10 tips with PowerShell in Windows Server 2008 - Part 2

In the previous article, I have shown you some of the basic functions and tricks that can be done with PowerShell in Windows Server 2008. And this time, we will continue with part 2, which is also the end. in PowerShell series in Windows Server environment ...

TipsMake.com - In the previous article , we have introduced some basic functions and tricks that can be done with PowerShell in Windows Server 2008. And this time, we will continue with part 2, is also the last part in the **PowerShell** series in the **Windows Server** environment **:**

- Collect information of the latest 10 errors

- Resetting control on the folder

- View server uptime

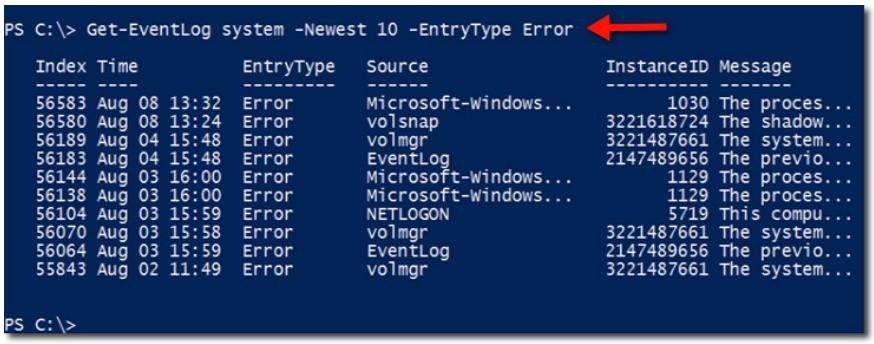
- Collect information Service Pack

- Delete unnecessary files and data

6. Collect information of the last 10 errors:

Every day, you may have to browse each log file in the system to find detailed information about the most recent error events on 1 or more computers. However, we can simplify this process by using the cmdlet **Get-EventLog** in **PowerShell** .

All you need to do here is specify the name of the log file and the corresponding component. The usual command structure will look like this:



In the screenshot above, the name of the log file is **system** , the participant component is **Error** . And so, **PowerShell** will collect all the information related to the 10 most recent errors from the system log. This command is executed on any local computer, so you will not need to specify the computer name. Note that if the above statement does not display the results as required, we can slightly edit the information according to the syntax below:

```
PS C:\> Get-EventLog system -newest 10 -Entry error | ft Timewritten,Source,EventID
,Message -wrap -auto
```

We simplify the process by converting the output of the previous statement to **ft** , with the alias **Format-Table** , and requesting the following properties: **Timewritten** , **Source** , **EventID** , and **Message** . At the same time, we add - **wrap** and - **auto** to make the results more 'easy to see'. Specifically, - the **wrap** activates the wrap mode for text, and - **auto information** is an auto-resizing feature.

And our results will be as follows:

```
TimeWritten          Source                      EventID Message
-----------          ------                      ------- -------
8/8/2011 1:32:15 PM  Microsoft-Windows-GroupPolicy    1030 The processing of Group
                                                           Policy failed. Windows
                                                           attempted to retrieve
                                                          new Group Policy settin
                                                          gs for this user or com
                                                          puter. Look in the deta
                                                          ils tab for error code
                                                          and description. Window
                                                          s will automatically re
                                                          try this operation at t
                                                          he next refresh cycle.
                                                          Computers joined to the
                                                           domain must have prope
                                                          r name resolution and n
                                                          etwork connectivity to
                                                          a domain controller for
                                                           discovery of new Group
                                                           Policy objects and set
                                                          tings. An event will be
                                                           logged when Group Poli
                                                          cy is successful.
8/8/2011 1:24:22 PM  volsnap                       36 The shadow copies of vo
                                                          lume C: were aborted be
                                                          cause the shadow copy s
                                                          torage could not grow d
                                                          ue to a user imposed li
```

Next, we will create another command, and this time the request is set to sort properties by the **Source** field, then merge them. The end result will be assigned more options to display each individual screen size, users will not have to drag down to see all:

```
PS C:\> Get-EventLog system -newest 10 -Entry error | sort Source | ft -GroupBy Sou
rce TimeWritten,EventID,Message -wrap -auto | more
```

And results:

```
   Source: EventLog

TimeWritten         EventID Message
-----------         ------- -------
8/4/2011 3:48:54 PM    6008 The previous system shutdown at 4:02:48 PM on ?8/?3/?2
                            011 was unexpected.
8/3/2011 3:59:12 PM    6008 The previous system shutdown at 3:32:46 PM on ?8/?3/?2
                            011 was unexpected.


   Source: Microsoft-Windows-GroupPolicy

TimeWritten         EventID Message
-----------         ------- -------
8/3/2011 4:00:02 PM    1129 The processing of Group Policy failed because of lack
                            of network connectivity to a domain controller. This m
                            ay be a transient condition. A success message would b
                            e generated once the machine gets connected to the dom
                            ain controller and Group Policy has succesfully proces
                            sed. If you do not see a success message for several h
                            ours, then contact your administrator.
8/3/2011 4:00:04 PM    1129 The processing of Group Policy failed because of lack
                            of network connectivity to a domain controller. This m
                            ay be a transient condition. A success message would b
                            e generated once the machine gets connected to the dom
                            ain controller and Group Policy has succesfully proces
                            sed. If you do not see a success message for several h
                            ours, then contact your administrator.
8/8/2011 1:32:15 PM    1030 The processing of Group Policy failed. Windows attempt
                            ed to retrieve new Group Policy settings for this user
                             or computer. Look in the details tab for error code a
-- More  --
```

Now is the time to include components according to the **source** . The first group of information has the same **EventLog** as the **source** , while the second group is **Microsoft-Windows-GroupPolicy.** You should note that - **More -** appears at the bottom of the screen, is responsible for informing users to press any key to view more information.

All **Get-EventLog statements** are executed on the **local** computer, but what if you want to do it on the **remote** computer?

For example, we want to see the results of the latest error report 5 that occur on some **domain controller computers** at a branch in **Chicago** . The computers named respectively **chi-dc01** and **chi-dc02** , assuming that the administrator wants to sort the results by **Machine Name.** To do this, we'll display the following properties: **Timewritten, Source, EventID, and Message.** Again, you should use the **- wrap** , **- auto** , and **more** parameters here:

```
PS C:\> Get-EventLog system -newest 5 -Entry error -comp "chi-dc01","chi-dc02" |
>> sort MachineName |
>> ft -GroupBy Machinename -prop TimeWritten,Source,EventID,Message -wrap -auto |
>> more
>>
```

And this is the result:

```
   MachineName: CHI-DC01.GLOBOMANTICS.local

TimeWritten         Source                        EventID Message
-----------         ------                        ------- -------
8/3/2011 3:58:18 PM DfsSvc                          14550 The description for Even
                                                          t ID '-1073727274' in So
                                                          urce 'DfsSvc' cannot be
                                                          found.  The local comput
                                                          er may not have the nece
                                                          ssary registry informati
                                                          on or message DLL files
                                                          to display the message,
                                                          or you may not have perm
                                                          ission to access them.
                                                          The following informatio
                                                          n is part of the event:
8/3/2011 3:57:52 PM DfsSvc                          14550 The description for Even
                                                          t ID '-1073727274' in So
                                                          urce 'DfsSvc' cannot be
                                                          found.  The local comput
                                                          er may not have the nece
                                                          ssary registry informati
                                                          on or message DLL files
                                                          to display the message,
                                                          or you may not have perm
                                                          ission to access them.
                                                          The following informatio
                                                          n is part of the event:
8/3/2011 3:58:38 PM DfsSvc                          14550 The description for Even
                                                          t ID '-1073727274' in So
                                                          urce 'DfsSvc' cannot be

-- More  --
```

7. Resetting control on the directory:

In fact, there are many cases where the **NTFS** permissions of a directory are not set up properly. And to fix this, we only need to reset the control corresponding to the **Set-Acl (Set-ACL)** cmdlet command on **PowerShell** .

The simplest way to do this is to use **Get-Acl** to collect **ACL** information from a completely normal directory, then assign it to the directory in question. Suppose that we have shared files named **sales** on the **CHI-FP01** computer, and that file has a fairly stable **ACL** . If you want to copy the **ACL** data of **sales** and then save the **$ acl** variable, then use the following command:

```
PS C:\> $acl=Get-Acl \\CHI-FP01\sales
PS C:\>
```

And here is the information inside the **ACL** :

```
PS C:\> $acl

    Directory:

Path                      Owner                  Access
----                      -----                  ------
                          BUILTIN\Administrators  CREATOR OWNER Allow  Fu...


PS C:\>
```

Do you see the **Access** attribute in the right corner? It is actually just another object, and to know the content inside, we use the command:

```
PS C:\> $acl.access
```

Results show:

```
InheritanceFlags  : ContainerInherit, ObjectInherit
PropagationFlags  : None

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : BUILTIN\Administrators
IsInherited       : False
InheritanceFlags  : ContainerInherit, ObjectInherit
PropagationFlags  : None

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : GLOBOMANTICS\Chicago Sales Managers
IsInherited       : False
InheritanceFlags  : ContainerInherit, ObjectInherit
PropagationFlags  : None

FileSystemRights  : Modify, Synchronize
AccessControlType : Allow
IdentityReference : GLOBOMANTICS\Chicago Sales Users
IsInherited       : False
InheritanceFlags  : ContainerInherit, ObjectInherit
PropagationFlags  : None

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : GLOBOMANTICS\Chicago IT
IsInherited       : False
InheritanceFlags  : ContainerInherit, ObjectInherit
PropagationFlags  : None
```

We can easily see that this is just a **collection** of many other **Access Control** components. If you just want to see the relevant information for **sales** , use the syntax:

```
PS C:\> $acl.access | where {$_.identityreference -match "Sales"}


FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : GLOBOMANTICS\Chicago Sales Managers
IsInherited       : False
InheritanceFlags  : ContainerInherit, ObjectInherit
PropagationFlags  : None

FileSystemRights  : Modify, Synchronize
AccessControlType : Allow
IdentityReference : GLOBOMANTICS\Chicago Sales Users
IsInherited       : False
InheritanceFlags  : ContainerInherit, ObjectInherit
PropagationFlags  : None


PS C:\> _
```

Next, use the same command to check the information inside the **Access** property, belonging to the newly created shared file named **chicagosales** , we won't receive anything.

```
PS C:\> (get-acl \\chi-FP01\chicagosales).Access | where {$_.identityreference -mat
ch "Sales"}
PS C:\> _    no output
```

The reason why the system does not display any information may be because the sharing mode has been set but the **NTFS** permissions are not applied correctly. The best solution is to copy **ACL** information from a stable file to a problem file. But first, we need to get **chicagosales** ' current **NTFS permissions** and save them as **XML** files. With this way, we can easily recover if an error occurs during the execution:

```
PS C:\> Get-Acl \\CHI-FP01\chicagosales | Export-Clixml .\chisales.xml
PS C:\> _
```

After that, you continue to implement and apply the **Set-Acl** command on **chicagosales** using the **$ acl** attribute **:**

```
PS C:\> Set-Acl -Path \\CHI-FP01\chicagosales  -AclObject $acl
PS C:\> _
```

To check if our implementation is successful, use the previous command to check **Sales** related information as shown below:

```
PS C:\> (get-acl \\CHI-FP01\chicagosales).Access | where {$_.identityreference -mat
ch "Sales"}


FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : GLOBOMANTICS\Chicago Sales Managers
IsInherited       : False
InheritanceFlags  : ContainerInherit, ObjectInherit
PropagationFlags  : None

FileSystemRights  : Modify, Synchronize
AccessControlType : Allow
IdentityReference : GLOBOMANTICS\Chicago Sales Users
IsInherited       : False
InheritanceFlags  : ContainerInherit, ObjectInherit
PropagationFlags  : None


PS C:\> _
```

And now **NTFS** permissions on the **chicagosales** folder are similar to **sales** .

---

8. Check server operation time:

To do this, we only need to apply the **WMI Win32_OperatingSystem** class in **PowerShell** , which can meet your needs, work directly on a **local** or **remote** machine - **remote** control. The technical property to search here is **LastBootUpTime** , but the original format is **WMI** , so we will have to convert the format - **convert** to another object that is easier to handle.

Let's start with an example of using a local computer with **Windows 7** operating system. First, save the result of **GetWmiObject** syntax into another variable named **$ wmi:**

```
PS C:\> $wmi=gwmi win32_operatingsystem
PS C:\> _
```

And from now on, we can work directly on $ **wmi** , namely **CSName - Computer Name** and **LastBootUpTime properties** :

```
PS C:\> $wmi | select CSName,LastBootUpTime

CSName                                   LastBootUpTime
------                                   --------------
CHI-WIN7-22                              20110806095247.520059-240


PS C:\>
```

As mentioned above, the **LastBootUpTime** attribute is **WMI timestamp** , which is not really useful in this case. Therefore, we will have to convert to another format, specifically here is another variable named **$ boot:**

```
PS C:\> $boot=$wmi.ConvertToDateTime($wmi.LastBootUptime)
PS C:\>
```

In this case, you should apply the **ConverToDateTime** function, which is included in the **WMI** object when using **GetWmiObject** . The parameter to pass here is the **LastBootUpTime** property of the $ **wmi** object of **WMI** . If we show the value of **$ boot** , the system will display as follows:

```
PS C:\> $boot

Saturday, August 06, 2011 9:52:47 AM


PS C:\>
```

Obviously, this information is much more useful than the original **LastBootUpTime attribute** . To find out how long the system worked, we just need to subtract $ **boot** from the current date / time - by using the **Get-Date** function **:**

```
PS C:\> (Get-Date) - $boot


Days              : 2
Hours             : 5
Minutes           : 45
Seconds           : 50
Milliseconds      : 467
Ticks             : 1935504676871
TotalDays         : 2.24016745008218
TotalHours        : 53.7640188019722
TotalMinutes      : 3225.84112811833
TotalSeconds      : 193550.4676871
TotalMilliseconds : 193550467.6871
```

The result here is another object like **TimeSpan** , if you want this information to be 'compact', then you just apply the formula to convert into string with the function **ToString ():**

```
PS C:\> ((Get-Date)-$boot).ToString()
2.05:46:11.6395621
```

Our system in this case has been active for 2 days, 5 hours, 46 minutes. The next thing we need to do here is to put the entire calculation function into a single function named **get-boot:**

```
PS C:\> Function get-boot {
>> Param([string]$computername=$env:computername)
>> Process {
>> if ($_) {$computername=$_}
>> gwmi win32_operatingsystem -computername $computername |
>> Select @{Name="Computername";Expression={$_.CSName}},
>> @{Name="LastBoot";Expression={$_.ConvertToDateTime($_.LastBootUpTime)}},
>> @{Name="Uptime";Expression={(Get-Date)-($_.ConvertToDateTime($_.LastBootUpTime)
)}}
>> }
>> }
>>
PS C:\> _
```

This **function** will have parameters to get the name of the computer, the default value will be the **local** machine:

```
>> Param([string]$computername=$env:computername)
>>
```

Besides the **Process** script to block the value of **Computer Name** , basically when any value is passed here, the $ **computername** variable will continue to pass the corresponding value into it. Besides, other components will be used if there is no value or equivalent to computername:

```
>> Process {
>> if ($_) {$computername=$_}
>>
```

Within the **Process** script is the **GetWmiObject** formula - applied to determine the name of the remote computer:

```
>> gwmi win32_operatingsystem -computername $computername |
>>
```

On the other hand, we have a few hashtable pairs here. The **CSName** attribute is not easy to use, so it needs to be replaced with the newly created value named **Computername** . And there is another attribute - **LastBoot** , which stores the **LastBootUpTime** value via **ConvertToDateTime ()** . Finally, the **Uptime** - attribute is in the form of a **TimeSpan** object, corresponding to the period of time the system has run:

```
>> Select @{Name="Computername";Expression={$_.CSName}},
>> @{Name="LastBoot";Expression={$_.ConvertToDateTime($_.LastBootUpTime)}},
>> @{Name="Uptime";Expression={(Get-Date)-($_.ConvertToDateTime($_.LastBootUpTime)
)}}
>> _
```

If we test on the local (for example, do not specify the computer name value), the system will apply directly to the local computer name value in the default mode. Examples are as follows:

```
PS C:\> Get-Boot

Computername            LastBoot                Uptime
------------            --------                ------
CHI-WIN7-22             8/6/2011 9:52:47 AM     2.05:48:02.5458121


PS C:\>
```

With what we have shown in part 1 of the article, in the section Turn off or restart the server, you can save the name of the server in a text file, applicable to the ping components, later. then pass those values ??into the function **get-boot:**

```
PS C:\> gc c:\work\servers.txt | where {Test-Connection $_ -quiet -count 2} | get-b
oot

Computername               LastBoot                  Uptime
------------               --------                  ------
CHI-DC01                   8/6/2011 9:37:37 AM       2.06:03:40.0473101
CHI-DC02                   8/6/2011 9:37:45 AM       2.06:03:32.9273411
CHI-FP01                   8/8/2011 1:37:49 PM       02:03:30.1127461
CHI-DB01                   8/8/2011 10:20:59 AM      05:20:22.0164441
CHI-EX01                   8/8/2011 10:20:47 AM      05:20:34.8494961
```

The screenshot above shows the list of remote computers, the total uptime and the most recent reboot.

9. Collect information Service Pack:

In fact, there are many reasons we need to know about the service pack information of the server. And to do this, you'll apply **WMI** and the **Win32_Operating System** class **,** the attributes used here include: **ServicePackMajorVersion** (with key values ??of 0, 1 or 2), **ServicePackMinorVersion** , **CSDVersion** (usually shown below **Service Pack 1**) .

Specifically, we will start by using **Get-WmiObject** and **Win32_operatingsystem** class to collect information in the system from **PowerShell** . The most important attributes here are: **CSName** (computer name), **Caption** (operating system), **CSDversion** and **ServicePackMajorVersion** .

The main syntax used here is of the form:

```
PS C:\> Get-WmiObject -class win32_operatingsystem |
>> select CSName,Caption,CSDVersion,ServicePackMajorVersion
>>

CSName               Caption              CSDVersion          ServicePackMajorVer
                                                                          sion
------               -------              ----------          -------------------
CHI-WIN7-22          Microsoft Windows...                                       0


PS C:\>
```

Looking at the screenshot above, we can see that this **Windows 7** operating system does not use any **Service Pack** version, so the **ServicePackMajorVersion** value is **0** , and the **CSDVersion** part is left blank. But our main job here is to create a new function named **Get-SP** , the first thing to do is to get the value of the computer name, the default is local machine:

```
PS C:\> Function Get-SP {
>> Param([string]$computername=$env:computername)
>>
```

Next is the **Process** script, if a computer name value is passed, the $ **computername** variable will initialize the value for that object. The main part of this syntax is the **Get-Wmiobject / Win32_operatingsystem class.**

Then, you need to deploy a pair of **Hashtable** , here we select the **CSName** property and assign it to another property named **ComputerName** . Similarly the attribute **Caption - Operating System** . Instead of using **CSDVersion** , we use **SPName** , finally **Version** replaces **ServicePackMajorVersion** .

```
>> Process {
>> if ($_) {$computername=$_}
>> Get-WmiObject -class win32_operatingsystem -computername $computername |
>> Select @{Name="Computername";Expression={$_.CSName}},
>> @{Name="OperatingSystem";Expression={$_.Caption}},
>> @{Name="SPName";Expression={$_.CSDVersion}},
>> @{Name="Version";Expression={$_.ServicePackMajorVersion}}
>> }
>> }
>>
```

And this is our full function when operating on the local:

```
PS C:\> Function Get-SP {
>> Param([string]$computername=$env:computername)
>> Process {
>> if ($_) {$computername=$_}
>> Get-WmiObject -class win32_operatingsystem -computername $computername |
>> Select @{Name="Computername";Expression={$_.CSName}},
>> @{Name="OperatingSystem";Expression={$_.Caption}},
>> @{Name="SPName";Expression={$_.CSDVersion}},
>> @{Name="Version";Expression={$_.ServicePackMajorVersion}}
>> }
>> }
>>
PS C:\> get-sp

Computername         OperatingSystem      SPName                          Version
------------         ---------------      ------                          -------
CHI-WIN7-22          Microsoft Windows...                                       0


PS C:\>
```

When we're done, we're ready to filter the list of computer names from the text file, our rest is just a ping issue and pass the corresponding computer name parameter to the newly created **get-sp** function. Our results here will look like this:

```
PS C:\> gc c:\work\servers.txt | where {Test-Connection $_ -quiet -count 2} | get-s
p | ft -auto

Computername OperatingSystem                              SPName         Version
------------ ---------------                              ------         -------
CHI-DC01     Microsoft Windows Server 2008 R2 Enterprise  Service Pack 1       1
CHI-DC02     Microsoft Windows Server 2008 R2 Standard                         0
CHI-FP01     Microsoft Windows Server 2008 R2 Datacenter  Service Pack 1       1
CHI-DB01     Microsoft Windows Server 2008 R2 Enterprise  Service Pack 1       1
CHI-EX01     Microsoft Windows Server 2008 R2 Enterprise  Service Pack 1       1


PS C:\> _
```

You can see that the **CHI-DC02** computer lacks **Service Pack 1** information, which was released for **Server 2008 R2.**

10. Delete unnecessary files in the system:

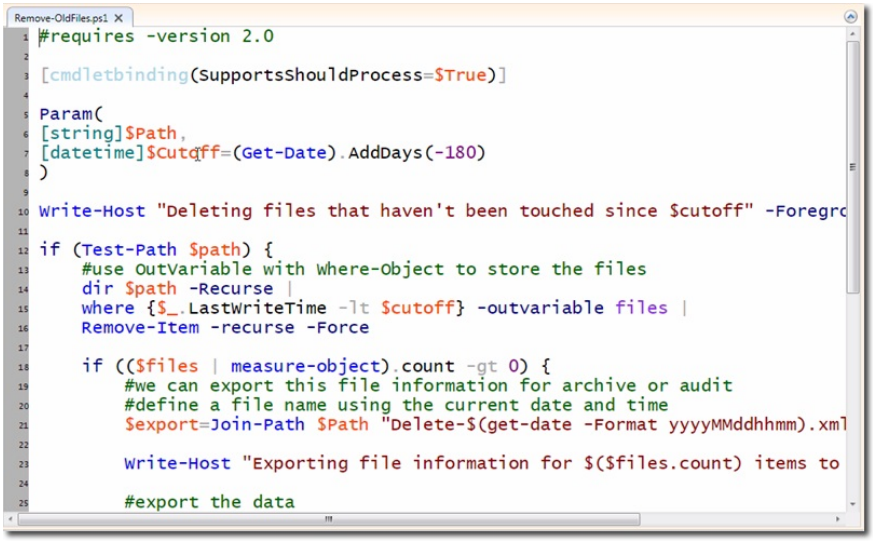
To do this, we will use the cmdlet **Get-ChildItem** , with the corresponding alias dir. Technically. The best way is to compare the **LastWriteTime** property of the file or directory within a certain time period. If **LastWriteTime** exceeds a certain time, it is the file to delete.

While **LastWriteTime** specifies the final file editing time, and **LastAccessTime** can be defined or changed by the application in the system, such as security software or defragment program - **Defragment** . Therefore, **LastAccessTime is** not really accurate at the time of using the file and is determined whether the action is performed by the user or the software.

Once you have found the files you want to remove, you can turn all of them into **Remove-Item** . Because **Remove-Item** syntax supports - **WhatIf** and - **Confirm** , so we will execute the command with those

parameters, save the result to a text file, and finally decide to apply the delete command with the file list .

However, the more optimal way is still the use of script. With this method, users can assign additional functions to save the log file and support whatif to confirm whether you really want to delete the file. Below is an example of a complete sample function called **Remove-OldFiles.ps1** . And this is part of the code:



This can be considered an advanced code, with the cmdlet **binding** structure set to **SupportShouldProcess = $ True** . That means, if the user specifies what - **whenif** scripts run, **whatif** - **whatif** will be chosen by **Remove-Item** (because there is a support mechanism - **whatif**).

Please note that the parameter is named $ **Cutoff** - with the initial default value of 180 days from the current time, but the user can still change and be considered an object of the date format - Other time in the system.

The body of the code:

```
dir $path -Recurse |
where {$_.LastWriteTime -lt $cutoff} -outvariable files |
Remove-Item -recurse -Force
```

This structure will list the recursive list of the specified path, and it will also find all files with the **LastWriteTime** shorter than the $ **cutoff** , then save the list as a The name is the file using the generic parameter - **outvariable** . And finally, any file with the **where** attribute selected is pushed to **Remove-Item.**

Assuming that the system has found files that meet the deletion criteria, the **XML** file will be created and stored in the current directory, and the file will be in the form **Delete-yyyyMMddhhmm.xml,** with **yyyyMMddhhmm** being the current **timestamp** , format - **year** - **month** - **day** - **hour** - **minute.** And this is the main code of the above function:

```
if (($files | measure-object).count -gt 0) {
    #we can export this file information for archive or audit
    #define a file name using the current date and time
    $export=Join-Path $Path "Delete-$(get-date -Format yyyyMMddhhmm).xml

    Write-Host "Exporting file information for $($files.count) items to

    #export the data
    $files | export-clixml $export
}
```

Next, we will run that code, assuming that the user wants to delete the file server's public directory on the **Chicago** remote computer - named **chi-fp01** , another requirement is to delete all the files long period of cutoff days. And this is the main statement, accompanied by a notification about the result:

```
PS C:\> S:\Remove-OldFiles.ps1 -path "\\chi-fp01\public" -Cutoff $cut
Deleting files that haven't been touched since 11/11/2010 17:09:16
Exporting file information for 90 items to \\chi-fp01\public\Delete-201108080510.xml.
PS C:\>
```

Note that we can see the path of the created **XML** file, for example there have been 4 files deleted. And to do this, please import the **XML** file content into the $ **data** variable and see the first 4 components in $ **data** :

```
PS C:\> $data=import-clixml \\chi-fp01\public\Delete-201108080510.xml
PS C:\> $data[0..3]


    Directory: \\chi-fp01\public


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         5/19/2007   9:52 AM     115712 0708SchoolCalendarTABLE.doc
-a---        11/15/2007   3:07 PM       4535 alias.csv
-a---        10/27/2004   1:38 PM       1298 csv.err
-a---        12/17/2006   5:03 PM      42496 DearMaria.doc


PS C:\> _
```

Good luck!

You finished reading the article "**10 tips with PowerShell in Windows Server 2008 - Part 2**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.