Kapitel 4

PowerShell-Pipeline

In diesem Kapitel:

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

Die PowerShell-Pipeline ist eines der wichtigsten Konzepte der PowerShell, mit dem sich mehrere Befehle zu einer Befehlskette zusammenfassen lassen. Obwohl der Aufbau und die Länge einer solchen »Pipeline« im Einzelfall ganz unterschiedlich sein kann, folgen Pipelines jedoch stets demselben Grundmuster:

```
Get-* | *-Object | [Format-*] | Out-*
Get-* | *-Object | Export-*
```

Die Pipeline funktioniert also im Grunde wie ein Fabrikfließband, auf dem nach und nach aus den Rohdaten die gewünschten Ergebnisse geformt werden. Sie beginnt deshalb in aller Regel mit einem Befehl, der Daten liefert, also entweder einem Cmdlet, dessen Name mit »Get« beginnt, oder mit den Rohdaten selbst, die beispielsweise aus einer Variable stammen können.

Danach werden so viele »Fabrikroboter« wie nötig mit dem Pipeline-Symbol »|« angefügt. Sie erhalten jeweils das Ergebnis vom vorangegangenen Befehl, erledigen ihre Arbeit und geben das eigene Ergebnis dann an den folgenden Befehl weiter. Diese Cmdlets stammen aus der Gruppe der **-Object*-Cmdlets:

Name	Beschreibung
Select-Object	Wählt die Eigenschaften (Spalten) aus, die sichtbar sein sollen
Sort-Object	Sortiert das Ergebnis basierend auf einer oder mehreren Eigenschaften
Group-Object	Gruppiert die Ergebnisse basierend auf einer oder auf mehreren Eigenschaften
Measure-Object	Zählt die Ergebnisse und kann optional numerische Eigenschaften auswerten
ForEach-Object	Führt einen Skriptblock für jedes einzelne Ergebnis aus. Innerhalb des Skriptblocks repräsentiert $_ das gerade über die Pipeline laufende Objekt. Dieses Cmdlet entspricht also einer Schleife.
Where-Object	Führt einen Skriptblock für jedes einzelne Ergebnis aus. Innerhalb des Skriptblocks repräsentiert *$_* das gerade über die Pipeline laufende Objekt. Ergibt der Skriptblock den Wert *$true*, wird das Objekt zum nächsten Pipelinebefehl weitergeleitet, sonst nicht. Dieses Cmdlet entspricht also einer Bedingung.

Tabelle 4.1 Wichtige Cmdlets innerhalb der PowerShell-Pipeline

Am Ende der Pipeline können die Ergebnisse mithilfe von Cmdlets der Gruppe *Out-** entweder in Text verwandelt und ausgegeben, mit Cmdlets der Gruppe *Export-** als Objekte serialisiert oder mit Cmdlets der Gruppe *ConvertTo-** in andere Formate konvertiert werden. Setzen Sie keine dieser Cmdlets ein, werden die Ergebnisse automatisch mit *Out-Default* in Text konvertiert und in die Konsole ausgegeben.

Wichtig hervorzuheben ist, dass die Pipeline die Daten in der Regel in Echtzeit bearbeitet. Sobald der erste Befehl in der Pipeline ein Ergebnis ausgibt, wird dieses durch die Pipeline geschickt und von den einzelnen Befehlen weiterbearbeitet. Noch während der erste Befehl also weitere Ergebnisse liefert, sind die übrigen Befehle in der Pipeline bereits damit beschäftigt, die schon gelieferten Ergebnisse zu bearbeiten. Zu Blockierungen kommt es nur, wenn Sie Befehle

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

einsetzen, die zuerst alle Ergebnisse sammeln, beispielsweise, weil die Ergebnisse anschließend sortiert werden sollen.

Informationen filtern

Sie möchten die Ergebnisse eines Befehls nach bestimmten Kriterien filtern.

Lösung

Leiten Sie die Ergebnisse an *Where-Object* weiter und legen Sie eine Bedingung fest. Die Bedingung wird in Form eines Filterskriptblocks formuliert. Darin repräsentiert die Variable *$_* das jeweils zu untersuchende Objekt.

Um alle Dateien im Windows-Ordner zu finden, die größer sind als 1 Mbyte, werten Sie beispielsweise die Eigenschaft *Length* aus:

```
PS > dir $env:windir | Where-Object { $_.Length -gt 1MB }
```

Möchten Sie alle Prozesse ermitteln, die mehr als 20 Sekunden Prozessorzeit verbraucht haben, lassen Sie die Eigenschaft *CPU* auswerten:

```
PS > Get-Process | Where-Object { $_.CPU -gt 20 }
```

Die folgende Zeile liefert nur Prozesse der Firma »Microsoft«:

```
PS > Get-Process | Where-Object { $_.Company -like '*microsoft*' }
```

Wollen Sie alle Dienste finden, die zurzeit nicht ausgeführt werden, überprüfen Sie die Eigenschaft *Status*:

```
PS > Get-Service | Where-Object { $_.Status -eq 'Stopped' }
```

Und möchten Sie aus einem Ordner nur die Dateien sehen, aber nicht die Unterordner, greifen Sie auf die Eigenschaft *PSIsContainer* zu:

```
PS > dir | Where-Object { $_.PSisContainer -eq $false }
```

Ausschließlich Unterordner, aber keine Dateien, erhalten Sie so:

```
PS > dir | Where-Object { $_.PSisContainer -ne $false }
```

Möchten Sie aus einem textbasierten Logbuch nur diejenigen Zeilen herausfiltern, die ein bestimmtes Stichwort enthalten, verfahren Sie so:

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

```
PS > Get-Content $env:windir\windowsupdate.log -Encoding UTF8 | Where-Object { $_ -like
'*successfully installed*' }
```

Und möchten Sie alle Updates sehen, deren KB-Artikelnummer mit »KB98« beginnt, werten Sie die Eigenschaft *HotfixID* aus:

```
PS > Get-Hotfix | Where-Object { $_.HotfixID -like 'KB98*' }
```

Auch das Ergebnis nativer Befehle lässt sich so filtern. Der folgende Code verwendet den nativen Befehl *ipconfig.exe* und gibt nur solche Zeilen zurück, in denen das Stichwort »IP« vorkommt:

```
PS > ipconfig | Where-Object { $_ -like '*IP*' }
```

Hintergrund

Where-Object untersucht jedes einzelne Objekt, das durch die Pipeline gesendet wird. Der Skriptblock, den Sie angeben, wird also für jedes Ergebnisobjekt erneut ausgewertet. Das gerade untersuchte Ergebnisobjekt wird darin durch die Variable *$_* repräsentiert. Ergibt der Skriptblock das Ergebnis *$true*, gilt das Objekt als erwünscht und wird nicht herausgefiltert, andernfalls schon.

Liefert *Get-Process* beispielsweise 20 laufende Prozesse und leiten Sie das Ergebnis an *Where-Object* weiter, wird der Skriptblock hinter *Where-Object* zwanzig Mal ausgeführt. Da *Get-Process* als Ergebnis *Process*-Objekte liefert, die über zahlreiche unabhängige Eigenschaften verfügen, wählen Sie im Skriptblock hinter dem Ausdruck »$_.« die Eigenschaft aus, die Sie überprüfen möchten. Welche Eigenschaften dabei zur Verfügung stehen, kann *Get-Member* für Sie ermitteln:

```
PS > Get-Process | Get-Member -memberType *property
```

Handelt es sich dagegen beim Objekt um einen primitiven Datentyp (einfache Zahlen oder Texte), kann *$_* direkt für den Vergleich herangezogen werden. Sie brauchen dann also nicht mit der Punktschreibweise hinter *$_* eine bestimmte Objekteigenschaft anzugeben. Die nächste Zeile würde beispielsweise nur Zahlen auflisten, die kleiner sind als 6:

```
PS> 1..10 | Where-Object { $_ -lt 6 }
1
2
3
4
5
```

Möchten Sie mehrere Kriterien kombinieren, verwenden Sie mehrere hintereinandergeschaltete *Where-Object*-Anweisungen. Dies entspricht einer logischen »Und«-Verknüpfung. Die folgende Zeile findet beispielsweise alle Prozesse der Firma Microsoft, die mehr als 20 Sekunden CPU-Zeit beansprucht haben:

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

```
PS > Get-Process | Where-Object { $_.CPU -gt 20 } | Where-Object ⟶
     { $_.Company -like '*Microsoft*' }
```

Alternativ könnten Sie auch den logischen Operator *–and* einsetzen und mit seiner Hilfe mehrere Bedingungen kombinieren. Das Hintereinanderschalten mehrerer *Where-Object*-Klauseln ist aber in den meisten Fällen empfehlenswerter, weil dadurch einfacherer Code entsteht und Sie sehr einfach nachträglich eine Bedingung wieder entfernen oder eine weitere hinzufügen können.

Wollen Sie dafür sorgen, dass alle Ergebnisse angezeigt werden, bei denen entweder die eine oder die andere Bedingung erfüllt ist, sind Sie allerdings gezwungen, mehrere Bedingungen mit dem logischen Operator *–or* zu kombinieren. Die folgende Zeile liefert alle Textzeilen einer Logbuchdatei, die entweder das Stichwort »No Network Connectivity« oder »Update is not allowed« enthalten:

```
PS > Get-Content $env:windir\windowsupdate.log | Where-Object ⟶
     { ($_ -like '*no network connectivity*') -or ($_ -like ⟶
     '*update is not allowed*') }
```

Diese Zeile liefert selbstverständlich keine Resultate, wenn die Logbuchdatei die gesuchten Stichwörter nicht enthält (oder Sie sich vertippt haben). Nutzen Sie den Operator *–clike* anstelle von *–like*, wenn Sie zwischen Groß- und Kleinschreibung unterscheiden müssen.

Eine Besonderheit ist das Filtern leerer Eigenschaften. Möchten Sie beispielsweise nur die Prozesse sehen, bei denen die Eigenschaft *Company* gefüllt ist, formulieren Sie so:

```
PS > Get-Process | Where-Object { $_.Company -eq $null }
...
PS > Get-Process | Where-Object { $_.Company -eq $null } | Select-Object Name, ⟶
     Company
...
```

Vollkommen leere Eigenschaften werden durch die vordefinierte Variable *$null* repräsentiert. Hin und wieder werden Sie vielleicht auch Codezeilen wie die folgende sehen:

```
PS > Get-Process | Where-Object { $_.Company } | Select-Object Name, Company
```

Hierbei macht man sich zunutze, dass ein vollkommen leerer Wert (also beispielsweise eine nicht festgelegte Objekteigenschaft) bei der Umwandlung in einen booleschen Wert zu *$false* wird, während jeder beliebige sonstige Datenwert (mit Ausnahme des Zahlenwerts *0*) zu *$true* konvertiert wird. Um also Objekte auszufiltern, die in einer bestimmten Eigenschaft keinen definierten Wert besitzen, geben Sie im Skriptblock hinter »$_.« den Namen dieser Eigenschaft an.

So erhalten Sie beispielsweise durch den folgenden Filter nur diejenigen Netzwerkadapter, bei denen tatsächlich eine IP-Adresse zugewiesen ist:

```
PS > Get-WmiObject Win32_NetworkAdapterConfiguration | Where-Object ⟶
     { $_.IPAddress }
```

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

Allerdings funktioniert diese Abkürzung nur, wenn die Eigenschaft, die Sie im Skriptblock auswerten, niemals den Wert 0 annehmen kann. Dieser Wert entspricht nämlich ebenfalls dem booleschen Wert *$false*, sodass Sie nicht nur vollkommen leere Eigenschaften ausfiltern, sondern auch solche, die genau den Wert 0 beinhalten.

HINWEIS Die Filterung mit *Where-Object* erfolgt clientseitig, was besonders bei Remotezugriffen große Bedeutung hat. Da bei einer clientseitigen Filterung zuerst alle Daten zum Client transportiert werden müssen, ist diese Filterung nicht sehr effizient und sollte nur dann eingesetzt werden, wenn eine serverseitige Filterung nicht möglich ist. Eine serverseitige Filterung setzt voraus, dass das Cmdlet, das die Daten abruft, selbst mithilfe eines Parameters die Datenmenge reduzieren kann. Im Falle der WMI verfügt *Get-WmiObject* dazu über den Parameter *–Filter*, sodass Sie die Netzwerkkarten auch über folgende Zeile effizienter serverseitig hätten filtern können:

```
PS > Get-WmiObject Win32_NetworkAdapterConfiguration -filter 'IPEnabled=true'
```

Dabei wird nicht etwa geprüft, ob die Eigenschaft *IPAddress* null ist, sondern auf eine andere, ebenso aussagekräftige Eigenschaft namens *IPEnabled* zurückgegriffen. Sie ist *$true*, wenn der jeweiligen Netzwerkkonfiguration eine IP-Adresse zugewiesen ist, sonst *$false*. Da es sich bei der serverseitigen Filterung um eine Leistung des Cmdlets und nicht der PowerShell handelt, übergeben Sie dabei keinen PowerShell-Code, sondern müssen sich nach dem Datentyp richten, den der zuständige Parameter von Ihnen erwartet. Der Parameter *–Filter* des Cmdlets *Get-WmiObject* erwartet beispielsweise eine Filterung im *WQL*-Format, also der Abfragesprache der WMI.

Aber auch ohne Remotezugriffe ist die serverseitige Filterung mittels Cmdlet-Parameter stets schneller. Die folgende Zeile liefert alle Ereignislogbucheinträge vom Typ *Error* aus dem Logbuch *System* und verwendet dazu die langsame clientseitige Filterung:

```
PS > Get-EventLog -LogName System | Where-Object { $_.EntryType -eq 'Error' }
```

Mehr als doppelt so schnell (und sehr viel simpler) ist die serverseitige Filterung, die sich zunutze macht, dass das Cmdlet *Get-EventLog* einen Parameter namens *–EntryType* besitzt:

```
PS > Get-EventLog -LogName System -EntryType Error
```

Grundsätzlich sollten Sie also zuerst versuchen, die Ergebnisse mithilfe der Parameter zu filtern, die das Cmdlet anbietet, das die Daten für Sie beschafft. Viele Cmdlets unterstützen Parameter, die genauso heißen wie die Spalten der Ergebnisdaten und mit denen man die Ergebnisse nach dieser Spalte filtern kann. Hier weitere Beispiele:

```
PS > Get-Command *service* -CommandType Cmdlet
PS > Get-Alias -Definition Get-ChildItem
PS > Get-Process | Get-Member -memberType *Property
PS > Get-ExecutionPolicy -Scope CurrentUser
```

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

Objekteigenschaften (Spalten) auswählen

Sie möchten nur bestimmte Eigenschaften (Spalten) eines Objekts weiterbearbeiten und die übrigen entfernen.

Lösung

Leiten Sie das Ergebnis des Befehls an *Select-Object* weiter und geben Sie mit dem Parameter *–property* an, welche Spalten Sie sehen möchten:

```
PS > Get-Process | Select-Object Name, *memory*

Name                       : AcroRd32
NonpagedSystemMemorySize   : 17960
NonpagedSystemMemorySize64 : 17960
PagedMemorySize            : 68947968
PagedMemorySize64          : 68947968
PagedSystemMemorySize      : 342640
PagedSystemMemorySize64    : 342640
...
PS > Get-Process | Select-Object Name, CPU, PeakVirtualMemorySize
Name                       CPU          PeakVirtualMemorySize
----                       ---          ---------------------
AcroRd32                   14,6328938   230084608
alg                        0            29872128
AppleMobileDeviceService   0,6552042    96903168
ApplicationUpdater         0,156001     71254016
...
```

PowerShell bestimmt dabei automatisch die Art der Formatierung: Geben Sie vier oder weniger Eigenschaften an, wird das Ergebnis als Tabelle nebeneinander angezeigt, andernfalls als Liste untereinander.

Hintergrund

PowerShell enthält mit dem Extended Type System (ETS) einen Mechanismus, der normalerweise vollautomatisch die wichtigsten Eigenschaften eines Objekttyps für Sie auswählt. Diese werden als Tabelle nebeneinander angezeigt, wenn es sich um vier oder weniger handelt, andernfalls als Liste untereinander. Von dieser Regel kann in Ausnahmefällen abgewichen werden, wenn im ETS abweichende Regeln für den Objekttyp hinterlegt worden sind.

Möchten Sie dagegen selbst bestimmen, welche Eigenschaften angezeigt werden, setzen Sie *Select-Object* ein und geben die Namen der gewünschten Eigenschaften als kommaseparierte Liste an. Ob die Ergebnisse als Tabelle nebeneinander oder als Liste untereinander angezeigt werden, hängt auch hier von der Anzahl der ausgewählten Eigenschaften ab.

Möchten Sie sämtliche Eigenschaften eines Objekts sehen, geben Sie als Eigenschaftenname den Platzhalter »*« an:

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

```
PS > Get-Process powershell | Select-Object *
```

WICHTIG Meistens entspricht die Spaltenüberschrift dem Namen der zugrundeliegenden Objekteigenschaft – aber nicht immer. Führen Sie *Get-Process* aus, wird eine Spalte namens »CPU(s)« genannt:

```
PS > Get-Process

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    244      23    58092      71408   208    17,18    548 AcroRd32
     76       8     1412        208    27             2656 alg
...
```

Dies entspricht aber nicht dem wahren Spaltennamen, und wenn Sie *Select-Object* beauftragen, die Spalte »CPU(s)« anzuzeigen, kommt es sogar zu einem Fehler. Hier hat das ETS aus Gründen der besseren Lesbarkeit für die Spaltenüberschrift einen anderen, abweichend Namen festgelegt. Die wahren Eigenschaftennamen erfahren Sie immer mit *Select-Object* *. Wie sich herausstellt, heißt die Eigenschaft in Wirklichkeit *CPU* und nicht *CPU(s)*.

```
PS > Get-Process | Select-Object *
...
Company                   : Adobe Systems Incorporated
CPU                       : 17,1757101
FileVersion               : 9.4.0.195
ProductVersion            : 9.4.0.195
...
```

Ähnlich ist es bei *Get-EventLog*:

```
PS > Get-EventLog System -Newest 2
   Index Time          EntryType   Source
   ----- ----          ---------   ------
  175275 Jan 31 21:03  Information Service Control M...
  175274 Jan 31 21:03  Information Service Control M...
```

Die Spalte *Time* heißt bei näherer Untersuchung in Wirklichkeit *TimeGenerated*. Bei den mitgelieferten Standard-Cmdlets sind solche Abweichungen nur selten. Bei einigen Befehlserweiterungen wie zum Beispiel den Cmdlets für Microsoft Exchange finden sich abweichenden Spaltenüberschriften dagegen häufig.

Alternativ können Sie auch *Get-Member* beauftragen, die Eigenschaften eines Objekttyps sichtbar zu machen:

```
PS > Get-Process powershell | Get-Member -memberType *Property
```

Platzhalterzeichen dürfen auch innerhalb von Eigenschaftennamen eingesetzt werden. Die folgende Zeile liefert die aktuelle Bildschirmauflösung eines physischen Systems (WMI liefert diese Informationen nicht für virtuelle Maschinen). Anstelle die beiden gewünschten Eigenschaften

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

vollständig als kommaseparierte Liste anzugeben (*CurrentHorizontalResolution*, *CurrentVerticalResolution*), wird nur die gemeinsame Endung **Resolution* festgelegt.

```
PS > Get-WmiObject Win32_VideoController | Select-Object *Resolution

        CurrentHorizontalResolution          CurrentVerticalResolution
        ---------------------------          -------------------------
                               1680                               1050
```

HINWEIS Möchten Sie selbst bestimmen, ob die Informationen als Tabelle oder als Liste dargestellt werden, greifen Sie anstelle von *Select-Object* zu *Format-Table* oder *Format-List*. Weil diese Cmdlets allerdings die Informationen umwandeln in Formatierungsobjekte, die nur noch von der Konsole korrekt dargestellt werden können, müssen *Format-*-Cmdlets stets am Ende einer Pipeline eingesetzt werden und dürfen höchstens noch an *Out-*-Cmdlets weitergeleitet werden. *Format-Table* hat eine besondere Bedeutung, wenn Spalteninhalte aus Platzgründen nicht vollständig in der Konsole angezeigt werden können. In diesem Fall kürzt PowerShell einzelne Spalteninhalte mit »...« ab. Um die Spalteninhalte möglichst vollständig anzuzeigen, verwendet man den Parameter *–AutoSize*:

```
PS > Get-Hotfix | Format-Table -AutoSize
```

Bei Angabe von *–AutoSize* sammelt *Format-Table* alle Ergebnisse zuerst, um danach die optimale Spaltenbreite zu bestimmen. So geht der Echtzeitcharakter der Pipeline zugunsten einer besseren Darstellung verloren. Auch mit *–AutoFormat* kann der Platz in der Konsole jedoch zu schmal sein, um Tabellen ungekürzt anzuzeigen. In diesem Fall kann der Parameter *–Wrap* helfen: Dieser bricht zu breite Spalteninhalte in mehrere Zeilen um:

```
PS > Get-Hotfix | Format-Table -AutoSize -Wrap
```

Möchten Sie Ergebnisse ungekürzt in eine Textdatei schreiben, verwenden Sie eine Kombination aus *Format-Table* und *Out-File*. Geben Sie bei *Out-File* den Parameter *–Width* an, damit PowerShell mehr Platz für die Ausgabe nutzen kann. Die folgende Zeile schreibt die Ergebnisse in eine Textdatei, die maximal 10.000 Zeichen breit sein darf, aber tatsächlich nur so breit sein wird wie nötig, um keine Spalte zu kürzen.

```
PS > Get-EventLog System -EntryType Error -Newest 10 | Format-Table -AutoSize | ⇢
     Out-File $home\report.txt -Width 10000
PS > Invoke-Item $home\report.txt
```

Ergebnisse einzeln verarbeiten

Sie möchten die Ergebnisse einer Pipeline einzeln in Echtzeit verarbeiten. Beispielsweise wollen Sie eine Liste mit Computernamen oder IP-Adressen anpingen.

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

Lösung

Setzen Sie *ForEach-Object* (Kurzform: »%«) ein. Das jeweilige Pipeline-Objekt steht in *$_* zur Verfügung. Die folgende Zeile listet jede einzelne Datei eines Ordnerlistings auf und gibt die Eigenschaften *Name* und *LastWriteTime* aus:

```
PS > dir | ForEach-Object { '{0} wurde am {1} zuletzt geändert' -f $_.Name, ⇢
     $_.LastWriteTime }
Application Data wurde am 19.11.2007 21:57:06 zuletzt geändert
Auswertung wurde am 13.05.2008 12:43:47 zuletzt geändert
bin wurde am 30.05.2008 07:16:48 zuletzt geändert
Bluetooth Software wurde am 16.11.2007 23:46:24 zuletzt geändert
Contacts wurde am 16.11.2007 18:07:55 zuletzt geändert
Desktop wurde am 30.05.2008 14:10:41 zuletzt geändert
(...)
```

Die Pipeline-Daten können auch aus einer Datei gelesen werden. Legen Sie eine Datei namens *Serverliste.txt* mit Servernamen an. Die folgende Zeile würde die in der Datei aufgeführten Server der Reihe nach abfragen:

```
PS > Get-Content c:\daten\Serverliste.txt | ForEach-Object { Get-WmiObject ⇢
     Win32_OperatingSystem -computer $_ }
```

Ebenso leicht könnten Sie die in der Datei aufgeführten Server herunterfahren:

```
PS > Get-Content c:\daten\Serverliste.txt | ForEach-Object ⇢
     { Stop-Computer -Computername $_-whatif }
```

Woher die Daten stammen, die die Pipeline verarbeitet, spielt keine Rolle. Sie müssen nicht von einem Befehl stammen. Die folgende Zeile generiert aus einem Zahlenbereich eine Liste mit IP-Adressen:

```
PS > 1..255 | ForEach-Object { "192.168.1.$_" }
192.168.1.1
192.168.1.2
192.168.1.3
192.168.1.4
```

Diese IP-Adressen könnten an *Test-Connection* weitergeleitet werden, um zu überprüfen, welche Computer gerade online und erreichbar sind:

```
PS > 1..255 | ForEach-Object { "192.168.1.$_" } | ForEach-Object ⇢
     { Test-Connection -computername $_ -Count 1 }
```

HINWEIS Falls diese Zeile Fehlermeldungen liefert, kann dies daran liegen, dass Sie augenblicklich nicht mit einem Netzwerk verbunden sind.

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

Auch formatierte Listen sind mithilfe des Formatierungsoperators »-f« möglich. Die folgende Zeile generiert eine Liste mit PC-Namen, wobei die laufende Nummer immer dreistellig ist:

```
PS > 1..100 | ForEach-Object { 'PC{0:000}' -f $_ }
PC001
PC002
PC003
...
PC010
PC011
```

Hintergrund

ForEach-Object erwartet einen Skriptblock, der für jedes Objekt ausgeführt wird, das vom vorangegangenen Befehl geliefert wurde. Innerhalb des Skriptblocks repräsentiert die Variable *$_* das jeweilige Objekt. Es liegt in Ihrer Verantwortung, innerhalb des Skriptblocks einen Rückgabewert zu liefern, den dann das nächstfolgende Cmdlet der Pipeline erhält. Im einfachsten Fall wird das empfangene Objekt in *$_* unverändert weitergereicht. So kann man beispielsweise Fortschrittsanzeigen realisieren:

```
PS > Dir $env:windir -filter *.log -recurse -ErrorAction SilentlyContinue | ⟶
     ForEach-Object { Write-Progress 'Suche Logbuchdateien...' $_.FullName; $_ }
```

Eine Aktualisierung der Fortschrittsanzeige in Echtzeit erreichen Sie über die folgende Zeile, die dafür sehr viel langsamer arbeitet:

```
PS > Dir $env:windir -recurse -ErrorAction SilentlyContinue | ForEach-Object ⟶
     { Write-Progress 'Suche Logbuchdateien...' (Split-Path $_.FullName); $_ } ⟶
     | Where-Object { $_.Extension -eq '*.log' }
```

Der Skriptblock muss allerdings nicht das empfangene Objekt unverändert weitergeben. Er kann es auch bearbeiten. Die folgende Zeile empfängt Textinformationen (IP-Adressen) und gibt DNS-Informationsobjekte zurück:

```
PS > '127.0.0.1', '127.0.0.2', '10.10.10.11' | ForEach-Object { $ip = $_; try ⟶
     { [System.Net.DNS]::GetHostByAddress($_) } catch { Write-Warning ⟶
     "Konnte $ip nicht auflösen: $_" } }
```

Grundsätzlich arbeitet *ForEach-Object* als Schleife. Sie könnten dieses Cmdlet also auch dazu verwenden, um Codeblöcke zu wiederholen. Die folgende Zeile startet 10 Instanzen von Notepad:

```
PS > 1..10 | ForEach-Object { notepad.exe }
```

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

ForEach-Object kann auch dazu eingesetzt werden, um nicht-Pipeline-fähige Befehle in der Pipeline auszuführen. Der Parameter *–computername* des Cmdlets *Stop-Computer* kann beispielsweise Informationen über die Pipeline nur von Objekten empfangen, die eine *ComputerName*-Eigenschaft besitzen. Möchten Sie einen Rechnerpark aber lieber über eine Liste mit Rechnernamen herunterfahren, setzen Sie *ForEach-Object* ein. Die folgende Zeile würde alle Rechner, die in der angegebenen Textdatei untereinander angegeben sind, herunterfahren:

```
PS > Get-Content c:\rechnerliste.txt | ForEach-Object ⟶
     { Stop-Computer -computername $_ -whatif }
```

Entfernen Sie den Parameter *–whatif*, wenn Sie die Rechner tatsächlich herunterfahren wollen.

Ist ein Cmdlet dagegen in der Lage, die Informationen direkt über die Pipeline zu empfangen, setzen Sie *ForEach-Object* nicht ein, sondern leiten die Informationen direkt an das Cmdlet weiter. Die folgende Zeile funktioniert zwar, ist aber nicht effizient:

```
PS > Get-Process notepad | ForEach-Object { Stop-Process -name $_.Name }
```

Formulieren Sie stattdessen:

```
PS > Get-Process notepad | Stop-Process
```

Überprüfen Sie stets, ob eine Pipeline für die Aufgabe überhaupt erforderlich ist:

```
PS > Stop-Process -name notepad
```

ForEach-Object kann am Beginn und am Ende der Pipeline weitere Aufgaben ausführen, wenn Sie anstelle eines Skriptblocks drei Skriptblöcke angeben. Die nächste Zeile färbt alle *.exe*-Dateien eines Ordnerlistings rot. Der *begin*- und der *end*-Block werden dazu verwendet, die ursprünglichen Konsolenfarben zu speichern und wiederherzustellen:

```
PS > dir $env:windir | ForEach-Object  -begin {
>> $farbeAlt = $host.UI.RawUI.ForegroundColor
>> } -process {
>> If ($_.Extension -eq '.exe') { $farbeNeu = 'Red' } else { $farbeNeu = 'Green' }
>> $host.UI.RawUI.ForegroundColor = $farbeNeu
>> $_
>> } -end {
>> $host.UI.RawUI.ForegroundColor = $farbeAlt
>> }
>>
```

ForEach-Object entspricht damit im Grunde einer Funktion mit den drei Skriptblöcken *begin*, *process* und *end*. Deshalb können Sie die Färbefunktion aus dem letzten Beispiel auch als Funktion formulieren:

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

```
PS > Function Markiere-Datei($extension='.exe') {
>> begin {
>> $farbeAlt = $host.UI.RawUI.ForegroundColor
>> }
>> process {
>> If ($_.Extension -eq $extension) { $farbeNeu = 'Red' } else →
   { $farbeNeu = 'Green' }
>> $host.UI.RawUI.ForegroundColor = $farbeNeu
>> $_
>> }
>> end {
>> $host.UI.RawUI.ForegroundColor = $farbeAlt
>> }
>> }
>>
```

Auf diese Weise heben Sie beliebige Dateien farblich hervor. Die nächste Zeile würde alle *.dll*-Dateien aus dem *System32*-Ordner farblich markieren:

```
PS > dir $env:windir\system32 | Markiere-Datei '.dll'
```

Der *begin*-Block wird ausgeführt, wenn die Pipeline startet. Hier merkt sich die Funktion die Ausgangsfarbe in *$farbeAlt*.

Der *process*-Block wird für jedes Element der Pipeline einmal aufgerufen. Hier prüft die Funktion den Dateityp und ändert dann die Vordergrundfarbe entsprechend. Anschließend wird das aktuelle Pipeline-Objekt in *$_* wieder auf die Pipeline gelegt, damit der nachfolgende Befehl es weiterbearbeiten kann.

ACHTUNG Legen Sie *$_* nicht zurück auf die Pipeline, wird dieses Ergebnis verschluckt. Das ist das Prinzip des Pipeline-Filters *Where-Object*.

Im *end*-Block schließlich finden die Aufräumungsarbeiten statt, nachdem die Pipeline beendet wurde. Hier restauriert die Funktion die Ausgangsfarben.

ACHTUNG Brechen Sie die Funktion mit Strg+C vorzeitig ab, wird der *end*-Block nicht ausgeführt und die Farben werden nicht auf die Ausgangseinstellung zurückgesetzt.

Ausgaben sortieren

Sie möchten die Ausgaben eines Befehls nach einem bestimmten Kriterium sortieren.

Lösung

Leiten Sie das Ergebnis über die Pipeline an *Sort-Object* weiter. Geben Sie dazu hinter *Sort-Object* an, nach welcher Eigenschaft Sie sortieren wollen. Die folgende Zeile liefert ein alphabetisches Ordnerlisting sämtlicher *.dll*-Dateien aus dem *System32*-Unterordner des *Windows*-Ordners:

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

```
PS > dir $env:windir\system32 *.DLL | Sort-Object Name

    Verzeichnis: Microsoft.PowerShell.Core\FileSystem::C:\Windows\system32

Mode           LastWriteTime       Length Name
----           -------------       ------ ----
-a---    18.01.2008    22:33       136192 aaclient.dll
-a---    18.01.2008    22:33      2515968 accessibilitycpl.dll
-a---    02.11.2006    08:28        39424 ACCTRES.dll
-a---    02.11.2006    10:46         7680 acledit.dll
-a---    18.01.2008    22:33       127488 aclui.dll
-a---    02.11.2006    10:46        38912 acppage.dll
-a---    02.11.2006    08:11         2048 acprgwiz.dll
(...)
```

Möchten Sie in absteigender Reihenfolge sortierten, geben Sie den Parameter *–descending* an. Möchten Sie zwischen Groß- und Kleinschreibung unterscheiden, geben Sie den Parameter *–caseSensitive* an.

Sort-Object kann auch nach berechneten Eigenschaften sortieren, falls keine vorhandene Eigenschaft abbildet, wonach Sie sortieren möchten. Die folgende Zeile sortiert einen Ordnerinhalt nach der Länge der Dateinamen:

```
PS > Dir $env:windir | Sort-Object { $_.Name.Length }
```

Hintergrund

Sort-Object sortiert Objekte nach einer oder mehreren Eigenschaften – sofern die Objekte überhaupt über Eigenschaften verfügen. Primitive Datentypen wie Zahlen oder Texte sortiert *Sort-Object* ohne zusätzliche Angaben:

```
PS > 1,4,2,7,6 | Sort-Object
```

Geben Sie zusätzlich den Parameter *–Unique* an, werden doppelte Resultate aus dem Ergebnis entfernt.

Stellt PowerShell die Ergebnisse in mehreren Spalten oder als Liste dar, wissen Sie, dass es sich um Objekte mit mehreren Eigenschaften handelt. In diesem Fall nennen Sie *Sort-Object* mit dem Parameter *–property* die Eigenschaft(en), nach denen sortiert werden soll. Die folgende Zeile sortiert die DLL-Dateien des Systemordners aufsteigend nach Größe (Eigenschaft *Length*):

```
PS > dir $env:windir\system32 *.dll | Sort-Object Length
```

Sort-Object kann auch nach mehreren Eigenschaften gleichzeitig sortieren. Die nächste Zeile sortiert laufende Prozesse zuerst nach Hersteller (Eigenschaft *Company*) und bei Gleichheit danach nach Name:

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

```
PS > Get-Process | Sort-Object Company, Name | Select-Object Name, Company
```

HINWEIS Als Nicht-Administrator dürfen Sie viele Prozesseigenschaften (wie zum Beispiel die Eigenschaft *Company*) nur von solchen Prozessen lesen, die Sie selbst gestartet haben. Bei allen anderen Prozessen ist die Eigenschaft dann leer.

Mit dem Parameter *Descending* drehen Sie die normalerweise aufsteigende Sortierreihenfolge um. Sortieren Sie nach mehreren Eigenschaften, legt der Parameter für alle Eigenschaften eine gemeinsame Sortierreihenfolge fest.

Möchten Sie die Sortierreihenfolge für einzelne Eigenschaften separat festlegen, verwenden Sie ein Hashtable. Die folgenden Zeilen sortieren alle laufenden Prozesse nach Hersteller in aufsteigender Reihenfolge. Der Speicherbedarf der Prozesse wird dagegen in absteigender Reihenfolge angegeben:

```
PS > $kriterium1 = @{expression='Company';Descending=$false}
PS > $kriterium2 = @{expression='VirtualMemorySize';Descending=$true}
PS > Get-Process | Sort-Object $kriterium1, $kriterium2 | Format-Table Name, ⟶
     Company, VirtualMem*
```

Sort-Object kann auch berechnete Eigenschaften zum Sortieren verwenden. Dazu geben Sie anstelle eines Eigenschaftsnamens einen Skriptblock an, der für jedes Objekt einzeln ausgewertet wird. Innerhalb des Skriptblocks repräsentiert *$_* das jeweils untersuchte Objekt. Auf diese Weise konnte *Sort-Object* ein Ordnerlisting nach der Länge des Dateinamens sortieren:

```
PS > Dir $env:windir | Sort-Object { $_.Name.Length }

    Verzeichnis: C:\Windows

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         06.10.2010     08:28            de
d----         14.07.2009     05:20            PLA
d----         05.02.2011     10:00            Logs
d----         01.11.2009     17:07            Help
d----         10.11.2009     10:19            en-US
d-r-s         12.11.2010     00:45            Fonts
d----         14.07.2009     04:36            system
d----         01.11.2009     09:27            Panther
d----         14.07.2009     07:32            Cursors
d----         14.07.2009     07:32            twain_32
d----         14.07.2009     04:35            SchCache
d----         14.07.2009     05:20            AppCompat
-a---         16.11.2009     11:30       6518 DPINST.LOG
```

Häufig kann man sich berechnete Eigenschaften aber auch zunutze machen, um den Datentyp zu korrigieren. Die folgende Zeile listet für alle DLL-Dateien aus dem Systemordner den Pfadnamen und die Dateiversion auf:

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

```
PS > dir $env:windir\system32 -filter *.dll | Select-Object -ExpandProperty ⇒
     VersionInfo | Select-Object FileName, ProductVersion
```

Würden Sie diese Zeile an *Sort-Object* weiterleiten und nach *ProductVersion* sortieren, wäre das Ergebnis nicht korrekt sortiert, denn die Eigenschaft *ProductVersion* verwendet den Datentyp *String* und wird von *Sort-Object* also alphabetisch sortiert. Das bedeutet, dass die hypothetische Versionsnummer »10.1.2.3« kleiner ist als »6.1.2.3«, weil bei der alphabetischen Sortierung zeichenweise vorgegangen wird und »1« kleiner ist als »6«.

Indem Sie *Sort-Object* einen Skriptblock übergeben, in dem die Eigenschaft in den für die Sortierung korrekten Datentyp umgewandelt wird, beheben Sie das Problem:

```
PS > dir $env:windir\system32 -filter *.dll | Select-Object -ExpandProperty ⇒
     VersionInfo | Select-Object FileName, ProductVersion | Sort-Object ⇒
     { try {[System.Version]$_.ProductVersion } catch { 0 }}
```

Ergebnisse gruppieren

Sie wollen die Ergebnisse eines Befehls nach einem oder mehreren Kriterien gruppieren. Sie möchten zum Beispiel Dienste nach ihrem Status gruppieren.

Lösung

Verwenden Sie das Cmdlet *Group-Object* und geben Sie dahinter das Kriterium an, nach dem Sie gruppieren wollen. Die folgende Zeile gruppiert Dienste nach ihrem Status:

```
PS > Get-Service | Group-Object Status

Count Name            Group
----- ----            -----
   89 Running         {AEADIFilters, AeLookupSvc, Appinfo, AudioEndpointBuilder...}
   67 Stopped         {ALG, AppMgmt, clr_optimization_v2.0.50727_32, COMSysApp...}
```

In der Spalte *Group* werden die gruppierten Objekte aufgeführt. Wenn Sie diese Information nicht benötigen, entfernen Sie sie mit dem Parameter *–noElement*. Dies spart erheblichen Speicherplatz.

Die nächste Zeile gruppiert Prozesse nach ihrem Hersteller:

```
PS > Get-Process  | Group-Object Company -noElement

Count Name
----- ----
    1 Adobe Systems Incorporated
   11 Microsoft Corporation
    1 Huawei Technologies Co., Ltd.
    1 Deutsche Post AG
```

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

Ebenso lassen sich Ereignislogbucheinträge nach *EntryType* gruppieren:

```
PS > Get-EventLog System | Group-Object EntryType -noElement

Count Name
----- ----
48724 Information
 3764 Error
 2985 Warning
```

Oder Sie gruppieren den Inhalt eines Dateiordners nach Dateierweiterung:

```
PS > Dir $env:windir | Group-Object Extension -noElement

Count Name
----- ----
   64
    1 .NET
    5 .ini
   13 .exe
    1 .dat
    8 .log
    1 .mif
(...)
```

Sie können auch nach mehreren Eigenschaften gleichzeitig sortieren. Doppelte Dateien finden Sie in Ihrem Benutzerprofil (in *$home*) beispielsweise, indem Sie nach *Length* und *CreationTime* gruppieren und alle Gruppen auflisten, die mehr als einmal vorkommen:

```
PS > Dir $home -recurse | Where-Object { $_.Length -gt 1KB } | Group-Object →
     Length, CreationTime | Where-Object { $_.Count -gt 1 } | ForEach-Object →
     { "Möglicherweise doppelt:"; $_.Group | ForEach-Object { '{0} ({1:0.0}KB)' →
     -f $_.FullName, ($_.Length/1KB)} }

Möglicherweise doppelt:
C:\Users\w7-pc9\video1.wmv (4930,5KB)
C:\Users\w7-pc9\video2.wmv (4930,5KB)

Möglicherweise doppelt:
C:\Users\w7-pc9\Documents\WindowsPowerShell\Modules\PSImageTools\FilterImagesIn
Directory.ps1 (3,9KB)
C:\Users\w7-pc9\Documents\WindowsPowerShell\Modules\PSImageTools\Example\Filter
ImagesInDirectory.ps1 (3,9KB)
```

HINWEIS Die identifizierten Dateien müssen nicht doppelt sein. Zwar ist es extrem unwahrscheinlich, dass unterschiedliche Dateien sowohl dieselbe Größe als auch dieselbe Erstellungszeit aufweisen, jedoch nicht unmöglich. Dieses Risiko ist umso größer, je kleiner die Dateien sind, weswegen der Code keine Dateien kleiner als *1KB*

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

berücksichtigt. Wirklich sicher identifiziert nur ein Dateihash identische Dateiinhalte. Die Erstellung von Hashwerten ist besonders bei größeren Dateien allerdings sehr zeitraubend und ressourcenintensiv und daher nur selten praktikabel.

Hintergrund

Group-Object gruppiert beliebige Objekte auf der Basis einer oder mehrerer Eigenschaften. Als Ergebnis liefert das Cmdlet *GroupInfo*-Objekte zurück. Jedes *GroupInfo*-Objekt meldet in *Count*, wie viele Objekte darin gruppiert wurden. Name meldet, welche Gemeinsamkeit in der zur Gruppierung verwendeten Eigenschaft gefunden wurde. *Group* schließlich enthält die Objekte, die in dieser Gruppe zusammengefasst wurden.

Geht es Ihnen nur darum, Häufigkeiten zu ermitteln, und brauchen Sie also die zugrunde liegenden Objekte nicht, sparen Sie erheblichen Speicherplatz mit dem Parameter *–noElement*. Wird er angegeben, verzichtet *Group-Object* darauf, die zugrunde liegenden Objekte aufzubewahren und in *Group* zurückzuliefern.

```
PS > dir $env:windir | Group-Object Extension -noElement | Sort-Object Name ⇢
     | ForEach-Object { '{0,30} = kam {1} mal vor' -f $_.Name, $_.Count }
                              = kam 64 mal vor
                          .bin = kam 1 mal vor
                          .dat = kam 1 mal vor
                          .dll = kam 2 mal vor
                          .exe = kam 13 mal vor
                          .ini = kam 5 mal vor
                          .log = kam 8 mal vor
                         .logs = kam 1 mal vor
                          .mif = kam 1 mal vor
                          .NET = kam 1 mal vor
                          .prx = kam 1 mal vor
                          .SCR = kam 1 mal vor
                          .txt = kam 2 mal vor
                          .xml = kam 2 mal vor
(...)
```

Group-Object kann auch nach berechneten Eigenschaften gruppieren. Dazu geben Sie die Eigenschaft als ausführbaren Skriptblock an. *Group-Object* gruppiert dann nach dem Ergebnis dieses Skriptblocks. Die nächste Zeile teilt Dateien zum Beispiel in zwei Gruppen, nämlich Dateien, die größer sind als 100 KB, und Dateien, die nicht größer sind als 100 KB:

```
PS > dir $env:windir | Group-Object {$_.Length -gt 100KB}

Count Name                      Group
----- ----                      -----
   94 False                     {AABBCC, addins, AppCompat, AppPatch...}
    9 True                      {explorer.exe, HelpPane.exe, notepad.exe, ntbtlog.txt...}
```

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

Liefert der Gruppierungsausdruck mehr als zwei Ergebnisse, erhalten Sie entsprechend mehr als zwei Gruppen. Die folgende Zeile bildet Gruppen auf der Basis der Anfangsbuchstaben der Dateinamen:

```
PS > dir $env:windir | Group-Object { $_.Name.SubString(0,1).toUpper() }

Count Name                      Group
----- ----                      -----
    6 A                         {AABBCC, addins, AppCompat, AppPatch...}
    4 B                         {Boot, Branding, bfsvc.exe, bootstat.dat}
    2 C                         {CSC, Cursors}
   10 D                         {de, de-DE, debug, diagnostics...}
(...)
    8 W                         {Web, winsxs, win.ini, WindowsUpdate.log...}
    2 N                         {notepad.exe, ntbtlog.txt}
    1 U                         {Ultimate.xml}
(...)
```

Weil jede Gruppe von einem *GroupInfo*-Objekt repräsentiert wird, das in seiner Eigenschaft *Group* die gruppierten Dateien noch enthält, könnten Sie sich auf diese Weise ein alphabetisch formatiertes Ordnerlisting generieren:

```
PS > dir $env:windir | Group-Object { $_.Name.SubString(0,1).toUpper() } ⇢
     | ForEach-Object { ''; "  - $($_.Name) - "; '---------'; ''; $_.Group }

  - A -
---------

    Verzeichnis: Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias


Mode           LastWriteTime       Length Name
----           -------------       ------ ----
d----    19.11.2007    21:57       <DIR> Application Data
d----    13.05.2008    12:43       <DIR> Auswertung
-a---    21.05.2008    09:16        8277 arbeitsblatt.xls

  - B -
---------

d----    30.05.2008    07:16       <DIR> bin
d----    16.11.2007    23:46       <DIR> Bluetooth Software
-a---    31.03.2008    17:03        1355 batch.vbs
-a---    31.03.2008    17:09        1355 batchersatz.vbs
(...)
```

Wollen Sie häufiger auf diese Weise Ordnerlistings gruppieren, schreiben Sie sich eine passende Funktion wie zum Beispiel *Group-Alphabetisch*:

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

```
PS > Function Group-Alphabetisch {
>> $input | Group-Object { $_.Name.SubString(0,1).toUpper() } | ForEach-Object ⇢
   { ''; "  - $($_.Name) - "; '---------'; ''; $_.Group }
>> }
>>
```

Diese Funktion blockiert anders als ein Filter die Pipeline, bis der Vorgängerbefehl sämtliche Ergebnisse geliefert hat. Die Ergebnisse stehen danach in *$input* zur Verfügung und können nun an *Group-Object* weitergeleitet werden.

```
dir | Group-Alphabetisch
```

TIPP Das Kriterium für die Gruppenbildung kann, wie Sie gesehen haben, frei berechnet werden. Im folgenden Beispiel werden alle Dateien im Windows-Ordner einer der drei Kategorien klein, mittel und groß zugeordnet:

```
PS > Dir $env:windir | Group-Object { Switch($_.Length) {
>> { $_ -eq $null } { 'Ordner'; continue }
>> { $_ -lt 1KB }   { 'klein'; continue }
>> { $_ -lt 1MB }   { 'mittel'; continue}
>> default          { 'gross' }
>> }}
>>

Count Name         Group
----- ----         -----
   65 Ordner       {AABBCC, addins, AppCompat, AppPatch...}
    8 klein        {avisplitter.ini, DirectX.log, iltwain.ini, setuperr.log...}
   27 mittel       {bfsvc.exe, bootstat.dat, DPINST.LOG, DtcInstall.log...}
    3 gross        {explorer.exe, Reflector.exe, WindowsUpdate.log}
```

Speichern Sie das Ergebnis in einer Variablen und geben die Parameter *–asHash* und *–asString* an, erhalten Sie damit einen sehr einfachen und übersichtlichen Weg, mit kleinen, mittelgroßen und großen Dateien zu arbeiten:

```
PS > $dateien = Dir $env:windir | Group-Object { switch($_.Length) {
>> { $_ -eq $null } { 'Ordner'; continue }
>> { $_ -lt 1KB }   { 'klein'; continue }
>> { $_ -lt 1MB }   { 'mittel'; continue}
>> default          { 'gross' }
>> }} -asHash -asString
>>

PS > $dateien.klein
```

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

```
    Verzeichnis: C:\Windows

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         14.03.2010     19:00         38 avisplitter.ini
-a---         06.10.2010     08:21        197 DirectX.log
-a---         16.04.2010     14:53         36 iltwain.ini
-a---         14.07.2009     06:51          0 setuperr.log
-a---         10.06.2009     23:08        219 system.ini
-a---         21.06.2010     12:40         16 test.logs
-a---         18.10.2010     16:52         16 test.txt
-a---         06.11.2009     14:06        478 win.ini

PS > $dateien.gross

    Verzeichnis: C:\Windows

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         31.10.2009     07:34    2870272 explorer.exe
-a---         12.11.2010     16:10    2854328 Reflector.exe
-a---         14.02.2011     08:48    1963673 WindowsUpdate.log
```

Ergebnisse in Text umwandeln

Sie möchten das Ergebnis eines Befehls in Text umwandeln, zum Beispiel, um den Text farbig auszugeben.

Lösung

Verwenden Sie *Out-String*. Die folgende Zeile wandelt das Ergebnis von *Get-Process* in Text um und gibt diesen in rot aus:

```
PS > Get-Process | Out-String -Stream  | Write-Host -foreground Red

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    244      23    58092      71480   208    17,21    548 AcroRd32
     76       8     1412        208    27             2656 alg
    211      19     3852       7040    82             1728 AppleMobi...
```

Ohne *Out-String* wären nur die Standardobjekteigenschaften in Text verwandelt worden:

```
PS > Get-Process | Write-Host -foreground Red
System.Diagnostics.Process (AcroRd32)
System.Diagnostics.Process (alg)
System.Diagnostics.Process (AppleMobileDeviceService)
...
```

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

Hintergrund

Leiten Sie Objekte an Befehle weiter, die eigentlich nur mit Text umgehen können, kommt es zu einer automatischen Umwandlung in Text. Diese Umwandlung wird nicht von PowerShell durchgeführt, sondern durch die *toString()*-Methode von .NET Framework. Objekte werden dabei meist durch ihren Datentypnamen ersetzt:

```
PS > (Get-Process -id $pid).toString()
System.Diagnostics.Process (powershell)
```

Sehr viel intelligenter arbeitet die Textumwandlung von PowerShell. Diese wird häufig automatisch aktiv, zum Beispiel, wenn Sie Objekte in die Konsole ausgeben. Leiten Sie dagegen Objekte an andere Befehle weiter, wird unter Umständen die automatische Konvertierung von .NET Framework aktiv. Damit dies nicht geschieht, wandeln Sie Objekte in solchen Fällen manuell mithilfe des Cmdlets *Out-String* in Text um. Der Parameter *–Stream* sorgt dafür, dass bei mehreren Objekten die einzelnen Objekte auch einzeln in Text umgewandelt werden.

```
PS > $text = Get-Process -id $pid | Out-String -Stream
PS > $text

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    867      35   123052     134532   594   156,66   4084 powershell

PS > $text[3]
    867      35   123052     134532   594   156,66   4084 powershell
```

Out-String bildet also die Brücke zwischen der objektorientierten internen PowerShell-Welt und der textorientierten Ausgabe-Welt. Dies können Sie sich zunutze machen, um ein einfaches, aber sehr effektives Werkzeug herzustellen. Der folgende Pipeline-Filter namens *grep* wandelt intern die Ergebnisse in Text um und prüft dann, ob ein beliebiges Stichwort darin vorkommt. Falls ja, wird das Objekt durch die Pipeline hindurchgelassen, andernfalls ausgefiltert:

```
PS > Filter grep($stichwort) { if ( ($_ | Out-String) -like "*$stichwort*") ⇢
     { $_ } }
```

Möchten Sie nun alle laufenden Dienste sehen, genügt:

```
PS > Get-Service | grep running
```

Alle *.exe*-Dateien im Windows-Ordner erhalten Sie nun so:

```
PS > Dir $env:windir | grep .exe
```

Alle Dateien, die am 14. Juli 2010 geändert wurden, finden Sie so:

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

```
PS > dir $env:windir | grep 14.07.2010
```

Und alle Aliasnamen für *Get-ChildItem* ermittelt diese Zeile:

```
PS > Get-Alias | grep child
```

Sie erhalten also auf diese Weise sehr einfach und leicht Filterergebnisse, denn Sie brauchen nicht länger anzugeben, in welcher Spalte Sie ein bestimmtes Ergebnis erwarten. Dafür allerdings können die Ergebnisse unscharf sein. Zurückgeliefert wird jedes Objekt, das in irgendeiner (sichtbaren) Spalte das Suchwort enthält.

Möchten Sie in sämtlichen Eigenschaften eines Objekts suchen, also auch in den normalerweise verborgenen, ändern Sie den Filter geringfügig und wandeln nicht nur die sichtbaren, sondern auch die unsichtbaren Eigenschaften in Text um, bevor Sie darin nach dem Stichwort suchen:

```
Filter grep($stichwort) { if ( ($_ | Select-Object * | Out-String) ⇾
-like "*$stichwort*") { $_ } }
```

Der Filter arbeitet nun langsamer, findet aber mehr. Alle Prozesse der Firma »Microsoft« finden Sie nun so:

```
PS > Get-Process | grep Microsoft
```

Im sichtbaren Ergebnis wird Ihr Suchwort möglicherweise nun gar nicht mehr vorkommen. Das Beispiel hat das Stichwort »Microsoft« beispielsweise in der normalerweise verborgenen Eigenschaft *Company* gefunden:

```
PS > Get-Process | grep Microsoft | Select-Object Name, Company
```

Zusammenfassung

Lediglich neun neue Cmdlets aus Tabelle 4.1 waren nötig, um mithilfe der PowerShell-Pipeline anspruchsvolle Aufgaben in einer einzigen Zeile Code zu lösen.

Aufgabe	Seite
Alle Dateien größer als 1MB auflisten	123
Alle Prozesse mit mehr als 20 Sekunden CPU-Belastung zeigen	123
Alle laufenden Prozesse der Firma »Microsoft« auflisten	123
Alle beendeten Dienste auflisten	123
Nur Dateien oder nur Unterordner eines Ordners anzeigen	123

Tabelle 4.2 Auswahl von Lösungen in diesem Kapitel, die durch die Pipeline ermöglicht wurden

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8

Aufgabe	Seite
Zeilen aus einer Logbuchdatei filtern, die ein bestimmtes Stichwort enthalten	123
Alle Hotfixes finden, die mit einer bestimmten KB-Artikelnummer beginnen	123
Nur die Zeilen des Befehls *ipconfig* anzeigen, die das Wort »IP« enthalten	124
Aktuelle Videoauflösung anzeigen	129
Die neuesten 10 Errorereignisse ungekürzt in eine Textdatei schreiben	129
Alle Computer aus der Datei serverliste.txt remote per WMI abfragen	130
Alle Computer aus der Datei serverliste.txt herunterfahren	130
Eine Liste mit IP-Adressen generieren	130
Ein IP-Adresssegment anpingen und auf Verfügbarkeit der Rechner prüfen	130
Eine Liste mit PC-Namen generieren	130
Ein IP-Adresssegment per DNS auflösen	131
Sortierte Ordnerlistings erzeugen	133
Doppelt vorkommende Dateien im Dateisystem finden	137
Einen eigenen »grep«-Filter zum leichteren Filtern von Befehlsergebnissen anlegen	142

Tabelle 4.2 Auswahl von Lösungen in diesem Kapitel, die durch die Pipeline ermöglicht wurden *(Fortsetzung)*

Das Prinzip dabei war bei jedem Beispiel identisch: Die Rohergebnisse eines Cmdlets wurden zuerst mit **-Object*-Cmdlets verfeinert, gefiltert und sortiert und anschließend mit *Format-**/*Out-**-Cmdlets ausgegeben.

Name	Beschreibung
ForEach-Object	Kurzform: »%«; bearbeitet alle Ergebnisse der Pipeline einzeln und in Echtzeit. Entspricht einer Schleife.
Format-List	Listet die Eigenschaften eines Objekts in Textdarstellung untereinander auf
Format-Table	Listet die Eigenschaften eines Objekts in Textdarstellung nebeneinander auf. Kann die Spaltenbreite optimieren und dafür sorgen, dass Zeilen umgebrochen und nicht abgeschnitten werden.
Group-Object	Gruppiert Objekte nach einer Eigenschaft oder einem Kriterium
Out-File	schreibt die Ergebnisse in eine Datei
Out-String	Wandelt ein Objekt in Text um
Select-Object	Beschränkt ein Objekt auf bestimmte gewünschte Eigenschaften und entfernt alle übrigen
Sort-Object	Sortiert das Ergebnis der Pipeline nach einem oder mehreren Kriterien
Where-Object	Kurzform: »?«; lässt nur diejenigen Objekte hindurch, die einem bestimmten Kriterium entsprechen, und filtert die übrigen heraus

Tabelle 4.3 Cmdlets in diesem Kapitel

Weltner, Dr. Tobias, Windows PowerShell 2.0 - Scripting für Administratoren, © 2011 Microsoft Press Deutschland, 978-3-86645-680-8