HANSER

Bruce Payette

Windows PowerShell im Einsatz

ISBN-10: 3-446-41239-5
ISBN-13: 978-3-446-41239-2

Leseprobe

Weitere Informationen oder Bestellungen unter
http://www.hanser.de/978-3-446-41239-2
sowie im Buchhandel

Kapitel 5

Erweiterte Operatoren und Variablen

Die größte Herausforderung für einen Denker ist es, ein Problem so zu formulieren, dass eine Lösung möglich ist.

– Bertrand Russell

Im Vorgängerkapitel haben wir die Basisoperatoren von PowerShell behandelt, in diesem Kapitel werden wir dieses Thema fortsetzen und dabei mit Operatoren Dinge vollbringen, welche die meisten Leute für unmöglich halten. Wir werden uns auch mit dem Aufbau komplexer Datenstrukturen unter Verwendung dieser Operatoren befassen. Das Kapitel schließt mit einer detaillierten Diskussion darüber, wie Variablen unter PowerShell funktionieren und wie sie zusammen mit Operatoren zur Lösung wichtiger Aufgaben eingesetzt werden können.

5.1 Operatoren für die Arbeit mit Typen

Der Typ eines Objekts ist ausschlaggebend für die Arten von Operationen die man mit diesem Objekt ausführen kann. Bis jetzt haben wir es dem Objekttyp erlaubt, die durchzuführenden Operationen *implizit* zu bestimmen. Manchmal möchten wir das aber *explizit* tun. Damit wir das können, stellt PowerShell für das Arbeiten mit Typen einige Operatoren bereit (siehe Tabelle 5.1). Diese Operatoren ermöglichen es uns zu testen, ob ein Objekt einen bestimmten Typ hat oder sie erlauben uns, ein Objekt in einen neuen Typ zu konvertieren.

Der *-is*-Operator liefert *True* falls das Objekt auf der linken Seite dem auf der rechten Seite spezifizierten Typ entspricht. Unter "is" verstehen wir, dass der linke Operand entweder vom angegebenen Typ ist oder davon abgeleitet wurde (zur Erklärung der Ableitung siehe "OOP-Auffrischkurs" in Kapitel 1, Abschnitt 1.3). Der *-isnot*-Operator liefert *True* wenn die linke Seite nicht dem Typ der rechten Seite entspricht. Die rechte Seite des Operators muss entweder einen Typ darstellen oder einen String der einen Typ benennt. Das bedeutet, dass Sie entweder ein Typliteral wie *[int]* benutzen können oder aber auch den Literalstring "int".

Der *-as*-Operator versucht, den linken Operanden in den durch den rechten Operanden spezifizierten Typ zu konvertieren. Auch hier kann entweder ein Typliteral oder ein String, welcher den Typen benennt, verwendet werden.

HINWEIS: PowerShells *-is*- und *-as*-Operatoren wurden wie die entsprechenden C#-Operatoren modelliert. Jedoch benutzt die PowerShell-Version von *-as* ein aggressiveres Typecasting. So wird z.B. unter C# der String "123" nicht in die Zahl 123 konvertiert, während PowerShell dies tut. Der *-as* Operator von PowerShell funktioniert mit jedem Typ, während der C# Operator auf Referenztypen beschränkt ist.

Möglicherweise wundern Sie sich, wozu wir den *-as*-Operator überhaupt brauchen, wenn wir doch stattdessen auch eine Typkonvertierung verwenden könnten. Der Grund ist, dass der *-as*-Operator die Verwendung eines Laufzeitausdrucks (runtime expression) erlaubt, während das Casting auf die Zeit des Parsens fixiert ist.

BEISPIEL: Anwenden des Laufzeitverhaltens:

```
PS C:\> foreach ($t in [float],[int],[string]) {"0123.45" -as $t}
123.45
123
0123.45
```

In diesem Beispiel wird mittels Schleife eine Liste von Typliteralen durchlaufen um den String in jeden der angegebenen Typen zu konvertieren. Dies aber wäre unmöglich wenn die Typen als Operatoren benutzt würden.

Schließlich gibt es da noch einen zusätzlichen Unterschied zwischen einem regulären Casting und der Verwendung des *-as*-Operators. Wenn die Konvertierung nicht erfolgreich ist, wird beim Casting ein Fehler erzeugt. Mit dem *-as*-Operator wird in diesem Fall *$null* anstatt eine Fehlermeldung zurückgegeben.

BEISPIEL: Das Casting von "abc" nach *[int]* generiert einen Fehler, aber der *-as*-Operator liefert stattdessen *$null*.

```
PS C:\> [int] "abc" -eq $null
Der Wert "abc" kann nicht in den Typ "System.Int32" konvertiert werden. Fehler: "Die Eing
abezeichenfolge hat das falsche Format."
Bei Zeile:1 Zeichen:6
+ [int]  <<<< "abc" -eq $null
PS C:\> ("abc" -as [int]) -eq $null
True
PS C:\>
```

Tabelle 5.1 zeigt verschiedene andere Beispiele zur Verwendung der Typoperatoren von PowerShell.

Operator	Beispiel	Ergebnis	Beschreibung
-is	*$true –is [bool]*	*$true*	*True* falls der Typ der linken Seite mit dem Typ der rechten Seite übereinstimmt.
	$true -is [object]	*$true*	Ist immer *True*, denn alles außer *$null* ist ein Objekt.
	$true -is [ValueType]	*$true*	Die linke Seite ist eine Instanz eines .NET-Wertetyps.
	"hi" -is [ValueType]	*$false*	Ein String ist kein Wertetyp sondern ein Referenztyp.
	"hi" –is [object]	*$true*	Aber ein String ist immer noch ein Objekt.
	12 –is [int]	*$true*	12 ist ein Integer.
	12 –is "int"	*$true*	Die rechte Seite des Operators kann entweder ein Typliteral oder ein String sein der den Typ benennt.
-isnot	*$true –isnot [string]*	*$true*	Das Objekt der linken Seite ist nicht vom gleichen Typ wie das Objekt der rechten Seite.
	$true –isnot [object]	*$true*	Der Nullwert ist das Einzige was kein Objekt ist.
-as	*"123" -as [int]*	*123*	Nimmt die linke Seite und konvertiert diese in den auf der rechten Seite spezifizierten Typ.
	123 –as "string"	*"123"*	Verwandelt die linke Seite in eine Instanz des Typs der durch den String auf der rechten Seite benannt wird.

Tabelle 5.1 PowerShells Operatoren für das Arbeiten mit Typen

In der Praxis können Sie sich die meiste Zeit auf den automatischen Mechanismus zur Typkonvertierung verlassen. Die übrigen Fälle lassen sich in der Regel mit expliziten Typecastings lösen.

Wozu also brauchen Sie dann diese Operatoren überhaupt? Meist benötigt man sie beim Schreiben von Skripten. Wenn Sie zum Beispiel ein Skript brauchen das sich bezüglich der Übergabe einer Zahl und eines Strings unterschiedlich verhält, so werden Sie den *-is* Operator zum Selektieren der auszuführenden Operation verwenden. Einfache Beispiele dafür sind die im Vorgängerkapitel beschriebenen binären Operatoren. So hat der Additionsoperator ein unterschiedliches Verhalten in Abhängigkeit vom Typ des linken Arguments. Um ein Skript zu schreiben welches dasselbe tut, müssten Sie *-is* zur Auswahl der Operation verwenden und *-as* zum Konvertieren des rechten Operanden in den korrekten Typ. Entsprechende Beispiele werden wir im Skripting-Kapitel erörtern.

5.2 Die unären Operatoren

Wir wollen nun die unären Operatoren detailliert betrachten (siehe Tabelle 5.2).

Operator	Beispiel	Ergebnis	Beschreibung
-	*- (2+2)*	-4	Negation. Versucht das Argument in eine Zahl zu konvertieren und negiert dann das Resultat.
+	*+ " "*	123	Unäres Plus. Versucht das Argument in eine Zahl zu konvertieren und gibt das Resultat zurück. Effektiv entspricht das einem Casting in eine Zahl.
--	*--$a ; $a--*	Hängt vom aktuellen Wert der Variablen ab.	Pre- und Postdekrement-Operator. Konvertiert den Inhalt der Variablen in eine Zahl und versucht 1 zu subtrahieren. Die Präfixversion liefert den neuen Wert, die Postfixversion den Originalwert.
++	*++$a; $a++*	Hängt vom aktuellen Wert der Variablen ab.	Pre- und Postinkrement. Konvertiert den Inhalt der Variablen in eine Zahl und versucht 1 zu addieren. Die Präfixversion liefert den neuen Wert, die Postfixversion den Originalwert.
[<type>]	*[int] "0x123"*	291	Typcasting. Konvertiert das Argument in eine Instanz des durch den Cast definierten Typs.
,	*, (1+2)*	1-Element Array welches den Wert des Ausdrucks enthält.	Unärer Komma-Operator. Erzeugt ein neues 1-Element Array des Typs *[object[]]* und speichert in diesem denOperanden.

Tabelle 5.2 PowerShells unäre Operatoren

Die meisten dieser Operatoren hatten wir bereits in den Vorgängerkapiteln kennen gelernt. Die unären Operatoren + und – funktionieren bei Zahlen erwartungsgemäß. Die Anwendung auf andere Typen endet mit einem Fehler. Die Verwendung von Typkonvertierungen als unäre Operatoren wurde ausgiebig im Kapitel 3 besprochen, sodass wir das hier nicht zu wiederholen brauchen. Die interessanten Operatoren des vorliegenden Abschnitts sind die für Inkrement und Dekrement. Bezüglich ihres Verhaltens entsprechen sie den äquivalenten Operatoren in C.

Noch einmal, diese Operatoren sind nur für Variablen bestimmt die Zahlen enthalten. Alles andere endet mit einem Fehler. Die Präfixform des ++ Operators inkrementiert die Variable mit 1 und liefert den neuen Wert. Die Postfixform inkrementiert die Variable mit 1, liefert aber den in der Variablen gespeicherten Originalwert. Der -- Operator tut dasselbe, nur dass er 1 subtrahiert anstatt zu addieren.

Fast wären Inkrement- und Dekrement-Operatoren nicht in PowerShell eingebunden worden, da sie ein Problem verursachten. Wenn Sie in Sprachen wie C und C# einen dieser Operatoren wie folgt benutzen:

```
$a++
```

... wird nichts angezeigt, weil diese Statements in C und C# keine Werte zurückgeben. In PowerShell hingegen liefern alle Statements einen Wert. Dies führt zur Konfusion.

BEISPIEL: Bei folgendem Skript:

```
$sum=0
$i=0
while ($i -lt 10) { $sum += $i; $i++ }
$sum
```

... wurden ursprünglich die Zahlen 1 bis 10 angezeigt, weil $a++ einen Wert zurückgab und PowerShell die Ergebnisse jedes Statements anzeigte. Dies war so verwirrend, dass wir dieses Operatoren fast aus der Sprache entfernt hätten. Dann kamen wir auf die Idee eines leeren Statements. Grundsätzlich ist darunter zu verstehen, dass bei bestimmten Typen von Ausdrücken, verwendet man sie als Statements, nichts angezeigt wird. Leere Statements betreffen Zuweisungen und die Inkrement/Dekrement-Operatoren. Verwendet man sie in einem Ausdruck, dann liefern sie einen Wert, verwendet man sie aber als alleinstehendes Statement, dann wird nichts zurückgegeben. Wieder ist das eines jener Details die zwar anders als erwartet funktionieren, die Verwendung von PowerShell aber nicht beeinträchtigen.

HINWEIS: Manchmal möchten Sie den Output eines Statements explizit verwerfen, d.h., Sie wollen ein reguläres Statement in ein leeres verwandeln. Der Weg dorthin führt über ein explizites Casting in den Typ *[void]* wie *[void] (write-object "discard me")*. Das Statement, dessen Wert Sie verwerfen wollen, ist eingeklammert und das ganze Ding wird in einen leeren Typ gecastet. Wir werden später in diesem Kapitel noch einen anderen Weg zum Erreichen desselben Effekts unter Verwendung des Umleitungs-Operators kennen lernen.

Ein Gebiet, wo leere Statements besonders interessant sind, bezieht sich auf Unterausdrücke (subexpressions) zum Gruppieren von Anweisungen. Im Folgenden werden wir uns dieser Art von Ausdrücken zuwenden.

5.3 Gruppieren, Unterausdrücke und Array-Unterausdrücke

Wir haben bereits eine Vielfalt von Situationen kennen gelernt, wo mehrere Ausdrücke oder Anweisungen miteinander gruppiert wurden. Wir haben diese Gruppenkonstrukte auch bei den in Kapitel 3 beschriebenen Stringexpansionen gesehen. Nun wollen wir sie detaillierter betrachten. Tatsächlich bietet PowerShell drei Wege zum Gruppieren von Ausdrücken (siehe Tabelle 5.3).

Operator	Beispiel	Resultate	Beschreibung
(...)	*(2 + 2) * 3* *(get-date).dayofweek*	12 Liefert den aktuellen Wochentag.	Die Klammern gruppieren Operationen mit Ausdrücken und können entweder eine simplen Ausdruck oder eine simple Pipeline enthalten.