PowerShell: часто задаваемые вопросы

Продолжение

Василий Гусев

В своей предыдущей статье [1] я уже ответил на многие популярные вопросы о PowerShell и о некоторых моментах работы с ним. Но, конечно, в рамках одной статьи сложно рассказать обо всём, поэтому продолжим.

Как вывести в строке какие-либо переменные или свойства объекта?

Думаю, все знают, как в PowerShell вывести строку на экран:

```
PS> "Hello world!"
Hello world!
```

И как объединить несколько строк или переменных в одну строку тоже:

```
PS> $w = "World"
PS> "Hello " + $w + "!"
Hello world!
```

Но все это можно делать куда более эффективно и удобно. Для того чтобы вставить значение переменной в строку, её достаточно поместить внутрь этой строки:

```
PS> "Hello $w!"
Hello world!
```

Логично, не правда ли? Но зачастую в строку надо вставить не простую переменную, а свойство какого-либо объекта, и тут возникает проблема:

```
PS> $file = Get-Item C:\test.zip
PS> "Размер файла $file составляет $file.Length байт"
Размер файла C:\test.zip составляет C:\test.zip.Length байт
```

Здесь я поместил в переменную $file объект, представляющий файл test.ps1, и затем пару раз упомянул его в строке. В первом вхождении переменная $file была преобразована в полный путь к файлу (всё равно как если бы мы выполнили метод $file.toString()). А во втором случае… Произошло то же самое! PowerShell посчитал, что .Length – это часть строки, не имеющая никакого отношения к переменной $file. Его тоже можно понять – вдруг пользователь пропустил пробел между предложениями? Но что же делать, если нам всё-таки надо поместить значение свойства в строку, а использовать временные переменные или использовать конкатенацию с помощью кучи плюсов не хочется? В таком случае нужно использовать конструкцию $(). Её можно вставить в строку, а между скобок поместить любое выражение. Это выражение будет выполнено, и его результат будет подставлен в строчку:

```
PS> "Размер файла $file составляет $($file.Length) байт"
Размер файла C:\test.zip составляет 1364964 байт
```

Неплохо? Но это еще не всё. Я не зря сказал, что внутрь $() можно поместить любое выражение, это действительно так:

```
PS> "Размер: $([Math]::Round($file.Length / 1kb)) килобайт"
Размер: 1333 килобайт
```

Тут я поделил размер файла в байтах на встроенную константу 1kb, и затем, воспользовавшись методом [Math]::Round() из .Net Framework, округлил полученный результат до целых. Есть и другой способ – воспользоваться оператором форматирования -f.

```
PS> "Размер: {0:n3} мегабайт" -f ($file.Length / 1mb)
Размер: 1,302 мегабайт
```

Те, кто знаком с программированием на языке C#, наверняка обрадуются знакомому синтаксису (http://msdn2.microsoft.com/en-us/library/fbxft59x.aspx). Для остальных же поясню:

Конструкция {0:n3} состоит из нескольких частей, первая из них – «0», это индекс элемента во втором операнде. В данном случае он один, но можно указать и несколько элементов, и при расстановке их внутри строки будет использоваться их порядковый номер, начиная с 0 у первого. Вторая часть конструкции – «n», указывает на то, что значение необходимо отформатировать как число (number), ну а следующая за нею цифра (в данном случае «3») определяет количество знаков после запятой.

Естественно, возможности оператора форматирования -f не ограничиваются обрезкой лишних знаков после запятой, к примеру, он обладает огромными возможностями форматирования дат. В следующем примере с помощью -f я получу путь к файлу, составленный из текущего каталога и сегодняшней даты:

```
PS> "{0}\{1:yyyy-MM-dd}.bak" -f $pwd, (Get-Date)
C:\backups\2008-04-12.bak
```

А какие в PowerShell маскирующие символы?

К счастью, этим маскирующим символом не является обратный слеш «\», как во многих языках программирования. Если бы так было в PowerShell, мы бы замучились набирать пути файловой системы, повторяя каждый слеш дважды.

В PowerShell роль маскирующего символа выполняет «`» – апостроф, символ, расположенный на большинстве клавиатур на клавише «Ё», под тильдой. С его помощью можно маскировать любые символы:

```
PS> "`$pwd = $pwd"
$pwd = C:\root
```

Здесь я замаскировал символ «$» в первом упоминании переменной, и она не была преобразована в значение. Еще можно использовать маскирующий символ для обозначения специальных символов. Так, например, «`n» будет означать переход на следующую строку:

```
PS> "Первая строка`nВторая строка"
Первая строка
Вторая строка
```

Вот некоторые из часто употребляемых специальных символов:

- **`n** – новая строка.
- **`a (alert)** – этот символ заставляет спикер компьютера издавать писк. Бывает полезно для того, чтобы привлечь внимание пользователя.
- **`t** – символ табуляции.

В чем отличия между разными типами кавычек в PowerShell?

Начнем с самых простых и популярных – двойных кавычек. В PowerShell, как и во множестве других языков, они служат для ограничения и обозначения строк. Все знают, что если набрать в командной строке PowerShell текст в кавычках, то он будет выведен на экран. Ну и, конечно, можно присвоить это текстовое значение переменной. Но, кроме того, точно так же можно работать и с многострочными текстами. Если, не закрыв кавычек, нажать <Enter>, то командная строка PowerShell переведет курсор на новую строку, и продолжит ожидание ввода. Так будет продолжаться, пока вы не закроете кавычки:

```
PS> $hw = "Hello
>> World!"
>>
PS>
```

Символы «>>» тут лишь означают, что ввод продолжается, в саму переменную они помещены не будут. Когда вы захотите использовать такую конструкцию в скрипте, просто делайте переносы строки:

```
$SqlCommand = "BACKUP DATABASE [$Base]
TO  DISK = '$Path'
WITH INIT"
```

Не правда ли, намного проще, чем создание многострочных переменных, например, в VBScript?

Перейдем ко второму типу кавычек, к одинарным. Их основное отличие от двойных – это то, что, если поместить внутри них название переменной, оно не будет преобразовано в её значение. Это хорошо видно на следующем примере:

```
PS> $var = '$pwd = ' + "'$pwd'"
PS> $var
$pwd = 'c:\root'
```

Имя переменной внутри одинарных кавычек осталось неизменным, а внутри двойных кавычек было подставлено значение переменной вместо её имени. А еще в этом примере видно, что один тип кавычек можно без проблем использовать внутри других кавычек, не волнуясь о какой-либо маскировке. То есть если вам необходимо составить строчку, внутри которой множество одинарных кавычек (к примеру, фильтр для WMI), то удобнее будет заключить эту строку в двойные кавычки, и наоборот.

Второе отличие одинарных кавычек от двойных – это игнорирование символа маскировки – «`»:

```
PS> 'Первая строка`nВторая строка'
Первая строка`nВторая строка
```

Впрочем, при необходимости можно использовать символ одинарной кавычки внутри строки, нужно повторить его дважды:

```
PS> 'Одинарная кавычка '' среди собратьев'
Одинарная кавычка ' среди собратьев
```

Но что делать, если в нашей строке используется множество кавычек обоих видов, например, если нужно поместить в переменную кусок кода PowerShell или SQL? Для та-

кого случая предусмотрена специальная разновидность кавычек, специально предназначенная для многострочных текстов (так же называемая HereString):

```
PS> $MyCode = @'
>> $Proc = Get-Process explorer
>> $Message = 'Переменная $Proc содержит сведения о процессе
>> например, в "$Proc.Path" содержится ' + "'$Path'."
>> '@
>>
PS> $MyCode
$Proc = Get-Process explorer
$Message = 'Переменная $Proc содержит сведения о процессе
например, в "$Proc.Path" содержится ' + "'$Path'."
```

Разумеется, есть и вариант HereString для двойных кавычек, в нём переменные преобразовываются в свои значения.

Как посчитать количество возвращенных командой объектов?

Очень часто встречающийся вопрос. Большинство командлетов в PowerShell в качестве результата возвращают несколько объектов, объединенных в массив. Ну и, разумеется, очень часто хочется посчитать количество этих результатов. Сделать это очень просто, достаточно приставить в конец конвейера командлет Measure-Object. Например, вот так можно посчитать количество журналов событий в системе:

```
PS> Get-EventLog -List | Measure-Object
Count    : 11
Average  :
Sum      :
Maximum  :
Minimum  :
Property :
```

Думаю, многим интересно, что это за строки – Average, Sum и т. д. Дело в том, что возможности Measure-Object не ограничиваются подсчетом количества элементов (хотя по умолчанию делает только это). Он может производить и некоторые другие вычисления, причем не только над самими объектами, но и над их свойствами:

```
PS> Get-Process | Measure-Object -Property WS -Sum -Average
Count    : 77
Average  : 13063952,6233766
Sum      : 1005924352
Maximum  :
Minimum  :
Property : WS
```

Так мы получили данные об используемой памяти (WorkingSet) – среднее значение на процесс и сумму по всем процессам.

Но вернёмся к нашему вопросу. Кроме использования Measure-Object, есть и другой способ, зачастую более удобный. У всех массивов в PowerShell есть свойство .Count, в котором и содержится количество элементов массива. Вот пример, как его можно использовать:

```
PS> $Shares = Get-WmiObject Win32_Share
PS> $shares.Count
5
```

Конечно, можно обойтись и без временной переменной, достаточно заключить выражение в скобки:

```
PS> (Get-WmiObject Win32_Share).count
5
PS> (Get-Process | where {$_.path -like "c:\win*"}).count
46
```

Здорово выполнять команды интерактивно в консоли или запускать из неё скрипты. Но для многих административных задач необходимо запускать скрипты из планировщика заданий. Как это сделать?

Сначала еще раз напомню про необходимость разрешить в системе выполнение неподписанных скриптов. Хоть это и всем известный шаг, при переходе в производственную среду о нем многие забывают. Либо, если вы серьезно относитесь к безопасности, стоит подумать о том, чтобы подписывать скрипты PowerShell, выполняющиеся на серверах. Обо всём этом можно подробнее прочитать, выполнив команду:

```
PS> Get-Help About_Signing
```

Ну а чтобы вызвать скрипт из планировщика, надо лишь в качестве запускаемой программы указать PowerShell.exe (полный путь – C:\Windows\system32\WindowsPowerShell\v1.0\powershell.exe), а в качестве аргумента – путь к файлу скрипта.

Если выполнить PowerShell.exe с ключом «/?», то можно узнать и о других полезных аргументах. Я опишу лишь ключи, полезные для автоматизированного запуска скриптов:

- **-NoLogo** – не будет выводиться приветственная строка;
- **-NoProfile** – не загружать профили PowerShell;
- **-NonInteractive** – не выдавать запросов пользователю, т.е. при вызове, например Read-Line, произойдет ошибка, и выполнение скрипта будет продолжено;
- **-Command** – указывается код PowerShell, выполняемый при запуске. Кстати, тут может быть не просто скрипт, но и полноценная команда.

У командлетов зачастую весьма длинные названия аргументов. Но иногда они вообще не указываются. Как это работает?

Возьмем в качестве образца командлет Set-Content. Полный его синтаксис предполагает следующую конструкцию:

```
PS> Set-Content -Path test.txt -Value "123" -Verbose
```

Тут мы говорим командлету поместить значение «123» в файл test.txt. Но обязательно ли указывать названия параметров Path и Value?

Если выполнить команду:

```
PS> Get-Help Set-Content -Parameter Path
```

то можно увидеть, что у этого параметра свойство Position равно 1. Это же свойство у параметра Value того же командлета равно 2. Это означает, что если мы не будем указывать имен параметров, то PowerShell посчитает первый аргумент значением для Path, а второй для Value. То есть можно выполнять команду вот так:

```
PS> Set-Content test.txt "123"
```

Зачем же тогда вообще может понадобиться указывать имена параметров, если всё работает и без них? Ну, во-первых, скрипт с полными именами параметров будет куда читабельнее, чем без них. Те, кто хоть раз ломал голову над своим скриптом многолетней давности, оценят. А во-вторых, если мы указываем названия параметров, – нам не нужно помнить их порядок, он может быть любым:

```
PS> Set-Content -Value "123" -Path test.txt
```

Но и в этом случае незачем писать полные имена параметров. Возьмем теперь команду с действительно длинными параметрами – Write-Host. У неё есть два параметра, позволяющие задать цвета выводимого текста и его фона: ForegroundColor и BackgroundColor. И свойство Position у этих параметров равно «Named» (в чем можно убедиться, выполнив команду:

```
PS> Get-Help Write-Host -Parameter ForegroundColor)
```

Named в данном случае означает, что для использования параметра необходимо указать его имя, вариант с помещением аргументов в правильном порядке не сработает. Но и полное имя указывать не обязательно, достаточно указать лишь первые несколько букв, чтобы PowerShell смог отличить имя параметра от остальных:

```
PS> Write-Host test -f red -b blue
test
```

Впрочем, чтобы было несколько понятнее, можно написать и так:

```
PS> Write-Host test -fore red -back blue
test
```

Но если мы, например, попробуем найти все команды, работающие с процессами, используя Get-Command, и вместо параметра -Noun укажем -n, то нас ждет сообщение об ошибке:

```
PS> Get-Command -n process
Get-Command : Не удается обработать параметр, так как имя
параметра "n" неоднозначно. Возможные совпадения: -Name -Noun.
В строка:1 знак:12
+ Get-Command  <<<< -n process
```

Дело в том, что в этом случае параметры отличаются со второй буквы, и для того чтобы PowerShell смог разобраться, какой параметр подразумевается, придется указывать на одну букву больше. В данном случае будет достаточно использовать -no.

Как получить значения параметров ключа реестра или, наоборот, задать их?

Несмотря на то что работа с реестром из PowerShell кажется очень простой, всё же существуют некоторые тонкости. Думаю, всем уже известно, что в PowerShell используется система так называемых «поставщиков» (provider), позволяющих работать с иерархическими системами хранения данных, как с обычной файловой системой (и даже автодополнение с помощью клавиши табуляции там тоже работает). И реестр как раз представлен в виде такого провайдера. Это дает возможность использовать для навигации по нему всё те же команды, как и для файловой системы: dir (Get-ChildItem), cd (Set-Location) или pwd (Get-Location). Но только этих команд для работы с реестром будет недостаточно.

Например, если нам понадобится посмотреть список автоматически запускаемых программ из HKCU:\Software\Microsoft\Windows\CurrentVersion\run, то одним Dir не обойтись. Дело в том, что параметры ключа не являются дочерними элементами по отношению к ключу. Они представляют собой его свойства, и для того чтобы получить их список, придётся использовать команду Get-ItemProperty (или её псевдоним – «gp»):

```
PS> gp HKCU:\Software\Microsoft\Windows\CurrentVersion\Run
Sidebar     : C:\Program Files\Windows Sidebar\sidebar.exe /autoRun
MsnMsgr     : "C:\Program Files\Windows Live\Messenger\MsnMsgr.Exe" /background
FolderShare : "C:\Program Files\FolderShare\FolderShare.exe" /background
Skype       : "C:\Program Files\Skype\Phone\Skype.exe" /nosplash /minimized
WMPNSCFG    : C:\Program Files\Windows Media Player\WMPNSCFG.exe
```

PowerShell попытается самостоятельно подобрать наилучший метод форматирования, основываясь на количестве свойств объекта. Так, если у вас в этом ключе реестра менее 5 параметров, то они будут выведены в виде таблицы с параметрами в роли столбцов. Чтобы этого избежать, следует перенаправить вывод в командлет Format-List, используя конвейер. Для создания параметра используется командлет New-ItemProperty или Set-ItemProperty (псевдоним – «sp»):

```
PS> cd HKCU:\Software\Microsoft\Windows\CurrentVersion\Run
PS> sp  -Path . -Name "Моя утилита" -Value "c:\ myutil.exe"
```

Тут я сначала установил в качестве текущего каталога ключ реестра, использовав cd (Set-Location). Затем с помощью командлета Set-ItemProperty создал параметр «Моя утилита» со значением «c:\myutil.exe» в текущем ключе (точка в качестве пути обозначает текущий каталог). Теперь можно проверить результат, снова использовав Get-ItemProperty, но на этот раз, указав ему конкретное свойство:

```
PS> gp . "Моя утилита" | Format-List
Моя утилита : c:\utils\myutil.exe
```

Ну и для завершения примера удалим этот созданный ключ. Не сложно догадаться, что для этого понадобится командлет «Remove-ItemProperty (“rp”)»:

```
PS> rp . "Моя утилита"
```

Как импортировать данные из Excel или, наоборот, поместить данные из PowerShell в Excel?

К сожалению, с PowerShell не поставляются командлеты для непосредственного импорта и экспорта файлов XLS. Но выход есть, и даже не один. Можно использовать для обмена данными с Excel файлы с разделителями запятыми – csv (Comma Separated File). По умолчанию этот формат даже открывается с помощью Excel и, разумеется, он может в него сохранять. Файлы CSV импортируются и экспортируются из PowerShell с помощью командлетов Import-Csv и Export-Csv соответственно.

Интересный момент – при импорте из csv-файла данные из первой его строки будут считаться заголовками, и в результате будут созданы объекты с такими же названиями свойств. Предположим, у нас есть файл следующего вида:

```
Имя,Отчество,Фамилия
Иван,Иванович,Иванов
Пётр,Петрович,Петров
Сидор,Сидорович,Сидоров
```

Импортировав этот файл, мы получим массив из трёх объектов, обладающих свойствами «Имя», «Отчество», и «Фамилия»:

```
PS> $fio = Import-Csv fio.csv
PS> $fio[0]

Имя  Отчество Фамилия
---  -------- -------
Иван Иванович Иванов

PS> $fio | foreach {$_.Фамилия}

Иванов
Петров
Сидоров
```

Ну и, конечно, если очень хочется работать напрямую с файлами XLS и XLSX, то можно воспользоваться сторонними командлетами, например, бесплатной оснасткой PowerData, которую можно скачать по адресу http://www.ultimate-projects.ru. Кроме командлетов Import-Excel и Export-Excel, в комплект входит Invoke-SQL для выполнения SQL-запросов и получения результатов в виде объектов PowerShell.

Каким образом в PowerShell можно перехватить ошибку?

Хорошие возможности в области обработки ошибок, несомненно, являются очень важным фактором для скриптового языка при применении в рабочем окружении. К счастью, у PowerShell в этом плане всё обстоит прекрасно. Есть и автоматические параметры для всех командлетов – ErrorAction и ErrorVariable, позволяющие определить поведение команды в случае ошибки и поместить объект ошибки в указанную переменную. Присутствуют специальные переменные – $ErrorActionPreference (глобально задает реакцию на ошибки) и $Error, содержащая массив последних произошедших ошибок (самая последняя $Error[0]). Но одним из самых полезных средств, конечно, является ключевое слово «trap». После этого слова задается скриптовый блок, который будет выполнен в случае ошибки. Кроме этого, в том же блоке можно обратиться к объекту ошибки (который внутри этого блока будет находится в переменной $_) и указать дальнейшие действия – break (прервать выполнение) или continue (продолжить выполнение скрипта дальше).

```
PS> trap {echo "Ошибка: $_"; break}; 1; 2/$null; 3

1
Ошибка: Попытка деления на нуль.
Попытка деления на нуль.
At line:1 char:39
+ trap {echo "Ошибка: $_"; break}; 1; 2/ <<<< $null; 3
```

Как видно из примера, после того как произошла ошибка деления на ноль, отработал код, указанный после ключевого слова trap, и затем выполнение скрипта было прервано. В случае же если указать continue, то после ошибки будут выполнены последующие команды:

```
PS> trap {echo "Ошибка: $_"; continue}; 1; 2/$null; 3

1
Ошибка: Попытка деления на нуль.
3
```

Trap очень удобен для применения в скриптах. Его можно поместить, к примеру, в начале файла, применив в скриптовом блоке командлет Export-Clixml для сохранения объекта ошибки в XML-файл:

```
trap {$_ | Export-Clixml Error.xml; stop}
```

Затем, при анализе причин проблемы, можно загрузить ошибку из этого файла в объект и детально разобраться в причинах:

```
PS> $Err = Import-Clixml Error.xml
PS> $Err

Copy-Item : Не найдено сетевое имя.
At line:1 char:10
+ copy-item <<<<  file.txt \\server\share

PS> $Err.InvocationInfo

MyCommand             : Copy-Item
CommandLineParameters : {Destination, Path}
ScriptLineNumber      : 1
OffsetInLine          : 10
ScriptName            :
Line                  : copy-item file.txt \\server\share
PositionMessage       :
                        At line:1 char:10
                        + copy-item <<<<  file.txt \\server\share
InvocationName        : copy-item
PipelineLength        : 1
PipelinePosition      : 1
ExpectingInput        : False
CommandOrigin         : Runspace
```

Жду новых вопросов на адрес. Ну и, конечно, заходите на мой блог – http://xaegr.wordpress.com.

Помните! Никакой FAQ не заменит чтения документации, так что для использования всей мощи PowerShell, надо знать команду Get-Help, и ознакомиться с содержимым прилагающейся к PowerShell документации.

1. Гусев В. PowerShell: часто задаваемые вопросы. //Системный администратор, №3, 2008 г. – С. 16-22.