3)
9
PS> [System.Math]::Truncate(-9.3)
-9
```

Eliminar objetos de la canalización (Where-Object)

En Windows PowerShell, a menudo se generan y pasan a una canalización más objetos de los deseados. Puede especificar las propiedades de objetos específicos que desee mostrar mediante los cmdlets **Format**, pero esto no ayuda a quitar objetos completos de la presentación. Quizá desee filtrar objetos antes del final de una canalización, para así poder realizar acciones sólo en un subconjunto de los objetos generados inicialmente.

Windows PowerShell incluye un cmdlet **Where-Object** que permite probar cada objeto de la canalización y pasarlo por ésta sólo si cumple una determinada condición de prueba. Los objetos que no superen la prueba se quitan de la canalización. La condición de prueba se proporciona como el valor del parámetro **Where-ObjectFilterScript**.

Realizar pruebas sencillas con Where-Object

El valor de **FilterScript** es un bloque de script (uno o más comandos de Windows PowerShell especificados entre llaves {}) que se evalúa como True o False. Estos bloques de script pueden ser muy sencillos, pero para crearlos hay que conocer otro concepto de Windows PowerShell: los operadores de comparación. Un operador de este tipo compara los elementos que aparecen a cada lado del mismo. Los operadores de comparación comienzan por un carácter '-' seguido de un nombre. Los operadores de comparación básicos se pueden usar con prácticamente cualquier tipo de objeto, mientras que algunos de los más avanzados sólo pueden utilizarse con texto o matrices.

Nota:

De manera predeterminada, cuando se trabaja con texto, los operadores de comparación de Windows PowerShell no distinguen entre mayúsculas y minúsculas.

Para facilitar el análisis del código no se utilizan los símbolos <, > y = como operadores de comparación. En su lugar, los operadores de comparación están formados por letras. En la siguiente tabla se muestran los operadores de comparación básicos:

Operador de comparación	Significado	Ejemplo (devuelve el valor True)
-eq	Es igual a	1 -eq 1
-ne	Es distinto de	1 -ne 2
-lt	Es menor que	1 -lt 2
-le	Es menor o igual que	1 -le 2
-gt	Es mayor que	2 -gt 1
-ge	Es mayor o igual que	2 -ge 1
-like	Es como (comparación de caracteres comodín para texto)	"file.doc" -like "f*.do?"
-notlike	No es como (comparación de caracteres comodín para texto)	"file.doc" -notlike "p*.doc"
-contains	Contiene	1,2,3 -contains 1

Operador de comparación	Significado	Ejemplo (devuelve el valor True)
-notcontains	No contiene	1,2,3 -notcontains 4

Los bloques de script de Where-Object usan la variable especial '$_' para hacer referencia al objeto actual en la canalización. A continuación se muestra un ejemplo de cómo funciona. Si tiene una lista de números y desea que sólo se devuelvan los que sean inferiores a 3, puede utilizar Where-Object para filtrar los números; para ello, escriba:

```
PS> 1,2,3,4 | Where-Object -FilterScript {$_ -lt 3}
1
2
```

Filtrado basado en propiedades de objetos

Dado que $_ hace referencia al objeto de canalización actual, podemos tener acceso a sus propiedades para nuestras pruebas.

A modo de ejemplo, examinemos la clase Win32_SystemDriver de WMI. Puede haber cientos de controladores en un sistema determinado, pero quizá sólo esté interesado en un conjunto específico de controladores del sistema, como los que se están ejecutando actualmente. Si utiliza Get-Member para ver los miembros de Win32_SystemDriver (**Get-WmiObject -Class Win32_SystemDriver | Get-Member -MemberType Property**), observará que la propiedad relevante es State y que tiene el valor "Running" cuando el controlador se está ejecutando. Para filtrar los controladores del sistema y seleccionar sólo los que se están ejecutando, escriba:

```
Get-WmiObject -Class Win32_SystemDriver | Where-Object -FilterScript {$_.State -eq
"Running"}
```

La lista generada seguirá siendo muy larga. Quizá desee filtrar de manera que se seleccionen sólo los controladores que se inician automáticamente mediante una prueba del valor de StartMode:

```
PS> Get-WmiObject -Class Win32_SystemDriver | Where-Object -FilterScript {$_.State
-eq "Running"} | Where-Object -FilterScript {$_.StartMode -eq "Auto"}

DisplayName : RAS Asynchronous Media Driver
Name        : AsyncMac
State       : Running
Status      : OK
Started     : True

DisplayName : Audio Stub Driver
Name        : audstub
```

```
State       : Running
Status      : OK
Started     : True
```

Esto proporciona mucha información que no necesitamos, puesto que sabemos que los controladores se están ejecutando. De hecho, probablemente la única información que necesitamos en este momento son el nombre común y el nombre para mostrar. El siguiente comando incluye únicamente estas dos propiedades, por lo que la salida es mucho más sencilla:

```
PS> Get-WmiObject -Class Win32_SystemDriver | Where-Object -FilterScript {$_.State
-eq "Running"} | Where-Object -FilterScript {$_.StartMode -eq "Manual"} | Format-
Table -Property Name,DisplayName

Name                                    DisplayName
----                                    -----------
AsyncMac                                RAS Asynchronous Media Driver
Fdc                                     Floppy Disk Controller Driver
Flpydisk                                Floppy Disk Driver
Gpc                                     Generic Packet Classifier
IpNat                                   IP Network Address Translator
mouhid                                  Mouse HID Driver
MRxDAV                                  WebDav Client Redirector
mssmbios                                Microsoft System Management BIOS Driver
```

El comando anterior contiene dos elementos Where-Object, pero se pueden expresar en un solo elemento Where-Object con el operador lógico -and, de esta manera:

```
Get-WmiObject -Class Win32_SystemDriver | Where-Object -FilterScript { ($_.State -
eq "Running") -and ($_.StartMode -eq "Manual") } | Format-Table -Property
Name,DisplayName
```

En la siguiente tabla se enumeran los operadores lógicos estándar:

Operador lógico	Significado	Ejemplo (devuelve el valor True)
-and	And lógico; devuelve True si ambos lados son True.	(1 -eq 1) -and (2 -eq 2)
-or	Or lógico; devuelve True si cualquiera de los lados es True.	(1 -eq 1) -or (1 -eq 2)
-not	Not lógico; invierte True y False.	-not (1 -eq 2)
!	Not lógico; invierte True y False.	!(1 -eq 2)

Repetir una tarea para varios objetos (ForEach-Object)

El cmdlet **ForEach-Object** usa bloques de script y el descriptor $_ para el objeto de canalización actual a fin de que pueda ejecutar un comando en cada objeto de la canalización. Esto se puede utilizar para realizar algunas tareas complicadas.

Una situación en la que puede resultar conveniente es cuando se manipulan datos para que sean más útiles. Por ejemplo, se puede usar la clase Win32_LogicalDisk de WMI para devolver información sobre el espacio libre en cada disco local. No obstante, los datos se devuelven en bytes, lo que dificulta su lectura:

```
PS> Get-WmiObject -Class Win32_LogicalDisk


DeviceID     : C:
DriveType    : 3
ProviderName :
FreeSpace    : 50665070592
Size         : 203912880128
VolumeName   : Local Disk
```

Podemos convertir el valor de FreeSpace en megabytes si dividimos cada valor entre 1024 dos veces; tras la primera división, los datos se muestran en kilobytes y, tras la segunda, en megabytes. Puede realizar esto con un bloque de script ForEach-Object:

```
Get-WmiObject -Class Win32_LogicalDisk | ForEach-Object -Process
{($_.FreeSpace)/1024,0/1024,0}
48318,01171875
```

Por desgracia, ahora el resultado son datos que no tienen asociadas etiquetas. Dado que las propiedades de WMI de este tipo son de sólo lectura, no puede convertir directamente FreeSpace. Si escribe esto:

```
Get-WmiObject -Class Win32_LogicalDisk | ForEach-Object -Process {$_.FreeSpace =
($_.FreeSpace)/1024,0/1024,0}
```

Aparecerá un mensaje de error:

```
"FreeSpace" es una propiedad de sólo lectura.
En línea:1 carácter:70
+ Get-WmiObject -Class Win32_LogicalDisk | ForEach-Object -Process {$_.F <<<< r
eeSpace = ($_.FreeSpace)/1024,0/1024,0}
```

Podría reorganizar los datos con algunas técnicas avanzadas, pero un método más sencillo es crear un nuevo objeto, con **Select-Object**.

Seleccionar partes de objetos (Select-Object)

El cmdlet **Select-Object** permite crear nuevos objetos personalizados de Windows PowerShell que contienen propiedades seleccionadas de los objetos utilizados para crearlos. Para crear un nuevo objeto que incluya sólo las propiedades Name y FreeSpace de la clase Win32_LogicalDisk de WMI, escriba el siguiente comando:

```
PS> Get-WmiObject -Class Win32_LogicalDisk | Select-Object -Property
Name,FreeSpace

Name                                   FreeSpace
----                                   ---------
C:                                     50664845312
```

No podrá ver el tipo de datos después de emitir este comando, pero si canaliza el resultado a Get-Member después de Select-Object, podrá ver que tiene un nuevo tipo de objeto, un PSCustomObject:

```
PS> Get-WmiObject -Class Win32_LogicalDisk | Select-Object -Property
Name,FreeSpace| Get-Member


   TypeName: System.Management.Automation.PSCustomObject

Name        MemberType   Definition
----        ----------   ----------
Equals      Method       System.Boolean Equals(Object obj)
GetHashCode Method       System.Int32 GetHashCode()
GetType     Method       System.Type GetType()
ToString    Method       System.String ToString()
FreeSpace   NoteProperty  FreeSpace=...
Name        NoteProperty System.String Name=C:
```

Select-Object tiene muchas aplicaciones. Una de ellas es replicar datos para modificarlos posteriormente. Ahora podemos resolver el problema que nos encontramos en la sección anterior. Podemos actualizar el valor de FreeSpace en nuestros objetos recién creados y el resultado incluirá la etiqueta descriptiva:

```
Get-WmiObject -Class Win32_LogicalDisk | Select-Object -Property Name,FreeSpace |
ForEach-Object -Process {$_.FreeSpace = ($_.FreeSpace)/1024,0/1024,0; $_}
Name                                                                   FreeSpace
----                                                                   ---------
C:                                                                  48317,7265625
```

Ordenar objetos

Podemos organizar los datos mostrados para facilitar su búsqueda con el cmdlet **Sort-Object**. Este cmdlet utiliza el nombre de una o más propiedades para ordenar y devuelve los datos ordenados por los valores de estas propiedades.

Considere el problema que supone enumerar las instancias de Win32_SystemDriver. Si desea ordenar por **State** y luego por **Name**, debe escribir:

```
Get-WmiObject -Class Win32_SystemDriver | Sort-Object -Property State,Name |
Format-Table -Property Name,State,Started,DisplayName -AutoSize -Wrap
```

Aunque la presentación es larga, puede observar que los elementos con el mismo estado están agrupados:

```
Name            State   Started DisplayName
----            -----   ------- -----------
ACPI            Running    True Microsoft ACPI Driver
AFD             Running    True AFD
AmdK7           Running    True AMD K7 Processor Driver
AsyncMac        Running    True RAS Asynchronous Media Driver
...
Abiosdsk        Stopped   False Abiosdsk
ACPIEC          Stopped   False ACPIEC
aec             Stopped   False Microsoft Kernel Acoustic Echo Canceller
...
```

También puede ordenar los objetos en orden inverso especificando el parámetro **Descending**. Esto invierte el criterio de ordenación de manera que los nombres se ordenan en orden alfabético inverso y los números en orden descendente.

```
PS> Get-WmiObject -Class Win32_SystemDriver | Sort-Object -Property State,Name -
Descending | Format-Table -Property Name,State,Started,DisplayName -AutoSize -Wrap

Name            State   Started DisplayName
----            -----   ------- -----------
WS2IFSL         Stopped   False Windows Socket 2.0 Non-IFS Service Provider Supp
                                ort Environment
wceusbsh        Stopped   False Windows CE USB Serial Host Driver...
...
wdmaud          Running    True Microsoft WINMM WDM Audio Compatibility Driver
Wanarp          Running    True Remote Access IP ARP Driver
...
```

Usar variables para almacenar objetos

Windows PowerShell trabaja con objetos. Permite crear variables, que son básicamente objetos con nombre, para almacenar los resultados y poder utilizarlos más adelante. Si está acostumbrado a trabajar con variables en otros shells, recuerde que las variables de Windows PowerShell son objetos, no texto.

Las variables se especifican siempre con el carácter inicial $ y pueden contener cualquier carácter alfanumérico o el carácter de subrayado en sus nombres.

Crear una variable

Para crear una variable debe escribir un nombre de variable válido:

```
PS> $loc
PS>
```

No se devuelven resultados porque **$loc** no tiene un valor. Puede crear una variable y asignarle un valor en el mismo paso. Windows PowerShell crea sólo la variable si no existe; de lo contrario, asigna el valor especificado a la variable existente. Para almacenar la ubicación actual en la variable **$loc**, escriba:

```
$loc = Get-Location
```

Cuando se escribe este comando, no se muestran resultados porque se envían a $loc. En Windows PowerShell, los resultados mostrados son un efecto secundario: los datos que no se redirigen a alguna ubicación siempre terminan enviándose a la pantalla. Si escribe $loc, se mostrará la ubicación actual:

```
PS> $loc

Path
----
C:\temp
```

Puede utilizar **Get-Member** para mostrar información acerca del contenido de variables. La canalización de $loc a Get-Member mostrará que se trata de un objeto **PathInfo**, el mismo resultado que se obtiene con Get-Location:

```
PS> $loc | Get-Member -MemberType Property


   TypeName: System.Management.Automation.PathInfo

Name         MemberType Definition
----         ---------- ----------
```

```
Drive        Property   System.Management.Automation.PSDriveInfo Drive {get;}
Path         Property   System.String Path {get;}
Provider     Property   System.Management.Automation.ProviderInfo Provider {...
ProviderPath Property   System.String ProviderPath {get;}
```

Manipular variables

Windows PowerShell incluye varios comandos que permiten manipular variables. Para ver una lista completa en un formato legible, escriba:

```
Get-Command -Noun Variable | Format-Table -Property Name,Definition -AutoSize -
Wrap
```

Además de las variables que cree en la sesión actual de Windows PowerShell, hay varias variables definidas por el sistema. Puede utilizar el cmdlet **Remove-Variable** para borrar todas las variables no controladas por Windows PowerShell. Para borrar todas las variables, escriba el siguiente comando:

```
Remove-Variable -Name * -Force -ErrorAction SilentlyContinue
```

Se generará la siguiente solicitud de confirmación:

```
Confirmar
¿Está seguro de que desea realizar esta acción?
Realizando la operación "Quitar variable" en el destino "Nombre: Error".
[S] Sí  [O] Sí a todo  [N] No  [T] No a todo  [S] Suspender  [?] Ayuda
(el valor predeterminado es "S"):O
```

Si después ejecuta el cmdlet **Get-Variable**, podrá ver las demás variables de Windows PowerShell. Como hay una unidad de Windows PowerShell para las variables, también puede mostrar todas las variables de Windows PowerShell mediante el siguiente comando:

```
Get-ChildItem variable:
```

Usar variables de Cmd.exe

Aunque Windows PowerShell no es Cmd.exe, se ejecuta en un entorno de shell de comandos y puede usar las mismas variables disponibles en cualquier entorno de Windows. Estas variables se exponen a través de la unidad **env**:. Para ver estas variables, escriba:

```
Get-ChildItem env:
```

Aunque los cmdlets de variables estándar no están diseñados para trabajar con variables **env:**, puede utilizarlos si especifica el prefijo **env:**. Por ejemplo, para ver el directorio raíz del sistema operativo, puede usar la variable **%SystemRoot%** del shell de comandos desde Windows PowerShell:

```
PS> $env:SystemRoot
C:\WINDOWS
```

También puede crear y modificar variables de entorno en Windows PowerShell. Las variables de entorno a las que se tiene acceso desde Windows PowerShell cumplen las normas habituales para las variables de entorno de cualquier otra ubicación de Windows.

Usar Windows PowerShell para tareas de administración

El objetivo fundamental de Windows PowerShell es proporcionar un control administrativo mejor y más sencillo de los sistemas, de forma interactiva o mediante scripts. En este capítulo se presentan soluciones a muchos problemas específicos de la administración de sistemas Windows con Windows PowerShell. Aunque no hemos hablado de scripts ni funciones en la Guía básica de Windows PowerShell, estas soluciones se pueden aplicar con scripts o funciones más adelante. Mostraremos ejemplos que incluyen funciones como parte de la solución para resolver los problemas.

En la descripción de las soluciones se incluyen combinaciones de soluciones que utilizan cmdlets específicos, el cmdlet general Get-WmiObject e incluso herramientas externas que forman parte de las infraestructuras de Windows y .NET. El uso de herramientas externas constituye un objetivo de diseño a largo plazo de Windows PowerShell. A medida que crezca el sistema, los usuarios seguirán encontrándose con situaciones en las que las herramientas disponibles no hacen todo lo que necesitan que hagan. En lugar de fomentar la dependencia exclusiva de las implementaciones de cmdlets, Windows PowerShell intenta admitir la integración de soluciones de todos los escenarios alternativos posibles.

Administrar procesos locales

Hay únicamente dos cmdlets Process principales: **Get-Process** y **Stop-Process**. Puesto que es posible inspeccionar y filtrar procesos utilizando parámetros o los cmdlets Object, puede realizar algunas tareas complejas con sólo estos dos cmdlets.

Mostrar la lista de procesos (Get-Process)

Para obtener una lista de todos los procesos que se ejecutan en el sistema local, ejecute **Get-Process** sin parámetros.

También puede devolver un único proceso si especifica el Id. del proceso (ProcessId) con el parámetro Id. En el siguiente ejemplo se devuelve el proceso Idle (inactivo) del sistema:

```
PS> Get-Process -Id 0
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
      0       0        0         16     0               0 Idle
```

Aunque es normal que los cmdlets no devuelvan datos en algunas situaciones, cuando se especifica un proceso por su ProcessId, **Get-Process** genera un error si no encuentra coincidencias, ya que lo que intenta normalmente es recuperar un proceso conocido que se esté ejecutando. Si no hay ningún proceso con ese Id., lo más probable es que el Id. no sea el correcto o que el proceso en cuestión ya se haya cerrado:

```
PS> Get-Process -Id 99
Get-Process : No se encuentra ningún proceso con el identificador 99.
En línea:1 carácter:12
+ Get-Process  <<<< -Id 99
```

El parámetro Name permite especificar un subconjunto de procesos según el nombre de proceso. Dado que algunos procesos pueden tener el mismo nombre, el resultado puede incluir varios procesos. Si no hay ningún proceso con ese nombre, **Get-Process** devolverá un error como cuando se especifica un ProcessId. Por ejemplo, si especifica el nombre de proceso **explore** en lugar de **explorer**:

```
PS> Get-Process -Name explore
Get-Process : No se encuentra ningún proceso con el nombre 'explore'.
En línea:1 carácter:12
+ Get-Process  <<<< -Name explore
```

El parámetro Name admite el uso de caracteres comodín, por lo que puede escribir los primeros caracteres de un nombre seguidos de un asterisco para obtener una lista:

```
PS> Get-Process -Name ex*
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    234       7     5572      12484   134     2.98   1684 EXCEL
    555      15    34500      12384   134   105.25    728 explorer
```

Nota:

Como la base para los procesos de Windows PowerShell es la clase System.Diagnostics.Process de .NET, Windows PowerShell sigue algunas de las convenciones que utiliza System.Diagnostics.Process. Una de estas convenciones es que el nombre de proceso correspondiente a un archivo ejecutable no puede contener nunca ".exe" al final del nombre del ejecutable.

Get-Process acepta también varios valores para el parámetro Name. Como cuando se especifica un único nombre, aparecerá un error si no se encuentran coincidencias con un nombre, aunque también obtendrá el resultado habitual para procesos que coincidan:

```
PS> Get-Process -Name exp*,power*,NotAProcess
Get-Process : No se encuentra ningún proceso con el nombre 'NotAProcess'.
En línea:1 carácter:12
+ Get-Process  <<<< -Name exp*,power*,svchost,NotAProcess
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    540      15    35172      48148   141    88.44    408 explorer
    605       9    30668      29800   155     7.11   3052 powershell
```

Detener procesos (Stop-Process)

Windows PowerShell proporciona flexibilidad a la hora de crear listas de procesos pero, ¿y para detener procesos?

El cmdlet **Stop-Process** usa un nombre o un Id. para especificar un proceso que se desea detener. La capacidad del usuario para detener procesos depende de los permisos que tenga. Algunos procesos no se pueden detener. Por ejemplo, si intenta detener el proceso inactivo, aparecerá un error:

```
PS> Stop-Process -Name Idle
Stop-Process : No se puede detener el proceso 'Idle (0)' debido al error
siguiente:
 Acceso denegado
En línea:1 carácter:13
+ Stop-Process  <<<< -Name Idle
```

También puede forzar la solicitud de confirmación con el parámetro **Confirm**. Este parámetro resulta especialmente útil si incluye un carácter comodín al especificar el nombre del proceso, ya que puede buscar por equivocación algunos procesos que no desea detener:

```
PS> Stop-Process -Name t*,e* -Confirm
Confirmar
¿Está seguro de que desea realizar esta acción?
```

```
Realizando la operación "Stop-Process" en el destino "explorer (408)".
[S] Sí  [O] Sí a todo  [N] No  [T] No a todo  [S] Suspender  [?] Ayuda
(el valor predeterminado es "S"):n
Confirmar
¿Está seguro de que desea realizar esta acción?
Realizando la operación "Stop-Process" en el destino "taskmgr (4072)".
[S] Sí  [O] Sí a todo  [N] No  [T] No a todo  [S] Suspender  [?] Ayuda
(el valor predeterminado es "S"):n
```

Puede manipular procesos complejos con algunos de los cmdlets para filtrar objetos. Dado que un objeto Process tiene una propiedad Responding con el valor True cuando ya no responde, puede detener todas las aplicaciones que dejen de responder con el siguiente comando:

```
Get-Process | Where-Object -FilterScript {$_.Responding -eq $false} | Stop-Process
```

Puede aplicar el mismo enfoque en otras situaciones. Por ejemplo, supongamos que una aplicación secundaria de la bandeja del sistema se ejecuta automáticamente cuando los usuarios inician otra aplicación. Puede que no desee mantener este comportamiento en sesiones de Servicios de Terminal Server, pero sí en sesiones que se ejecuten en la consola del equipo físico. Las sesiones conectadas al escritorio de un equipo físico tienen siempre el Id. de sesión 0; por tanto, puede detener todas las instancias del proceso que se encuentren en otras sesiones con **Where-Object** y el proceso, **SessionId**:

```
Get-Process -Name BadApp | Where-Object -FilterScript {$_.SessionId -neq 0} |
Stop-Process
```

Detener todas las demás sesiones de Windows PowerShell

En algunas ocasiones, puede resultar útil poder detener todas las sesiones de Windows PowerShell que se estén ejecutando, menos la sesión actual. Si una sesión está utilizando demasiados recursos o no es posible tener acceso a ella (puede estar ejecutándose de forma remota o en otra sesión del escritorio), es posible que no pueda detenerla directamente. No obstante, si intenta detener todas las sesiones que se están ejecutando, es posible que lo que termine sea la sesión actual.

Cada sesión de Windows PowerShell tiene un PID de variable de entorno que contiene el Id. del proceso de Windows PowerShell. Puede comparar el $PID con el Id. de cada sesión y terminar únicamente las sesiones de Windows PowerShell que tengan un Id. distinto. El siguiente comando de canalización realiza esta tarea y devuelve una lista de las sesiones terminadas (debido a que se usa el parámetro **PassThru**):

```
PS> Get-Process -Name powershell | Where-Object -FilterScript {$_.Id -ne $PID} |
Stop-Process -
```

```
PassThru
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    334       9    23348      29136   143     1.03    388 powershell
    304       9    23152      29040   143     1.03    632 powershell
    302       9    20916      26804   143     1.03   1116 powershell
    335       9    25656      31412   143     1.09   3452 powershell
    303       9    23156      29044   143     1.05   3608 powershell
    287       9    21044      26928   143     1.02   3672 powershell
```

Administrar servicios locales

Hay ocho cmdlets Service principales, diseñados para una amplia variedad de tareas relacionadas con servicios. Explicaremos únicamente cómo se enumeran y cambian los estados de ejecución de servicios, pero puede obtener una lista de los cmdlets Service con **Get-Help *-Service** y también puede obtener información acerca de cada cmdlet Service con **Get-Help<nombreCmdlet>** (p. ej., **Get-Help New-Service**).

Mostrar la lista de servicios

Puede mostrar los servicios locales de un equipo con **Get-Service**. Al igual que con **Get-Process**, el uso del comando **Get-Service** sin parámetros devuelve todos los servicios. Puede filtrarlos por nombre e incluso utilizar un asterisco como carácter comodín:

```
PS> Get-Service -Name se*
Status   Name               DisplayName
------   ----               -----------
Running  seclogon           Secondary Logon
Running  SENS               System Event Notification
Stopped  ServiceLayer       ServiceLayer
```

Dado que no siempre es evidente cuál es el nombre real del servicio, puede encontrarse con que necesita buscar los servicios por el nombre para mostrar. Puede hacerlo por un nombre específico, con caracteres comodín o utilizando una lista de nombres para mostrar:

```
PS> Get-Service -DisplayName se*
Status   Name               DisplayName
------   ----               -----------
Running  lanmanserver       Server
Running  SamSs              Security Accounts Manager
Running  seclogon           Secondary Logon
Stopped  ServiceLayer       ServiceLayer
Running  wscsvc             Security Center
PS> Get-Service -DisplayName ServiceLayer,Server
```

```
Status   Name               DisplayName
------   ----               -----------
Running  lanmanserver       Server
Stopped  ServiceLayer       ServiceLayer
```

Detener, iniciar, suspender y reiniciar servicios

Los cmdlets Service tienen todos el mismo formato general. Puede especificar los servicios por el nombre común o el nombre para mostrar, así como usar listas y caracteres comodín como valores. Para detener la cola de impresión, utilice:

```
Stop-Service -Name spooler
```

Para iniciar la cola de impresión después de haberla detenido, utilice:

```
Start-Service -Name spooler
```

Para suspender la cola de impresión, utilice:

```
Suspend-Service -Name spooler
```

El cmdlet **Restart-Service** funciona de la misma manera que los otros cmdlets Service, pero mostraremos algunos ejemplos más complejos de este cmdlet. Con el uso más sencillo, puede especificar el nombre del servicio:

```
PS> Restart-Service -Name spooler
ADVERTENCIA: Esperando a que termine de iniciarse el servicio 'Print Spooler
(Spooler)'...
ADVERTENCIA: Esperando a que termine de iniciarse el servicio 'Print Spooler
(Spooler)'...
PS>
```

Observará que aparece varias veces un mensaje de advertencia relativo al inicio de la cola de impresión. Si realiza una operación de servicio que tarda cierto tiempo en completarse, Windows PowerShell le informará de que todavía está intentando realizar la tarea.

Si desea reiniciar varios servicios, puede obtener una lista de los servicios, filtrarlos y, a continuación, efectuar el reinicio:

```
PS> Get-Service | Where-Object -FilterScript {$_.CanStop} | Restart-Service
ADVERTENCIA: Esperando a que termine de detenerse el servicio 'Computer Browser
(Browser)'...
ADVERTENCIA: Esperando a que termine de detenerse el servicio 'Computer Browser
(Browser)'...
Restart-Service : No se puede detener el servicio 'Logical Disk Manager
(dmserver)' porque
 tiene servicios dependientes. Sólo se puede detener si está establecida la marca
```

```
Force.
En línea:1 carácter:57
+ Get-Service | Where-Object -FilterScript {$_.CanStop} | Restart-Service <<<<
ADVERTENCIA: Esperando a que termine de iniciarse el servicio 'Print Spooler
(Spooler)'...
ADVERTENCIA: Esperando a que termine de iniciarse el servicio 'Print Spooler
(Spooler)'...
```

Recopilar información acerca de equipos

Get-WmiObject es el cmdlet más importante para tareas generales de administración de sistemas. Toda la configuración fundamental de los subsistemas se expone mediante WMI. Además, WMI trata los datos como objetos incluidos en conjuntos de uno o más elementos. Dado que Windows PowerShell también trabaja con objetos y cuenta con una canalización que permite tratar uno o varios objetos de la misma manera, el acceso general a WMI permite realizar algunas tareas avanzadas con muy poco esfuerzo.

En los siguientes ejemplos se muestra cómo recopilar información específica con **Get-WmiObject** respecto a un equipo arbitrario. El valor especificado del parámetro **ComputerName** es un punto (**.**), que representa al equipo local. Puede especificar un nombre o una dirección IP correspondiente a cualquier equipo al que pueda tener acceso mediante WMI. Para recuperar información acerca del equipo local, puede omitir **-ComputerName.**

Mostrar la lista de configuraciones de escritorio

Empezamos con un comando que recopila información acerca de los escritorios del equipo local.

```
Get-WmiObject -Class Win32_Desktop -ComputerName .
```

Este comando devuelve información de todos los escritorios, estén o no en uso.

Nota:

La información que devuelven algunas clases de WMI puede ser muy detallada e incluye a menudo metadatos acerca de dicha clase. Dado que la mayoría de estas propiedades de metadatos tienen nombres que comienzan por dos caracteres de subrayado, puede filtrar las propiedades con Select-Object. Puede especificar únicamente las propiedades que comiencen por caracteres alfabéticos si utiliza **[a-z]*** como el valor de Property. Por ejemplo:

```
Get-WmiObject -Class Win32_Desktop -ComputerName . | Select-Object -Property [a-z]*
```

Para filtrar los metadatos, utilice un operador de canalización (|) para enviar los resultados del comando Get-WmiObject a **Select-Object -Property [a-z]***.

Mostrar información del BIOS

La clase Win32_BIOS de WMI devuelve información bastante compacta y completa acerca del BIOS del sistema en el equipo local:

```
Get-WmiObject -Class Win32_BIOS -ComputerName .
```

Mostrar información de procesadores

Puede recuperar información general de procesadores con la clase **Win32_Processor** de WMI, aunque es probable que desee filtrar la información:

```
Get-WmiObject -Class Win32_Processor -ComputerName . | Select-Object -Property [a-z]*
```

Para obtener una cadena de descripción general de la familia de procesadores, basta con que se devuelva la propiedad **Win32_ComputerSystemSystemType**:

```
PS> Get-WmiObject -Class Win32_ComputerSystem -ComputerName . | Select-Object -Property SystemType
SystemType
----------
X86-based PC
```

Mostrar el fabricante y el modelo del equipo

La información del modelo del equipo también está disponible en **Win32_ComputerSystem**. No es necesario filtrar el resultado estándar mostrado para proporcionar los datos de OEM:

```
PS> Get-WmiObject -Class Win32_ComputerSystem
Domain              : WORKGROUP
Manufacturer        : Compaq Presario 06
Model               : DA243A-ABA 6415cl NA910
Name                : MyPC
PrimaryOwnerName    : Jane Doe
TotalPhysicalMemory : 804765696
```

El resultado de comandos como éste, que devuelven información directamente de algunos componentes de hardware, será tan válido como los datos de los que usted disponga. Algunos fabricantes de hardware pueden no haber configurado correctamente

algún tipo de información y, por tanto, es posible que esta información no esté disponible.

Mostrar las revisiones instaladas

Puede mostrar todas las revisiones instaladas con **Win32_QuickFixEngineering**:

```
Get-WmiObject -Class Win32_QuickFixEngineering -ComputerName .
```

Esta clase devuelve una lista de revisiones similar a la siguiente:

```
Description         : Update for Windows XP (KB910437)
FixComments         : Update
HotFixID            : KB910437
Install Date        :
InstalledBy         : Administrator
InstalledOn         : 12/16/2005
Name                :
ServicePackInEffect : SP3
Status              :
```

Para que el resultado sea más conciso, quizá desee excluir algunas propiedades. Aunque puede utilizar el parámetro **Get-WmiObject Property** para seleccionar sólo el **HotFixID**, en realidad esta acción devuelve más información, ya que todos los metadatos se muestran de manera predeterminada:

```
PS> Get-WmiObject -Class Win32_QuickFixEngineering -ComputerName . -Property
HotFixId
HotFixID         : KB910437
__GENUS          : 2
__CLASS          : Win32_QuickFixEngineering
__SUPERCLASS     :
__DYNASTY        :
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
```

Se devuelven los datos adicionales porque el parámetro Property de **Get-WmiObject** limita las propiedades devueltas de instancias de clases de WMI, no el objeto devuelto a Windows PowerShell. Para reducir el resultado, utilice **Select-Object**:

```
PS> Get-WmiObject -Class Win32_QuickFixEngineering -ComputerName . -Property Hot
FixId | Select-Object -Property HotFixId
HotFixId
--------
KB910437
```

Mostrar información de versión del sistema operativo

Las propiedades de la clase **Win32_OperatingSystem** incluyen información sobre la versión del sistema operativo y los Service Pack instalados. Puede seleccionar explícitamente estas propiedades únicamente para obtener un resumen de la información de versión de **Win32_OperatingSystem**:

```
Get-WmiObject -Class Win32_OperatingSystem -ComputerName . | Select-Object -
Property
BuildNumber,BuildType,OSType,ServicePackMajorVersion,ServicePackMinorVersion
```

También puede utilizar caracteres comodín con el parámetro **Select-Object Property**. Como todas las propiedades que comienzan por **Build** o **ServicePack** son importantes en este caso, hemos reducido el resultado de la siguiente manera:

```
PS> Get-WmiObject -Class Win32_OperatingSystem -ComputerName . | Select-Object -
Property Build*,OSType,ServicePack*

BuildNumber             : 2600
BuildType               : Uniprocessor Free
OSType                  : 18
ServicePackMajorVersion : 2
ServicePackMinorVersion : 0
```

Mostrar los usuarios y propietarios locales

La información general sobre los usuarios locales (número de usuarios con licencia, número actual de usuarios y nombre del propietario) está disponible mediante una selección de propiedades de **Win32_OperatingSystem**. Puede seleccionar explícitamente las propiedades para que se muestre lo siguiente:

```
Get-WmiObject -Class Win32_OperatingSystem -ComputerName . | Select-Object -
Property NumberOfLicensedUsers,NumberOfUsers,RegisteredUser
```

Una versión más concisa con caracteres comodín es la siguiente:

```
Get-WmiObject -Class Win32_OperatingSystem -ComputerName . | Select-Object -
Property *user*
```

Obtener el espacio en disco disponible

Para ver el espacio en disco y el espacio disponible para unidades locales, puede utilizar la clase Win32_LogicalDisk de WMI. Necesita ver únicamente las instancias con el valor 3 para DriveType, que es el valor que WMI usa para los discos duros fijos.

```
Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3" -ComputerName .

DeviceID     : C:
DriveType    : 3
ProviderName :
FreeSpace    : 65541357568
Size         : 203912880128
VolumeName   : Local Disk

DeviceID     : Q:
DriveType    : 3
ProviderName :
FreeSpace    : 44298250240
Size         : 122934034432
VolumeName   : New Volume

PS> Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3" -ComputerName . |
Measure-Object -Property FreeSpace,Size -Sum

Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3" -ComputerName . |
Measure-Object -Property FreeSpace,Size -Sum | Select-Object -Property
Property,Sum
```

Obtener información de sesiones iniciadas

Puede obtener información general acerca de sesiones iniciadas que estén asociadas a usuarios mediante la clase Win32_LogonSession de WMI:

```
Get-WmiObject -Class Win32_LogonSession -ComputerName .
```

Obtener el usuario que ha iniciado una sesión en un equipo

Puede mostrar el usuario que ha iniciado una sesión en un determinado equipo con Win32_ComputerSystem. Este comando devuelve únicamente el usuario que ha iniciado una sesión en el escritorio del sistema:

```
Get-WmiObject -Class Win32_ComputerSystem -Property UserName -ComputerName .
```

Obtener la hora local de un equipo

Puede recuperar la hora local actual que se muestra en un equipo específico con la clase Win32_LocalTime de WMI. Como esta clase muestra todos los metadatos de manera predeterminada, es posible que desee filtrarlos con **Select-Object**:

```
PS> Get-WmiObject -Class Win32_LocalTime -ComputerName . | Select-Object -Property
```

```
[a-z]*


Day          : 15
DayOfWeek    : 4
Hour         : 12
Milliseconds :
Minute       : 11
Month        : 6
Quarter      : 2
Second       : 52
WeekInMonth  : 3
Year         : 2006
```

Mostrar el estado de un servicio

Para ver el estado de todos los servicios en un equipo específico, puede utilizar localmente el cmdlet **Get-Service** como se ha mencionado anteriormente. Para sistemas remotos, puede usar la clase Win32_Service de WMI. Si usa también **Select-Object** para filtrar los resultados a **Status**, **Name** y **DisplayName**, el formato de los resultados será prácticamente igual que el de **Get-Service**:

```
Get-WmiObject -Class Win32_Service -ComputerName . | Select-Object -Property
Status,Name,DisplayName
```

Para que se muestren completos los nombres de servicio que sean muy largos, quizá desee usar **Format-Table** con los parámetros **AutoSize** y **Wrap**, lo que optimiza el ancho de columna y permite ajustar los nombres largos en lugar de truncarlos:

```
Get-WmiObject -Class Win32_Service -ComputerName . | Format-Table -Property
Status,Name,DisplayName -AutoSize -Wrap
```

Trabajar con instalaciones de software

Se puede tener acceso a aplicaciones diseñadas correctamente para utilizar Windows Installer mediante la clase **Win32_Product** de WMI, pero actualmente no todas las aplicaciones usan Windows Installer. Dado que Windows Installer proporciona la mayor variedad de técnicas estándar para trabajar con aplicaciones que se pueden instalar, nos centraremos principalmente en este tipo de aplicaciones. Las aplicaciones que utilizan otras rutinas de instalación no se suelen administrar con Windows Installer. Las técnicas concretas para trabajar con estas aplicaciones dependen del software de instalación y de las decisiones que adopte el programador de la aplicación.

Nota:

Las aplicaciones que se instalan con el procedimiento de copiar los archivos de la aplicación en el equipo no se pueden administrar normalmente con las técnicas descritas aquí. Estas aplicaciones se pueden administrar como si fueran archivos y carpetas utilizando las técnicas descritas en la sección "Trabajar con archivos y carpetas".

Mostrar las aplicaciones instaladas con Windows Installer

Las aplicaciones instaladas con Windows Installer en un sistema local o remoto se pueden enumerar fácilmente mediante una simple consulta de WMI:

```
PS> Get-WmiObject -Class Win32_Product -ComputerName .
IdentifyingNumber : {7131646D-CD3C-40F4-97B9-CD9E4E6262EF}
Name              : Microsoft .NET Framework 2.0
Vendor            : Microsoft Corporation
Version           : 2.0.50727
Caption           : Microsoft .NET Framework 2.0
```

Quizá desee usar **Select-Object**, ya que puede resultar útil otro tipo de información que no se muestra de manera predeterminada. Si desea buscar la ubicación de la caché de paquetes de Microsoft .NET Framework 2.0, puede utilizar lo siguiente:

```
PS> Get-WmiObject -Class Win32_Product -ComputerName . | Where-Object -
FilterScript {$_.Name -eq "Microsoft .NET Framework 2.0"} | Select-Object -
Property [a-z]*
Name              : Microsoft .NET Framework 2.0
Version           : 2.0.50727
InstallState      : 5
Caption           : Microsoft .NET Framework 2.0
Description       : Microsoft .NET Framework 2.0
IdentifyingNumber : {7131646D-CD3C-40F4-97B9-CD9E4E6262EF}
InstallDate       : 20060506
InstallDate2      : 20060506000000.000000-000
InstallLocation   :
PackageCache      : C:\WINDOWS\Installer\619ab2.msi
SKUNumber         :
Vendor            : Microsoft Corporation
```

Como alternativa, puede usar el parámetro **Get-WmiObject Filter** para seleccionar sólo Microsoft .NET Framework 2.0. Este comando aplica un filtro de WMI, por lo que no se usa la sintaxis de filtrado de Windows PowerShell. En su lugar, se usa el lenguaje de consultas de WMI (WQL):

```
Get-WmiObject -Class Win32_Product -ComputerName . -Filter "Name='Microsoft .NET
Framework 2.0'"| Select-Object -Property [a-z]*
```

Nota:

Las consultas de WQL suelen contener caracteres, como espacios o el signo igual, que tienen un significado especial en Windows PowerShell. Por este motivo, es aconsejable especificar siempre el valor del parámetro Filter entre comillas. También se puede usar el carácter de escape de Windows PowerShell, un acento grave (`), aunque no mejore la legibilidad. El siguiente comando equivale al comando anterior y devuelve los mismos resultados, pero utiliza el acento grave para omitir caracteres especiales, en lugar de especificar entre comillas la cadena de filtro completa:

```
Get-WmiObject -Class Win32_Product -ComputerName . -Filter
Name`=`'Microsoft` .NET` Framework` 2.0`' | Select-Object -Property [a-z]*
```

Otra manera de generar una lista reducida es seleccionar explícitamente el formato de presentación. El siguiente resultado contiene algunas de las propiedades más útiles para identificar determinados paquetes instalados:

```
Get-WmiObject -Class Win32_Product -ComputerName . | Format-List
Name,InstallDate,InstallLocation,PackageCache,Vendor,Version,IdentifyingNumber
...
Name              : HighMAT Extension to Microsoft Windows XP CD Writing Wizard
InstallDate       : 20051022
InstallLocation   : C:\Archivos de programa\HighMAT CD Writing Wizard\
PackageCache      : C:\WINDOWS\Installer\113b54.msi
Vendor            : Microsoft Corporation
Version           : 1.1.1905.1
IdentifyingNumber : {FCE65C4E-B0E8-4FBD-AD16-EDCBE6CD591F}
...
```

Por último, para buscar únicamente los nombres de las aplicaciones instaladas, una sencilla instrucción **Format-Wide** reducirá el resultado:

```
Get-WmiObject -Class Win32_Product -ComputerName .  | Format-Wide -Column 1
```

Aunque ahora disponemos de varias maneras de examinar las aplicaciones instaladas con Windows Installer, no hemos considerado otras aplicaciones. Puesto que la mayoría de las aplicaciones estándar registran su programa de desinstalación en Windows, podemos trabajar con estos programas localmente buscándolos en el Registro de Windows.

Mostrar todas las aplicaciones que se pueden desinstalar

Aunque no hay manera de garantizar que se pueda encontrar cada aplicación del sistema, se pueden buscar todos los programas que aparecen en el cuadro de diálogo

Agregar o quitar programas. Agregar o quitar programas busca estas aplicaciones en las listas que mantiene en la clave del Registro **HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall**, pero también podemos examinar manualmente esta clave para buscar aplicaciones. Para poder ver más fácilmente la clave Uninstall, podemos asignar una unidad de Windows PowerShell a esta ubicación del Registro:

```
PS> New-PSDrive -Name Uninstall -PSProvider Registry -Root
HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall

Name       Provider      Root                                    CurrentLocation
----       --------      ----                                    ---------------
Uninstall  Registry      HKEY_LOCAL_MACHINE\SOFTWARE\Micr...
```

Nota:

La unidad **HKLM:** está asignada a la raíz de **HKEY_LOCAL_MACHINE**, motivo por el que hemos utilizado esta unidad en la ruta de acceso a la clave Uninstall. En lugar de **HKLM:**, podríamos haber especificado la ruta de acceso al Registro con **HKLM** o **HKEY_LOCAL_MACHINE**. La ventaja de usar una unidad del Registro existente es que podemos utilizar el procedimiento para completar con el tabulador para completar los nombres de las claves, de manera que no es necesario escribirlas.

Ahora tenemos una unidad llamada "Uninstall" que podemos usar para buscar aplicaciones instaladas con comodidad y rapidez. Para saber el número de aplicaciones instaladas, puede contar el número de claves del Registro que aparecen en la unidad Uninstall: de Windows PowerShell:

```
PS> (Get-ChildItem -Path Uninstall:).Length
459
```

Podemos realizar búsquedas más refinadas en esta lista de aplicaciones mediante diversas técnicas, empezando por **Get-ChildItem**. Para obtener una lista de las aplicaciones en la variable **$UninstallableApplications**, podemos realizar lo siguiente:

```
$UninstallableApplications = Get-ChildItem -Path Uninstall:
```

Nota:

Usamos un nombre de variable largo por mayor claridad. En la realidad, no hay ningún motivo para usar nombres largos. Aunque puede usar el procedimiento para completar con el tabulador para los nombres de variables, también puede usar nombres de 1 ó 2 caracteres para ir más rápido. Los nombres descriptivos

largos son muy útiles cuando se programa código que se va a volver a utilizar más adelante.

Podemos buscar los nombres para mostrar de las aplicaciones en la clave Uninstall con el siguiente comando:

```
PS> Get-ChildItem -Path Uninstall: | ForEach-Object -Process {
$_.GetValue("DisplayName") }
```

No hay ninguna garantía de que estos valores sean únicos. En el siguiente ejemplo, dos elementos instalados aparecen como "Windows Media Encoder 9 Series":

```
PS> Get-ChildItem -Path Uninstall: | Where-Object -FilterScript {
$_.GetValue("DisplayName") -eq "Windows Media Encoder 9 Series"}


   Hive: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Micros
oft\Windows\CurrentVersion\Uninstall

SKC  VC Name                           Property
---  -- ----                           --------
  0   3 Windows Media Encoder 9        {DisplayName, DisplayIcon, UninstallS...
  0  24 {E38C00D0-A68B-4318-A8A6-F7... {AuthorizedCDFPrefix, Comments, Conta...
```

Instalar aplicaciones

La clase **Win32_Product** permite instalar paquetes de Windows Installer, ya sea de forma remota o local. En lo que respecta a las instalaciones remotas, se debe especificar la ruta de acceso al paquete .msi para la instalación como una ruta de red UNC (Convención de nomenclatura universal) típica, ya que el subsistema WMI no reconoce las rutas de Windows PowerShell. Por ejemplo, para instalar el paquete de MSI NewPackage.msi ubicado en el recurso compartido de red \\AppServ\dsp, en el equipo remoto PC01, deberá escribir lo siguiente en el símbolo del sistema de Windows PowerShell:

```
(Get-WMIObject -ComputerName PC01 -List | Where-Object -FilterScript {$_.Name -eq
"Win32_Product"}).InvokeMethod("Install","\\AppSrv\dsp\NewPackage.msi")
```

Las aplicaciones que no utilizan la tecnología Windows Installer pueden disponer de métodos específicos para la implementación automatizada. Para averiguar si hay un método para la implementación automatizada, probablemente necesite consultar la documentación de la aplicación o ponerse en contacto con el servicio de soporte técnico del proveedor de la aplicación. En algunos casos, aunque el proveedor no haya

diseñado específicamente la aplicación para la instalación automatizada, es posible que el fabricante del software de instalación disponga de técnicas para la automatización.

Eliminar aplicaciones

El procedimiento para quitar un paquete de Windows Installer con Windows PowerShell es prácticamente el mismo que se utiliza para instalar un paquete con **InvokeMethod**. En el siguiente ejemplo, se selecciona el paquete que se va a desinstalar según su propiedad Name; en algunos casos, puede resultar más fácil filtrar con **IdentifyingNumber**:

```
(Get-WmiObject -Class Win32_Product -Filter "Name='ILMerge'" -ComputerName .
).InvokeMethod("Uninstall",$null)
```

Quitar otras aplicaciones no es sencillo, ni siquiera localmente. Podemos encontrar las cadenas de desinstalación de la línea de comandos correspondientes a estas aplicaciones mediante la extracción de la propiedad **UninstallString**. Este método se puede usar para aplicaciones instaladas con Windows Installer y para programas antiguos mostrados en la clave Uninstall:

```
Get-ChildItem -Path Uninstall: | ForEach-Object -Process {
$_.GetValue("UninstallString") }
```

Si lo desea, puede filtrar el resultado por el nombre para mostrar:

```
Get-ChildItem -Path Uninstall: | Where-Object -FilterScript {
$_.GetValue("DisplayName") -like "Win*"} | ForEach-Object -Process {
$_.GetValue("UninstallString") }
```

No obstante, es posible que estas cadenas no se puedan utilizar directamente desde el símbolo del sistema de Windows PowerShell sin alguna modificación.

Actualizar aplicaciones instaladas con Windows Installer

Para actualizar una aplicación se utilizan dos tipos de información: necesita saber el nombre de la aplicación instalada que se va a actualizar, así como la ruta de acceso al paquete de actualización de la aplicación. Con esta información, se puede realizar una actualización desde Windows PowerShell con una única línea de comandos:

```
(Get-WmiObject -Class Win32_Product -ComputerName . -Filter
"Name='OldAppName'").InvokeMethod("Upgrade","\\AppSrv\dsp\OldAppUpgrade.msi")
```

Cambiar el estado del equipo: bloquear, cerrar la sesión, apagar y reiniciar

Puede restablecer un equipo de varias maneras distintas desde Windows PowerShell pero en la primera versión debe usar una herramienta estándar de línea de comandos o WMI. Aunque está utilizando Windows PowerShell sólo para invocar una determinada herramienta, recorrer el proceso para cambiar el estado de energía de un equipo ilustra algunas partes importantes del uso de herramientas externas.

Bloquear un equipo

La única manera de bloquear directamente un equipo con las herramientas estándar disponibles es llamar directamente a la función **LockWorkstation()** en **user32.dll**:

```
rundll32.exe user32.dll,LockWorkStation
```

Este comando bloquea de forma inmediata la estación de trabajo. En sistemas operativos como Windows XP, con Cambio rápido de usuario activado, el equipo volverá a la pantalla de inicio de sesión del usuario en lugar de iniciar el protector de pantalla del usuario actual. En un servidor Terminal Server, donde quizá desee desconectar sesiones específicas, también puede utilizar la herramienta **tsshutdn.exe** de la línea de comandos.

Cerrar la sesión actual

Puede cerrar una sesión de varias maneras distintas en el sistema local. La manera más sencilla es usar la herramienta de línea de comandos logoff.exe de Escritorio remoto/Servicios de Terminal Server (escriba **logoff /?** en el símbolo del sistema de Windows PowerShell o del shell de comandos para ver la información de uso). Para cerrar la sesión activa actualmente, escriba **logoff** sin argumentos.

Otra opción es utilizar la herramienta shutdown.exe con la opción de cierre de sesión correspondiente:

```
shutdown.exe -l
```

Una tercera opción es usar WMI. La clase Win32_OperatingSystem incluye el método Win32Shutdown; si se invoca este método con el argumento 0, se iniciará el cierre de la sesión:

```
(Get-WmiObject -Class Win32_OperatingSystem -ComputerName
.).InvokeMethod("Win32Shutdown",0)
```

Apagar o reiniciar un equipo

Apagar y reiniciar equipos son generalmente tareas del mismo tipo. Las herramientas utilizadas para apagar equipos suelen reiniciarlos también, y viceversa. Hay dos opciones sencillas para reiniciar un equipo desde Windows PowerShell: tsshutdn.exe o shutdown.exe, con los argumentos adecuados. Puede obtener información de uso detallada con **tsshutdn.exe /?** o **shutdown.exe /?**.

Asimismo, es posible apagar y reiniciar un equipo utilizando **Win32_OperatingSystem** directamente desde Windows PowerShell. No obstante, la información detallada de este tipo de implementación queda fuera del ámbito de esta Guía básica de Windows PowerShell.

Trabajar con impresoras

Las tareas de administración de impresoras pueden realizarse en Windows PowerShell con WMI y con el objeto COM **WScript.Network** de WSH. Utilizaremos una combinación de ambas herramientas para demostrar la forma de realizar tareas específicas.

Crear una lista de conexiones de impresora

La manera más sencilla de crear una lista de las impresoras instaladas en un equipo es usar la clase **Win32_Printer** de WMI:

```
Get-WmiObject -Class Win32_Printer -ComputerName .
```

También puede crear una lista de las impresoras con el objeto COM **WScript.Network** utilizado habitualmente en scripts de WSH:

```
(New-Object -ComObject WScript.Network).EnumPrinterConnections()
```

Este comando devuelve un simple conjunto de cadenas de los nombres de puertos y los nombres de dispositivos de impresora sin etiquetas distintivas, lo que no resulta útil para facilitar la inspección.

Agregar una impresora de red

La manera más fácil de agregar una impresora de red es utilizar **WScript.Network**:

```
(New-Object -ComObject
WScript.Network).AddWindowsPrinterConnection("\\Printserver01\Xerox5")
```

Configurar una impresora predeterminada

Para establecer la impresora predeterminada con WMI, debe filtrar la colección **Win32_Printer** hasta obtener la impresora deseada y, a continuación, invocar el método **SetDefaultPrinter**:

```
(Get-WmiObject -ComputerName . -Class Win32_Printer -Filter "Name='HP LaserJet 5Si'").InvokeMethod("SetDefaultPrinter",$null)
```

WScript.Network es un poco más fácil de utilizar; incluye también el método **SetDefaultPrinter** y sólo es necesario especificar el nombre de la impresora como argumento:

```
(New-Object -ComObject WScript.Network).SetDefaultPrinter('HP LaserJet 5Si')
```

Quitar una conexión de impresora

Puede quitar una conexión de impresora con el método **WScript.Network RemovePrinterConnection**:

```
(New-Object -ComObject WScript.Network).RemovePrinterConnection("\\Printserver01\Xerox5")
```

Realizar tareas de red

La mayoría de las tareas básicas de administración de protocolos de red implican el uso de TCP/IP, ya que éste es el protocolo de red más utilizado. Veremos ahora cómo efectuar una selección de estas tareas desde Windows PowerShell con WMI.

Crear una lista de direcciones IP utilizadas en un equipo

Puede devolver todas las direcciones IP que se estén utilizando en un equipo con el siguiente comando:

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=TRUE -ComputerName . | Select-Object -Property IPAddress
```

El resultado de este comando es diferente de la mayoría de las listas de propiedades, ya que los valores se especifican entre llaves:

```
IPAddress

---------

{192.168.1.80}
```

```
{192.168.148.1}

{192.168.171.1}

{0.0.0.0}
```

Para entender por qué se especifican entre llaves, examine detenidamente la propiedad **IPAddress** con **Get-Member**:

```
PS> Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=TRUE
-ComputerName . | Get-Member -Name IPAddress



   TypeName: System.Management.ManagementObject#root\cimv2\Win32_NetworkAdapter
Configuration


Name      MemberType Definition
----      ---------- ----------
IPAddress Property   System.String[] IPAddress {get;}
```

La propiedad IPAddress de cada adaptador de red es en realidad una matriz. Las llaves de la definición indican que **IPAddress** no es un valor de **System.String**, sino una matriz de valores de **System.String**.

Puede expandir estos valores con el parámetro **Select-Object ExpandProperty**:

```
PS> Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=TRUE
-ComputerName . | Select-Object -ExpandProperty IPAddress
192.168.1.80
192.168.148.1
192.168.171.1
0.0.0.0
```

Mostrar los datos de configuración de IP

Para mostrar los datos de configuración de IP correspondientes a cada adaptador de red, puede utilizar el siguiente comando:

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=TRUE -
ComputerName .
```

Nota:

De manera predeterminada, se muestra un conjunto muy reducido de la información de configuración disponible de los adaptadores de red. Para una inspección más exhaustiva y para la solución de problemas, **Select-Object**

permite forzar la presentación de más propiedades. Si no está interesado en las propiedades de IPX o WINS (lo que probablemente sea el caso en una red TCP/IP moderna), también puede excluir todas las propiedades que comiencen por WINS o IPX:

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter
IPEnabled=TRUE -ComputerName . | Select-Object -Property [a-z]* -
ExcludeProperty IPX*,WINS*
```

Este comando devolverá información detallada acerca de DHCP, DNS, el enrutamiento y otras propiedades de configuración de IP menos importantes.

Hacer ping en equipos

Puede hacer un sencillo ping a un equipo con **Win32_PingStatus**. El siguiente comando hará el ping, pero devuelve un resultado largo:

```
Get-WmiObject -Class Win32_PingStatus -Filter "Address='127.0.0.1'" -ComputerName
.
```

Un formato más útil para la información de resumen es mostrar sólo las propiedades Address, ResponseTime y StatusCode:

```
PS> Get-WmiObject -Class Win32_PingStatus -Filter "Address='127.0.0.1'" -
ComputerName . | Select-Object -Property Address,ResponseTime,StatusCode

Address                                  ResponseTime               StatusCode
-------                                  ------------               ----------
127.0.0.1                                           0                          0
```

El código de estado 0 indica que el ping ha sido correcto.

Puede hacer ping a toda una serie de equipos utilizando una matriz. Como se utilizan varias direcciones, hay que usar **ForEach-Object** para hacer ping a cada dirección por separado:

```
"127.0.0.1","localhost","research.microsoft.com" | ForEach-Object -Process {Get-
WmiObject -Class Win32_PingStatus -Filter ("Address='" + $_ + "'") -ComputerName
.} | Select-Object -Property Address,ResponseTime,StatusCode
```

Se puede utilizar el mismo proceso para hacer ping a toda una subred. Por ejemplo, si desea comprobar una red privada con el número de red 192.168.1.0 y usa una máscara de subred estándar de clase C (255.255.255.0), únicamente las direcciones entre 192.168.1.1 y 192.168.1.254 serán direcciones locales válidas (0 se reserva siempre para el número de red y 255 es una dirección de emisión de subred).

Puede obtener una matriz de los números del 1 al 254 en Windows PowerShell con la instrucción **1..254**; por tanto, se puede hacer un ping de subred completo si se genera la matriz y se agregan los valores a una dirección parcial en la instrucción ping:

```
1..254| ForEach-Object -Process {Get-WmiObject -Class Win32_PingStatus -Filter
("Address='192.168.1." + $_ + "'") -ComputerName .} | Select-Object -Property
Address,ResponseTime,StatusCode
```

Nota:

Esta técnica para generar un intervalo de direcciones se puede usar también en otras partes. Puede generar un conjunto completo de direcciones de esta manera:

```
$ips = 1..254 | ForEach-Object -Process {"192.168.1." + $_}
```

Recuperar propiedades de adaptadores de red

En una sección anterior de esta Guía básica, hemos mencionado que podía recuperar propiedades de configuración generales con **Win32_NetworkAdapterConfiguration**. Aunque no se trata estrictamente de información de TCP/IP, la información relativa a los adaptadores de red, como las direcciones MAC y los tipos de adaptadores, puede resultar útil para entender lo que está pasando en un equipo. Puede obtener un resumen de esta información con este comando:

```
Get-WmiObject -Class Win32_NetworkAdapter -ComputerName .
```

Asignar el dominio DNS para un adaptador de red

La asignación del dominio DNS para utilizarlo en la resolución automática de nombres se puede automatizar con el método **Win32_NetworkAdapterConfiguration SetDNSDomain**. Dado que se asigna el dominio DNS por separado para la configuración de cada adaptador de red, es necesario usar una instrucción **ForEach-Object** para asegurarse de que se seleccionan los adaptadores de forma individual:

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=true -
ComputerName . | ForEach-Object -Process { $_.InvokeMethod("SetDNSDomain",
"fabrikam.com")}
```

Hemos utilizado aquí la instrucción de filtrado IPEnabled=true porque, incluso en una red que usa sólo TCP/IP, varias de las configuraciones de adaptadores de red que hay en un equipo no son adaptadores TCP/IP reales; se trata de elementos de software generales que admiten RAS, PPTP, QoS y otros servicios para todos los adaptadores y, por tanto, no tienen una dirección propia.

Podría filtrar el comando con **Where-Object,** en lugar de usar **Get-WmiObject Filter**:

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -ComputerName . | Where-
Object -FilterScript {$_.IPEnabled} | ForEach-Object -Process
{$_.InvokeMethod("SetDNSDomain", "fabrikam.com")}
```

Realizar tareas de configuración de DHCP

La modificación de la información de DHCP conlleva trabajar con un conjunto de adaptadores, como en la configuración de DNS. Se pueden realizar varias acciones distintas con WMI; recorreremos paso a paso algunas de las más comunes.

Determinar los adaptadores con DHCP habilitado

Puede usar el siguiente comando para buscar los adaptadores con DHCP habilitado en un equipo:

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter "DHCPEnabled=true"
-ComputerName .
```

Si no está buscando adaptadores con problemas de configuración de IP, puede restringir esta búsqueda a los adaptadores con IP habilitado:

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter "IPEnabled=true and
DHCPEnabled=true" -ComputerName .
```

Recuperar propiedades de DHCP

Las propiedades relativas a DHCP de un adaptador comienzan generalmente por DHCP; por tanto, puede agregar un elemento de canalización **Select-Object -Property DHCP*** para ver información de DHCP de resumen de los adaptadores:

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter "DHCPEnabled=true"
-ComputerName . | Select-Object -Property DHCP*
```

Habilitar DHCP en cada adaptador

Si desea habilitar DHCP globalmente en todos los adaptadores, utilice el siguiente comando:

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=true -
ComputerName . | ForEach-Object -Process {$_.InvokeMethod("EnableDHCP", $null)}
```

También podría utilizar la instrucción **Filter** "IPEnabled=true and DHCPEnabled=false" para no intentar habilitar DHCP donde ya esté habilitado, pero omitir este paso no producirá ningún error.

Liberar y renovar concesiones DHCP en adaptadores específicos

Win32_NetworkAdapterConfiguration incluye los métodos **ReleaseDHCPLease** y **RenewDHCPLease**, y ambos se utilizan de la misma manera. En general, estos métodos se usan si sólo es necesario liberar o renovar direcciones para un adaptador en una subred específica. La manera más sencilla de filtrar los adaptadores en una subred es seleccionar únicamente las configuraciones de adaptadores que utilicen la puerta de enlace de esa subred. Por ejemplo, el siguiente comando libera todas las concesiones DHCP en adaptadores del equipo local que obtengan estas concesiones a partir de 192.168.1.254:

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter "IPEnabled=true and
DHCPEnabled=true" -ComputerName . | Where-Object -FilterScript {$_.DHCPServer -
contains "192.168.1.254"} | ForEach-Object -Process
{$_.InvokeMethod("ReleaseDHCPLease",$null)}
```

El único cambio para renovar concesiones DHCP es invocar **RenewDHCPLease** en lugar de **ReleaseDHCPLease**:

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter "IPEnabled=true and
DHCPEnabled=true" -ComputerName . | Where-Object -FilterScript {$_.DHCPServer -
contains "192.168.1.254"} | ForEach-Object -Process
{$_.InvokeMethod("ReleaseDHCPLease",$null)}
```

Nota:

Cuando se utilizan estos métodos en un equipo remoto, se puede perder el acceso al sistema remoto si se está conectado al mismo a través del adaptador con la concesión liberada o renovada.

Liberar y renovar concesiones DHCP en todos los adaptadores

Puede liberar o renovar direcciones DHCP globalmente en todos los adaptadores con los métodos **Win32_NetworkAdapterConfiguration**, **ReleaseDHCPLeaseAll** y **RenewDHCPLeaseAll**. No obstante, es necesario aplicar el comando a la clase de WMI, y no a un adaptador concreto, ya que la liberación y la renovación global de concesiones se realizan en la clase, no en un adaptador concreto.

Puede obtener una referencia a una clase de WMI, y no a instancias de la clase, si crea una lista de todas las clases de WMI y, a continuación, selecciona sólo la clase que

desea por su nombre. Por ejemplo, este comando devuelve la clase Win32_NetworkAdapterConfiguration:

```
Get-WmiObject -List | Where-Object -FilterScript {$_.Name -eq
"Win32_NetworkAdapterConfiguration"}
```

Puede tratar el comando completo como la clase y, a continuación, invocar el método **ReleaseDHCPAdapterLease** incluido. En el siguiente comando, los elementos de canalización **Get-WmiObject** y **Where-Object** aparecen entre paréntesis y esto fuerza que se evalúen primero:

```
( Get-WmiObject -List | Where-Object -FilterScript {$_.Name -eq
"Win32_NetworkAdapterConfiguration"} ) .InvokeMethod("ReleaseDHCPLeaseAll", $null)
```

Puede utilizar el mismo formato de comando para invocar el método **RenewDHCPLeaseAll**:

```
( Get-WmiObject -List | Where-Object -FilterScript {$_.Name -eq
"Win32_NetworkAdapterConfiguration"} ) .InvokeMethod("RenewDHCPLeaseAll", $null)
```

Crear un recurso compartido de red

Puede crear un recurso compartido de red con el método **Win32_Share Create**:

```
(Get-WmiObject -List -ComputerName . | Where-Object -FilterScript {$_.Name -eq
"Win32_Share"}).InvokeMethod("Create",("C:\temp","TempShare",0,25,"test share of
the temp folder"))
```

También puede crear el recurso compartido con **net share** en Windows PowerShell:

```
net share tempshare=c:\temp /users:25 /remark:"test share of the temp folder"
```

Eliminar un recurso compartido de red

Puede quitar un recurso compartido de red con **Win32_Share**, pero el proceso es ligeramente distinto del que se usa para crear un recurso compartido, ya que necesita recuperar el recurso compartido específico que desea quitar, en lugar de la clase **Win32_Share**. La siguiente instrucción elimina el recurso compartido "TempShare":

```
 (Get-WmiObject -Class Win32_Share -ComputerName . -Filter
"Name='TempShare'").InvokeMethod("Delete",$null)
```

También se puede utilizar el comando **net share**:

```
PS> net share tempshare /delete
tempshare se ha eliminado.
```

Conectar una unidad de red accesible desde Windows

New-PSDrive permite crear una unidad de Windows PowerShell, pero las unidades creadas de esta manera están disponibles únicamente en Windows PowerShell. Para crear una unidad en red, puede usar el objeto COM **WScript.Network**. El siguiente comando asigna el recurso compartido \\FPS01\users a la unidad local B:

```
(New-Object -ComObject WScript.Network).MapNetworkDrive("B:", "\\FPS01\users")
```

También se puede utilizar el comando **net use**:

```
net use B: \\FPS01\users
```

Las unidades asignadas con **WScript.Network** o **net use** están disponibles inmediatamente en Windows PowerShell.

Trabajar con archivos y carpetas

Los procedimientos para desplazarse por las unidades de Windows PowerShell y manipular los elementos que contienen son similares a los que se utilizan para manipular archivos y carpetas en unidades de disco físicas de Windows. En esta sección explicaremos cómo realizar tareas específicas de manipulación de archivos y carpetas.

Mostrar todos los archivos y carpetas que contiene una carpeta

Puede obtener todos los elementos incluidos directamente en una carpeta con **Get-ChildItem**. Agregue el parámetro opcional **Force** para mostrar los elementos ocultos o del sistema. Por ejemplo, este comando muestra el contenido directo de la unidad C de Windows PowerShell (que es el mismo que la unidad C física de Windows):

```
Get-ChildItem -Force C:\
```

Este comando, que es muy parecido al comando **DIR** de Cmd.exe o **ls** de un shell de UNIX, muestra sólo los elementos contenidos directamente. Para que se muestren los elementos contenidos, debe especificar también el parámetro **- Recurse** (esta operación puede tardar mucho tiempo en completarse). Para mostrar todo el contenido de la unidad C:

```
Get-ChildItem -Force C:\ -Recurse
```

Get-ChildItem puede filtrar elementos con los parámetros **Path**, **Filter**, **Include** y **Exclude**, pero éstos se basan normalmente sólo en el nombre. Puede aplicar filtros complejos basándose en otras propiedades de los elementos con **Where-Object**.

El siguiente comando busca todos los archivos ejecutables en la carpeta Archivos de programa que se modificaron por última vez después del 1 de octubre de 2005 y cuyo tamaño se encuentra entre 1 megabyte y 10 megabytes:

```
Get-ChildItem -Path $env:ProgramFiles -Recurse -Include *.exe | Where-Object -
FilterScript {($_.LastWriteTime -gt "2005-10-01") -and ($_.Length -ge 1m) -and
($_.Length -le 10m)}
```

Copiar archivos y carpetas

Las operaciones de copia se realizan con **Copy-Item**. El siguiente comando crea una copia de seguridad de C:\boot.ini en C:\boot.bak:

```
Copy-Item -Path c:\boot.ini -Destination c:\boot.bak
```

Si el archivo de destino ya existe, la copia no se podrá realizar. Para sobrescribir un destino que ya existe, utilice el parámetro Force:

```
Copy-Item -Path c:\boot.ini -Destination c:\boot.bak -Force
```

Este comando se puede usar incluso con un destino de sólo lectura.

La copia de carpetas se realiza de la misma manera. Este comando copia recursivamente la carpeta C:\temp\test1 en la nueva carpeta c:\temp\DeleteMe:

```
Copy-Item C:\temp\test1 -Recurse c:\temp\DeleteMe
```

También puede copiar una selección de elementos. El siguiente comando copia todos los archivos .txt incluidos en cualquier ubicación de c:\data en c:\temp\text:

```
Copy-Item -Filter *.txt -Path c:\data -Recurse -Destination c:\temp\text
```

También se pueden usar otras herramientas para realizar copias del sistema de archivos. En Windows PowerShell se pueden utilizar objetos XCOPY, ROBOCOPY y COM, como **Scripting.FileSystemObject**. Por ejemplo, puede usar la clase **Scripting.FileSystem COM** de Windows Script Host para crear una copia de seguridad de C:\boot.ini en C:\boot.bak:

```
(New-Object -ComObject Scripting.FileSystemObject).CopyFile("c:\boot.ini",
"c:\boot.bak")
```

Crear archivos y carpetas

La creación de nuevos elementos se realiza de la misma manera en todos los proveedores de Windows PowerShell. Si un proveedor de Windows PowerShell incluye varios tipos de elementos (por ejemplo, el proveedor FileSystem de Windows PowerShell distingue entre directorios y archivos), deberá especificar el tipo de elemento.

Este comando crea una nueva carpeta C:\temp\New Folder:

```
New-Item -Path 'C:\temp\New Folder' -ItemType "directory"
```

Este comando crea un nuevo archivo vacío C:\temp\New Folder\file.txt:

```
New-Item -Path 'C:\temp\New Folder\file.txt' -ItemType "file"
```

Eliminar todos los archivos y carpetas que contiene una carpeta

Puede usar **Remove-Item** para quitar elementos contenidos, pero se le pedirá que confirme la eliminación si el elemento contiene algo más. Por ejemplo, si intenta eliminar la carpeta C:\temp\DeleteMe que contiene otros elementos, Windows PowerShell le solicitará confirmación antes de eliminar la carpeta:

```
Remove-Item C:\temp\DeleteMe

Confirmar
El elemento situado en C:\temp\DeleteMe tiene elementos secundarios y no se
especificó el parámetro -recurse
Si continúa, se quitarán todos los elementos secundarios junto con el elemento.
¿Está seguro de que desea continuar?
[S] Sí  [O] Sí a todo  [N] No  [T] No a todo  [S] Suspender  [?] Ayuda
(el valor predeterminado es "S"):
```

Si no desea que se le pregunte por cada elemento contenido, especifique el parámetro **Recurse**:

```
Remove-Item C:\temp\DeleteMe -Recurse
```

Asignar una carpeta local como una unidad accesible desde Windows

También puede asignar una carpeta local con el comando **subst**. El siguiente comando crea una unidad local P: cuya raíz es el directorio local Archivos de programa:

```
subst p: $env:programfiles
```

Al igual que las unidades de red, las unidades asignadas en Windows PowerShell con **subst** se muestran inmediatamente en la sesión de Windows PowerShell.

Leer un archivo de texto en una matriz

Uno de los formatos más comunes de almacenamiento de datos de texto es un archivo con líneas independientes tratadas como elementos de datos distintos. El cmdlet **Get-Content** permite leer un archivo completo en un solo paso, como se muestra a continuación:

```
PS> Get-Content -Path C:\boot.ini
[boot loader]
timeout=5
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"
 /noexecute=AlwaysOff /fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS=" Microsoft Windows XP Professional
with Data Execution Prevention" /noexecute=optin /fastdetect
```

Get-Content trata los datos leídos del archivo como una matriz, con un elemento por línea de contenido del archivo. Para confirmar esto, compruebe el valor de **Length** del contenido devuelto:

```
PS> (Get-Content -Path C:\boot.ini).Length
6
```

Este comando es muy útil para obtener listas de información directamente en Windows PowerShell. Por ejemplo, puede almacenar una lista de nombres de equipos o direcciones IP en un archivo C:\temp\domainMembers.txt, con un nombre en cada línea del archivo. **Get-Content** permite recuperar el contenido del archivo y colocarlo en la variable **$Computers**:

```
$Computers = Get-Content -Path C:\temp\DomainMembers.txt
```

$Computers es ahora una matriz que contiene un nombre de equipo en cada elemento.

Trabajar con claves del Registro

Dado que las claves del Registro son elementos de las unidades de Windows PowerShell, trabajar con ellas es muy parecido a trabajar con archivos y carpetas. Una diferencia fundamental es que todo elemento de una unidad de Windows PowerShell basada en el Registro es un contenedor, como lo es una carpeta en una unidad del

sistema de archivos. No obstante, las entradas del Registro y sus valores asociados son propiedades de los elementos, no elementos distintos.

Mostrar todas las subclaves de una clave del Registro

Puede mostrar todos los elementos incluidos directamente en una clave del Registro con **Get-ChildItem**. Agregue el parámetro opcional **Force** para mostrar los elementos ocultos o del sistema. Por ejemplo, este comando muestra los elementos incluidos directamente en la unidad HKCU: de Windows PowerShell, que corresponde al subárbol del Registro HKEY_CURRENT_USER:

```
PS> Get-ChildItem -Path hkcu:\


   Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER

SKC  VC Name                           Property
---  -- ----                           --------
  2   0 AppEvents                     {}
  7  33 Console                        {ColorTable00, ColorTable01, ColorTab...
 25   1 Control Panel                 {Opened}
  0   5 Environment                   {APR_ICONV_PATH, INCLUDE, LIB, TEMP...}
  1   7 Identities                    {Last Username, Last User ...
  4   0 Keyboard Layout               {}
...
```

Se trata de las claves de nivel superior mostradas en HKEY_CURRENT_USER en el Editor del Registro (Regedit.exe).

También puede especificar esta ruta de acceso al Registro si especifica el nombre del proveedor del Registro seguido de "**::**". El nombre completo del proveedor del Registro es **Microsoft.PowerShell.Core\Registry**, pero se puede abreviar como **Registry**. Cualquiera de los siguientes comandos enumerará el contenido que hay directamente en la unidad HKCU:

```
Get-ChildItem -Path Registry::HKEY_CURRENT_USER
Get-ChildItem -Path Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
Get-ChildItem -Path Registry::HKCU
Get-ChildItem -Path Microsoft.PowerShell.Core\Registry::HKCU
Get-ChildItem HKCU:
```

Estos comandos enumeran sólo los elementos contenidos directamente, muy parecido al comando **DIR** de Cmd.exe o **ls** de un shell de UNIX. Para que se muestren los elementos contenidos, debe especificar el parámetro **Recurse**. Para enumerar todas las claves del Registro en la unidad HKCU, utilice el siguiente comando (esta operación puede tardar mucho tiempo en completarse):

```
Get-ChildItem -Path hkcu:\ -Recurse
```

Get-ChildItem permite aplicar filtros complejos con los parámetros **Path**, **Filter**, **Include** y **Exclude**, pero éstos se basan normalmente sólo en el nombre. Puede aplicar filtros complejos basándose en otras propiedades de los elementos con el cmdlet **Where-Object**. El siguiente comando busca todas las claves incluidas en HKCU:\Software que tengan una subclave como máximo y que tengan exactamente cuatro valores:

```
Get-ChildItem -Path HKCU:\Software -Recurse | Where-Object -FilterScript
{($_.SubKeyCount -le 1) -and ($_.ValueCount -eq 4) }
```

Copiar claves

Las operaciones de copia se realizan con **Copy-Item**. El siguiente comando copia HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion y todas sus propiedades a HKCU:\, creando una nueva clave llamada "CurrentVersion":

```
Copy-Item -Path 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion' -Destination
hkcu:
```

Si examina esta nueva clave en el Editor del Registro o con **Get-ChildItem**, observará que no hay copias de las subclaves en la nueva ubicación. Para copiar todo el contenido de un contenedor, debe especificar también el parámetro **- Recurse**. Para aplicar el comando de copia anterior de forma recursiva, utilice este comando:

```
Copy-Item -Path 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion' -Destination
hkcu: -Recurse
```

También puede usar otras herramientas de las que ya dispone para realizar copias del sistema de archivos. Cualquier herramienta de modificación del Registro (entre ellas, reg.exe, regini.exe y regedit.exe) y objetos COM que admitan la modificación del Registro (como WScript.Shell y la clase StdRegProv de WMI) se pueden utilizar en Windows PowerShell.

Crear claves

La creación de nuevas claves en el Registro es una operación más sencilla que crear nuevos elementos en un sistema de archivos. Dado que todas las claves del Registro son contenedores, no es necesario especificar el tipo de elemento; basta con proporcionar una ruta de acceso explícita, como:

```
New-Item -Path hkcu:\software\_DeleteMe
```

También puede utilizar una ruta basada en un proveedor para especificar una clave:

```
New-Item -Path Registry::HKCU\_DeleteMe
```

Eliminar claves

La eliminación de elementos es prácticamente igual para todos los proveedores. Los siguientes comandos quitan elementos de forma silenciosa:

```
Remove-Item -Path hkcu:\Software\_DeleteMe
Remove-Item -Path 'hkcu:\key with spaces in the name'
```

Eliminar todas las claves contenidas en una clave específica

Puede quitar elementos contenidos con **Remove-Item**, pero se le pedirá que confirme la eliminación si el elemento contiene algo más. Por ejemplo, si intentamos eliminar la subclave HKCU:\CurrentVersion que creamos, podremos ver lo siguiente:

```
Remove-Item -Path hkcu:\CurrentVersion

Confirmar
El elemento situado en HKCU:\CurrentVersion\AdminDebug tiene elementos secundarios
y no se especificó el parámetor -recurse. Si continúa, se quitarán todos los
secundarios junto con elemento. ¿Está seguro de que desea continuar?
[S] Sí  [O] Sí a todo  [N] No  [T] No a todo  [S] Suspender  [?] Ayuda
(el valor predeterminado es "S"):
```

Para eliminar elementos contenidos sin pedir confirmación, especifique el parámetro **-Recurse**:

```
Remove-Item -Path HKCU:\CurrentVersion -Recurse
```

Si desea quitar todos los elementos incluidos en HKCU:\CurrentVersion, pero no HKCU:\CurrentVersion, puede utilizar:

```
Remove-Item -Path HKCU:\CurrentVersion\* -Recurse
```

Trabajar con entradas del Registro

Dado que las entradas del Registro son propiedades de claves y, como tales, no se pueden examinar directamente, hay que adoptar un enfoque ligeramente distinto para trabajar con ellas.

Mostrar las entradas del Registro

Hay muchas maneras distintas de examinar entradas del Registro. La manera más sencilla es obtener los nombres de propiedades asociados a una clave. Por ejemplo, para ver los nombres de las entradas de la clave del Registro **HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion**, utilice **Get-Item**. Las claves del Registro tienen una propiedad con el nombre genérico "Property", que muestra una lista de las entradas del Registro que contiene la clave. El siguiente comando selecciona la propiedad Property y expande los elementos para que se muestren en una lista:

```
PS> Get-Item -Path
Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion | Select-
Object -ExpandProperty Property
DevicePath
MediaPathUnexpanded
ProgramFilesDir
CommonFilesDir
ProductId
```

Para facilitar la lectura de las entradas del Registro, utilice **Get-ItemProperty**:

```
PS> Get-ItemProperty -Path
Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion


PSPath              : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SO
                      FTWARE\Microsoft\Windows\CurrentVersion
PSParentPath        : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SO
                      FTWARE\Microsoft\Windows
PSChildName         : CurrentVersion
PSDrive             : HKLM
PSProvider          : Microsoft.PowerShell.Core\Registry
DevicePath          : C:\WINDOWS\inf
MediaPathUnexpanded : C:\WINDOWS\Media
ProgramFilesDir     : C:\Archivos de programa
CommonFilesDir      : C:\Archivos de programa\Archivos comunes
ProductId           : 76487-338-1167776-22465
WallPaperDir        : C:\WINDOWS\Web\Wallpaper
MediaPath           : C:\WINDOWS\Media
ProgramFilesPath    : C:\Archivos de programa
PF_AccessoriesName  : Accesorios
(default)           :
```

Todas las propiedades de la clave relativas a Windows PowerShell tienen el prefijo "PS", como **PSPath**, **PSParentPath**, **PSChildName** y **PSProvider**.

Puede usar la notación "**.**" para hacer referencia a la ubicación actual. Puede utilizar **Set-Location** para cambiar primero al contenedor del Registro **CurrentVersion**:

```
Set-Location -Path
Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
```

Como alternativa, puede utilizar la unidad HKLM de PowerShell integrada con **Set-Location**:

```
Set-Location -Path hklm:\SOFTWARE\Microsoft\Windows\CurrentVersion
```

Después puede usar la notación "**.**" para la ubicación actual y crear una lista de las propiedades sin especificar una ruta de acceso completa:

```
PS> Get-ItemProperty -Path .
...
DevicePath          : C:\WINDOWS\inf
MediaPathUnexpanded : C:\WINDOWS\Media
ProgramFilesDir     : C:\Archivos de programa
...
```

Las rutas de acceso se pueden expandir de la misma manera que en el sistema de archivos; por tanto, desde esta ubicación puede obtener la lista de **ItemProperty** para **HKLM:\SOFTWARE\Microsoft\Windows\Help** mediante **Get-ItemProperty -Path ..\Help**.

Obtener una sola entrada del Registro

Si desea recuperar una entrada específica de una clave del Registro, puede aplicar uno de varios enfoques posibles. En este ejemplo, se busca el valor de **DevicePath** en **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion**.

Con **Get-ItemProperty**, utilice el parámetro **Path** para especificar el nombre de la clave y el parámetro **Name** para especificar el nombre de la entrada **DevicePath**.

```
PS> Get-ItemProperty -Path HKLM:\Software\Microsoft\Windows\CurrentVersion -Name
DevicePath


PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software\
               Microsoft\Windows\CurrentVersion
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software\
               Microsoft\Windows
PSChildName  : CurrentVersion
PSDrive      : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry
DevicePath   : C:\WINDOWS\inf
```

Este comando devuelve las propiedades estándar de Windows PowerShell, así como la propiedad **DevicePath**.

Nota:

Aunque **Get-ItemProperty** cuenta con los parámetros **Filter**, **Include** y **Exclude**, no se pueden usar para filtrar por nombre de propiedad. Estos parámetros hacen referencia a claves del Registro (que son rutas de acceso a elementos) y no a entradas del Registro (que son propiedades de elementos).

Otra opción es utilizar la herramienta Reg.exe de la línea de comandos. Para obtener Ayuda acerca de reg.exe, escriba **reg.exe /?** en el símbolo del sistema. Para buscar la entrada DevicePath, utilice reg.exe como se muestra en el siguiente comando:

```
PS> reg query HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion /v DevicePath

! REG.EXE VERSION 3.0

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
    DevicePath  REG_EXPAND_SZ   %SystemRoot%\inf
```

También puede usar el objeto **WshShell COM** para buscar algunas entradas del Registro, aunque este método no se puede usar con datos binarios de gran tamaño ni con nombres de entradas del Registro que contengan caracteres como "\"). Anexe el nombre de la propiedad a la ruta de acceso al elemento con un separador \:

```
PS> (New-Object -ComObject
WScript.Shell).RegRead("HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\DevicePath")
%SystemRoot%\inf
```

Crear nuevas entradas del Registro

Para agregar una nueva entrada llamada "PowerShellPath" a la clave **CurrentVersion**, utilice **New-ItemProperty** con la ruta de acceso a la clave, el nombre de la entrada y el valor de la entrada. En este ejemplo, vamos a utilizar el valor de la variable **$PSHome** de Windows PowerShell, que almacena la ruta de acceso al directorio de instalación de Windows PowerShell.

Puede agregar la nueva entrada a la clave con el siguiente comando, que también devuelve información acerca de la nueva entrada:

```
PS> New-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion -Name
PowerShellPath -PropertyType String -Value $PSHome


PSPath         : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWAR
                 E\Microsoft\Windows\CurrentVersion
PSParentPath   : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWAR
                 E\Microsoft\Windows
```

```
PSChildName    : CurrentVersion
PSDrive        : HKLM
PSProvider     : Microsoft.PowerShell.Core\Registry
PowerShellPath : C:\Archivos de programa\Windows PowerShell\v1.0
```

El valor de **PropertyType** debe ser el nombre de un miembro de enumeración de **Microsoft.Win32.RegistryValueKind** de los que se muestran en la siguiente tabla:

Valor de PropertyType	Significado
Binary	Datos binarios
DWord	Número de tipo UInt32 válido
ExpandString	Cadena que puede contener variables de entorno expandidas dinámicamente
MultiString	Cadena de varias líneas
String	Cualquier valor de cadena
QWord	8 bytes de datos binarios

Nota:

Para agregar una entrada del Registro a varias ubicaciones debe especificar una matriz de valores para el parámetro **Path**:

```
New-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion,
HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion -Name PowerShellPath -PropertyType
String -Value $PSHome
```

También puede sobrescribir el valor de una entrada del Registro que ya existe si agrega el parámetro **Force** a cualquier comando **New-ItemProperty**.

Cambiar el nombre de entradas del Registro

Para cambiar el nombre de la entrada **PowerShellPath** a "PSHome", utilice **Rename-ItemProperty**:

```
Rename-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion -Name
PowerShellPath -NewName PSHome
```

Para mostrar el valor con el nuevo nombre, agregue el parámetro **PassThru** al comando:

```
Rename-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion -Name
PowerShellPath -NewName PSHome -passthru
```

Eliminar entradas del Registro

Para eliminar las entradas del Registro PSHome y PowerShellPath, utilice **Remove-ItemProperty**:

```
Remove-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion -Name
PSHome
Remove-ItemProperty -Path HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion -Name
PowerShellPath
```

Apéndice 1: Alias de compatibilidad

Windows PowerShell incluye varios alias de transición que permiten a los usuarios de Unix y Cmd usar nombres de comandos familiares en Windows PowerShell. En la siguiente tabla se muestran los alias más comunes, junto con el comando de Windows PowerShell al que corresponde el alias y el alias estándar de Windows PowerShell, si existe.

Nota:

Puede buscar el comando de PowerShell al que corresponde cualquier alias en PowerShell con Get-Alias. Por ejemplo:

PS> Get-Alias cls

```
CommandType     Name                                Definition
-----------     ----                                ----------
Alias           cls                                 Clear-Host
```

Comando de CMD	Comando de Unix	Comando de PS	Alias de PS
dir	**ls**	**Get-ChildItem**	**gci**
cls	**clear**	**Clear-Host** (función)	N/D
del, erase, rmdir	**rm**	**Remove-Item**	**ri**
copy	**cp**	**Copy-Item**	**ci**
move	**mv**	**Move-Item**	**mi**
rename	**mv**	**Rename-Item**	**rni**
type	**cat**	**Get-Content**	**gc**

Comando de CMD	Comando de Unix	Comando de PS	Alias de PS
cd	**cd**	**Set-Location**	**sl**
md	**mkdir**	**New-Item**	**ni**
N/D	**pushd**	**Push-Location**	N/D
N/D	**popd**	**Pop-Location**	N/D

Apéndice 2: Crear accesos directos personalizados de PowerShell

El siguiente procedimiento le guiará paso a paso en la creación de un acceso directo de PowerShell con varias opciones útiles personalizadas:

1. Cree un acceso directo a powershell.exe.
2. Haga clic con el botón secundario del mouse en el acceso directo y seleccione Propiedades.
3. Haga clic en la ficha Opciones.
4. Para habilitar la selección y copia con el mouse, en Opciones de edición, active la casilla Edición rápida. A continuación, puede seleccionar texto en la ventana de la consola de PowerShell; para ello, arrastre el botón primario del mouse y copie el texto en el Portapapeles con la tecla Entrar o con el botón secundario del mouse.
5. En Opciones de edición, active la casilla Modo de inserción. Después, puede hacer clic con el botón secundario del mouse en la ventana de la consola para pegar automáticamente el texto del Portapapeles.
6. En Historial de comandos, en el cuadro de número Tamaño del búfer, escriba o seleccione un número entre 1 y 999. Éste será el número de comandos utilizados que se mantendrán en el búfer de la consola.
7. En Historial de comandos, active la casilla Descartar duplicados antiguos para eliminar comandos repetidos del búfer de la consola.
8. Haga clic en la ficha Diseño.
9. En el cuadro de grupo Tamaño del búfer de pantalla, escriba un número entre 1 y 9999 en el cuadro de desplazamiento Alto. El alto representa el número de líneas de resultados almacenadas en el búfer. Se trata del número máximo de líneas que se pueden ver al desplazarse por la ventana de la consola. Si este número es menor

que el alto mostrado en el cuadro Tamaño de la ventana, este alto se reducirá automáticamente al mismo valor.

10. En el cuadro de grupo Tamaño de la ventana, escriba un número entre 1 y 9999 para el ancho. Este valor representa el número de caracteres que se muestran a lo ancho de la ventana de la consola. El ancho predeterminado es 80 y el formato de los resultados de PowerShell está diseñado teniendo en cuenta este ancho.
11. Si desea colocar la consola en una parte determinada en el escritorio cuando se abra, en el cuadro de grupo Posición de la ventana, desactive la casilla El sistema ubica la ventana y, a continuación, en Posición de la ventana, cambie los valores que se muestran en Izquierda y Arriba.
12. Cuando haya terminado, haga clic en Aceptar.