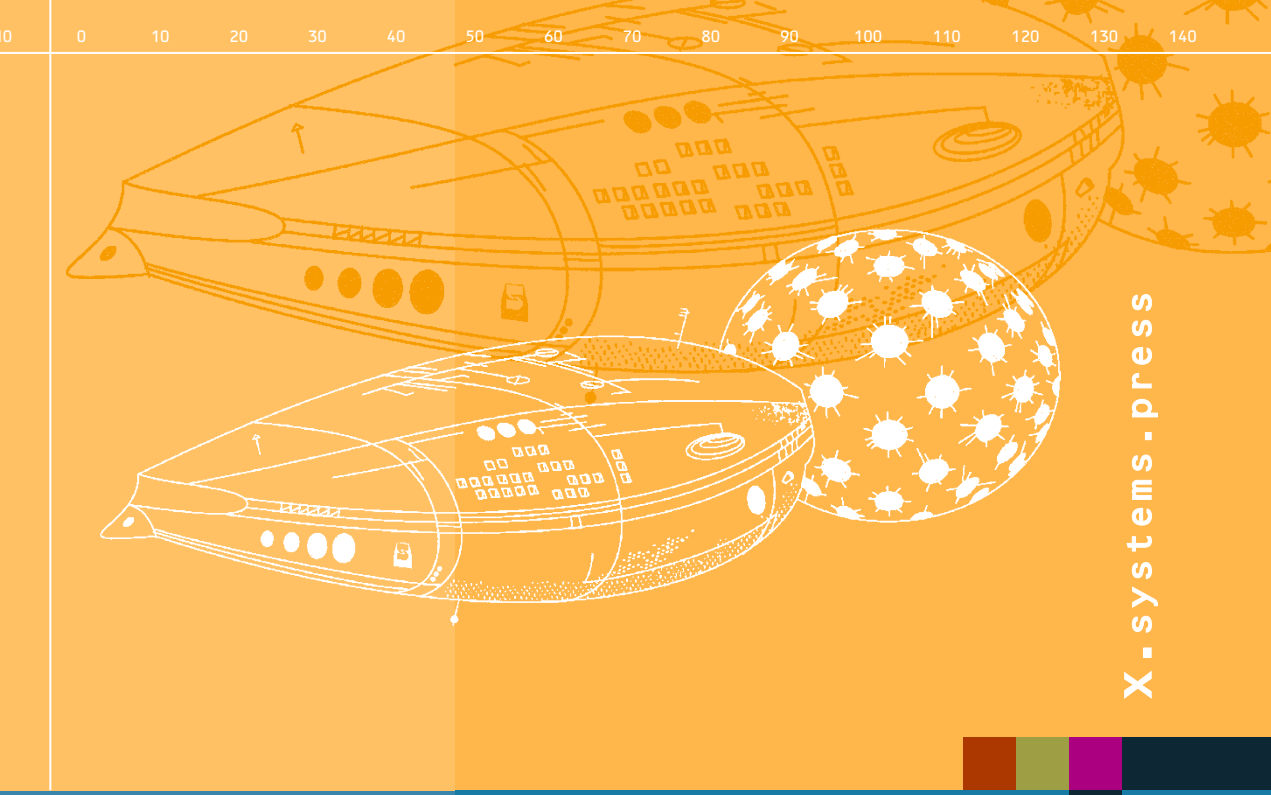

X.systems.press

Peter Monadjemi

Windows Server-Administration mit PowerShell 5.1

Eine kompakte und praxisorientierte Einführung

EBOOK INSIDE

Springer Vieweg

X.systems.press ist eine praxisorientierte Reihe zur Entwicklung und Administration von Betriebssystemen, Netzwerken und Datenbanken.

Weitere Bände in dieser Reihe
http://www.springer.com/series/5189

Peter Monadjemi

Windows Server-Administration mit PowerShell 5.1

Eine kompakte und praxisorientierte Einführung

Springer Vieweg

Peter Monadjemi
Esslingen, Deutschland

ISSN 1611-8618 ISSN 2363-9059 (electronic)
X.systems.press
ISBN 978-3-658-17665-5 ISBN 978-3-658-17666-2 (eBook)
DOI 10.1007/978-3-658-17666-2

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Springer Vieweg

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Vieweg ist Teil von Springer Nature
Die eingetragene Gesellschaft ist Springer Fachmedien Wiesbaden GmbH
Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Vorwort

Der Satz „Veränderungen machen auch vor den IT-Abteilungen nicht Halt“ mag zunächst widersinnig erscheinen, steht doch die IT-Branche selber wie kaum eine andere Branche für die Veränderung schlechthin. Doch bei etwas genauerer Betrachtung ergibt sich ein differenzierteres Bild. In vielen großen Organisationen dominieren statische Prozesse, eine strikte Aufgaben- und Kompetenzaufteilung und damit verbunden auch Formalismen, Bürokratie und eine gewisse „Das haben wir immer schon so gemacht“-Mentalität. Es versteht sich von selber, dass diese Strukturen für die modernen Herausforderungen im Cloud-Zeitalter nicht optimal sind. Eine dieser Herausforderungen besteht darin, dass die in den letzten zehn Jahren und länger im Bereich der Software-Entwicklung entstandene Agilität nicht durch starre Strukturen in der IT-Abteilung ausgebremst werden soll.

Das exorbitante Wachstum der Internet-Giganten, allen voran Amazon, Facebook, Twitter, aber auch viele nicht ganz so in der Öffentlichkeit bekannte Größen wie Flickr, Booking.com, Ueber und viele, viele mehr sind nicht nur in kurzer Zeit gewachsen, sondern haben zwangsläufig ihre eigenen IT-Prozesse definiert. In diesen Prozessen fließen Entwicklung und Bereitstellung zwangsläufig zusammen und erfordern ein Umdenken. Auch Microsoft hat einen enormen Aufwand, verbunden mit einer tiefgreifenden internen Umstrukturierung, betrieben, um den Anschluss nicht zu verlieren, und die Ergebnisse in Gestalt neuer Cloud-Produkte, allen voran Office 365, und der Aktienkurs scheinen zu bestätigen, dass man auf dem richtigen Weg ist und die Dominanz auf dem Desktop in der Cloud behaupten kann.

Um das Zusammenwachsen von „Development“ und „Operations“ zu beschreiben, wurde bereits vor vielen Jahren der Begriff „DevOps“ geprägt. Anders als man es vermuten könnte, entstand dieser Begriff weder in der Marketingabteilung eines Internet-Giganten, noch stammt er aus der Doktorarbeit eines Informatikers. Der Begriff entstand, ganz unspektakulär, eher zufällig, als eine belgische Usergroup einen griffigen Namen für eine geplante Veranstaltungsserie suchte.

Auch wenn ich den Begriff „DevOps“ nicht überbewerten möchte, steht das Kürzel doch stellvertretend für den grundlegenden Wandel, der in der IT-Branche seit einigen Jahren im vollen Gange ist, und der noch längst nicht abgeschlossen ist (dazu ist die weltweite IT-Landschaft auch zu groß). Es ist auch nicht so, dass DevOps für alle IT-Abteilungen

einen nicht zu vermeidenden Paradigmenwechsel bedeutet. Es hängt sehr stark davon ab, ob zum Beispiel das Thema Bereitstellen von (Web-)Anwendungen überhaupt ein Thema ist. Der kleinste gemeinsame Nenner von DevOps steht einfach für mehr Agilität, schlankere Prozesse (das klingt natürlich immer gut) und damit verbunden für die Aufgabe starrer Strukturen.

Alles schön und gut, doch welche Rolle spielt die PowerShell? Die PowerShell ist natürlich kein „DevOps"-Tool, das es genauso wenig gibt wie ein „TCO-Tool", mit dem sich die „Total Cost of Ownership" reduzieren lässt.

Die mit der Version 4.0 eingeführte *Desired State Configuration* (DSC) ist eine Technik, die einen zentralen Aspekt von DevOps adressiert: Das schnelle Bereitstellen von Konfigurationsänderungen. Das moderne Schlagwort lautet „Infrastruktur als Code", also die textuelle Beschreibung von herzustellenden Zuständen in einer modernen IT-Infrastruktur. Insbesondere im Zusammenspiel mit den Cloud-Plattformen von Microsoft aber auch Amazon lassen sich agile Prozesse wie das Bereitstellen von Anwendungen und Konfigurationsänderungen flexibel anwenden. DSC wurde mit den Versionen 5.0 und 5.1 weiterentwickelt und steht auch im Rahmen der Cloud-Plattform Azure zur Verfügung. DSC ist „work in progress". Microsoft plant für die kommenden Versionen wichtige Verbesserungen beim *Local Configuration Manager* (LCM), dem Dreh- und Angelpunkt von DSC, die zu dem Zeitpunkt als dieses Buch entstand, aber noch nicht angekündigt waren.

Open Source und PowerShell „on every System"

Am 16. August 2016 wurde vom PowerShell-Team die vielleicht wichtigste Ankündigung in der inzwischen zehn Jahre dauernden Geschichte der PowerShell gemacht. An diesem Tag wurde aus der PowerShell ein Open Source-Projekt. Zwar bezieht sich diese Ankündigung „nur" auf die PowerShell Core Edition, die in der Version 6.0 zu diesem Zeitpunkt lediglich in einer frühen Alpha-Version vorliegt und voraussichtlich irgendwann in naher Zukunft offiziell werden wird, dennoch markiert diese Ankündigung einen wichtigen Richtungswechsel. In naher Zukunft wird es die PowerShell in der Core Edition nicht nur für Windows, Mac OS X und die wichtigsten Linux-Distributionen geben, sondern für jede Plattform, auf der sich die Laufzeitumgebung .NET Core implementieren lässt.

Was lesen Sie in diesem Buch?

Eine „Warnung" gleich vorweg: Dieses Buch ist kein typisches PowerShell-Einsteigerbuch. Falls Sie als Administrator oder IT-Pro die PowerShell zunächst kennenlernen möchten, ist dieses Buch vermutlich keine gute Wahl, da es bereits Grundkenntnisse im Umgang mit der PowerShell voraussetzt. Begriffe wie Cmdlet, Parameter, Pipeline, Functions und vor

allem Objekte sollten beim Lesen einen vertrauten Klang besitzen und keine virtuellen Fragezeichen auslösen. Ich möchte mit diesem Buch eine Lücke schließen zwischen der umfangreichen Einstiegsliteratur, in der die Grundlagen der PowerShell leicht verständlich erklärt werden, und den Blog-Einträgen der PowerShell-Experten, in denen in der Regel Techniken verwendet werden, die scheinbar „nirgendwo“ erklärt werden (vieles findet man in der umfangreichen PowerShell-Hilfe und den weit über 100-About-Themen, doch seien wir einmal ehrlich, wer liest gerne die Hilfe?). Auf insgesamt 19 Kapiteln dieses Buches erhalten Sie einen Querschnitt über die wichtigsten Themen, die in die Kategorie Grundlagen und Praxis für den etwas fortgeschrittenen Einsatz der PowerShell fallen. Dass das Buch mit dem Kapitel „Spaß mit der PowerShell“ endet, geschah nicht ohne Grund. Die PowerShell ist nicht nur ein Werkzeug für die tägliche Arbeit, sondern auch eine universelle Skriptsprache in der Tradition von Perl, Python, Bash usw., mit der sich einfach eine Menge machen lässt. Und wer möchte, kann mit ihr auch Spaß haben.

Die Beispiele zu diesem Kapitel

Einige Beispiele, die ich in diesem Buch vorstelle, sind aus Platzgründen nicht vollständig abgebildet. Sie erhalten alle Beispiele als Download. Die Adresse finden Sie in meinem Blog http://poshadmin.de. Sollten Sie dort wider Erwarten nicht fündig werden, schicken Sie mir bitte eine E-Mail und schreiben Sie einen Eintrag in das Forum.

Kontakt zum Autor

Ich freue mich über Lob, Kritik und Fragen zum Buch oder allgemein zur PowerShell. Sie erreichen mich per E-Mail unter pm@activetraining.de, per Twitter (@pemo09) oder über die Kommentarfunktion meines Blog http://poshadmin.de.

Danksagungen

Ich möchte mich bei meiner Lektorin, Frau Dr. Kathke vom Springer-Vieweg-Verlag, für die wirklich gute Zusammenarbeit bedanken (ursprünglich hätte das Buch natürlich deutlich eher erscheinen sollen). Die persönliche Widmung ist für meine Frau Andrea in tiefer Dankbarkeit für die vergangenen 13 Jahre. Ohne ihre Unterstützung und ihre unglaubliche Fähigkeit, in jedem Chaos eine Struktur zu erkennen und in verfahrenen Situationen einen klaren Kopf zu bewahren, wäre aus diesem Buch nichts geworden.

Peter Monadjemi
Esslingen, Mai 2017

Inhaltsverzeichnis

PowerShell für Kurzentschlossene

1

Zusammenfassung

Dieses Kapitel gibt eine kompakte Einführung in die Grundlagen der PowerShell für alle Leser, die die PowerShell zwar kennen, aber mit diesen Grundlagen noch nicht in allen Details vertraut sind. Außerdem werden in diesem Kapitel die wichtigsten Begriffe vorgestellt, die den theoretischen Unterbau der PowerShell-Infrastruktur betreffen.

1.1 Das WMF im Überblick

Das *Windows Management Framework* (WMF) ist der Überbau, der nicht nur die beiden PowerShell-Hostanwendungen PowerShell-Konsole und PowerShell ISE, sondern auch eine Reihe von Komponenten (in Gestalt von Assembly-Dateien) umfasst, die für die Ausführung von PowerShell-Funktionalitäten eine Rolle spielen. Die aktuelle Version ist WMF 5.1. Wenn diese Version nicht bereits Teil des Betriebssystems ist, wie bei Windows Server 2016 und Windows 10 Anniversary Update (Version 1607), kann es als Update nachträglich installiert werden.[1] Da sich Microsoft dazu entschieden hat, das WMF für alle aktuellen Windows-Versionen anzubieten, ist es kein Problem, WMF 5.1 unter Windows Server 2008 R2 und Windows 7 zu installieren. Im Unterschied zur Installation von WMF 5.0 ist es bei einem Update von den Versionen 2.0 und 3.0 nicht erforderlich, dass zuerst WMF 4.0 als Zwischenschritt installiert wird. Tab. 1.1 stellt die Bestandteile von WMF 5.1 zusammen. Abb. 1.1 zeigt die Bestandteile des WMF in einem Schaubild.

[1] Auch wenn sich solche Downloadlinks mit der favorisierten Suchmaschine im Allgemeinen schnell finden lassen und Erweiterungen in der Regel über die PowerShell Gallery hinzugefügt werden, habe ich in meinem Blog http://poshadmin.de die Links der wichtigsten PowerShell-Downloads zusammengestellt.

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_1

Tab. 1.1 Die Komponenten des Windows Management Frameworks

Komponente	Bedeutung
Windows-PowerShell	Zwei Host-Anwendungen für die interaktive Eingabe von Befehlen und das Ausführen von Skripten.
DSC	Über die *Desired State Configuration* ist das Übertragen von Konfigurationsänderungen auf der Grundlage einer Beschreibungssprache möglich.
WinRM	Ermöglicht das Ausführen von Befehlen auf anderen Computern im Netzwerk auf der Grundlage von Ws-Management.
Managed WMI	Implementierung von WMI/CIM auf der Grundlage der .NET-Laufzeit.
PowerShell Webdienste	Ermöglicht das Ausführen von Cmdlets über REST-basierte Webservice-Aufrufen im Zusammenspiel mit der IIS-Erweiterung und Management OData.
SIL	*Software Inventory Logging*. SIL wurde mit Windows Server 2012 R2 vorgestellt.
CIM-Provider	Ein CIM/WMI-Provider umfasst die Definitionen von CIM-Klassen in Gestalt von MOF-Dateien. Traditionell basieren Provider auf COM-Komponenten. WMF ermöglicht es, dass ein CIM-Provider auch in der Programmiersprache C umgesetzt werden kann. Damit wird der Aufwand für das Erstellen eines Providers für die Hersteller von Hardwarekomponenten deutlich reduziert und die Provider werden plattformunabhängig.

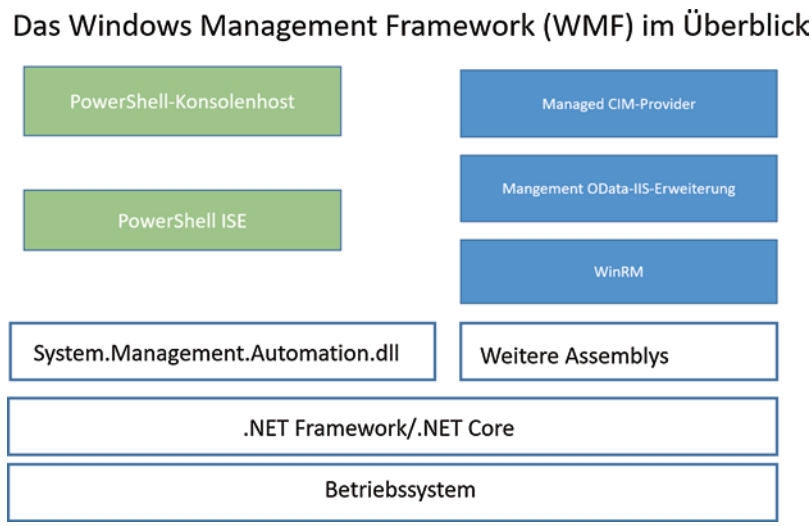

Abb. 1.1 Das WMF im Überblick

▶ **Tipp** Wie sich WMF 5.1 per SCCM verteilen lässt, wird in einem Blog-Eintrag von *Anders Rodland* anschaulich beschrieben: https://www.andersrodland.com/deploy-windows-management-framework-51-with-sccm.

1.2 Die .NET-Laufzeit

Die .NET-Laufzeit, auch .NET Framework oder einfach nur .Net (ausgesprochen als „dot-net"), ist eine Laufzeitumgebung für Anwendungen und der Unterbau der PowerShell. Sie ist nicht Teil des WMF, sondern muss separat installiert werden. Windows besitzt ab Windows Server 2008 R2 und Windows 7 zwar von Anfang an die .NET-Laufzeit, oft aber nicht in der passenden Version. WMF 5.1 setzt bereits .NET 4.6.1 voraus. Die .NET-Laufzeit ist nicht für die Ausführung der PowerShell-Assemblys zuständig, sondern bietet eine Fülle von Funktionalitäten, die natürlich direkt in der PowerShell-Konsole oder in einem Skript verwendet werden können. Dazu gleich eine kleine Kostprobe. Der folgende Befehl gibt über einen true/false-Wert an, ob der PowerShell-Host als Administrator gestartet wurde:

```
[System.Security.Principal.WindowsIdentity]::GetCurrent().Groups.Value -
Contains "S-1-5-32-544"
```

Keine Lust, ein solches „Befehlsmonster" abzutippen? Kein Problem. Zum einen gibt es in der PowerShell-Konsole auch für solche Befehle eine Eingabevervollständigung per [Tab]-Taste, so dass das Eintippen relativ schnell geht. Zum anderen gibt es Erweiterungen wie das *Carbon*-Modul, das bereits fertige Befehle dafür enthält. Wo gibt es das *Carbon*-Modul? Zum Beispiel in der PowerShell Gallery. Und wie erhalte ich das Modul? Ganz einfach über das *Install-Module*-Cmdlet der PowerShell.

Der folgende Befehl fügt das *Carbon*-Modul von der PowerShell Gallery hinzu:

```
Install-Module -Name Carbon -Scope CurrentUser -Force
```

Der *Scope*-Parameter sorgt dafür, dass das Modulverzeichnis nicht im Programm-Verzeichnis (dazu müsste die PowerShell als Administrator gestartet werden), sondern im Dokumente-Verzeichnis abgelegt wird. Anschließend steht der Befehl „Test-AdminPrivilege" zur Verfügung, der einen *true*-Wert zurückgibt, wenn die PowerShell-Hostanwendung mit Administratorberechtigungen gestartet wurde.

1.2.1 Assemblys

Der Begriff Assembly (engl. für Versammlung) steht bei der .NET-Laufzeit lediglich für eine Datei, die „Managed Code" enthält. Managed Code wiederum ist ein Befehlscode, der von der *Common Language Runtime* (CLR) ausgeführt wird. Die CLR ist ein

Kernbestandteil der .NET-Laufzeit. Die PowerShell besteht aus einer Reihe solcher Assembly-Dateien. Die wichtigste ist *System.Management.Automation.dll*. Diese Dateien sind keine abstrakten Größen, sondern liegen auf der Festplatte des Computers. Der folgende PowerShell-Befehl gibt die Pfade der aktuell geladenen Assembly-Dateien aus:

```
[AppDomain]::GetAssemblies().CurrentDomain.Location
```

Eine Assembly(-Datei) enthält in der Regel eine Vielzahl von Typendefinitionen, in der Regel in Gestalt von Klassendefinitionen. Jede Klasse definiert die Members eines Objekts. Jedes PowerShell-Cmdlet basiert auf einer Klassendefinition. Die Klassendefinition selber lässt sich nicht so ohne weiteres ausgeben (dazu wird ein sogenannter IL-Disassembler wie zum Beispiel das Programm *ILSpy* benötigt), wohl aber die Members und die genaue Bezeichnung der Klassendefinition.

Der folgende Befehl gibt den Namen der Klassendefinition und die Members jener Klasse aus, auf der das *Get-Command*-Cmdlet basiert:

```
Get-Command –Name Get-Command | Get-Member
```

Zuerst holt das *Get-Command*-Cmdlet das *Get-Command*-Cmdlet als Objekt (daher kommt „Get-Command“ zwei Mal vor). Das daraus resultierende CmdletInfo-Objekt wird per Pipe-Operator dem *Get-Member*-Cmdlet übergeben, das dann die Members des Objekts ausgibt.

Die besondere Bedeutung von Assemblys für die PowerShell besteht darin, dass eine Assembly-Datei in der Regel viele Definitionen von Klassen, Schnittstellen und Konstantenlisten enthält, die allgemein unter dem Betriff „Typen“ zusammengefasst werden.

Es ist faszinierend, wie einfach sich zum Beispiel die Klassendefinitionen einer Assembly auflisten lassen. Dafür ist die unscheinbare Eigenschaft *Assembly* zuständig, die es bei jedem Typobjekt gibt. Sie liefert einen Verweis auf jenes Objekt, das die Assembly-Datei repräsentiert, in der der Typ definiert ist.

Beispiel

Der folgende Befehl gibt die Namen aller öffentlichen Klassen aus, die in der PowerShell-Assembly *System.Management.Automation.dll* enthalten sind:

```
[PSObject].Assembly.GetTypes() | Where { $_.IsPublic -and $_.IsClass } |
Select Namespace, Name
```

PSObject ist der Name einer Klasse, die in *System.Management.Automation.dll* definiert ist. Würden Sie stattdessen „Object“ schreiben, würde die *Assembly*-Eigenschaft eine andere Assembly ansprechen und eine ganz andere Ausgabe wäre die Folge. Eine kleine Änderung mit großer Wirkung. Mehr über den Umgang mit Assemblys erfahren Sie im Anhang dieses Buches, in dem die PowerShell aus der Perspektive eines Software-Entwicklers vorgestellt wird.

1.3 Die Objekt-Pipeline

Das sicherlich wichtigste Unterscheidungsmerkmal zwischen der PowerShell und anderen Shells wie der *Bash*, die seit *Windows 10 Anniversary Update* ein optionaler Bestandteil des Betriebssystems ist, dem *Windows Scripting Host* oder den guten alten Stapeldateien ist der Umstand, dass bei der PowerShell ausschließlich Objekte über die Pipeline übertragen werden. Ein Objekt beschreibt einen Gegenstand wie einen Prozess oder einen Systemdienst und stellt die Details über diesen Gegenstand über Namen zur Verfügung, die allgemein Properties (engl. für Eigenschaften) heißen. Eine Property ist nicht nur ein Name, sondern immer mit einem Datentyp verbunden, so dass die Information darüber, wie der Wert behandelt werden muss, in die Eigenschaft eingebaut ist. Das bedeutet konkret, dass die Weiterverarbeitung einer Ausgabe sehr einfach wird. Sollen zum Beispiel die laufenden VMs nach ihrer Laufzeit sortiert werden, erledigt dies eine Kombination aus *Get-VM* und *Sort-Object*:

```
Get-VM | Sort-Object Uptime
```

„Uptime“ ist der Name einer der zahlreichen Eigenschaften, die ein Objekt besitzt, das von *Get-VM* in die Pipeline gelegt wird, und die für die bisherige „Uptime“ der VM steht. Da dieser Wert nicht einfach eine Zahlfolge oder eine Zeichenkette, sondern selber ein Objekt vom Typ *TimeSpan* ist, wird der Wert jedes einzelnen Objekts so interpretiert, dass daraus automatisch die richtige Reihenfolge resultiert.

Während in diesem Fall ein Wert wie 11:45:00, der für eine Laufzeit von 11 Stunden, 45 Minuten und 0 Sekunden steht, auch als Zeichenfolge passend interpretiert werden würde, sieht dies bei einem Datumszeitwert etwas anders aus. Hier ist der Wert „1/1/2017“ immer dann kleiner als der Wert „2/1/2007“, wenn beide Werte als Zeichenfolge verglichen würden. Da die Eigenschaft *StartTime* bei einem Prozessobjekt vom Typ *DateTime* ist, kann auch hier das *Sort-Object*-Cmdlet den Inhalt der Pipeline so sortieren wie es einem Datumszeitwert entspricht:

```
Get-Process | Sort-Object StartTime
```

Um es salopp zu formulieren: Dem *Sort-Object* muss nicht gesagt werden, wie es die Werte einer Eigenschaft sortieren muss. Es erfährt dies aus dem Typ der jeweiligen Eigenschaft, in dem die Sortierfähigkeit „eingebaut“ ist. Wäre die Rückgabe eines Cmdlets lediglich Text, wäre ein Sortieren der Ausgabe deutlich aufwändiger. Abb. 1.2 veranschaulicht das Prinzip der Objekt-Pipeline.

Nicht nur das Sortieren profitiert davon, dass über die Pipeline Objekte übergeben werden. Jede Operation wird deutlich einfacher oder überhaupt erst mit vertretbarem Aufwand möglich. Sollen die Eckdaten aller Prozesse, die aktuell mehr als 100MB im Arbeitsspeicher belegen, sortiert nach dem Wert der Arbeitsspeicherbelegung im HTML-Format ausgegeben werden, erledigt das der folgende Befehl:

```
Get-Process | Where-Object WS -gt 100MB | Sort-Object WS -Descending |
Select-Object Name, WS, StartTime | ConvertTo-Html
```

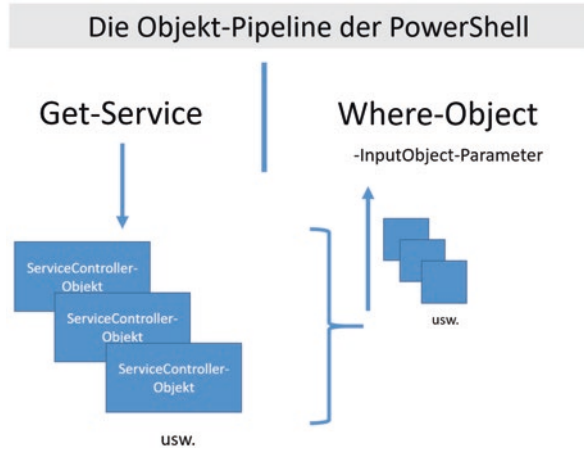

Abb. 1.2 Das Prinzip der Objekt-Pipeline – die Ausgabe eines Cmdlets wird in Gestalt von Objekten an den Parameter eines anderen Cmdlets gebunden

Soll der Output in einer Textdatei gespeichert werden, muss die Ausgabe entweder per > umgeleitet oder per *Out-File*-Cmdlet gespeichert werden:

```
Get-Process | Where-Object WS -gt 100MB | Sort-Object WS -Descending |
Select-Object Name, WS, StartTime | ConvertTo-Html > Prozesse.htm
```

1.4 PSProvider und PSDrives

Ein weitere Innovation bei der PowerShell war bei ihrer Einführung im Jahr 2006 der Umstand, dass der Laufwerksbegriff neu definiert wurde. Ein PSDrive ist bei der PowerShell ein virtuelles Laufwerk, über das beliebige Ablagen einheitlich, also mit demselben Satz an Cmdlets, angesprochen werden. Das *Get-PSDrive*-Cmdlet listet alle Laufwerke auf. Neben den Dateisystemlaufwerken gibt es unter anderem die PSDrives *Function* für die PowerShell-Functions, *Variable* für die PowerShell-Variablen, *env* für die Umgebungsvariablen des PowerShell-Prozesses und *cert* für die lokalen Zertifikate. Damit listet ein „dir C:\" das Stammverzeichnis von Laufwerk C:, ein „dir env:" die Umgebungsvariablen des Prozesses auf. Ein „Get-Content – Path C:\Windows\Win.ini" gibt den Inhalt der angesprochenen Datei, ein „Get-Content – Path Function:Get-FileHash" den Inhalt der angesprochenen Function-Definition aus. Was genau geholt wird, legen stets der *Path*-Parameter des Cmdlets fest und das Laufwerk, das über den Pfad angesprochen wird.

Alle PSDrive-Laufwerke werden durch PSProvider zur Verfügung gestellt. Das *Get-PSProvider*-Cmdlet listet die aktuell geladenen PSProvider auf. Über das Importieren von Modulen kommen weitere PSProvider hinzu. Ein Beispiel ist das *ActiveDirectory*-Modul, das den gleichnamigen PSProvider lädt, der ein Laufwerk mit dem Namen „AD" hinzufügt. Über das AD-Laufwerk kann das Active Directory-Verzeichnis wie ein Laufwerk angesprochen werden. Ein „dir AD: -Recurse" gibt den gesamten Inhalt des Verzeichnisses aus.

1.4.1 Dynamische Parameter

Da ein Cmdlet wie *Get-ChildItem* für jeden Laufwerkstyp passen muss, umfasst seine Parameterliste offiziell nur Parameter, die für sämtliche Laufwerkstypen geeignet sind. Das bedeutet im Umkehrschluss, dass es Parameter gibt, die nur auf bestimmte Laufwerkstypen angewendet werden können. Ob zum Beispiel die Parameter *Filter* oder *Recurse* bei einem Laufwerk verwendet werden können, hängt davon ab, ob diese Option durch den PSProvider, der das Laufwerk zur Verfügung stellt, implementiert wurde.

Damit Parameter in Abhängigkeit des Laufwerkstyps zur Verfügung stehen und z. B. das *Get-ChildItem*-Cmdlet je nach Laufwerkstyp, der über den *Path*-Parameter angegeben wird, einen *Recurse*-Parameter besitzt oder nicht, gibt es die dynamischen Parameter. Die dynamischen Parameter sind Teil des Providers und stehen immer dann zur Verfügung, wenn über den *Path*-Parameter des Cmdlets ein Laufwerk angesprochen wird, das von diesem PSProvider zur Verfügung gestellt wird. Ein Beispiel ist der Parameter *Directory*, den es beim *Get-ChildItem*-Cmdlet nur dann gibt, wenn ein hierarchisches Laufwerk angesprochen wird. Den Parameter *CodeSigningCert* gibt es nur dann, wenn über den *Path*-Parameter das *Cert*-Laufwerk angesprochen wird.

In der PowerShell-Hilfe werden die dynamischen Parameter nicht in der Hilfe zu dem jeweiligen Cmdlet, sondern in jeder Hilfe zu dem jeweiligen Provider beschrieben. Ein „Help Certificate“ listet zum Beispiel die Beschreibung des *Cert*-Providers auf, in der auch alle dynamischen Parameter enthalten sind.

▶ **Tipp** Eine Übersicht über alle zur Auswahl stehenden Parameter liefert die Tastenkombination [Strg]+[Leertaste] (sofern das *PSReadline*-Modul geladen ist). Diese richtet sich danach, welches Laufwerk über den *Path*-Parameter ausgewählt wurde. Abb. 1.3 zeigt die Parameterausgabe beim *Get-ChildItem*-Cmdlet, wenn über den *Path*-Parameter des Cmdlets das Cert-Laufwerk angesprochen wird.

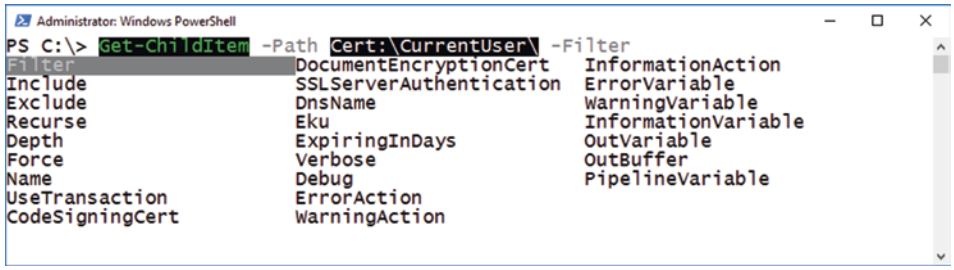

Abb. 1.3 Dank PSReadLine werden alle Parameter eines Cmdlets inklusive der dynamischen Parameter übersichtlich angezeigt

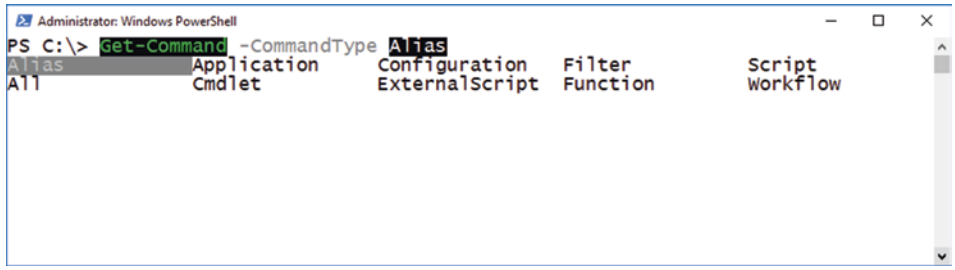

Abb. 1.4 Dank PSReadline werden auch die Parameterwerte eines Parameters übersichtlich angezeigt

1.5 Functions, Aliase, Workflows und Configurations

Cmdlets sind nicht der einzige Typ von Kommandos, der bei der PowerShell zur Auswahl steht. Darüber hinaus gibt es Functions, Aliase, Workflows, Skripte und Konfigurationen. Das *Get-Command*-Cmdlet zeigt per Voreinstellung nur Cmdlets, Aliase und Functions an. Möchte man weitere oder alle Commandtypen sehen, gibt es dafür den Parameter *CommandType*, dem die Namen der Commandtypen per Komma getrennt übergeben werden. In der Regel sind dies *Alias*, *Cmdlet*, *Function* und *Application*. Die Ausgabe kann dabei so umfangreich werden, dass in der Konsole die Puffertiefe auf den Maximalwert 9999 gesetzt werden muss.[2] Der Commandtyp *Application* gibt alle Programmdateien zurück, die sich in einem der Verzeichnisse befinden, die Teil der *Path*-Umgebungsvariablen sind. Dazu zählen nicht nur Exe-Dateien, sondern auch Dateien mit den Erweiterungen *.Cmd* und *.Bat*.

▶ **Tipp** Auch bei der Anzeige der möglichen Werte für einen Parameter bietet das *PSReadline*-Modul eine praktische Eingabehilfe an. Nach der Eingabe des Parameternamens und eines Leerzeichens zeigt ein [Strg]+[Leerzeichen] die für den Parameter zur Auswahl stehenden Werte an. Abb. 1.4 zeigt eine *PsReadline*-Ausgabe für den *CommandType*-Parameter von *Get-Command*.

1.6 Erweiterbarkeit

Ein weiterer Pluspunkt der PowerShell ist ihre Erweiterbarkeit. Eine Erweiterung ist entweder ein Snap-In oder ein Modul. Snap-Ins spielen als Erweiterungen nur noch in Ausnahmefällen eine Rolle. Eine dieser Ausnahmen sind die Cmdlets aus dem *Xen App SDK* von *Citrix*. Es wird noch weitere Ausnahmen geben. Die typische PowerShell-Erweiterung liegt in Gestalt eines Moduls vor. Ein Modul basiert auf einem Verzeichnis, das eine oder

[2] Auf meinem Windows 10-PC sind es 8079 Commands. Wer bietet mehr?

mehrere PowerShell-Dateien enthält. Damit ein Verzeichnis als Modulverzeichnis erkannt wird, muss es eine Psd1-, Psm1- oder eine Dll-Datei enthalten, deren Name dem Modulverzeichnis entspricht. Spielt die Versionsnummer des Moduls eine Rolle, enthält das Modulverzeichnis auf der obersten Ebene ein Verzeichnis mit der Versionsnummer als Name, in dem sich die Psd1- oder Psm1-Datei befindet. Damit können mehrere Versionen eines Moduls parallel vorliegen. Alle „offiziellen" Modulverzeichnisse sind in der Umgebungsvariable *PSModulePath* enthalten. Dazu zählen *$PSHome\Modules*, *$Home\Documents\WindowsPowerShell\Modules* und *$env:ProgramFiles\WindowsPowerShell\Modules*. Auf einem 64-Bit-Windows kommt *$env:ProgramFiles(x86)\WindowsPowerShell\Modules* als weiteres Modulverzeichnis hinzu, in dem sich die Modulverzeichnisse für 32-Bit-PowerShell-Hosts befinden. Grundsätzlich spielt es keine Rolle, in welchem Verzeichnis sich ein Modulverzeichnis befindet. Modulverzeichnisse, die in keinem der offiziellen Modulverzeichnisse liegen, müssen per *Import-Module* direkt geladen werden. Alle anderen Module werden dadurch geladen, dass ein Command (Cmdlet oder Function) ausgeführt wird, das in dem Modul definiert wird. Die PowerShell legt im Benutzerprofil einen Cache an, in dem die Namen aller Commands in allen verfügbaren Modulen enthalten sind, so dass ein Modul relativ schnell nach Eingabe eines Commandnamens lokalisiert wird.

Beispiel

Der folgende Befehl lädt das Modul *PsKurs*, das sich im Verzeichnis *C:\MeineModule* befindet:

```
Import-Module –Name C:\MeineModule\PsKurs
```

Die Psd1- oder Psm1-Datei muss beziehungsweise soll nicht explizit angegeben werden. Befinden sich in dem Verzeichnis mehrere Versionen des Moduls, wird automatisch die aktuellste Version geladen. Ansonsten muss die Version über den Parameter *RequiredVersion* explizit angefordert werden.

Beispiel

Der folgende Befehl importiert die Version 1.1 des Moduls *PsKurs*:

```
Import-Module –Name C:\MeineModule\PsKurs –RequiredVersion 1.1
```

Ein im Praxisalltag nicht selten vorkommendes Missverständnis besteht darin, dass davon ausgegangen wird, dass ein *Get-Module* alle verfügbaren Module auflistet. Das ist aber nicht der Fall. Das *Get-Module*-Cmdlet zeigt nur die aktuell geladenen Module an. Möchte man alle verfügbaren Module sehen, muss der Parameter *ListAvailable* angehängt werden. Auch hier kann ein Missverständnis auftreten. Da die Module nach Verzeichnissen gruppiert ausgegeben werden, kann es passieren, dass man ein Modul nicht entdeckt, da es sich im Benutzerprofilverzeichnis befindet und daher als erstes ausgegeben wurde.

Möchte man die Modulliste etwas übersichtlicher erhalten, muss ein *Select-Object* angehängt werden:

```
Get-Module -ListAvailable | Select Name, Version, ModuleType | Sort-
Object Name
```

Was auch nicht jeder erfahrenere PowerShell-Anwender weiß: Die Ausgabe in der Konsole ist nicht in Stein gemeißelt, über das *Out-GridView*-Cmdlet kann jede Ausgabe in einem Fenster angezeigt werden:

```
Get-Module -ListAvailable | Select Name, Version, ModuleType | Sort-
Object Name | Out-GridView
```

Jetzt sieht die Ausgabe doch gleich etwas freundlicher aus.

Und es gibt noch einen weiteren Grund für Missverständnisse. Unter einer 32-Bit-PowerShell werden unter einem 64-Bit-Windows nur die Verzeichnisse in *C:\Program Files(86)\WindowsPowerShell\Modules* berücksichtigt, nicht jene, die sich in *C:\ProgramFiles\WindowsPowerShell\Modules* befinden und umgekehrt.

Was macht *Get-InstalledModule*? Die Function aus dem *PowerShellGet*-Modul listet nur jene Module auf, die über das *Install-Module*-Cmdlet hinzugefügt wurden.

1.6.1 Ein Modulmanager in Gestalt des PowerShellGet-Moduls

Die Zeiten, in denen ein Modul als Zip-Datei von der Webseite des Modulautors heruntergeladen, ausgepackt und der Inhalt in ein Verzeichnis kopiert werden musste, sind seit der Version 5.0 vorbei. Seit dieser Version umfasst die PowerShell offiziell einen „Paketmanager", der auch das Laden von Modulen übernimmt.[3] Die Befehle für die Modulverwaltung enthält das Modul PowerShellGet. Für das Aufspüren von Modulen gibt es die Function *Find-Module*. Sie durchsucht die von Microsoft betriebene PowerShell Gallery (PSGallery) unter http://powershellgallery.com.

Beispiel

Der folgende Befehl gibt alle Module aus, in deren Namen das Wort „Excel" enthalten ist:

```
Find-Module *excel*
```

Beim ersten Aufruf werden Sie aufgefordert, Provider für den NuGet-Paketmanager zu installieren, über den die Zugriffe abgewickelt werden. Per *Install-Module* wird ein Modul lokal hinzugefügt. Je nach Wert für den *Scope*-Parameter wird das Modulverzeichnis entweder unter *C:\Programfiles\WindowsPowerShell\Modules* oder unter *$Home\Documents\WindowsPowerShell\Modules* abgelegt.

[3] Für die Versionen 3.0 und 4.0 steht der Paketmanager-Manager ebenfalls zur Verfügung, aktuell (Stand: März 2017) aber immer noch in Gestalt der „Package Management Preview".

Über *Get-InstalledModule* werden alle per PowerShellGet installierten Module aufgelistet. Auch Skripte lassen sich auf diese Weise von der PoweShell Gallery laden. Ein „Find-Script *“ gibt alle verfügbaren Skripte aus, per *Install-Script* wird ein Skript lokal abgelegt.

So elegant das Hinzufügen von Modulen und Skripten über die PowerShell Gallery ist, möchte und kann nicht jeder Administrator fremden Programmcode laden. Auch wenn die Wahrscheinlichkeit sehr gering ist, kann es nicht ausgeschlossen werden, dass ein Modul oder Skript „Schadcode“ enthält. Anders als man es vermuten würde, wird ein von einem grundsätzlich anonymen Nutzer hochgeladenes Modul nicht geprüft. Es steht unmittelbar nach dem Upload zum Download zur Verfügung. Das PowerShell-Team beabsichtigt zwar die Einführung von „High Quality-Modulen“, die geprüft, versioniert und signiert sind, doch wird es eine Weile dauern bis sich dieser Modultyp verbreitet. Die Alternative zur öffentlichen PowerShell Gallery besteht aktuell darin, ein Repository selber zu hosten, entweder im Internet, zum Beispiel unter Azure, oder im Intranet. In diesem Repository werden nur ausgewählte Module abgelegt. Eine Option ist die „Private PSGallery“, die Microsoft als GitHub-Projekt zur Verfügung stellt. Mehr zu diesem Thema in Kap. 7, in dem es um das Bereitstellen von Skripten und Modulen geht.

1.7 Befehlssyntax

Die Syntax der PowerShell-Befehle wirkt auf einen Anfänger nicht gerade konsistent. Das betrifft auch die Ausgabe eines Befehls. Dabei war die Konsistenz eines der obersten Ziele bei der Planung der PowerShell gewesen. Ziel verfehlt? Das natürlich nicht. Man muss nur wissen, dass es bei der PowerShell-Syntax verschiedene Abkürzungen gibt:

- Für Namen eines Command können Aliase verwendet werden.
- Ist ein Parameter ein Positionsparameter, kann sein Name entfallen.
- Auch für Parameter gibt es Aliase (zum Beispiel „ea“ für den Parameter *ErrorAction*).
- Generell kann der Name eines Parameters so stark verkürzt werden, dass er noch eindeutig ist in Bezug auf die übrigen Parameter des Commands. Der Parameter *ForegroundColor* beim *Write-Host*-Cmdlet kann daher durch „Fore“, „Fo“ und sogar „f“ abgekürzt werden. Dass sich dadurch eine bunte Mischung unterschiedlicher Schreibweisen ergibt, wirkt bei oberflächlicher Betrachtung natürlich nicht gerade konsistent. Dahinter stecken aber einfache Namensregeln.
- Diese Besonderheit dürfte auch so mancher erfahrene PowerShell-Admin nicht kennen. Bereits seit der Version 1.0 kann bei den Get-Cmdlets das Get-Verb weggelassen werden. Ein „Service“ gibt daher die Eckdaten zu allen Systemdiensten aus, ein „localuser“ die Eckdaten zu allen lokalen Benutzerkonten. Ein „Process“ gibt allerdings nicht die Eckdaten zu allen Prozessen aus, da „Process“ ein reserviertes Wort ist.

Beispiel

Die folgenden Befehle sind identisch bezüglich ihrer Syntax:

```
Get-Process -Name Svchost | Select-Object -Property StartTime, Id, Ws
Get-Process Svchost | Select StartTime, Id, Ws
gps Svchost | Select StartTime, Id, Ws
```

Name und Property sind jeweils Positionsparameter. Sie erhalten ihren Wert daher auch aufgrund der Position eines Wertes, dem kein Parametername vorausgeht. Ob ein Parameter ein Positionsparameter ist, erfährt man aus der Hilfe zu dem Parameter. Mit anderen Worten: Auch wenn der Name eines Parameters entfallen kann, ist der Parameter trotzdem im Spiel.

Während innerhalb von Skripten Aliase aufgrund der Lesbarkeit von Befehlen nicht verwendet werden sollten, spricht (natürlich) nichts gegen die Verwendung von Aliasen in der Konsole. Im Gegenteil, je kürzer desto besser.

1.8 Die moderne Konsole

Viele Anwender, die die PowerShell unter Windows Server 2016 oder Windows 10 starten, denken sich: Toll, die Konsole ist farbig geworden und selbstverständliche Kleinigkeiten wie ein Kopieren und Einfügen von Texten über [Strg]+[C] und [Strg]+[V] sind (endlich) möglich. Sie installieren die PowerShell 5.0 oder 5.1 unter Windows Server 2012 und stellen fest: Alles sieht aus wie immer und [Strg]+[V] fügt nichts ein. Der Grund für die auf Anhieb nicht ganz nachvollziehbare Diskrepanz ist einfach: Ab Windows Server 2016 und Windows 10 wird automatisch das Modul *PSReadline* geladen, das unter älteren Windows-Versionen zunächst hinzugefügt werden muss. *PSReadline* ist ein Modul, das den Standard-Befehlszeileneditor der Konsole durch eine neue Version, die deutlich mehr kann, ersetzt. Dazu gehören unter anderem eine Fülle von Tastaturshortcuts, die an den Unix-Editor EMACS angelehnt wurden, neue Möglichkeiten wie das Markieren der kompletten Befehlszeile über ein vertrautes [Strg]+[A], das Kopieren und Einfügen über [Strg]+[C] und [Strg]+[V], eine automatische Syntaxkontrolle während der Eingabe mit einer Fehleranzeige in Gestalt eines roten Promptzeichens und eben eine Syntaxeinfärbung. Der größte Komfortgewinn, der mit *PSReadline* einhergeht, ist der Umstand, dass sich in der Konsole auch eine mehrzeilige Eingabe komfortabel editieren lassen (siehe Abb. 1.5).

PSReadline setzt lediglich die Version 3.0 der PowerShell voraus, so dass grundsätzlich nichts dagegen sprechen sollte, es bei jeder PowerShell-Installation hinzuzufügen. Aber wie? Am einfachsten natürlich über ein *Install-Module*. Der folgende Befehl setzt voraus, dass die PowerShell-Konsole als Administrator gestartet wurde:

```
Install-Module PSReadline -Force
```

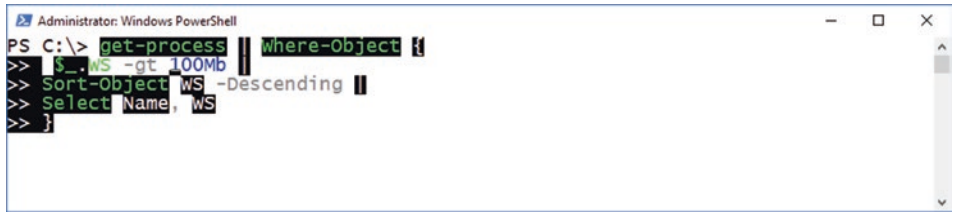

Abb. 1.5 PSReadline vereinfacht das Editieren einer mehrzeiligen Eingabe

Das Modul umfasst fünf Cmdlets. Damit lassen sich allgemeine Einstellungen ändern und Tastaturshortcuts (neu) belegen. Die aktuelle Tastaturbelegung wird über das *Get-PSReadLineKeyHandler* ausgegeben.

Beispiel

Der folgende Befehl belegt die Funktionstaste [F7] mit dem Start des Editors, der eine Profilskriptdatei lädt:

```
Set-PSReadlineKeyHandler -ScriptBlock { Notepad
$Profile.CurrentUserAllHosts } -Chord "F7"
```

PSReadline besitzt auch einen kleinen „Nachteil“. Die [F7]-Taste zeigt nicht mehr den Inhalt des Befehlspuffers der Konsole an, die Alternative ist die Eingabe von „h“ (dem Alias des *Get-History*-Cmdlets).

Oft möchte oder muss man die Farbeinstellungen ändern, da sich einzelne Vorder- und Hintergrundfarben mit der Hintergrundfarbe des Konsolenfensters „beißen“. Das ist natürlich ebenfalls kein Problem. Man muss dazu lediglich wissen, dass eine Farbe per *Set-PSReadlineOption*-Cmdlet pro Syntaxelement eingestellt werden muss und das Syntaxelement über den *TokenKind*-Parameter ausgewählt wird.

Beispiel

Der folgende Befehl setzt die Hintergrundfarbe für Operatoren auf weiß:

```
Set-PSReadlineOption -BackgroundColor White -TokenKind Operator
```

Ein solcher Befehl wird in der Regel in der Profilskriptdatei untergebracht.

1.9 Hilfe

Die Hilfe zur PowerShell muss am Anfang einmal vom Microsoft-Server über das Internet abgerufen werden. Ansonsten steht nur eine Rumpfhilfe zur Verfügung, die lediglich aus einer Syntaxbeschreibung der Cmdlets besteht. Für das Abrufen der Hilfe gibt es zwei Cmdlets: *Update-Help* und *Save-Help*. Während *Update-Help* die Hilfe in das zuständige