about_Splatting - PowerShell

sdwheeler ⋮ 14-18 minutes

about_Splatting

- Article
- 01/29/2024
-

In this article

Short description

Describes how to use splatting to pass parameters to commands in PowerShell.

Long description

Splatting is a method of passing a collection of parameter values to a command as a unit. PowerShell associates each value in the collection with a command parameter. Splatted parameter values are stored in named splatting variables, which look like standard variables, but begin with an At symbol (`@`) instead of a dollar sign (`$`). The At symbol tells PowerShell that you are passing a collection of values, instead of a single value.

Splatting makes your commands shorter and easier to read. You can reuse the splatting values in different command calls and use splatting to pass parameter values from the `$PSBoundParameters` automatic variable to other scripts and functions.

Beginning in Windows PowerShell 3.0, you can also use splatting to represent all parameters of a command.

Syntax

```
<CommandName> <optional parameters> @<HashTable> <optional parameters>
<CommandName> <optional parameters> @<Array> <optional parameters>
```

To provide parameter values for positional parameters, in which parameter names are not required, use the array syntax. To provide parameter name and value pairs, use the hash table syntax. The splatted value can appear anywhere in the parameter list.

When splatting, you do not need to use a hash table or an array to pass all parameters. You may pass some parameters by using splatting and pass others by position or by parameter name. Also, you can splat multiple objects in a single command so you don't pass more than one value for each parameter.

As of PowerShell 7.1, you can override a splatted parameter by explicitly defining a parameter in a command.

Splatting with hash tables

Use a hash table to splat parameter name and value pairs. You can use this format for all parameter types, including positional and switch parameters. Positional parameters must be assigned by name.

The following examples compare two `Copy-Item` commands that copy the Test.txt file to the Test2.txt file in the same directory.

The first example uses the traditional format in which parameter names are included.

```
Copy-Item -Path "test.txt" -Destination "test2.txt" -WhatIf
```

The second example uses hash table splatting. The first command creates a hash table of parameter-name and parameter-value pairs and stores it in the `$HashArguments` variable. The second command uses the `$HashArguments` variable in a command with splatting. The At symbol (`@HashArguments`) replaces the dollar sign (`$HashArguments`) in the command.

To provide a value for the **WhatIf** switch parameter, use `$True` or `$False`.

```
$HashArguments = @{
  Path = "test.txt"
  Destination = "test2.txt"
  WhatIf = $true
}
Copy-Item @HashArguments
```

Note

In the first command, the At symbol (`@`) indicates a hash table, not a splatted value. The syntax for hash tables in PowerShell is: `@{<name>=<value>; <name>=<value>; ...}`

Splatting with arrays

Use an array to splat values for positional parameters, which do not require parameter names. The values must be in position-number order in the array.

The following examples compare two `Copy-Item` commands that copy the Test.txt file to the Test2.txt file in the same directory.

The first example uses the traditional format in which parameter names are omitted. The parameter values appear in position order in the command.

```
Copy-Item "test.txt" "test2.txt" -WhatIf
```

The second example uses array splatting. The first command creates an array of the parameter values and stores it in the `$ArrayArguments` variable. The values are in position order in the array. The second command uses the `$ArrayArguments` variable in a command in splatting. The At symbol (`@ArrayArguments`) replaces the dollar sign (`$ArrayArguments`) in the command.

```
$ArrayArguments = "test.txt", "test2.txt"
Copy-Item @ArrayArguments -WhatIf
```

Using the ArgumentList parameter

Several cmdlets have an **ArgumentList** parameter that is used to pass parameter values to a script block that is executed by the cmdlet. The **ArgumentList** parameter takes an array of values that is passed to the script block. PowerShell is effectively using array splatting to bind the values to the parameters of the script block. When using **ArgumentList**, if you need to pass an array as a single object bound to a single parameter, you must wrap the array as the only element of another array.

The following example has a script block that takes a single parameter that is an array of strings.

```
$array = 'Hello', 'World!'
Invoke-Command -ScriptBlock {
  param([string[]]$words) $words -join ' '
  } -ArgumentList $array
```

In this example, only the first item in `$array` is passed to the script block.

```
Hello
```

```
$array = 'Hello', 'World!'
Invoke-Command -ScriptBlock {
  param([string[]]$words) $words -join ' '
} -ArgumentList (,$array)
```

In this example, `$array` is wrapped in an array so that the entire array is passed to the script block as a single object.

```
Hello World!
```

Examples

Example 1: Reuse splatted parameters in different commands

This example shows how to reuse splatted values in different commands. The commands in this example use the `Write-Host` cmdlet to write messages to the host program console. It uses splatting to specify the foreground and background colors.

To change the colors of all commands, just change the value of the `$Colors` variable.

The first command creates a hash table of parameter names and values and stores the hash table in the `$Colors` variable.

```
$Colors = @{ForegroundColor = "black"; BackgroundColor = "white"}
```

The second and third commands use the `$Colors` variable for splatting in a `Write-Host` command. To use the `$Colors variable`, replace the dollar sign (`$Colors`) with an At symbol (`@Colors`).

```
#Write a message with the colors in $Colors
Write-Host "This is a test." @Colors

#Write second message with same colors. The position of splatted
#hash table does not matter.
Write-Host @Colors "This is another test."
```

Example 2: Forward parameters using $PSBoundParameters

This example shows how to forward their parameters to other commands using splatting and the `$PSBoundParameters` automatic variable.

The `$PSBoundParameters` automatic variable is a dictionary object (System.Collections.Generic.Dictionary) that contains all the parameter names and values that are used when a script or function is run.

In the following example, we use the `$PSBoundParameters` variable to forward the parameters values passed to a script or function from `Test2` function to the `Test1` function. Both calls to the `Test1` function from `Test2` use splatting.

```
function Test1
{
    param($a, $b, $c)

    "a = $a"
    "b = $b"
    "c = $c"
}

function Test2
{
    param($a, $b, $c)

    #Call the Test1 function with $a, $b, and $c.
    Test1 @PSBoundParameters

    #Call the Test1 function with $b and $c, but not with $a
    Test1 -b $PSBoundParameters.b -c $PSBoundParameters.c
}

Test2 -a 1 -b 2 -c 3
```

```
a = 1
b = 2
c = 3
a =
b = 2
c = 3
```

Example 3: Override splatted parameters with explicitly defined parameters

This example shows how to override a splatted parameter using explicitly defined parameters. This is useful when you don't want to build a new hashtable or change a value in the hashtable you are using to splat.

The `$commonParams` variable stores the parameters to create virtual machines in the `East US` location. The `$allVms` variable is a list of virtual machines to create. We loop through the list and use `$commonParams` to splat the parameters to create each virtual machine. However, we want `myVM2` to be created in a different region than the other virtual machines. Instead of adjusting the `$commonParams` hashtable, you can explicitly define the **Location** parameter in `New-AzVm` to supersede the value of the `Location` key in `$commonParams`.

```
$commonParams = @{
    ResourceGroupName = "myResourceGroup"
    Location = "East US"
    VirtualNetworkName = "myVnet"
    SubnetName = "mySubnet"
    SecurityGroupName = "myNetworkSecurityGroup"
    PublicIpAddressName = "myPublicIpAddress"
}

$allVms = @('myVM1','myVM2','myVM3',)

foreach ($vm in $allVms)
```

```
{
    if ($vm -eq 'myVM2')
    {
        New-AzVm @commonParams -Name $vm -Location "West US"
    }
    else
    {
        New-AzVm @commonParams -Name $vm
    }
}
```

Example 4: Using multiple splatted objects in a single command

You can use multiple splatted objects in a single command. In this example, different parameters are defined in separate hashtables. The hashtables are splatted in a single `Write-Host` command.

```
$a = @{
    Message         = 'Hello', 'World!'
}
$b = @{
    Separator       = '|'
}
$c = @{
    BackgroundColor = 'Cyan'
    ForegroundColor = 'Black'
}
Write-Host @a @b @c
```

Splatting command parameters

You can use splatting to represent the parameters of a command. This technique is useful when you are creating a proxy function, that is, a function that calls another command. This feature is introduced in Windows PowerShell 3.0.

To splat the parameters of a command, use `@Args` to represent the command parameters. This technique is easier than enumerating command parameters and it works without revision even if the parameters of the called command change.

The feature uses the `$Args` automatic variable, which contains all unassigned parameter values.

For example, the following function calls the `Get-Process` cmdlet. In this function, `@Args` represents all the parameters of the `Get-Process` cmdlet.

```
function Get-MyProcess { Get-Process @Args }
```

When you use the `Get-MyProcess` function, all unassigned parameters and parameter values are passed to `@Args`, as shown in the following commands.

```
Get-MyProcess -Name PowerShell
```

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    463      46   225484     237196   719    15.86   3228 powershell
```

```
Get-MyProcess -Name PowerShell_Ise -FileVersionInfo
```

```
ProductVersion   FileVersion      FileName
--------------   -----------      --------
6.2.9200.16384   6.2.9200.1638...
C:\Windows\system32\WindowsPowerShell\...
```

You can use `@Args` in a function that has explicitly declared parameters. You can use it more than once in a function, but all parameters that you enter are passed to all instances of `@Args`, as shown in the following example.

```
function Get-MyCommand
{
    Param ([switch]$P, [switch]$C)
    if ($P) { Get-Process @Args }
    if ($C) { Get-Command @Args }
}

Get-MyCommand -P -C -Name PowerShell
```

```
 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
     50   112.76      78.52      16.64    6880   1 powershell

Path               :
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Extension          : .exe
Definition         :
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Source             :
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Version            : 10.0.22621.3085
Visibility         : Public
OutputType         : {System.String}
Name               : powershell.exe
CommandType        : Application
```

```
ModuleName          :
Module              :
RemotingCapability : PowerShell
Parameters          :
ParameterSets       :
HelpUri             :
FileVersionInfo     : File:
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
                      InternalName:     POWERSHELL
                      OriginalFilename: PowerShell.EXE.MUI
                      FileVersion:      10.0.22621.1
(WinBuild.160101.0800)
                      FileDescription:  Windows PowerShell
                      Product:          Microsoft&reg; Windows&reg;
Operating System
                      ProductVersion:   10.0.22621.1
                      Debug:            False
                      Patched:          False
                      PreRelease:       False
                      PrivateBuild:     False
                      SpecialBuild:     False
                      Language:         English (United States)
```

Notes

If you make a function into an advanced function by using either the **CmdletBinding** or **Parameter** attributes, the `$args` automatic variable is no longer available in the function. Advanced functions require explicit parameter definition.

PowerShell Desired State Configuration (DSC) was not designed to use splatting. You cannot use splatting to pass values into a DSC resource. For more information, see Gael Colas' article Pseudo-Splatting DSC Resources.

See also

- about_Arrays
- about_Automatic_Variables
- about_Hash_Tables
- about_Parameters

In this article

Previous Chapter Next Chapter