KAPITEL 8

Pipelining

Die Pipeline ist eines der grundlegendsten Elemente der PowerShell. Über den Pipeline-Operator | werden Objekte zwischen Cmdlets übergeben. Die Ausgabe des ersten Cmdlets wird zur Eingabe des zweiten Cmdlets. Dabei kann es sich um einfache Datentypen, einzelne Objekte oder um eine Vielzahl von Objekten handeln. Genau dieses Prinzip unterscheidet die PowerShell von anderen Shells, wie man sie gegebenenfalls von Unix kennt. Während dort auf der Basis von Text Informationen aus einem Verarbeitungsschritt über die Pipeline an den nächsten Befehl übergeben werden, sind es bei der PowerShell in aller Regel typisierte Objekte auf der Basis des .NET Framework. Eine komplexere Textanalyse kann entfallen.

```
# Beispiel 1:#
Ein einzelner Wert wird an einen Parameter übergeben.
Get-Date -date '1961-04-12'

# Mehrere Werte werden über die Pipeline übergeben.
'1961-04-12','1969-07-21' | Get-Date

# Die Pipeline erweitern.
'1961-04-12','1969-07-21' | Get-Date |
  Select-Object -Property DayOfWeek
```

Das Prinzip des Förderbands wird häufig genutzt, um die Arbeitsweise der Pipeline zu verdeutlichen. Im Beispiel oben zeigt sich dies ganz exemplarisch: Ein Werkstoff (unsere Daten) werden von einem ersten Cmdlet (`Get-Date`) verarbeitet, das Ergebnis wird an ein zweites Cmdlet (`Select-Object`) weitergereicht. Sie benötigen die Pipeline nicht zwingend, wenn Sie nur ein einzelnes Objekt (in unserem Fall: ein Datum) verarbeiten wollen. Sobald Sie mehrere

Objekte/mehrere Daten verarbeiten möchten, kann sie die Arbeit allerdings stark vereinfachen.

Natürlich können Sie auch bei vielen Objekten andere Lösungswege finden:

- Sie könnten eine Batch-Datei erstellen und den Befehl n-mal wiederholen (nicht sehr elegant).
- Sie schreiben eine verbesserte Version von `Get-Date`, die Ihnen erlaubt, mehrere Daten in der Art `Get-Date -date '1961-04-12','1969-07-20'` zu übergeben (viele andere Cmdlets erlauben dies).
- Sie könnten eine beliebige Schleife verwenden.

Ein zentraler Vorteil der Pipeline ist der intuitiv verständliche Aufbau und die simple Tatsache, dass man in einer kommandozeilenorientierten Konsole die Pipeline Stück für Stück aufbauen kann – sehr viel einfacher als mit einer `for`-Anweisung. Auch die Anzahl der Eingabeobjekte kann beliebig variieren von 1 bis zu 1.000 Elementen. Die Syntax ändert sich nicht.

Ob das Cmdlet `Get-Date` überhaupt in der Lage ist, Pipeline-Eingaben zu verarbeiten, entnehmen Sie der Hilfe.

```
Ausgabe des Befehls: Get-Help -name Get-Date -parameter date
-Date <datetime>
    Required?                   false
    Position?                   0
    Accept pipeline input?      true (ByValue, ByPropertyName)
    Parameter set name          (All)
    Aliases                     LastWriteTime
    Dynamic?                    false
```

Beispiel 1 auf Seite 69 zeigt ein sogenanntes *Pipelining ByValue*: Einfache Zeichenfolgen werden an die Pipeline übergeben. Oftmals werden Sie komplexere Informationen verarbeiten wollen. Angenommen, Sie hätten eine Datei mit strukturierten Daten (zum Beispiel eine CSV-Datei wie in Abbildung 8-1) vorliegen, die Sie zur Eingabe verwenden wollen. Zu diesem Zweck muss eindeutig gekennzeichnet sein, welche Teile der Datensätze für die Weiterverarbeitung relevant sind.

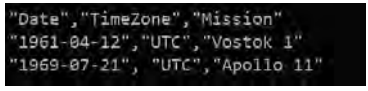

Abbildung 8-1: Die Datei »missions.csv«

```
# Beispiel 2: Pipelining by PropertyName
Import-Csv -Path 'missions.csv' | Get-Date |
  Select-Object -Property DayOfWeek
```

Wir konnten der Hilfe von `Get-Date` entnehmen, dass Pipeline-Eingaben für den Parameter `-date` sowohl `ByValue` als auch `ByPropertyName` möglich sind. *Pipelining ByPropertyName* erfordert es, dass die Eingabedaten auf den korrespondierenden Parameter verweisen. Exakt dies nutzten wir oben in Beispiel 2 aus.

ForEach-Object

Leider kann nicht jedes Cmdlet von Haus aus Pipeline-Eingaben verarbeiten. Möglicherweise sind Sie in Einzelfällen auch auf externe Programme angewiesen. Für diesen Fall bringt die PowerShell eine Art Schweizer Taschenmesser mit: `ForEach-Object` (alias `%`, `foreach`)

```
# Beispiel 1: Ein externes Programm wird in die
# Pipeline eingebunden.
'www.oreilly.de','www.dpunkt.de' |
  ForEach-Object -Process { nslookup $_ }

# Beispiel 2: Der Alias % für ForEach-Object.
'www.oreilly.de','www.dpunkt.de' | % { nslookup $_ }

# Beispiel 3: Suche mithilfe des Active Directory
# (nur Windows).
Get-ADComputer -Filter * | ForEach-Object -Process {
    Test-Connection -ComputerName $_.DNSHostname
      -Count 1 -ErrorAction SilentlyContinue
}
```

Beispiel 3 erfordert ein Active Directory und die Installation der RSAT (siehe auch Kapitel 10, Den Funktionsumfang der PowerShell erweitern).

Die besondere Stärke von *ForEach-Object* liegt darin begründet, dass Sie einen Skriptblock {} verwenden können, in dem Sie beliebig viele Anweisungen mit jeder Iteration ausführen lassen. Innerhalb des Skriptblocks steht Ihnen die automatische Variable `$_` zur Verfügung, sie repräsentiert das aktuelle Objekt in der Pipeline.

`$_` wird häufig als *Special Pipeline Variable* bezeichnet. Entgegen dieser Bezeichnung findet sich die Variable auch außerhalb der Pipeline, z.B. in einer `switch`-Anweisung (siehe Abschnitt »Die Anweisung switch« auf Seite 62). Seit PowerShell 3 kann man alternativ auch `$psitem` verwenden.

ForEach-Object vs. foreach

Das Cmdlet `ForEach-Object` lässt sich leicht mit dem Schlüsselwort `foreach` verwechseln, das Sie im Abschnitt »Die foreach-Schleife« auf Seite 65 bereits kennengelernt haben.

```
# Pipelining mit ForEach-Object
'Tick','Trick','Track' |
   ForEach-Object -process {
     "$_ ist ein Neffe von Donald!"
}

# Eine Schleife mit foreach
foreach ($neffe in 'Tick','Trick','Track') {
  "$neffe ist ein Neffe von Donald!"
}
```

Sie sollten diese beiden Verfahrensweisen trotz aller Ähnlichkeit sauber voneinander trennen. Sie können im ersten Beispiel die Pipeline weiter fortführen, was im zweiten Beispiel nicht möglich ist. Dafür können Sie Schleifen mit `foreach` sehr viel einfacher verschachteln, da Sie die Laufvariable explizit benennen. Darüber hinaus ist die `foreach`-Schleife, wie bereits in Kapitel 7, *Flusskontrolle*, erläutert, allgemein schneller.

Mein Tipp: Verzichten Sie grundsätzlich auf den Alias `ForEach` für `ForEach-Object`, um Ihren Code fehlerfrei zu halten.

Seit PowerShell 7 unterstützt `ForEach-Object` die gleichzeitige Ausführung des Skriptblocks mit dem Paramater `-parallel`. Dabei nutzt PowerShell 7, ebenso wie das neu entwickelte Modul *ThreadJobs*, sogenannte *Runspaces*, die es erlauben, multiple Threads innerhalb des PowerShell-Prozesses zu erzeugen.

```
# Beispiel 1: Parallele Ausführung
'www.oreilly.de','www.dpunkt.de' | ForEach-Object -Process {
    Test-Connection -ComputerName $_ -Count 4
      -ErrorAction SilentlyContinue
  }

# Beispiel 2: Zeit messen
Measure-Command -Expression {
  $count = 4
  'www.oreilly.de','www.dpunkt.de' |
    ForEach-Object -ThrottleLimit 2 -Parallel {
      Test-Connection -ComputerName $_
        -Count $using:count -ErrorAction SilentlyContinue
  }
} | Select-Object -ExpandProperty TotalSeconds
```

Prinzipiell kann für jedes Eingabeobjekt ein eigener Runspace erzeugt und der gesamte Skriptblock kann parallelisiert werden. In den Beispielen oben werden zwei Elemente über die Pipeline transportiert, nämlich die Rechnernamen. Mit dem Parameter `-Throttle Limit` geben Sie die maximale Anzahl von Runspaces vor. Verringern Sie den Wert auf 1, arbeitet `ForEach-Object` ebenso sequenziell wie mit dem Parameter `-Process`. Erhöhen Sie den Wert, haben Sie keinen positiven Effekt: Mehr als zwei Runspaces werden hier nicht benötigt.

Wenn Sie sehr viele Eingabeobjekte haben, sollten Sie dennoch vorsichtig mit der Erhöhung des ThrottleLimits sein. Das Erzeugen der Ressourcen kostet Rechenzeit, und die de facto verfügbaren Runspaces richten sich nach den verfügbaren CPU-Kernen! Diese Kerne müssen aber alle möglichen Aufgaben erledigen, sodass das ThrottleLimit kein beliebig skalierbarer Wert ist. Der Standardwert ist 5 – aus gutem Grund.

Unterm Strich können Sie bei allzu leichtfertiger Verwendung Ihre Befehle sogar ausbremsen. Eine schnelle `foreach`-Schleife kann bei vielen kurzlebigen Aufgaben die wesentlich bessere Wahl sein.

Where-Object

Die Arbeit mit der PowerShell erfordert häufig den Umgang mit großen Datenmengen und mit zahlreichen Objekten. In vielen Fällen werden Sie eine Auswahl der für Sie relevanten Daten treffen wollen und nur diese Daten an einen nachfolgenden Befehl in der Pipeline weitergeben. Genau diesen Zweck erfüllt das Cmdlet `Where-Object` (alias `?`, `where`), das im Wesentlichen nichts anderes ist als die elegante Kombination von `ForEach-Object` und einer `if`-Anweisung.

```
# Beispiel 1: Finde Dateien, die größer als 10 MB sind.
Get-ChildItem -Path ~ -File -Recurse |
  Where-Object -FilterScript { $_.Length -gt 10MB } |
    Sort-Object -Property Length -Descending

# Beispiel 2: Finde Dateien, die in den letzten 6 Monaten
geändert wurden.
Get-ChildItem -Path ~ -File -Recurse |
  Where-Object -FilterScript {
   $_.LastWriteTime -gt (Get-Date).AddMonths(-6)
 }

# Beispiel 3: Finde Prozesse, die mehr als 42 s CPU-
Zeit verbraucht haben.
Get-Process | Where-Object -FilterScript { $_.CPU -gt 42 } |
  Sort-Object -Property CPU -Descending

# Beispiel 4: Finde lokale Benutzer, deren SID auf "-
500" endet (nur Windows).
Get-LocalUser | ? { $_.SID -like '*-500' } |
  Format-List -Property Name, Enabled, Description, SID

# Beispiel 5: Prüfe die Erreichbarkeit (nur PowerShell 7)
'www.oreilly.de','www.dpunkt.de' | ? {
    Test-Connection -TargetName $_ -TcpPort 80
}
```

Obgleich ich in diesem Buch weitgehend auf verkürzte Schreibweisen verzichte, habe ich in den letzten Beispielen einige Male den Alias ? verwendet, da Sie diese Kurzschreibweise in vielen Skriptbeispielen finden werden – sei es in der Arbeit von Kolleginnen und Kollegen, im Internet, in den Hilfedokumenten oder anderen Büchern. Das Gleiche gilt für den Alias % und `ForEach-Object`.

Tipp

`Where-Object` ist ein großartiges Werkzeug, dennoch sollte es immer Ihre zweite Wahl sein. Wann immer ein Cmdlet über einen Parameter verfügt, mit dem Sie Daten unmittelbar filtern können, sollten Sie diesen bevorzugen.

```
# Erste Wahl
Get-ChildItem -path ~ -Filter '*.ps1'

# Zweite Wahl
Get-ChildItem -path ~ | Where-Object
-FilterScript {
  $_.Extension -eq '.ps1'
}
```

Das Filtern mithilfe des für das Dateisystem spezialisierten Cmdlets `Get-ChildItem` wird signifikant schneller sein als das generische Filtern mittels `Where-Object`. In vielen Fällen bieten Cmdlets spezielle Filter für den jeweiligen Anwendungsfall; so ermöglichen die Get-AD*-Cmdlets das sehr mächtige Verwenden von LDAP-Filtern zur Suche im Active Directory.

Calculated Properties

In der Pipeline verwenden Sie häufig Cmdlets wie `Select-Object`, `Format-Table` oder `Format-List`, um Ausgaben an Ihre eigenen Bedürfnisse anzupassen. Sie müssen sich hierbei aber nicht auf die Auswahl von Eigenschaften beschränken, Sie können viele weitergehende Anpassungen durchführen.

Ein Beispiel: Dateisystemobjekte geben die Dateigröße unter der Bezeichnung `Length` in Byte aus, was nicht sonderlich aussagekräf-

tig ist! Sie können mit einer Hash Table die Ausgabe modifizieren. Dabei tritt die Hash Table an die Stelle der Eigenschaft in dieser allgemeinen Form: `@{ name=''; expression ={} }`

```
# Ersetzen des Eigenschaftsnamens
Get-Item -Path 'helloworld.ps1' |
  Select-Object -Property Mode, LastWriteTime,
    @{ Name='Size'; expression={$_.Length}}, Name

# Umrechnen in Kilobyte (KiB)
Get-Item -Path 'helloworld.ps1' |
  Select-Object -Property Mode, LastWriteTime,
    @{ Name='Size(KiB)'; expression={$_.Length/1024}}, Name
```

Die Schlüssel Name/Label[1] und Expression werden von verschiedenen Cmdlets unterstützt. Weitergehende Auszeichnungen sind insbesondere mit `Format-Table` möglich. Alle Schlüssel lassen sich mit einem einzelnen Buchstaben abkürzen, siehe Tabelle 8-1.

Tabelle 8-1: Auszeichnungsschlüssel

Kürzel	Schlüssel	Werte	CmdLets
n, l	Name, Label	beliebig	Format-Table, Format-List, Select-Object
e	Expression	{Scriptblock}	Format-Table, Format-List, Select-Object
f	Format-String	.NET Format-Strings[2]	Format-Table, Format-List
a	Align(ment)	left, center, left	Format-Table
w	Width	Anzahl Zeichen	Format-Table

`Format-Table` bietet die am weitesten reichenden Möglichkeiten. Bedenken Sie aber, dass die Pipeline mit einem Format-Cmdlet enden sollte, eine weitergehende Verarbeitung ist in der Regel nicht möglich (wie zum Beispiel der Export in eine CSV-Datei). Eine weitere Besonderheit ist, dass eine Breitenangabe mit `width` nur dann

1 Ursprünglich verwendete ausschließlich das Cmdlet `Select-Object` den Schlüssel »Name« im Gegensatz zum gebräuchlichen »Label«. Um Irritationen zu vermeiden, kann man seit PowerShell 2 beide Schlüssel verwenden.

2 Siehe »Format types in .NET«: *https://docs.microsoft.com/en-us/dotnet/standard/base-types/formatting-types*

funktioniert, wenn die fragliche Eigenschaft als erstes Element genannt wird.

```
# Weitergehende Anpassungen mit Format-Table
Get-Item -Path 'helloworld.ps1' |
  Format-Table -Property @{
    n='Size(KiB)';w=42; e={$_.Length/1024};f='N2';a='left'
  }, Mode, LastWriteTime, Name
```

Sie sollten sich im Alltag möglichst mit `Select-Object` und den Schlüsseln *Name/Label* und *Expression* begnügen. Die Einschränkungen, die `Format-Table` mit sich bringt, wiegen stärker als der Nutzen. Den Formatschlüssel können Sie gleichwertig ersetzen, wie Ihnen die abschließenden Beispiele zeigen.

```
# Formatierung mit dem Format-Operator
Get-Item -Path 'helloworld.ps1' |
  Select-Object -Property Mode, LastWriteTime, @{
    n='Size(KiB)'
    e={'{0:N3}' -f ($_.Length/1024)}
  }, Name

# Runden statt formatieren
Get-Item -Path 'helloworld.ps1' |
  Select-Object -Property Mode, LastWriteTime,  @{
    n='Size(KiB)'
    e={ [Math]::Round($_.Length/2,3)}
  }, Name
```