about_Parsing - PowerShell

sdwheeler ⋮ 18-23 minutes

---

about_Parsing

- Article
- 05/06/2024
-

In this article

Short description

Describes how PowerShell parses commands.

Long description

When you enter a command at the command prompt, PowerShell breaks the command text into a series of segments called *tokens* and then determines how to interpret each token.

For example, if you type:

```
Write-Host book
```

PowerShell breaks the command into two tokens, `Write-Host` and `book`, and interprets each token independently using one of two major parsing modes: expression mode and argument mode.

Note

As PowerShell parses command input it tries to resolve the command names to cmdlets or native executables. If a command name doesn't have an exact match, PowerShell prepends `Get-` to the command

as a default verb. For example, PowerShell parses `Service` as `Get-Service`. It's not recommended to use this feature for the following reasons:

- It's inefficient. This causes PowerShell to search multiple times.
- External programs with the same name are resolved first, so you may not execute the intended cmdlet.
- `Get-Help` and `Get-Command` don't recognize verb-less names.
- The command name may be a reserved word or a language keyword. `Process` is both, and can't be resolved to `Get-Process`.

Expression mode

Expression mode is intended for combining expressions, required for value manipulation in a scripting language. Expressions are representations of values in PowerShell syntax, and can be simple or composite, for example:

Literal expressions are direct representations of their values:

```
'hello'
32
```

Variable expressions carry the value of the variable they reference:

```
$x
$script:path
```

Operators combine other expressions for evaluation:

```
-12
-not $Quiet
3 + 7
$input.Length -gt 1
```

- *Character string literals* must be contained in quotation marks.
- *Numbers* are treated as numerical values rather than as a series of characters (unless escaped).
- *Operators*, including unary operators like `-` and `-not` and binary operators like `+` and `-gt`, are interpreted as operators and apply their respective operations on their arguments (operands).
- *Attribute and conversion expressions* are parsed as expressions and applied to subordinate expressions. For example: `[int] '7'`.
- *Variable references* are evaluated to their values but *splatting* is forbidden and causes a parser error.
- Anything else is treated as a command to be invoked.

Argument mode

When parsing, PowerShell first looks to interpret input as an expression. But when a command invocation is encountered, parsing continues in argument mode. If you have arguments that contain spaces, such as paths, then you must enclose those argument values in quotes.

Argument mode is designed for parsing arguments and parameters for commands in a shell environment. All input is treated as an expandable string unless it uses one of the following syntaxes:

- Dollar sign (`$`) followed by a variable name begins a variable reference, otherwise it's interpreted as part of the expandable string. The variable reference can include member access or indexing.

    - Additional characters following simple variable references, such as `$HOME`, are considered part of the same argument. Enclose the variable name in braces (`{}`) to separate it from subsequent characters. For example, `${HOME}`.
    - When the variable reference includes member access, the first of any additional characters is considered the start of a new argument. For example `$HOME.Length-more` results in two arguments: the value of `$HOME.Length` and string literal `-more`.

- Quotation marks (`'` and `"`) begin strings
- Braces (`{}`) begin a new script blocks
- Commas (`,`) introduce lists passed as arrays, unless the command being called is a native application, in which case they're interpreted as part of the expandable string. Initial, consecutive or trailing commas aren't supported.
- Parentheses (`()`) begin a new expression
- Subexpression operator (`$()`) begins an embedded expression
- Initial at sign (`@`) begins expression syntaxes such as splatting (`@args`), arrays (`@(1,2,3)`), and hash table literals (`@{a=1;b=2}`).
- `()`, `$()`, and `@()` at the start of a token create a new parsing context that can contain expressions or nested commands.
    - When followed by additional characters, the first additional character is considered the start of a new, separate argument.
    - When preceded by an unquoted literal `$()` works like an expandable string, `()` starts a new argument that's an expression, and `@()` is taken as literal `@` with `()` starting a new argument that's an expression.
- Everything else is treated as an expandable string, except metacharacters that still need escaping. See Handling special characters.
    - The argument-mode metacharacters (characters with special syntactic meaning) are: `<space> ' " ` , ; ( ) { } | & < > @ #`. Of these, `< > @ #` are only special at the start of a token.
- The stop-parsing token (`--%`) changes the interpretation of all remaining arguments. For more information, see the stop-parsing token section below.

Examples

The following table provides several examples of tokens processed in expression mode and argument mode and the evaluation of those tokens. For these examples, the value of the variable `$a` is 4.

Example	Mode	Result
`2`	Expression	2 (integer)
`` `2 ``	Expression	"2" (command)
`Write-Output 2`	Expression	2 (integer)
`2+2`	Expression	4 (integer)
`Write-Output 2+2`	Argument	"2+2" (string)
`Write-Output(2+2)`	Expression	4 (integer)

Example	Mode	Result
`$a`	Expression	4 (integer)
`Write-Output $a`	Expression	4 (integer)
`$a+2`	Expression	6 (integer)
`Write-Output $a+2`	Argument	"4+2" (string)
`$-`	Argument	"$-" (command)
`Write-Output $-`	Argument	"$-" (string)
`a$a`	Expression	"a$a" (command)
`Write-Output a$a`	Argument	"a4" (string)
`a'$a'`	Expression	"a$a" (command)
`Write-Output a'$a'`	Argument	"a$a" (string)
`a"$a"`	Expression	"a$a" (command)
`Write-Output a"$a"`	Argument	"a4" (string)
`a$(2)`	Expression	"a$(2)" (command)
`Write-Output a$(2)`	Argument	"a2" (string)

Every token can be interpreted as some kind of object type, such as **Boolean** or **String**. PowerShell attempts to determine the object type from the expression. The object type depends on the type of parameter a command expects and on whether PowerShell knows how to convert the argument to the correct type. The following table shows several examples of the types assigned to values returned by the expressions.

Example	Mode	Result
`Write-Output !1`	argument	"!1" (string)
`Write-Output (!1)`	expression	False (Boolean)
`Write-Output (2)`	expression	2 (integer)
`Set-Variable AB A,B`	argument	'A','B' (array)
`CMD /CECHO A,B`	argument	'A,B' (string)
`CMD /CECHO $AB`	expression	'A B' (array)
`CMD /CECHO :$AB`	argument	':A B' (string)

Handling special characters

The backtick character (`) can be used to escape any special character in an expression. This is most useful for escaping the argument-mode metacharacters that you want to use as literal characters rather than as a metacharacter. For example, to use the dollar sign ($) as a literal in an expandable string:

```
"The value of `$ErrorActionPreference is '$ErrorActionPreference'."
```

```
The value of $ErrorActionPreference is 'Continue'.
```

Line continuation

The backtick character can also be used at the end of a line to allow you to continue the input on the next line. This is improve the readability of a command that takes several parameters with long names and argument values. For example:

```
New-AzVm `
    -ResourceGroupName "myResourceGroupVM" `
    -Name "myVM" `
    -Location "EastUS" `
    -VirtualNetworkName "myVnet" `
    -SubnetName "mySubnet" `
    -SecurityGroupName "myNetworkSecurityGroup" `
    -PublicIpAddressName "myPublicIpAddress" `
    -Credential $cred
```

However, you should avoid using line continuation.

- The backtick characters can be hard to see and easy to forget.
- An extra space after the backtick breaks the line continuation. Since the space is hard to see it can be difficult to find the error.

PowerShell provides several ways break lines at natural points in the syntax.

- After pipe characters (`|`)
- After binary operators (`+`, `-`, `-eq`, etc.)
- After commas (`,`) in an array
- After opening characters such as `[`, `{`, `(`

For large parameter set, use splatting instead. For example:

```
$parameters = @{
    ResourceGroupName = "myResourceGroupVM"
    Name = "myVM"
    Location = "EastUS"
    VirtualNetworkName = "myVnet"
    SubnetName = "mySubnet"
    SecurityGroupName = "myNetworkSecurityGroup"
    PublicIpAddressName = "myPublicIpAddress"
    Credential = $cred
}
New-AzVm @parameters
```

Passing arguments to native commands

When running native commands from PowerShell, the arguments are first parsed by PowerShell. The parsed arguments are then joined into a single string with each parameter separated by a space.

For example, the following command calls the `icacls.exe` program.

```
icacls X:\VMS /grant Dom\HVAdmin:(CI)(OI)F
```

To run this command in PowerShell 2.0, you must use escape characters to prevent PowerShell from misinterpreting the parentheses.

```
icacls X:\VMS /grant Dom\HVAdmin:`(CI`)`(OI`)F
```

The stop-parsing token

Beginning in PowerShell 3.0, you can use the *stop-parsing* (`--%`) token to stop PowerShell from interpreting input as PowerShell commands or expressions.

Note

The stop-parsing token is only intended for use native commands on Windows platforms.

When calling a native command, place the stop-parsing token before the program arguments. This technique is much easier than using escape characters to prevent misinterpretation.

When it encounters a stop-parsing token, PowerShell treats the remaining characters in the line as a literal. The only interpretation it performs is to substitute values for environment variables that use standard Windows notation, such as `%USERPROFILE%`.

```
icacls X:\VMS --% /grant Dom\HVAdmin:(CI)(OI)F
```

PowerShell sends the following command string to the `icacls.exe` program:

```
X:\VMS /grant Dom\HVAdmin:(CI)(OI)F
```

The stop-parsing token is effective only until the next newline or pipeline character. You can't use the line continuation character (`` ` ``) to extend its effect or use a command delimiter (`;`) to terminate its effect.

Other than `%variable%` environment-variable references, you can't embed any other dynamic elements in the command. Escaping a % character as %%, the way you can do inside batch files, isn't supported. `%<name>%` tokens are invariably expanded. If `<name>` doesn't refer to a defined environment variable the token is passed through as-is.

You can't use stream redirection (like `>file.txt`) because they're passed verbatim as arguments to the target command.

In the following example, the first step runs a command without using the stop-parsing token. PowerShell evaluates the quoted string and passes the value (without quotes) to `cmd.exe`, which results in an error.

```
PS> cmd /c echo "a|b"
'b' is not recognized as an internal or external command,
operable program or batch file.
PS> cmd /c --% echo "a|b"
"a|b"
```

Note

The stop-parsing token isn't needed when using PowerShell cmdlets. However, it could be useful to pass arguments to a PowerShell function that is designed to call a native command with those arguments.

Passing arguments that contain quote characters

Some native commands expect arguments that contain quote characters. PowerShell 7.3 changed the way the command line is parsed for native commands.

Caution

The new behavior is a **breaking change** from the Windows PowerShell 5.1 behavior. This may break scripts and automation that work around the various issues when invoking native applications. Use the stop-parsing token (`--%`) or the `Start-Process` cmdlet to avoid the native argument passing when needed.

The new `$PSNativeCommandArgumentPassing` preference variable controls this behavior. This variable allows you to select the behavior at runtime. The valid values are `Legacy`, `Standard`, and `Windows`. The default behavior is platform specific. On Windows platforms, the default setting is `Windows` and non-Windows platforms default to `Standard`.

`Legacy` is the historic behavior. The behavior of `Windows` and `Standard` mode are the same except, in `Windows` mode, invocations of the following files automatically use the `Legacy` style argument passing.

- `cmd.exe`
- `cscript.exe`
- `wscript.exe`
- ending with `.bat`
- ending with `.cmd`
- ending with `.js`
- ending with `.vbs`
- ending with `.wsf`

If the `$PSNativeCommandArgumentPassing` is set to either `Legacy` or `Standard`, the parser doesn't check for these files.

Note

The following examples use the `TestExe.exe` tool. You can build `TestExe` from the source code. See TestExe in the PowerShell source repository.

New behaviors made available by this change:

- Literal or expandable strings with embedded quotes the quotes are now preserved:

```
PS> $a = 'a" "b'
PS> TestExe -echoargs $a 'c" "d' e" "f
Arg 0 is <a" "b>
Arg 1 is <c" "d>
Arg 2 is <e f>
```

- Empty strings as arguments are now preserved:

```
PS> TestExe -echoargs '' a b ''
Arg 0 is <>
Arg 1 is <a>
```

```
Arg 2 is <b>
Arg 3 is <>
```

The goal of these examples is to pass the directory path (with spaces and quotes) `"C:\Program Files (x86)\Microsoft\"` to a native command so that it received the path as a quoted string.

In `Windows` or `Standard` mode, the following examples produce the expected results:

```
TestExe -echoargs """${env:ProgramFiles(x86)}\Microsoft\"""
TestExe -echoargs '"C:\Program Files (x86)\Microsoft\"'
```

To get the same results in `Legacy` mode, you must escape the quotes or use the stop-parsing token (`--%`):

```
TestExe -echoargs """""${env:ProgramFiles(x86)}\Microsoft\\""""""
TestExe -echoargs "\""C:\Program Files (x86)\Microsoft\\"""
TestExe -echoargs --% ""\""C:\Program Files (x86)\Microsoft\\"\"""
TestExe -echoargs --% """C:\Program Files (x86)\Microsoft\\""
TestExe -echoargs --% """%ProgramFiles(x86)%\Microsoft\\""
```

Note

The backslash (`\`) character isn't recognized as an escape character by PowerShell. It's the escape character used by the underlying API for ProcessStartInfo.ArgumentList.

PowerShell 7.3 also added the ability to trace parameter binding for native commands. For more information, see Trace-Command.

Passing arguments to PowerShell commands

Beginning in PowerShell 3.0, you can use the *end-of-parameters* token (`--`) to stop PowerShell from interpreting input as PowerShell parameters. This is a convention specified in the POSIX Shell and Utilities specification.

The end-of-parameters token

The end-of-parameters token (`--`) indicates that all arguments following it are to be passed in their actual form as though double quotes were placed around them. For example, using `--` you can output the string `-InputObject` without using quotes or having it interpreted as a parameter:

```
Write-Output -- -InputObject
```

```
-InputObject
```

Unlike the stop-parsing (`--%`) token, any values following the `--` token can be interpreted as expressions by PowerShell.

```
Write-Output -- -InputObject $env:PROCESSOR_ARCHITECTURE
```

```
-InputObject
AMD64
```

This behavior only applies to PowerShell commands. If you use the `--` token when calling an external command, the `--` string is passed as an argument to that command.

```
TestExe -echoargs -a -b -- -c
```

The output shows that `--` is passed as an argument to `TestExe`.

```
Arg 0 is <-a>
Arg 1 is <-b>
Arg 2 is <-->
Arg 3 is <-c>
```

Tilde (~)

The tilde character (~) has special meaning in PowerShell. When it's used with PowerShell commands at the beginning of a path, the tilde character is expanded to the user's home directory. If the tilde character is used anywhere else in a path, it's treated as a literal character.

```
PS D:\temp> $PWD

Path
----
D:\temp

PS D:\temp> Set-Location ~
PS C:\Users\user2> $PWD

Path
----
C:\Users\user2
```

In this example, the **Name** parameter of the `New-Item` expects a string. The tilde character is treated as a literal character. To change to the newly created directory, you must qualify the path with the tilde character.

```
PS D:\temp> Set-Location ~
PS C:\Users\user2> New-Item -Type Directory -Name ~


    Directory: C:\Users\user2

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----           5/6/2024  2:08 PM                ~

PS C:\Users\user2> Set-Location ~
```

```
PS C:\Users\user2> Set-Location .\~
PS C:\Users\user2\~> $PWD

Path
----
C:\Users\user2\~
```

When you use the tilde character with native commands, PowerShell passes the tilde as a literal character. Using the tilde in a path causes errors for native commands on Windows that don't support the tilde character.

```
PS D:\temp> $PWD

Path
----
D:\temp

PS D:\temp> Get-Item ~\repocache.clixml

    Directory: C:\Users\user2

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---           4/29/2024  3:42 PM          88177 repocache.clixml

PS D:\temp> more.com ~\repocache.clixml
Cannot access file D:\temp\~\repocache.clixml
```

See also

- about_Command_Syntax

In this article

Previous Chapter Next Chapter