about_Scopes - PowerShell

sdwheeler ⋮ 26-33 minutes

about_Scopes

- Article
- 07/22/2024
-

In this article

Short description

Explains the concept of scope in PowerShell and shows how to set and change the scope of elements.

Long description

PowerShell protects access to variables, aliases, functions, and PowerShell drives (PSDrives) by limiting where they can be read and changed. PowerShell uses scope rules to ensure that you don't make unintentional changes to items in other scopes.

Scope rules

When you start PowerShell, the host (`pwsh.exe`) creates a PowerShell runspace. Host processes can have multiple runspaces. Each runspace has its own session state and scope containers. Session state and scopes can't be accessed across runspace instances.

The following are the basic rules of scope:

- Scopes may nest. An outer scope is referred to as a parent scope. Any nested scopes are child scopes of that parent.
- An item is visible in the scope that it was created and in any child scopes, unless you explicitly make it private.
- You can declare variables, aliases, functions, and PowerShell drives for a scope outside of the current scope.
- An item that you created within a scope can be changed only in the scope in which it was created, unless you explicitly specify a different scope.
- When code running in a runspace references an item, PowerShell searches the scope hierarchy, starting with the current scope and proceeding through each parent scope.
  - If the item isn't found, a new item is created in the current scope.
  - If it finds a match, the value of the item is retrieved from the scope where is was found.
  - If you change the value, the item is copied to the current scope so that the change only affects the current scope.
- If you explicitly create an item that shares its name with an item in a different scope, the original item might be hidden by the new item, but it isn't overridden or changed.

Parent and child scopes

You can create a new child scope by calling a script or function. The calling scope is the parent scope. The called script or function is the child scope. The functions or scripts you call may call other functions, creating a hierarchy of child scopes whose root scope is the global scope.

Note

Functions from a module don't run in a child scope of the calling scope. Modules have their own session state that's linked to the scope in which the module was imported. All module code runs in a module-specific hierarchy of scopes that has its own root scope. For more information, see the Modules section of this article.

When a child scope is created, it includes all the aliases and variables that have the **AllScope** option, and some automatic variables. This option is discussed later in this article.

Unless you explicitly make the items private, the items in the parent scope are available to the child scope. Items that you create or change in a child scope don't affect the parent scope, unless you explicitly specify the scope when you create the items.

To find the items in a particular scope, use the Scope parameter of `Get-Variable` or `Get-Alias`.

For example, to get all the variables in the local scope, type:

```
Get-Variable -Scope local
```

To get all the variables in the global scope, type:

```
Get-Variable -Scope global
```

When a reference is made to a variable, alias, or function, PowerShell searches the current scope. If the item isn't found, the parent scope is searched. This search is repeated all they way up to the global scope. If a variable is private in a parent scope, the search through continues through the scope chain. Example 4 shows the effect of a private variable in a scope search.

PowerShell scopes names

PowerShell defines names for some scopes to allow easier access to that scope. PowerShell defines the following named scopes:

- **Global**: The scope that's in effect when PowerShell starts or when you create a new session or runspace. Variables and functions that are present when PowerShell starts, such as automatic variables and preference variables, are created in the global scope. The variables, aliases, and functions in your PowerShell profiles are also created in the global scope. The global scope is the root parent scope in a runspace.
- **Local**: The current scope. The local scope can be the global scope or any other scope.
- **Script**: The scope that's created while a script file runs. The commands in the script run in the script scope. For the commands in a script, the script scope is the local scope.

For cmdlets that support scopes, scopes can be referred to by a number that describes the relative position of one scope to another. Scope 0 denotes the current (local) scope, scope 1 is the current scope's parent, scope 2 is the current scope's grandparent. This pattern continues until you reach the root scope.

Scope modifiers

A variable, alias, or function name can include any one of the following optional scope modifiers:

- `global:` - Specifies that the name exists in the **Global** scope.

- `local:` - Specifies that the name exists in the **Local** scope. The current scope is always the **Local** scope.

- `private:` - Specifies that the name is **Private** and only visible to the current scope.

  Note

  `private:` isn't a scope. It's an option that changes the accessibility of an item outside of the scope in which it's defined.

- `script:` - Specifies that the name exists in the **Script** scope. **Script** scope is the nearest ancestor script file's scope or **Global** if there is no nearest ancestor script file.

- `using:` - Used to access variables defined in another scope while running in remote sessions, background jobs, or thread jobs.

- `workflow:` - Specifies that the name exists within a workflow. Note: Workflows aren't supported in PowerShell v6 and higher.

- `<variable-namespace>` - A modifier created by a PowerShell **PSDrive** provider. For example:

Namespace	Description
`Alias:`	Aliases defined in the current scope
`Env:`	Environment variables defined in the current scope
`Function:`	Functions defined in the current scope
`Variable:`	Variables defined in the current scope

The default scope for scripts is the script scope. The default scope for functions and aliases is the local scope, even if they're defined in a script.

Using scope modifiers

To specify the scope of a new variable, alias, or function, use a scope modifier.

The syntax for a scope modifier in a variable is:

```
$[<scope-modifier>:]<name> = <value>
```

The syntax for a scope modifier in a function is:

```
function [<scope-modifier>:]<name> {<function-body>}
```

The following command, which doesn't use a scope modifier, creates a variable in the current or **local** scope:

```
$a = "one"
```

To create the same variable in the **global** scope, use the scope `global:` modifier:

```
$global:a = "one"
Get-Variable a | Format-List *
```

Notice the **Visibility** and **Options** property values.

```
Name        : a
Description :
Value       : one
Visibility  : Public
Module      :
ModuleName  :
Options     : None
Attributes  : {}
```

Compare that to a private variable:

```
$private:pVar = 'Private variable'
Get-Variable pVar | Format-List *
```

Using the `private` scope modifier sets the **Options** property to `Private`.

```
Name        : pVar
Description :
Value       : Private variable
Visibility  : Public
```

```
Module       :
ModuleName   :
Options      : Private
Attributes   : {}
```

To create the same variable in the **script** scope, use the `script:` scope modifier:

```
$script:a = "one"
```

You can also use a scope modifier with functions. The following function definition creates a function in the **global** scope:

```
function global:Hello {
  Write-Host "Hello, World"
}
```

You can also use scope modifiers to refer to a variable in a different scope. The following command refers to the `$test` variable, first in the local scope and then in the global scope:

```
$test
$global:test
```

The `using:` scope modifier

Using is a special scope modifier that identifies a local variable in a remote command. Without a modifier, PowerShell expects variables in remote commands to be defined in the remote session.

The `using` scope modifier is introduced in PowerShell 3.0.

For any script or command that executes out of session, you need the `using` scope modifier to embed variable values from the calling session scope, so that out of session code can access them. The `using` scope modifier is supported in the following contexts:

- Remotely executed commands, started with `Invoke-Command` using the **ComputerName**, **HostName**, **SSHConnection** or **Session** parameters (remote session)
- Background jobs, started with `Start-Job` (out-of-process session)
- Thread jobs, started via `Start-ThreadJob` or `ForEach-Object -Parallel` (separate thread session)

Depending on the context, embedded variable values are either independent copies of the data in the caller's scope or references to it. In remote and out-of-process sessions, they're always independent copies.

For more information, see about_Remote_Variables.

A `$using:` reference only expands to a variable's value. If you want to change the value of a variable in the caller's scope, you must have a reference to the variable itself. You can create a reference to a variable by getting the **PSVariable** instance of the variable. The following example show how to create a reference and make changes in a thread job.

```
$Count = 1
$refOfCount = Get-Variable Count

Start-ThreadJob {
    ($using:refOfCount).Value = 2
} | Receive-Job -Wait -AutoRemoveJob

$Count
```

```
2
```

Note

This is not a thread-safe operation. You can cause data corruption if you try to change the value from multiple threads at the same time. You should use thread-safe data types or synchronization primitives to protect shared data. For more information, see Thread-Safe collections.

Serialization of variable values

Remotely executed commands and background jobs run out-of-process. Out-of-process sessions use XML-based serialization and deserialization to make the values of variables available across the process boundaries. The serialization process converts objects to a **PSObject** that contains the original objects properties but not its methods.

For a limited set of types, deserialization rehydrates objects back to the original type. The rehydrated object is a copy of the original object instance. It has the type properties and methods. For simple types, such as **System.Version**, the copy is exact. For complex types, the copy is imperfect. For example, rehydrated certificate objects don't include the private key.

Instances of all other types are **PSObject** instances. The **PSTypeNames** property contains the original type name prefixed with **Deserialized**, for example, **Deserialized.System.Data.DataTable**

The AllScope Option

Variables and aliases have an **Option** property that can take a value of **AllScope**. Items that have the **AllScope** property become part of any child scopes that you create, although they aren't retroactively inherited by parent scopes.

An item that has the **AllScope** property is visible in the child scope, and it's part of that scope. Changes to the item in any scope affect all the scopes in which the variable is defined.

Managing scope

Several cmdlets have a **Scope** parameter that lets you get or set (create and change) items in a particular scope. Use the following command to find all the cmdlets in your session that have a **Scope** parameter:

```
Get-Help * -Parameter scope
```

To find the variables that are visible in a particular scope, use the `Scope` parameter of `Get-Variable`. The visible variables include global variables, variables in the parent scope, and variables in the current scope.

For example, the following command gets the variables that are visible in the local scope:

```
Get-Variable -Scope local
```

To create a variable in a particular scope, use a scope modifier or the **Scope** parameter of `Set-Variable`. The following command creates a variable in the global scope:

```
New-Variable -Scope global -Name a -Value "One"
```

You can also use the Scope parameter of the `New-Alias`, `Set-Alias`, or `Get-Alias` cmdlets to specify the scope. The following command creates an alias in the global scope:

```
New-Alias -Scope global -Name np -Value Notepad.exe
```

To get the functions in a particular scope, use the `Get-Item` cmdlet when you are in the scope. The `Get-Item` cmdlet doesn't have a **Scope** parameter.

Note

For the cmdlets that use the **Scope** parameter, you can also refer to scopes by number. The number describes the relative position of one scope to another. Scope 0 represents the current, or local, scope. Scope 1 indicates the immediate parent scope. Scope 2 indicates the parent of the parent scope, and so on. Numbered scopes are useful if you have created many recursive scopes.

Using dot-source notation with scope

Scripts and functions follow the rules of scope. You create them in a particular scope, and they affect only that scope unless you use a cmdlet parameter or a scope modifier to change that scope.

But, you can add the contents of a script or function to the current scope using dot-source notation. When you run a script or function using dot-source notation, it runs in the current scope. Any functions, aliases, and variables in the script or function are added to the current scope.

For example, to run the `Sample.ps1` script from the `C:\Scripts` directory in the script scope (the default for scripts), just enter the full path to the script file on the command line.

```
c:\scripts\sample.ps1
```

A script file must have a `.ps1` file extension to be executable. Files that have spaces in their path must be enclosed in quotes. If you try to execute the quoted path, PowerShell displays the contents of the quoted string instead of running the script. The call operator (&) allows you to execute the contents of the string containing the filename.

Using the call operator to run a function or script runs it in script scope. Using the call operator is no different than running the script by name.

```
& c:\scripts\sample.ps1
```

You can read more about the call operator in about_Operators.

To run the `Sample.ps1` script in the local scope type a dot and a space (`. `) before the path to the script:

```
. c:\scripts\sample.ps1
```

Now, any functions, aliases, or variables defined in the script are added to the current scope.

Restricting without scope

PowerShell has some options and features that are similar to scope and may interact with scopes. These feature may be confused with scope or the behavior of scope.

Sessions, modules, and nested prompts are self-contained environments, not child scopes of the global scope in the session.

Sessions

A session is an environment in which PowerShell runs. When you create a session on a remote computer, PowerShell establishes a persistent connection to the remote computer. The persistent connection lets you use the session for multiple related commands.

Because a session is a contained environment, it has its own scope, but a session isn't a child scope of the session in which it was created. The session starts with its own global scope. This scope is independent of the global scope of the session. You can create child scopes in the session. For example, you can run a script to create a child scope in a session.

Modules

You can use a PowerShell module to share and deliver PowerShell tools. A module is a unit that can contain cmdlets, scripts, functions, variables, aliases, and other useful items. Unless explicitly exported (using `Export-ModuleMember` or the module manifest), the items in a module aren't accessible outside the module. Therefore, you can add the module to your session and use the public items without worrying that the other items might override the cmdlets, scripts, functions, and other items in your session.

By default, modules are loaded into the root-level (global) scope of the runspace. Importing a module doesn't change the scope. Within the session, modules have their own scope. Consider the following module `C:\temp\mod1.psm1`:

```
$a = "Hello"

function foo {
    "`$a = $a"
    "`$global:a = $global:a"
}
```

Now we create a global variable `$a`, give it a value and call the function **foo**.

```
$a = "Goodbye"
foo
```

The module declares the variable `$a` in the module scope then the function **foo** outputs the value of the variable in both scopes.

```
$a = Hello
$global:a = Goodbye
```

Modules create parallel scope containers linked to the scope in which they were imported. Items exported by the module are available starting at the scope-level in which they are imported. Items not exported from the module are only available within the module's scope container. Functions in the module can access items in the scope in which they were imported as well as items in the module's scope container.

If you load **Module2** from *within* **Module1**, **Module2** is loaded into the scope container of Module1. Any exports from **Module2** are placed in the current module scope of **Module1**. If you use `Import-Module -Scope local`, then the exports are placed into the current scope object rather than at the top level. If you are *in a module* and load another module using `Import-Module -Scope global` (or `Import-Module -Global`), that module and its exports are loaded into the global scope instead of the module's local scope. The **WindowsCompatibility** feature does this to import proxy modules into the global session state.

Nested prompts

Nested prompts don't have their own scope. When you enter a nested prompt, the nested prompt is a subset of the environment. But, you remain within the local scope.

Scripts do have their own scope. If you are debugging a script, and you reach a breakpoint in the script, you enter the script scope.

Private option

Aliases and variables have an **Option** property that can take a value of `Private`. Items that have the `Private` option can be viewed and changed in the scope in which they're created, but they can't be viewed or changed outside that scope.

For example, if you create a variable that has a private option in the global scope and then run a script, `Get-Variable` commands in the script don't display the private variable. Using the global scope modifier in this instance doesn't display the private variable.

You can use the **Option** parameter of the `New-Variable`, `Set-Variable`, `New-Alias`, and `Set-Alias` cmdlets to set the value of the Option property to Private.

Visibility

The **Visibility** property of a variable or alias determines whether you can see the item outside the container, in which it was created. A container could be a module, script, or snap-in. Visibility is designed for containers in the same way that the `Private` value of the **Option** property is designed for scopes.

The **Visibility** property takes the `Public` and `Private` values. Items that have private visibility can be viewed and changed only in the container in which they were created. If the container is added or imported, the items that have private visibility can't be viewed or changed.

Because visibility is designed for containers, it works differently in a scope.

- If you create an item that has private visibility in the global scope, you can't view or change the item in any scope.
- If you try to view or change the value of a variable that has private visibility, PowerShell returns an error message.

You can use the `New-Variable` and `Set-Variable` cmdlets to create a variable that has private visibility.

Examples

Example 1: Change a variable value only in a script

The following command changes the value of the `$ConfirmPreference` variable in a script. The change doesn't affect the global scope.

First, to display the value of the `$ConfirmPreference` variable in the local scope, use the following command:

```
PS>  $ConfirmPreference
High
```

Create a Scope.ps1 script that contains the following commands:

```
$ConfirmPreference = "Low"
"The value of `$ConfirmPreference is $ConfirmPreference."
```

Run the script. The script changes the value of the `$ConfirmPreference` variable and then reports its value in the script scope. The output should resemble the following output:

```
The value of $ConfirmPreference is Low.
```

Next, test the current value of the `$ConfirmPreference` variable in the current scope.

```
PS>  $ConfirmPreference
High
```

This example shows that changes to the value of a variable in the script scope doesn't affect the variable`s value in the parent scope.

Example 2: View a variable value in different scopes

You can use scope modifiers to view the value of a variable in the local scope and in a parent scope.

First, define a `$test` variable in the global scope.

```
$test = "Global"
```

Next, create a `Sample.ps1` script that defines the `$test` variable. In the script, use a scope modifier to refer to either the global or local versions of the `$test` variable.

In `Sample.ps1`:

```
$test = "Local"
"The local value of `$test is $test."
"The global value of `$test is $global:test."
```

When you run `Sample.ps1`, the output should resemble the following output:

```
The local value of $test is Local.
The global value of $test is Global.
```

When the script is complete, only the global value of `$test` is defined in the session.

```
PS> $test
Global
```

Example 3: Change the value of a variable in a parent scope

Unless you protect an item using the Private option or another method, you can view and change the value of a variable in a parent scope.

First, define a `$test` variable in the global scope.

```
$test = "Global"
```

Next, create a Sample.ps1 script that defines the `$test` variable. In the script, use a scope modifier to refer to either the global or local versions of the `$test` variable.

In Sample.ps1:

```
$global:test = "Local"
"The global value of `$test is $global:test."
```

When the script is complete, the global value of `$test` is changed.

```
PS> $test
```

```
Local
```

Example 4: Creating a private variable

A variable can be made private by using the `private:` scope modifier or by creating the variable with the **Option** property set to `Private`. Private variables can only be viewed or changed in the scope in which they were created.

In this example, the `ScopeExample.ps1` script creates five functions. The first function calls the next function, which creates a child scope. One of the functions has a private variable that can only be seen in the scope in which it was created.

```
PS> Get-Content ScopeExample.ps1
# Start of ScopeExample.ps1
function funcA {
    "Setting `$funcAVar1 to 'Value set in funcA'"
    $funcAVar1 = "Value set in funcA"
    funcB
}

function funcB {
    "In funcB before set -> '$funcAVar1'"
    $private:funcAVar1 = "Locally overwrite the value - child scopes
can't see me!"
    "In funcB after set  -> '$funcAVar1'"
    funcC
}

function funcC {
    "In funcC before set -> '$funcAVar1' - should be the value set in
funcA"
    $funcAVar1 = "Value set in funcC - Child scopes can see this
change."
    "In funcC after set  -> '$funcAVar1'"
    funcD
}

function funcD {
    "In funcD before set -> '$funcAVar1' - should be the value from
funcC."
    $funcAVar1 = "Value set in funcD"
    "In funcD after set  -> '$funcAVar1'"
    '-------------------'
    ShowScopes
}

function ShowScopes {
    $funcAVar1 = "Value set in ShowScopes"
    "Scope [0] (local)  `$funcAVar1 = '$(Get-Variable funcAVar1 -Scope
0 -ValueOnly)'"
    "Scope [1] (parent) `$funcAVar1 = '$(Get-Variable funcAVar1 -Scope
1 -ValueOnly)'"
    "Scope [2] (parent) `$funcAVar1 = '$(Get-Variable funcAVar1 -Scope
2 -ValueOnly)'"
    "Scope [3] (parent) `$funcAVar1 = '$(Get-Variable funcAVar1 -Scope
3 -ValueOnly)'"
    "Scope [4] (parent) `$funcAVar1 = '$(Get-Variable funcAVar1 -Scope
4 -ValueOnly)'"
}
funcA
# End of ScopeExample.ps1
PS> .\ScopeExample.ps1
```

The output shows the value of the variable in each scope. You can see that the private variable is only visible in `funcB`, the scope in which it was created.

```
Setting $funcAVar1 to 'Value set in funcA'
In funcB before set -> 'Value set in funcA'
In funcB after set  -> 'Locally overwrite the value - child scopes
can't see me!'
In funcC before set -> 'Value set in funcA' - should be the value set
in funcA
In funcC after set  -> 'Value set in funcC - Child scopes can see this
change.'
In funcD before set -> 'Value set in funcC - Child scopes can see this
change.' - should be the value from funcC.
In funcD after set  -> 'Value set in funcD'
-------------------
Scope [0] (local)  $funcAVar1 = 'Value set in ShowScopes'
Scope [1] (parent) $funcAVar1 = 'Value set in funcD'
Scope [2] (parent) $funcAVar1 = 'Value set in funcC - Child scopes can
see this change.'
Scope [3] (parent) $funcAVar1 = 'Locally overwrite the value - child
scopes can't see me!'
Scope [4] (parent) $funcAVar1 = 'Value set in funcA'
```

As shown by the output from `ShowScopes`, you can access variables from other scopes using `Get-Variable` and specifying a scope number.

Example 5: Using a local variable in a remote command

For variables in a remote command created in the local session, use the `using` scope modifier. PowerShell assumes that the variables in remote commands were created in the remote session.

The syntax is:

```
$using:<VariableName>
```

For example, the following commands create a `$Cred` variable in the local session and then use the `$Cred` variable in a remote command:

```
$Cred = Get-Credential
Invoke-Command $s {Remove-Item .\Test*.ps1 -Credential $using:Cred}
```

The `using` scope modifier was introduced in PowerShell 3.0.

See also

- about_Environment_Variables
- about_Functions
- about_Script_Blocks
- about_Variables
- ForEach-Object
- Start-ThreadJob

In this article