



# Эффективное программирование в Windows PowerShell

Разбираться в Windows PowerShell и получать от него больше.

<http://vam.in.ua>

Keith Hill Copyright © 2007-2009

3/8/2009

## Содержание

Введение.....	1
Часть 1: Четыре командлета - ключи, открывающие PowerShell.....	1
Ключ #1: Get-Command.....	1
Ключ #2: Get-Help.....	2
Ключ #3: Get-Member.....	5
Ключ #4: Get-PSDrive.....	7
Дополнение для PowerShell 2.0.....	8
Часть 2: Понимание вывода объектов.....	9
Вывод - это всегда объект .NET .....	9
Функция возвращает все, что не попало в поток вывода.....	10
Другие типы вывода, которые не могут быть захвачены.....	13
Часть 3: Как объекты передаются по конвейеру.....	14
Часть 4: Разнообразие вывода - скаляры, коллекции и пустые наборы - о, боже!.....	17
Скаляры.....	17
Работа с коллекциями.....	18
Работа с пустыми наборами.....	19
Часть 5: Используйте объекты, Люк. Используйте объекты!.....	21
Часть 6: Как форматируется вывод.....	25
Часть 7: Режимы синтаксического разбора PowerShell.....	35
Часть 8: Параметры привязки элементов конвейера ByPropertyName (по имени).....	40
Часть 9: Параметры привязки элементов конвейера ByValue (по значению).....	43
Часть 10: Регулярные выражения – один из мощнейших инструментов PowerShell.....	48
Дополнение для PowerShell 2.0.....	49
Часть 11: Сравнение массивов.....	50
Часть 12: Старайтесь использовать Set-PSDebug -Strict в своих сценариях .....	53
Примечание для PowerShell 2.0.....	54
Часть 13: Комментирование строк в файле сценария.....	55
Дополнение для PowerShell 2.0.....	56
Дополнительные материалы.....	56

## Введение

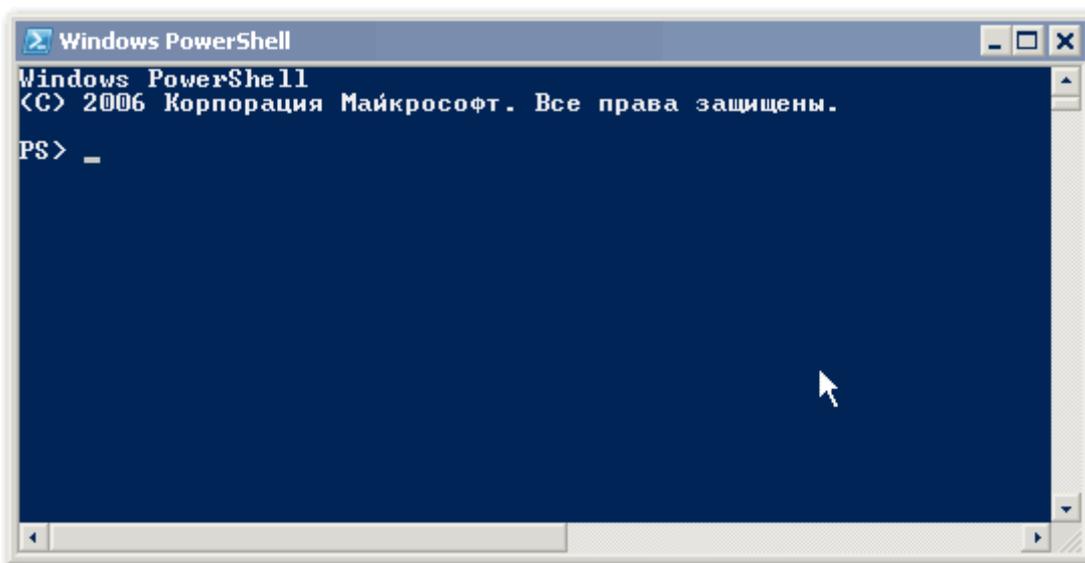
Я большой поклонник серии книг по эффективному программированию, таких как *Effective COM* и *Effective XML*. Не пытаюсь быть слишком самонадеянным, я хочу продемонстрировать некоторые приёмы, накопленные мной за последние пару лет использования Windows PowerShell в интерактивном режиме и написания производственных и тестовых скриптов. Эти приемы были созданы на PowerShell 1.0. Там, где это необходимо, добавлен раздел *Обновление для PowerShell 2.0*, в котором поясняются изменения, необходимые с предстоящим выходом версии 2.0. В заключение, ряд примеров иллюстрирует повышение функциональности PowerShell с помощью расширений, разработанных сообществом PowerShell Community Extensions. Эти расширения можно загрузить с <http://www.codeplex.com/PowerShellCX>.

## Часть 1: Четыре командлета - ключи, открывающие PowerShell

Эта часть относится скорее к основам PowerShell, чем к его эффективному применению. Тем не менее, эти четыре командлета имеют жизненно важное значение для выяснения того, что может PowerShell, и делает их заслуживающими внимания. Эти четыре командлета - то, что вы должны выучить в первую очередь. Они просты в использовании, но помогут вам начать использовать PowerShell эффективно.

### Ключ #1: Get-Command

Этот командлет - лекарство от пустоты, приглашения PowerShell в никуда. В самом деле, вы только что установили PowerShell, запустили его, и теперь любуетесь этим:



А что дальше-то? Многие приложения страдают этим - "пустой экран смерти". Вы загрузили приложение, установили его, запустили - и теперь перед вами пустая рамка окна или документ, в котором ничего нет. Зачастую нет ничего, что могло бы вам подсказать, как начать пользоваться новым приложением. Командлет *Get-Command* позволит вам начать работу с PowerShell, показав список всех доступных команд. Это касается и ваших старых консольных приложений, пакетных файлов, сценариев VBScript и т. д. По существу, любой исполняемый файл может быть запущен из PowerShell. Конечно же, вы устанавливали PowerShell не только для того, чтобы запускать свои старые скрипты и приложения. Вы хотите узнать, что может PowerShell. Попробуйте вот так:

```
PS> Get-Command
CommandType      Name                Definition
-----
Cmdlet           Add-Content        Add-Content [-Path] <String[]> [-Value] <Object[...
...
```

По умолчанию, *Get-Command* выводит список всех командлетов, предоставляемых PowerShell. Отметим, что *Get-Command* является одним из них. *Get-Command* может предоставить и больше информации, однако как заставить его сделать это? Чтобы выяснить это, мы перейдем ко второй команде, с которой вам необходимо познакомиться, и которая очень часто используется в PowerShell.

## Ключ #2: Get-Help

Командлет *Get-Help* предоставляет справочные материалы по различным разделам, включая то, что делает определенный командлет, какие параметры он может иметь и обычно содержит примеры использования команд. Он также предоставляет общую справочную информацию, например, о масках в именах файлов или об операторах. Допустим, вы хотите посмотреть все разделы справки PowerShell. Это легко, достаточно ввести:

```
PS> Get-Help *
```

Name	Category	Synopsis
ac	Alias	Add-Content
asnp	Alias	Add-PSSnapin
...		
Get-Command	Cmdlet	Возвращает базовые сведения о команд...
Get-Help	Cmdlet	Отображает сведения о командлетах и ...
...		
Alias	Provider	Предоставляет доступ к псевдонимам W...
Environment	Provider	Предоставляет доступ к переменным ср...
FileSystem	Provider	Поставщик PowerShell для доступа к ф...
Function	Provider	Предоставляет доступ к функциям, опр...
Registry	Provider	Предоставляет доступ к разделам и зн...
Variable	Provider	Предоставляет доступ к переменным Wi...
Certificate	Provider	Обеспечивает доступ к хранилищам сер...
...		
about_globbing	HelpFile	См. справку по подстановочным знакам
about_history	HelpFile	Получение команд, введенных в команд...
about_if	HelpFile	Команда языка, используемая для выпо...
about_line_editing	HelpFile	Редактирование команд в командной ст...
about_location	HelpFile	Доступ к объектам из рабочего местоп...
about_logical_operator	HelpFile	Операторы, которые можно использовать...
...		

А если вы хотите посмотреть общие справочные разделы, наберите

```
PS> Get-Help about*
```

Name	Category	Synopsis
-----	-----	-----
about_alias	HelpFile	Использование альтернативных имен ко...
about_arithmetic_operators	HelpFile	Операторы, которые используются в ко...
about_array	HelpFile	Компактная структура размещения элем...
...		

Давайте теперь попробуем применить *Get-Help* к командлету *Get-Command* и посмотрим, что мы еще можем получить с помощью *Get-Command*:

```
PS> Get-Help get-command -detailed
```

ИМЯ

Get-Command

ОПИСАНИЕ

Возвращает базовые сведения о командлетах и о других элементах команд Windows PowerShell.

СИНТАКСИС

```
Get-Command [[-argumentList] <Object[]>] [-verb <string[]>] [-noun <string[]>]  
[-totalCount <int>] [-syntax] [-pSSnapIn <string[]>] [<CommonParameters>]
```

```
Get-Command [[-name] <string[]>] [[-argumentList] <Object[]>] [-commandType  
{<Alias> | <Function> | <Filter> | <Cmdlet> | <ExternalScript> | <Application>  
| <Script> | <All>}] [-totalCount <int>] [-syntax] [<CommonParameters>]
```

ПОДРОБНОЕ ОПИСАНИЕ

Командлет *Get-Command* возвращает базовые сведения о командлетах и других элементах команд

Windows PowerShell, таких как файлы, функции и поставщики Windows PowerShell.

ПАРАМЕТРЫ

**-name <string[]>**

Получает сведения только о командлетах или командных элементах с указанным именем. <Строка> представляет целое имя или часть имени командлета или командного элемента. Подстановочные знаки разрешены.

**-verb <string[]>**

Получает данные о командлетах с именами, в которых содержится указанный глагол. <Строка> представляет один или несколько глаголов или шаблонов глаголов, таких как "remove" или \*et". Подстановочные знаки разрешены.

**-noun <string[]>**

Получает командлеты с именами, в которых содержится указанное существительное. <Строка> представляет одно или несколько существительных или шаблонов

существительных,  
таких как "process" или "\*item\*". Подстановочные знаки разрешены.

`-commandType <CommandTypes>`

Получает только указанные типы командных объектов. Допустимые значения для <Типы\_команд>:

Alias	ExternalScript
All	Filter
Application	Function
Cmdlet (по умолчанию)	Script

*Совет: используйте параметр -Detailed при вызове Get-Help, иначе вы получите минимум информации. Надеюсь, в PowerShell V3 разработчики исправят это, и справочные материалы, выводимые по умолчанию, будут более информативными.*

Существует несколько способов получения разделов справочных материалов. Во-первых, вы можете выполнить `Get-Command` с параметром `-CommandType` для вывода списка других типов команд. Давайте взглянем, какие функции PowerShell доступны по умолчанию:

```
PS> Get-Command -commandType function
```

CommandType	Name	Definition
Function	A:	Set-Location A:
Function	B:	Set-Location B:
Function	C:	Set-Location C:
Function	Clear-Host	\$spaceType = [System.Management.Automation.Host....
...		
Function	help	param([string]\$Name,[string[]]\$Category=@('All')...
...		
Function	man	param([string]\$Name,[string[]]\$Category=@('All')...
Function	md	param([string[]]\$paths); New-Item -type director...
Function	mkdir	param([string[]]\$paths); New-Item -type director...
Function	more	param([string[]]\$paths); if((\$paths -ne \$null) ...
...		
Function	prompt	'PS ' + \$(Get-Location) + \$(if (\$nestedpromptlev...
...		

Отлично. Мы можем сделать то же самое для псевдонимов, внешних и внутренних скриптов, приложений и фильтров. Также необходимо отметить, что `Get-Command` позволяет вести поиск командлетов по глаголу или существительному. Это позволит получить более компактное представление:

```
PS> Get-Command write-*
```

CommandType	Name	Definition
Cmdlet	Write-Debug	Write-Debug [-Message] <String> [-Verbose] [-Deb...
Cmdlet	Write-Error	Write-Error [-Message] <String> [-Category <Erro...
Cmdlet	Write-Host	Write-Host [[-Object] <Object>] [-NoNewline] [-S...
Cmdlet	Write-Output	Write-Output [-InputObject] <PSObject[]> [-Verbo...
Cmdlet	Write-Progress	Write-Progress [-Activity] <String> [-Status] <S...

```

Cmdlet      Write-Verbose      Write-Verbose [-Message] <String> [-Verbose] [-D...
Cmdlet      Write-Warning      Write-Warning [-Message] <String> [-Verbose] [-D...

```

Вы можете поставить подстановочный знак вместо глагола, чтобы найти все глаголы, связанные с заданным существительным (как правило, такой поиск более полезен):

```
PS> Get-Command *-object
```

CommandType	Name	Definition
Cmdlet	Compare-Object	Compare-Object [-ReferenceObject] <PSObject[]> [...
Cmdlet	ForEach-Object	ForEach-Object [-Process] <ScriptBlock[]> [-Inpu...
Cmdlet	Group-Object	Group-Object [[-Property] <Object[]>] [-NoElemen...
Cmdlet	Measure-Object	Measure-Object [[-Property] <String[]>] [-InputO...
Cmdlet	New-Object	New-Object [-TypeName] <String> [[-ArgumentList]...
Cmdlet	Select-Object	Select-Object [[-Property] <Object[]>] [-InputOb...
Cmdlet	Sort-Object	Sort-Object [[-Property] <Object[]>] [-Descendin...
Cmdlet	Tee-Object	Tee-Object [-FilePath] <String> [-InputObject <P...
Cmdlet	Where-Object	Where-Object [-FilterScript] <ScriptBlock> [-Inp...

И, наконец, мы можем передать *Get-Command* имя, чтобы узнать, будет ли оно интерпретироваться как команда, и если это так, то какого она типа: псевдоним, приложение, командлет, фильтр, функция или сценарий. Такое использование *Get-Command* схоже с UNIX-командой *which*, которой добавили больше возможностей. Позвольте мне показать вам, что я хочу сказать:

```
PS> Get-Command more
```

CommandType	Name	Definition
Function	more	param([string[]]\$paths); if((\$paths -ne \$null) ...
Application	more.com	C:\WINDOWS\system32\more.com

Заметьте, что PowerShell показывает мне не только тип каждой команды (функция и приложение), но и показывает путь для приложения *more.com* и определение для функции *more*.

*Примечание: Порядок вывода в PowerShell 1.0 не указывает, что именно будет выполнено, если команды имеют одинаковое название. Это исправлено в версии 2.*

Если вы хотите запустить исполняемый файл *more.com*, вы должны ввести команду *more.com*. Однако здесь можно найти более подробную информацию, чем кажется на первый взгляд. Для этого мы перейдем к третьему ключевому командлету - *Get-Member*.

### Ключ #3: Get-Member

Основная концепция, вызывающая затруднения у большинства людей, впервые сталкивающихся с PowerShell, заключается в том, что почти всё является .NET-объектом (или может им являться). Это значит, что если вы отправляете информацию от одного командлета по конвейеру другому, как правило, она не является текстом. Даже если передается текст, он по-прежнему является объектом типа *System.String*. Впрочем, довольно часто тип объекта бывает другим, и начинающий пользователь PowerShell вряд ли будет знать тип объекта и то, что с ним можно сделать. Давайте посмотрим, какую информацию (и в том числе объекты) предоставляет вывод *Get-Command*. Для этого мы используем *Get-Member*:

```
PS> Get-Command more.com | Get-Member
```

```
TypeName: System.Management.Automation.ApplicationInfo
```

Name	MemberType	Definition
-----	-----	-----
Equals	Method	System.Boolean Equals(Object obj)
GetHashCode	Method	System.Int32 GetHashCode()
GetType	Method	System.Type GetType()
get_CommandType	Method	System.Management.Automation.CommandTypes get_CommandType()
get_Definition	Method	System.String get_Definition()
get_Extension	Method	System.String get_Extension()
get_Name	Method	System.String get_Name()
get_Path	Method	System.String get_Path()
ToString	Method	System.String ToString()
CommandType	Property	System.Management.Automation.CommandTypes CommandType {get;}
Definition	Property	System.String Definition {get;}
Extension	Property	System.String Extension {get;}
Name	Property	System.String Name {get;}
Path	Property	System.String Path {get;}
FileVersionInfo	ScriptProperty	System.Object FileVersionInfo {get=[System.Diagnostics.Fil.

Разве это не интересно? В отличие от UNIX-команды *which*, показывающей лишь путь к приложению, PowerShell предоставляет немного больше информации. Давайте рассмотрим свойство *FileVersionInfo*, связанное с объектом *ApplicationInfo*:

```
PS> Get-Command more.com | Foreach {$_.FileVersionInfo}
```

ProductVersion	FileVersion	FileName
-----	-----	-----
5.1.2600.5512	5.1.2600.5512...	C:\WINDOWS\system32\more.com

Это лишь намёк на мощь доступа к объектам, в отличие от неструктурированной информации, представленной в текстовой форме. *Get-Member* также может оказать помощь в выяснении доступных свойств и методов для объектов .NET.

```
PS> Get-Date | Get-Member
```

```
TypeName: System.DateTime
```

Name	MemberType	Definition
-----	-----	-----
Add	Method	System.DateTime Add(TimeSpan value)
AddDays	Method	System.DateTime AddDays(Double value)
AddHours	Method	System.DateTime AddHours(Double value)
AddMilliseconds	Method	System.DateTime AddMilliseconds(Double value)

```
AddMinutes          Method          System.DateTime AddMinutes(Double value)
```

```
...
```

Вы также можете получить информацию о статических свойствах и методах, например, так:

```
PS> [System.Math] | Get-Member -static
```

```
TypeName: System.Math
```

Name	MemberType	Definition
-----	-----	-----
Abs	Method	static System.Single Abs(Single value), static
Acos	Method	static System.Double Acos(Double d)
Asin	Method	static System.Double Asin(Double d)
Atan	Method	static System.Double Atan(Double d)
Atan2	Method	static System.Double Atan2(Double y, Double x)
BigMul	Method	static System.Int64 BigMul(Int32 a, Int32 b)

#### Ключ #4: Get-PSDrive

Ещё одна важная идея, которую необходимо понять, чтобы глубоко разбираться в PowerShell, заключается в том, что файловая система является лишь одним из видов дисков, которыми могут оперировать командлеты, работающие с файлами. Как узнать, какие диски доступны в PowerShell? Используйте команду Get-PSDrive:

```
PS> Get-PSDrive
```

Name	Provider	Root
-----	-----	-----
Alias	Alias	
C	FileSystem	C:\
cert	Certificate	\
D	FileSystem	D:\
E	FileSystem	E:\
Env	Environment	
F	FileSystem	F:\
Function	Function	
G	FileSystem	G:\
HKCU	Registry	HKEY_CURRENT_USER
HKLM	Registry	HKEY_LOCAL_MACHINE
Variable	Variable	

Использовать все эти диски могут любые командлеты, работающие с файловой системой. Какие именно из них? Для ответа на этот вопрос введите команду:

```
PS> Get-Command *-Item*
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Clear-Item	Clear-Item [-Path] <String[]>...
Cmdlet	Clear-ItemProperty	Clear-ItemProperty [-Path] <St...
Cmdlet	Copy-Item	Copy-Item [-Path] <String[]> [...
Cmdlet	Copy-ItemProperty	Copy-ItemProperty [-Path] <Str...
Cmdlet	Get-Item	Get-Item [-Path] <String[]> [-...
Cmdlet	Get-ItemProperty	Get-ItemProperty [-Path] <Stri...
Cmdlet	Invoke-Item	Invoke-Item [-Path] <String[]>...
Cmdlet	Move-Item	Move-Item [-Path] <String[]> [...
Cmdlet	Move-ItemProperty	Move-ItemProperty [-Path] <Str...
Cmdlet	New-Item	New-Item [-Path] <String[]> [-...
Cmdlet	New-ItemProperty	New-ItemProperty [-Path] <Stri...
Cmdlet	Remove-Item	Remove-Item [-Path] <String[]>...
Cmdlet	Remove-ItemProperty	Remove-ItemProperty [-Path] <S...
Cmdlet	Rename-Item	Rename-Item [-Path] <String> [...
Cmdlet	Rename-ItemProperty	Rename-ItemProperty [-Path] <S...
Cmdlet	Set-Item	Set-Item [-Path] <String[]> [...
Cmdlet	Set-ItemProperty	Set-ItemProperty [-Path] <Stri...

Теперь вы знаете их - четыре командлета, которые позволят вам начать эффективно использовать Windows PowerShell. *Get-Command* поможет узнать, какими командами вы можете воспользоваться. *Get-Help* подскажет, как их использовать. *Get-Member* пояснит, какие свойства, методы и события доступны для тех объектов .NET, с которыми вы столкнётесь в PowerShell. Наконец, используйте *Get-PSDrive*, чтобы выяснить, какими типами дисков кроме файловой системы, вы можете оперировать.

### Дополнение для PowerShell 2.0

*Get-Command* показывает команды с совпадающими именами в том порядке, в котором PowerShell их будет выполнять. Если *Get-Help* не сможет обнаружить название раздела справочной системы с заданным именем, он выведет список разделов, в которых обнаружит заданное слово. *Get-Member* больше не выводит по умолчанию методы, генерируемые компилятором (наподобие *get\_Name/set\_Name*). Если вам необходимо вывести эти методы, используйте параметр *-Force*.

## Часть 2: Понимание вывода объектов

*Прим. переводчика*

*В UNIX-подобных системах широко используется понятие стандартных потоков ввода/вывода. Эти потоки имеют зарезервированные номера (дескрипторы). Поток с дескриптором 1 называется `stdout` - поток стандартного вывода. Он используется для вывода данных (обычно текстовых, хотя это и не обязательно), как правило, на устройство отображения - например, монитор. Поток с дескриптором 2 называют `stderr` - это вывод отладочной информации и диагностических сообщений системы.*

В оболочках, которыми вы, возможно, пользовались ранее, всё, что появляется в потоках `stdout` и `stderr`, считается выводом. В этих оболочках вы, как правило, можете перенаправить вывод `stdout` в файл, используя оператор перенаправления `>`. И в некоторых оболочках вы можете присвоить выводимые данные некоторой переменной, например в `Korn` это выглядит так:

```
DIRS=$(find . | sed.exe -e 's/\\/\\/g')
```

Если необходимо в дополнение к выводу получить `stderr`, используется оператор перенаправления вывода, например, следующим образом:

```
DIRS=$(find . | sed.exe -e 's/\\/\\/g' 2>&1)
```

Вы можете сделать то же самое в PowerShell:

```
PS> $dirs = Get-ChildItem -recurse
PS> $dirs = Get-ChildItem -recurse 2>&1
```

Согласитесь, в PowerShell всё выглядит почти так же. Так в чём же дело? Что ж, в PowerShell существует ряд различий и нюансов, и вам их необходимо знать.

### Вывод - это всегда объект .NET

Во-первых, запомните, что PowerShell всегда выводит .NET-объект. Он может быть, например, объектом `System.IO.FileInfo`, `System.Diagnostics.Process` или `System.String`. В принципе, он может быть любым .NET-объектом, сборка которого загружена в PowerShell, включая ваши собственные объекты .NET. Вы не должны путать вывод PowerShell с текстом, который отображается на экране. Позднее, в [части 6: Как форматируется вывод](#) я поясню, что при подготовке объекта .NET к выводу на экран используются различные технологии, пытающиеся определить наилучший формат для текстового представления объекта. Однако, когда вы присваиваете переменной результат работы команды, вы получаете не текст, который вы видите на экране. Вы получаете .NET объект(ы). Давайте взглянем на это на следующем примере:

```
PS> Get-Process PowerShell
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	--	-----
376	6	32848	11776	138	6,05	2432	powershell

Теперь поместим вывод в переменную и запросим ее тип:

```
PS> $procs = Get-Process PowerShell
PS> $procs.GetType().FullName
System.Diagnostics.Process
```

Как видите, в переменной *\$procs* был сохранен объект типа *System.Diagnostics.Process*, а вовсе не тот текст, который мы видели на экране. Ну хорошо, а если мы действительно хотим получить именно тот текст, который был выведен на экран? В этом случае можно использовать командлет *Out-String*, позволяющий получить вывод в качестве строки. Взгляните:

```
PS> $procs = Get-Process PowerShell | Out-String
PS> $procs.GetType().FullName
System.String
PS> $procs
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
408	6	32852	13120	138	6,09	2432	powershell

Другим полезным свойством *Out-String* является параметр *Width*, позволяющий задать максимальную ширину отображаемого текста. Это удобно, когда выводится много информации, и вы не хотите, чтобы она была урезана из-за недостаточной ширины экрана.

## Функция возвращает все, что не попало в поток вывода

Судя по группе новостей PowerShell, эта проблема появляется снова и снова. Как правило, она появляется у тех из нас, кто привык к написанию функций в стиле C. Вы должны знать, что функции в PowerShell имеют ряд отличий. Хотя функции в PowerShell предусматривают отдельную область видимости переменных, и на них удобно ссылаться несколько раз, не нарушая принципа *DRY (Don't Repeat Yourself)*, способ, которым они осуществляют вывод, может ввести в заблуждение. По сути, функция обрабатывает вывод так же, как и любой скрипт PowerShell, который не является функцией. Что это значит? Позвольте вам продемонстрировать:

```
PS> function bar {
>> $procs = Get-Process svchost
>> "Returning svchost process objects"
>> return $procs
>> }
>>
```

Этот код должен вернуть массив объектов типа *System.Diagnostics.Process*, верно? Мы говорим оболочке - "вернуть переменную *\$procs*". Давайте проверим:

```
PS> $result = bar
PS> $result | foreach {$_.GetType().FullName}
System.String
System.Diagnostics.Process
System.Diagnostics.Process
System.Diagnostics.Process
```

...

Стоп, стоп! Почему тип первого возвращенного объекта - *System.String*? Давайте проверим, что он содержит, и вы поймете ответ:

```
PS> $result[0] Returning svchost process objects
```

Заметьте, что информационное сообщение, которое мы просто собирались вывести на экран, на самом деле было возвращено как часть результата работы функции. Здесь важно понять несколько тонкостей. Во-первых, ключевое слово *return* позволяет выйти из функции в любой указанной точке. Вы можете также дополнительно указать аргумент для оператора возврата, который будет выведен непосредственно перед возвратом. "*return \$procs*" не значит, что функция вернет лишь переменную *\$procs*. В действительности эта конструкция семантически эквивалентна такой - "*\$procs; return*".

Во-вторых, строка

```
"Returning svchost process objects"
```

эквивалентна такой:

```
Write-Output "Returning svchost process objects"
```

Это ясно дает понять, что строка рассматривается как часть вывода функции.

Хорошо, а если мы хотим вывести эту информацию пользователю, но при этом бы она не являлась частью данных, возвращаемых функцией? Тогда следует использовать командлет *Write-Host* таким образом:

```
PS> function bar {  
>> $Proc = Get-Process svchost  
>> Write-Host "Returning svchost process objects"  
>> return $Proc  
>> }  
>>
```

*Write-Host* выводит свое содержимое не в результат функции, а прямо и непосредственно на экран. Это может показаться банальным, но при создании функции в PowerShell вы должны тщательно проверять, что на выходе будут именно те данные, которые вам нужны. Обычно используется перенаправление нежелательных данных в *\$null* (или явное указание типа *[void]* для таких данных). Вот еще пример:

```
PS> function LongNumericString {  
>>     $strBld = new-object System.Text.StringBuilder  
>>     for ($i=0; $i -lt 20; $i++) {  
>>         $strBld.Append($i)  
>>     }  
>>     $strBld.ToString()  
>> }  
>>
```

Заметьте, что мы не используем ключевое слово *return*, так, как это принято в функциях C. Любые выражения и операторы, возвращающие значение, будут выводить данные в результат функции. В этом смысле функции PowerShell поступают точно так же, как обычный сценарий PowerShell. В вышеприведенной функции, мы, несомненно, хотим получить *\$strBld.ToString()*, который будет результатом функции, но вместо этого мы получим следующий результат:

```
PS> LongNumericString
```

Capacity	MaxCapacity	Length
-----	-----	-----
16	2147483647	1
16	2147483647	2
16	2147483647	3
16	2147483647	4
16	2147483647	5
16	2147483647	6
16	2147483647	7
16	2147483647	8
16	2147483647	9
16	2147483647	10
16	2147483647	12
16	2147483647	14
16	2147483647	16
32	2147483647	18
32	2147483647	20
32	2147483647	22
32	2147483647	24
32	2147483647	26
32	2147483647	28
32	2147483647	30

```
012345678910111213141516171819
```

Ну как? Это, вероятно, больше, чем вы ожидали. Проблема в том, что метод *StringBuilder.Append()* возвращает объект *StringBuilder*, который допускает каскадный вызов метода *Append*. К сожалению, теперь наша функция возвращает 20 объектов типа *StringBuilder* и один объект *System.String*. Это просто исправить, достаточно отбросить ненужные результаты, например, так:

```
PS> function LongNumericString {
>>     $strBld = new-object System.Text.StringBuilder
>>     for ($i=0; $i -lt 20; $i++) {
>>         [void]$strBld.Append($i)
>>     }
>>     $strBld.ToString()
>> }
>> LongNumericString
>>
012345678910111213141516171819
```

## Другие типы вывода, которые не могут быть захвачены

В предыдущей части мы видели один случай особого типа вывода - *Write-Host*, который не участвует в потоке вывода *stdout*. По сути, этот тип вывода никак не может быть получен, кроме вывода на экран. То, что является параметром *-object* командлета *Write-Host*, выводится прямо в консоль хоста, минуя поток вывода *stdout*. Поэтому в отличие от *stderr* вывода, который может быть захвачен, как показано ниже, вывод *Write-Host* не использует потоки и, следовательно, не может быть перенаправлен.

```
PS> $result = remove-item ThisFilenameDoesntExist 2>&1
PS> $result | foreach {$_.GetType().FullName}
System.Management.Automation.ErrorRecord
```

Вывод *Write-Host* может быть получен только использованием командлета *Start-Transcript*. *Start-Transcript* записывает все события, происходящие в течение сессии PowerShell, за исключением, к сожалению, вывода унаследованных приложений (то есть записываются все команды, вводимые пользователем, и все выходные данные, которые отображаются в консоли). Имейте в виду, что *Start-Transcript* предназначен больше для записи сессии, чем записи событий отдельного сценария. Например, если вы обычно запускаете *Start-Transcript* в вашем профиле для записи сеанса PowerShell, сценарий, содержащий команду *Start-Transcript* будет вызывать ошибку, потому что нельзя запустить "вложенную" запись. В этом случае необходимо сначала остановить предыдущую запись сеанса.

Вот список типов вывода, которые не могут быть захвачены, кроме как используя *Start-Transcript*:

1. Прямой вывод на экран с использованием *Write-Host* & *Out-Host*
2. Вывод отладочной информации в консоль с помощью *Write-Debug* или параметра *-Debug* командлета
3. Вывод предупреждающих сообщений с помощью *Write-Warning*
4. Вывод дополнительных подробных сведений, которые могут осуществлять многие командлеты при указании параметра *-Verbose*
5. Потоки *stdout* или *stderr* непосредственно исполняемого файла

Вот и все. Просто всегда помните о том, что операторы и выражения участвуют в выводе функций в PowerShell. Вы должны всегда проверять, что вывод происходит именно так, как вы планировали.

## Часть 3: Как объекты передаются по конвейеру

Для эффективного использования конвейеров в PowerShell, необходимо знать, как объекты передаются по конвейеру. Иногда при передаче объект может изменить свой тип. Не имея возможности проверить, какой тип объекта используется на каждом этапе конвейера, вы можете получить абсолютно непредсказуемый результат.

Например, вот вопрос, с группы новостей на *microsoft.public.windows.powershell*:

"В известном каталоге существует набор подпапок, мне необходимо в каждой из них выполнить команду".

Одним из способов решения может быть такой:

```
PS> Get-Item * | Where {$_.PSIsContainer} | Push-Location -passthru |  
>> Foreach {du .; Pop-Location}
```

Это работает замечательно в случае использования утилиты *du* (В данном случае идет речь о применении утилиты *du*, поскольку она работает с текущим каталогом. Тем не менее, в порядке эксперимента, я попробовал указать полный путь. Результат меня удивил:

```
PS> Get-Item * | Where {$_.PSIsContainer} | Push-Location -passthru |  
>> Foreach {du $_.Fullname; Pop-Location}
```

```
Du v1.31 - report directory disk usage  
Copyright (C) 2005-2006 Mark Russinovich  
Sysinternals - www.sysinternals.com
```

```
No matching files were found.  
...
```

Чтобы выяснить, что произошло, используем команду *Get-Member*:

```
PS> Get-Item * | Where {$_.PSIsContainer} | Get-Member
```

```
TypeName: System.IO.DirectoryInfo
```

Name	MemberType	Definition
-----	-----	-----
Create	Method	System.Void Create(), System.Void Create(Directo
...		

*Get-Member* показывает, что после условия *Where* объект в конвейере содержит тот тип, который и ожидался. Давайте смотреть далее:

```
PS> Get-Item * | Where {$_.PSIsContainer} | Set-Location -PassThru | Get-Member
```

```
TypeName: System.Management.Automation.PathInfo
```

Name	MemberType	Definition
Equals	Method	System.Boolean Equals(Object obj)
GetHashCode	Method	System.Int32 GetHashCode()
GetType	Method	System.Type GetType()
get_Drive	Method	System.Management.Automation.PSDriveInfo get_Drive()
get_Path	Method	System.String get_Path()
get_Provider	Method	System.Management.Automation.ProviderInfo get_Provider()
get_ProviderPath	Method	System.String get_ProviderPath()
ToString	Method	System.String ToString()
Drive	Property	System.Management.Automation.PSDriveInfo Drive {get;}
Path	Property	System.String Path {get;}
Provider	Property	System.Management.Automation.ProviderInfo Provider {get;}
ProviderPath	Property	System.String ProviderPath {get;}

Теперь на этапе конвейера *Set-Location* мы неожиданно получили объект типа *PathInfo*. Что же произошло? Очевидно, командлет *Set-Location* получил наш объект *DirectoryInfo*, превратил его в объект *PathInfo*, и отправил дальше по конвейеру, так как мы установили параметр *-PassThru*. Однако в данном случае, *Set-Location* не передает первоначальный объект - мы получаем совершенно новый объект! Как видим, в *PathInfo* отсутствует параметр *Fullname*, но он имеет несколько других параметров, имеющих отношение к названию пути. Ну а какой из них мы сможем использовать? Давайте добавим командлет *Format-List*, чтобы посмотреть все значения объекта *PathInfo*, передаваемого *Set-Location*.

```
PS> Get-Item * | Where {$_.PSIsContainer} | Set-Location -PassThru |
>> Select -First 1 | Format-List *
```

```
Drive      :
Provider   : Microsoft.PowerShell.Core\FileSystem
ProviderPath : C:\Bin
Path       : Microsoft.PowerShell.Core\FileSystem:C:\Bin
```

Теперь становится понятно, что для последующего использования в приложении следует использовать свойство *ProviderPath*, потому что свойство *Path* приложение вряд ли сможет правильно интерпретировать. Отмечу, что в этом примере я использовал конструкцию *Select -First 1*, чтобы выбрать только первый каталог. Это полезно в случае, если результат команды возвращает набор объектов. Нам нет смысла ждать возможного вывода свойств всех объектов - их ведь может быть несколько тысяч - когда необходимо узнать значения свойств для любого из них.

Еще одна вещь, которую необходимо отметить - в данном сценарии *Get-Member* выводит информацию о разных типах, и это может мешать, если необходимо узнать лишь имя типа объекта. *Get-Member* также показывает информацию один раз для каждого уникального типа объекта. Это не поможет вам выяснить, сколько объектов различных типов передается по конвейеру. Такую информацию легко получить, используя метод *GetType()*, применимый к любому объекту .NET:

```
PS> Get-ChildItem | Foreach {$_.GetType().FullName}
System.IO.DirectoryInfo
System.IO.DirectoryInfo
System.IO.DirectoryInfo
System.IO.DirectoryInfo
System.IO.DirectoryInfo
```

```
System.IO.DirectoryInfo
System.IO.FileInfo
System.IO.FileInfo
System.IO.FileInfo
```

*GetType()* возвращает объект *System.RuntimeType*, который содержит всю интересующую информацию. Нас интересует свойство *FullName*. Если бы я использовал *Get-Member*, я бы получил две строки, содержащих название типа, плюс еще около 125 строк текста. Поэтому очень удобно создать фильтр, и даже стоит поместить его в свой профиль:

```
PS> filter Get-TypeName {if ($_ -eq $null) {'<null>'} else {$_ .GetType().Fullname }}
PS> Get-Date | Get-TypeName
System.DateTime
```

Сообщество *PowerShell Community Extensions* предоставляет этот фильтр в реализации, более устойчивой к ошибкам. Например, в случае, когда важно знать, что элементу конвейера не передан вообще никакой объект. Наш простой фильтр в этом случае не поможет:

```
PS> @() | Get-TypeName
```

В ответ мы не получим ничего, что могло бы подсказать, почему объект не передан по конвейеру. Однако реализация этого фильтра *PSCX* позволяет получить гораздо больше информации:

```
PS> @() | Get-TypeName
WARNING: Get-TypeName did not receive any input. The input may be an empty collection.
You can either prepend the collection expression with the comma operator e.g.
",$collection | gtn" or you can pass the variable or expression to Get-TypeName as an
argument e.g. "gtn $collection".
```

```
PS> ,@() | Get-TypeName -full
System.Object[]
```

Итак, при отладке передачи объектов по конвейеру, используйте *Get-Member*, чтобы узнать свойства и методы, доступные для этих объектов. Используйте *Format-List \**, чтобы узнать значения свойств объектов. И используйте наш небольшой, но полезный фильтр, показывающий тип каждого объекта, проходящего через элемент конвейера в том порядке, в каком увидит их следующий командлет.

## Часть 4: Разнообразие вывода - скаляры, коллекции и пустые наборы - о, боже!

В *части 2: Понимание вывода объектов*, мы уже рассматривали некоторые основные понятия вывода PowerShell. Однако, для эффективного использования PowerShell необходимо понимать еще кое-что. В этой части мы поговорим о разнообразии вывода. Когда вывод является скалярным (то есть имеет одно значение), а когда - коллекцией? А в некоторых случаях вывод может вообще отсутствовать, образуя пустое множество. Я использую термин "коллекция" в широком смысле, для различных типов, включая массивы.

### Скаляры

Работа со скалярами проста. Все следующие примеры создают скалярные значения:

```
PS> $num = 1
PS> $str = "Hi"
PS> $flt = [Math]::Pi
PS> $proc = (get-process)[0]
PS> $date = Get-Date
```

Тем не менее, вы можете иметь дело со скалярами в тот момент, когда думаете, что работаете с коллекцией. Например, когда вы отправляете коллекцию на конвейер, PowerShell автоматически "разбирает" ее на части, отправляя дальше по конвейеру каждый отдельный элемент коллекции один за другим. Вот пример:

```
PS> filter Get-TypeName {$_ .GetType().FullName}
PS> $array = "hi",1,[Math]::Pi,$false
PS> $array | Get-TypeName
System.String
System.Int32
System.Double
System.Boolean
```

Фактически, конвейер не оперирует начальной коллекцией целиком. В подавляющем большинстве случаев разбор коллекции на элементы внутри конвейера - это то, что вам и требуется. В противном случае для принудительного расщепления коллекции вам пришлось бы писать такой код:

```
PS> foreach ($item in $array) {$item} | Get-TypeName
```

Заметим, что это потребовало включения дополнительного оператора *foreach* в конвейере. Поскольку конвейер, как правило, работает с элементами последовательности, а не с целой последовательностью, довольно разумно, что разбиение на элементы происходит автоматически. Тем не менее, случаются ситуации, когда разделение на элементы недопустимо. Для этого случая есть две новости - хорошая и плохая. Для начала сообщим плохую - технически, такое поведение изменить невозможно. PowerShell всегда расщепляет коллекции. Хорошая новость в том, что мы можем обойти это поведение, создавая новую коллекцию, которая будет содержать только один элемент - начальную коллекцию. Например, именно так я бы изменил предыдущий пример, чтобы отправить неизменённый массив далее по конвейеру, а не каждый его отдельный элемент:

```
PS> ,$array | Get-TypeName
System.Object[]
```

Едва различимое изменение - запятая перед *\$array*. Эта запятая - унарный оператор, заставляющий PowerShell "обернуть" объект, следующий за ней, в новый массив, содержащий единственный элемент - начальный объект. Таким образом, на выходе мы получим именно тот результат, который нам нужен.

Еще одна особенность обработки скаляров PowerShell заключается в том, как оператор *foreach* обрабатывает скаляры. Например, следующий сценарий может удивить разработчиков *C#*:

```
PS> $vars = 1
PS> foreach ($var in $vars) { "`$var is $var" }
$var is 1
```

В языках типа *C#* переменная *\$vars* должна представлять коллекцию (*IEnumerable*), иначе возникнет ошибка выполнения. Для PowerShell это не является проблемой. Если переменная *\$vars* окажется скалярной, PowerShell просто обработает ее так же, как и коллекцию с единственным элементом. Опять же, это хорошая вещь в PowerShell иначе, если бы мы писали код, подобный этому:

```
PS> $files = Get-ChildItem *.sys
PS> foreach ($file in $files) { "File is: $file" }
File is: C:\config.sys
```

то его бы пришлось модифицировать в случае, если *Get-ChildItem* обнаружит только один *.sys*-файл. Наш сценарий не страдает от избытка строк, осуществляющих проверку на тип данных - являются они скаляром или коллекцией. Теперь проницательный читатель может спросить: "Хорошо, а если *Get-ChildItem* не найдет вообще ни одного *.sys*-файла?". Немного подождите с ответом.

## Работа с коллекциями

Работа с коллекциями в PowerShell также проста. Все примеры ниже создают коллекции:

```
PS> $nums = 1,2,3+7..20
PS> $strs = "Hi", "Mom"
PS> $flts = [Math]::Pi, [Math]::E
PS> $procs = Get-Process
```

Иногда необходимо, чтобы результат рассматривался как коллекция, даже если возвращена может быть лишь одна величина (скаляр). PowerShell предлагает для этого случая оператор массива. Давайте посмотрим на нашу команду *Get-ChildItem* ещё раз. В этот раз мы принудительно заставим результат быть коллекцией:

```
PS> $files = @(Get-ChildItem *.sys)
PS> $files.GetType().FullName System.Object[]
PS> $files.length
1
```

В нашем случае найден один файл. Здесь также важно знать, что и коллекция, и *FileInfo* в случае единичного файла имеют свойство *Length*. Это может ввести в заблуждение. Учитывая, что запятая

(оператор) заносит объект в новый массив, как с ней работает оператор, создающий массив? Давайте посмотрим:

```
PS> $array = @(1,2,3,4)
```

```
PS> $array.rank
```

```
1
```

```
PS> $array.length
```

```
4
```

Как видим, в этом случае запятая не оказывает влияния - размерность массива не изменяется. Ну и всё-таки, а что будет, если *Get-ChildItem* не вернёт ничего?

## Работа с пустыми наборами

Давайте посмотрим, что произойдет, если команда не возвращает ничего. Это довольно сложная тема, и необходимо понимать несколько вещей, чтобы избежать ошибок в сценариях. Сначала запишем несколько правил:

1. Результатом команды может являться поток вывода, но его может и не быть - то, что я назвал пустым набором
2. Если переменной присваивается результат команды, для представления пустого набора используется *\$null*
3. Оператор *foreach* будет выполнен с каждым скаляром, даже если его значение - *\$null*

Кажется простым, верно? Однако, их сочетания могут быть столь удивительными, что вызовут проблемы при написании сценариев. Вот пример:

```
PS> function GetSysFiles { }
```

```
PS> foreach ($file in GetSysFiles) { "File: $file" }
```

```
PS>
```

*GetSysFiles* не содержит какого-либо результата, и оператору *foreach*, нечего перебирать, так как вызов *GetSysFiles* ничего не возвращает. Что ж, давайте попробуем немного изменить задачу. Предположим, что вызов функции содержит длинный список аргументов, и мы решили выделить её вызов в отдельную строку, вот так:

```
PS> $files = GetSysFiles SomeReallyLongSetOfArguments
```

```
PS> foreach ($file in $files) { "File: $file" }
```

```
File:
```

Хм, теперь мы получили вывод. Хотя всё, что мы сделали - добавили промежуточную переменную, содержащую вывод функции. Если честно, мне кажется это нарушением принципа "[как можно меньше сюрпризов](#)". Позвольте мне объяснить, что случилось.

Используя временную переменную, мы вызвали правило №2. При присваивании переменной результата, который явился пустым множеством, он был конвертирован в *\$null* и назначен *\$files*. Пока кажется разумным, верно? К несчастью, оператор *foreach* следует правилу №3 - и выполняет итерацию, так как он получил уже скаляр, пусть и со значением *\$null*. В общем, сам PowerShell обрабатывает ссылки на *\$null* довольно правильно. Отметим, что изменение строки кода в вышеприведённом примере не вызвало никаких ошибок при обнаружении *\$null*. Однако методы .NET Framework могут вести себя не столь "безобидно":

```
PS> foreach ($file in $files) { "Basename: $($file.Substring(0,$file.Length-4))" }
Нельзя вызвать метод для выражения со значением NULL.
В строка:1 знак:16
+ $file.Substring( <<<< 0,$file.Length-4)
Basename:
```

"Хьюстон, у нас проблема". Это означает, что необходимо быть осторожным при использовании *forEach* для перебора результатов команды в том случае, когда не ясно, какое количество элементов будет возвращено, а сценарий не сможет обработать перебор *\$null*. Использование оператора массива может помочь в этом случае, но крайне важно использовать его в правильном месте. Например, вот такая конструкция тоже не работает:

```
PS> foreach ($file in @($files)) { "Basename: $($file.Substring(0,$file.Length-4))" }
Нельзя вызвать метод для выражения со значением NULL.
В строка:1 знак:16
+ $file.Substring( <<<< 0,$file.Length-4)
Basename:
```

Так как *\$files* уже имеет значение *\$null*, оператор создания массива просто создает массив с одним элементом - *\$null*, который *forEach* "успешно" обрабатывает.

Я рекомендую помещать вызов функции полностью в конструкцию *forEach*, если её вызов достаточно лаконичен. Оператор *forEach* прекрасно знает что делать, если функция не возвратит результат. Если же вызов функции достаточно длинный, его можно сделать таким образом:

```
PS> $files = @(GetSysFiles SomeReallyLongSetOfArguments)
PS> foreach ($file in $files) { "Basename: $($file.Substring(2))" }
PS>
```

При применении оператора массива непосредственно к функции, которая ничего не возвращает, вы получите пустой массив, а не массив с *\$null* в нём. Если вы хотите, чтобы ваши функции, имели возможность возвращать пустые массивы, используйте оператор-запятую, как показано ниже, для обеспечения возврата результатов в форме массива.

```
function ReturnArrayAlways {
    $result = @()
    # Здесь что-то, что может добавить 0, 1 или более элементов к массиву $result
    # $result = 1
    # or
    # $result = 1,2
    , $result
}
```

## Часть 5: Используйте объекты, Люк. Используйте объекты!

Использование Windows PowerShell требует изменений в вашем отношении к тому, как оболочка обрабатывает информацию. В большинстве других оболочек вы имеете дело в первую очередь с информацией в виде текста. PowerShell предоставляет много полезных функций для манипуляций с текстом:

- ◆ *-like*
- ◆ *-notlike*
- ◆ *-match*
- ◆ *-notmatch*
- ◆ *-replace*
- ◆ *-eq*
- ◆ *-ne*
- ◆ *-ceq (case-sensitive)*
- ◆ *-cne (case-sensitive)*

По умолчанию, PowerShell обрабатывает текст (точнее, объекты типа *System.String*), не обращая внимания на регистр в операциях сравнения, поиска или замены регулярных выражений. Из-за удобств этих функций легко продолжать пользоваться методами разбора и сравнения строк. Иногда даже этого не избежать. Но лучше стараться использовать свойства предоставляемых объектов. Вот основные причины для этого:

- простое понимание кода
- легче избежать ошибок (связанных с изменением формата, неправильных регулярных выражений, неверной техники сравнения)
- более высокая производительность

Вот, например, вопрос из группы новостей *public.microsoft.windows.powershell*: "Как проверить вывод *dir* или *Get-ChildItem*, так, чтобы отфильтровать каталоги, а файлы отправить на конвейер?"

Вот подход, основанный на прежнем методе:

```
PS> Get-ChildItem | Where {$_.mode -ne "d"}
```

Во-первых, скажу сразу, что он не работает. Во-вторых, и это более важно, он полагается на сравнение строк при определении, отправлять или нет элемент на конвейер. Если вы уж хотите осуществлять проверку таким способом, можно попробовать следующий вариант:

```
PS> Get-ChildItem | Where {$_.mode -notlike "d*"}
```

Однако в случае неосторожного использования легко получить неверные результаты. Более лучший подход к решению подобных проблем - способ PowerShell. PowerShell присваивает каждому элементу на выходе командлета *Get-ChildItem* (или других *\*-Item* командлетов) дополнительные свойства. Это не зависит от используемого провайдера: файловой системы, реестра, функций и т. д. Мы можем увидеть эти дополнительные свойства, которые содержат приставку PS, с помощью "нашего старого друга" *Get-Member* таким образом:

```
PS> cd function:
```

```
PS Function:\> New-Item -type function "foo" -value {} | Get-Member
```

```
TypeName: System.Management.Automation.FunctionInfo
```

Name	MemberType	Definition
-----	-----	-----
Equals	Method	System.Boolean Equals(Object obj)
GetHashCode	Method	System.Int32 GetHashCode()
GetType	Method	System.Type GetType()
ToString	Method	System.String ToString()
PSDrive	NoteProperty	System.Management.Automation.PSDriveInfo PSDrive=Function
PSIsContainer	NoteProperty	System.Boolean PSIsContainer=False
PSPath	NoteProperty	System.String PSPath=Microsoft.PowerShell.Core\Function::foo
PSPProvider	NoteProperty	System.Management.Automation.ProviderInfo PSPProvider=Microsof
CommandType	Property	System.Management.Automation.CommandTypes CommandType {get;}
Definition	Property	System.String Definition {get;}
Name	Property	System.String Name {get;}
Options	Property	System.Management.Automation.ScopedItemOptions Options {get;s
ScriptBlock	Property	System.Management.Automation.ScriptBlock ScriptBlock {get;}

Одно из этих дополнительных свойств - *PSIsContainer*, сообщает нам, что объект является контейнером. В случае с реестром это значит ключ реестра, для файловой системы - это означает каталог (объект *DirectoryInfo*). Поэтому задача может быть решена так:

```
PS> Get-ChildItem | Where {!$_.PSIsContainer}
```

Это значительно короче и менее подвержено ошибкам. А что у нас с производительностью? Давайте сравним оба метода (я даже добавлю ещё один с проверкой регулярного выражения и параметром *-notmatch*) и измерим производительность:

```
PS> $oldWay1 = 1..20 | Measure-Command {Get-ChildItem | Where {$_.mode -notlike "d*"}}
PS> $oldWay2 = 1..20 | Measure-Command {Get-ChildItem | Where {$_.mode -notmatch "d"}}
PS> $poshWay = 1..20 | Measure-Command {Get-ChildItem | Where {!$_.PSIsContainer}}
```

Вот результат:

```
PS> $oldWay1 | Measure-Object TotalSeconds -ave
```

```
Count      : 1
Average    : 169.2571743
Sum        :
Maximum    :
Minimum    :
Property   : TotalSeconds
```

```
PS> $oldWay2 | Measure-Object TotalSeconds -ave
```

```
Count      : 1
Average    : 181.929144
Sum        :
Maximum    :
Minimum    :
Property   : TotalSeconds
```

```
PS> $poshWay | Measure-Object TotalSeconds -ave
```

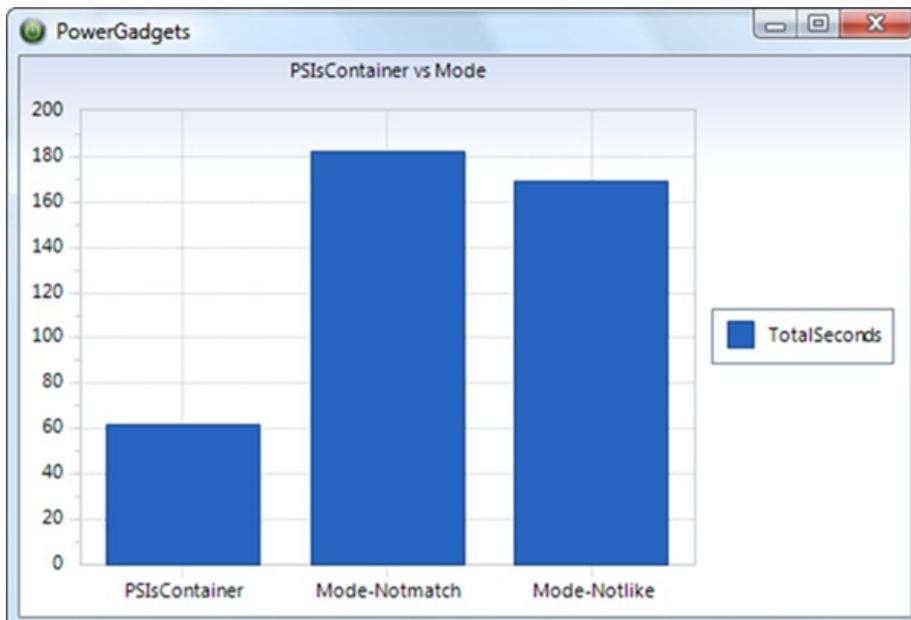
```
Count      : 1
Average    : 61.5349126
Sum        :
Maximum    :
Minimum    :
Property   : TotalSeconds
```

Немного математики, в PowerShell конечно же, и мы получим:

```
PS> "{0:P0}" -f ((169.26 - 61.53) / 61.53)
175 %
```

Надо же! Использование сравнения строк медленнее на 175%, чем использование свойства *PSIsContainer*. С помощью *PowerGadgets*, разработанных *SoftwareFX*, это можно представить наглядно:

```
PS> $data = @{
>> 'Mode-Notlike' = $oldWay1.TotalSeconds
>> 'Mode-Notmatch' = $oldWay2.TotalSeconds
>> PSIsContainer = $poshWay.TotalSeconds
>> }
>>
PS> $data.Keys | Select @{n='Method';e={$_}},@{n='TotalSeconds';e={$data[$_]}} |
>> Out-Chart -Title "PSIsContainer vs Mode"
>>
```



*PowerGadgets* - очень удобный инструмент. Я использую его для презентации использования управлениями версиями менеджерам проекта.

Консоль PowerShell создаёт иллюзию, что вы работаете с текстом - на самом деле вы имеете дело с .NET-объектами, даже если они представлены текстом. Вы часто имеете дело с объектами, содержащими больше информации, чем *System.String* и часто эти объекты содержат необходимую вам информацию в виде свойств. Можно извлекать эту информацию, не прибегая к помощи синтаксического разбора текста. Дополнительный пример работы со свойствами объектов вместо операций с текстом я описал здесь - (<http://tinyurl.com/PsSortIP>).

## Часть 6: Как форматируется вывод

Я уже говорил ранее, практически всё в Windows PowerShell возвращает объекты .NET почти для всего. *Get-ChildItem* выводит последовательность объектов *System.IO.FileInfo* и *System.IO.DirectoryInfo*. *Get-Date* выводит объект *System.DateTime*. *Get-Process* выводит объекты *System.Diagnostics.Process*, *Get-Content* выводит объект *System.String* (или их массив, в зависимости от параметра *-ReadCount*). PowerShell всегда имеет дело с .NET-объектами. Отображение текста в консоли делает это не всегда очевидным. Представим себе на минуту, что нам бы самим понадобилось преобразовать объект в текст, который необходимо вывести на экран.

Вероятно, первым делом мы бы рассмотрели метод *ToString()*, имеющийся для каждого .NET-объекта. Это отлично работает для некоторых объектов .NET:

```
PS> (Get-Date).ToString()
9/3/2007 10:21:23 PM
```

Но не очень подходит для других:

```
PS> (Get-Process)[0].ToString()
System.Diagnostics.Process (audiodg)
```

Как видим, нам предоставляется очень мало информации. Давайте посмотрим, как разработчики PowerShell решали эту проблему. Они придумали понятие "вида" для популярных типов .NET, которые могут отображать объекты таблицей, списком, столбцами или пользовательским способом. Для известных типов .NET у PowerShell есть описание по умолчанию, позволяющее выводить текстовую информацию в подходящем виде без необходимости указания форматирования. Для типов, с которыми PowerShell "не знаком", необходимо выбрать способ форматирования. Если вы не зададите форматирование, PowerShell сам выберет один из стандартных видов форматирования.

Краткое определение для типов и объектов. Класс *System.DateTime* это лишь один из типов .NET. Командлет *Get-Date* выводит объект, который является экземпляром типа *System.DateTime*. Может существовать много объектов *DateTime*, основанных на *System.DateTime*. PowerShell определяет вид для всех экземпляров объектов этого типа.

Что делать, если PowerShell не сможет определить вид для какого-либо типа? Это вполне возможно, поскольку типов в .NET может быть бесконечное множество. Я могу прямо сейчас создать тип *Plan9FromOuterSpace*, скомпилировать его в сборку .NET и загрузить её в PowerShell. Как будет работать с ним PowerShell, если он ничего не знает о таком типе? Давайте посмотрим:

```
@'
public class Plan9FromOuterSpace {
    public string Director = "Ed Wood";
    public string Genre = "Science Fiction B Movie";
    public int NumStars = 0;
}
'@ > C:\temp\Plan9.cs
```

```
PS> csc /t:library Plan9.cs
PS> [System.Reflection.Assembly]::LoadFrom('c:\temp\Plan9.dll')
PS> New-Object Plan9FromOuterSpace
Director      Genre                NumStars
-----
Ed Wood      Science Fiction B Movie  0
```

Эксперименты показывают, что если объект содержит менее четырех свойств, PowerShell использует для его отображения таблицу. Если у объекта пять и более свойств, PowerShell выводит их в виде списка.

Возможно определить несколько видов отображения для одного типа .NET. Отображение определяется в XML-файлах форматирования, расположенных в папке установки PowerShell:

```
PS> Get-ChildItem $PSHOME\*format*
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\Windows\System32\
WindowsPowerShell\v1.0
```

Mode	LastWriteTime	Length	Name
-a---	1/24/2007 11:23 PM	22120	Certificate.format.ps1xml
-a---	1/24/2007 11:23 PM	60703	DotNetTypes.format.ps1xml
-a---	1/24/2007 11:23 PM	19730	FileSystem.format.ps1xml
-a---	1/24/2007 11:23 PM	250197	Help.format.ps1xml
-a---	1/24/2007 11:23 PM	65283	PowerShellCore.format.ps1xml
-a---	1/24/2007 11:23 PM	13394	PowerShellTrace.format.ps1xml
-a---	1/24/2007 11:23 PM	13540	Registry.format.ps1xml

Содержание этих файлов выглядит следующим образом:

```
<View>
  <Name>process</Name>
  <ViewSelectedBy>
    <TypeName>System.Diagnostics.Process</TypeName>
    <TypeName>Deserialized.System.Diagnostics.Process</TypeName>
  </ViewSelectedBy>
  <TableControl>
    <TableHeaders>
      <TableColumnHeader>
        <Label>Handles</Label>
        <Width>7</Width>
        <Alignment>right</Alignment>
      </TableColumnHeader>
      <TableColumnHeader>
        <Label>NPM(K)</Label>
        <Width>7</Width>
        <Alignment>right</Alignment>
      </TableColumnHeader>
      <TableColumnHeader>
        <Label>PM(K)</Label>
        <Width>8</Width>
        <Alignment>right</Alignment>
      </TableColumnHeader>
      <TableColumnHeader>
        <Label>WS(K)</Label>
        <Width>10</Width>
```

```

    <Alignment>right</Alignment>
</TableColumnHeader>
<TableColumnHeader>
    <Label>VM(M)</Label>
    <Width>5</Width>
    <Alignment>right</Alignment>
</TableColumnHeader>
<TableColumnHeader>
    <Label>CPU(s)</Label>
    <Width>8</Width>
    <Alignment>right</Alignment>
</TableColumnHeader>
<TableColumnHeader>
    <Width>6</Width>
    <Alignment>right</Alignment>
</TableColumnHeader>
<TableColumnHeader />
</TableHeaders>
<TableRowEntries>
<TableRowEntry>
    <TableColumnItems>
        <TableColumnItem>
            <PropertyName>HandleCount</PropertyName>
        </TableColumnItem>
        <TableColumnItem>
            <ScriptBlock>[int]($_NPM / 1024)</ScriptBlock>
        </TableColumnItem>
        <TableColumnItem>
            <ScriptBlock>[int]($_PM / 1024)</ScriptBlock>
        </TableColumnItem>
        <TableColumnItem>
            <ScriptBlock>[int]($_WS / 1024)</ScriptBlock>
        </TableColumnItem>
        <TableColumnItem>
            <ScriptBlock>[int]($_VM / 1048576)</ScriptBlock>
        </TableColumnItem>
        <TableColumnItem>
            <ScriptBlock>
                if ($_CPU -ne $()) {
                    $_CPU.ToString("N")
                }
            </ScriptBlock>
        </TableColumnItem>
        <TableColumnItem>
            <PropertyName>Id</PropertyName>
        </TableColumnItem>
        <TableColumnItem>
            <PropertyName>ProcessName</PropertyName>
        </TableColumnItem>
    </TableColumnItems>

```

```

    </TableRowEntry>
  </TableRowEntries>
</TableControl>
</View>

```

Это XML-определение табличного отображения для типа *Process*. Оно определяет атрибуты отображаемых колонок, данных которые в них отображаются и в некоторых случаях производит преобразование данных к более удобному отображению (например, перевод байтов в *Кб* или *Мб*). А вот определение для того же типа в для отображения в виде столбцов (*Format-Wide*):

```

<View>
  <Name>process</Name>
  <ViewSelectedBy>
    <TypeName>System.Diagnostics.Process</TypeName>
  </ViewSelectedBy>
  <WideControl>
    <WideEntries>
      <WideEntry>
        <WideItem>
          <PropertyName>ProcessName</PropertyName>
        </WideItem>
      </WideEntry>
    </WideEntries>
  </WideControl>
</View>

```

В этом типе отображения выводится лишь одно свойство - *ProcessName*. Поиск в *DotNetTypes.format.ps1xml*, мы найдем больше определений. Например, именованное форматирование *StartTime* не вызывается по умолчанию. Вы можете применить его, указав имя в командлете *Format-Table*:

```

<View>
  <Name>StartTime</Name>
  <ViewSelectedBy>
    <TypeName>System.Diagnostics.Process</TypeName>
  </ViewSelectedBy>
  <GroupBy>
    <ScriptBlock>$_.StartTime.ToShortDateString()</ScriptBlock>
    <Label>StartTime.ToShortDateString()</Label>
  </GroupBy>
  <TableControl>
    <TableHeaders>
      <TableColumnHeader>
        <Width>20</Width>
      </TableColumnHeader>
      <TableColumnHeader>
        <Width>10</Width>
        <Alignment>right</Alignment>
      </TableColumnHeader>
      <TableColumnHeader>

```

```

        <Width>13</Width>
        <Alignment>right</Alignment>
    </TableColumnHeader>
    <TableColumnHeader>
        <Width>12</Width>
        <Alignment>right</Alignment>
    </TableColumnHeader>
</TableHeaders>
<TableRowEntries>
    <TableRowEntry>
        <TableColumnItems>
            <TableColumnItem>
                <PropertyName>ProcessName</PropertyName>
            </TableColumnItem>
            <TableColumnItem>
                <PropertyName>Id</PropertyName>
            </TableColumnItem>
            <TableColumnItem>
                <PropertyName>HandleCount</PropertyName>
            </TableColumnItem>
            <TableColumnItem>
                <PropertyName>WorkingSet</PropertyName>
            </TableColumnItem>
        </TableColumnItems>
    </TableRowEntry>
</TableRowEntries>
</TableControl>
</View>

```

Зачем я всё это показываю вам? Я думаю, важно понять то "волшебство", благодаря которому объекты .NET из двоичных сущностей превращаются в текст, отображаемый в вашей консоли. Обладая этими знаниями, вы никогда не должны забывать о том, что вы имеете дело в первую очередь с объектами .NET.

Существует ли простой способ выяснить, какие выводы отображения доступны для того или иного типа .NET? Да, если у вас установлены расширения *PowerShell Community Extensions*. *PSCX* предлагает удобный сценарий, написанный *Joris van Lier*, который называется *Get-ViewDefinition* и его можно использовать вот так:

```
PS> Get-Viewdefinition System.Diagnostics.Process
```

```

Name       : process
Path       : C:\Windows\System32\WindowsPowerShell\v1.0\DotNetTypes.format.ps1xml
TypeName   : System.Diagnostics.Process
SelectedBy : {System.Diagnostics.Process, Deserialized.System.Diagnostics.Process}
GroupBy    :
Style      : Table

```

```

Name       : Priority
Path       : C:\Windows\System32\WindowsPowerShell\v1.0\DotNetTypes.format.ps1xml
TypeName   : System.Diagnostics.Process

```

```
SelectedBy : System.Diagnostics.Process
GroupBy    : PriorityClass
Style      : Table
```

```
Name       : StartTime
Path       : C:\Windows\System32\WindowsPowerShell\v1.0\DotNetTypes.format.ps1xml
TypeName   : System.Diagnostics.Process
SelectedBy : System.Diagnostics.Process
GroupBy    :
Style      : Table
```

```
Name       : process
Path       : C:\Windows\System32\WindowsPowerShell\v1.0\DotNetTypes.format.ps1xml
TypeName   : System.Diagnostics.Process
SelectedBy : System.Diagnostics.Process
GroupBy    :
Style      : Wide
```

Эти данные могут показать все доступные формы отображения. Возможно, о каких-то вы не были осведомлены. Давайте проверим эти дополнительные формы:

```
PS> Get-Process | Format-Wide
```

```
AcroRd32          ADCDLicSvc
alg               ati2evxx
ati2evxx          BTNTService
ccApp             CCC
ccEvtMgr          ccSetMgr
csrss             ctfmon
DefWatch          editplus
explorer          GoogleUpdate
Idle              lsass
mdm               mDNSResponder
miranda32         MOM
opera             PnkBstrA
powershell       RTHDCPL
Rtvsan            scsiaccess
```

```
PS> Get-Process | Format-Table -View Priority
```

PriorityClass: Normal

ProcessName	Id	HandleCount	WorkingSet
-----	--	-----	-----
AcroRd32	3388	239	59809792
ADCDLicSvc	496	36	1212416
alg	2440	107	3584000

```

csrss                764          667          5844992
ctfmon               2732           80          3575808
DefWatch             580           55          5087232
editplus            2508           85          7794688
explorer            1716           690         17539072
...

```

PriorityClass: High

```

ProcessName          Id  HandleCount  WorkingSet
-----
winlogon             800           465         4407296

```

PS> Get-Process | Format-Table -View StartTime

StartTime.ToShortDateString(): 21.03.2009

```

ProcessName          Id  HandleCount  WorkingSet
-----
AcroRd32             3388           239         60645376
ADCDClicSvc          496            36         1212416
alg                  2440           107         3584000
ati2evxx             1036            88         3440640
ati2evxx             1292           107         3805184
BTNtService          528            82         2449408
ccApp                2700           285         7659520
...

```

StartTime.ToShortDateString(): 01.01.1601

```

ProcessName          Id  HandleCount  WorkingSet
-----
System                4           744         262144

```

StartTime.ToShortDateString(): 21.03.2009

```

ProcessName          Id  HandleCount  WorkingSet
-----
winlogon             800           465         4407296

```

Что делать, если вы забыли, какие способы форматирования доступны? Главное - не забыть, что вы можете использовать *Get-Command*:

```
PS> Get-Command Format-*
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Format-Custom	Format-Custom [[-Property] <Object[]>] [-Depth <...>]
Cmdlet	Format-List	Format-List [[-Property] <Object[]>] [-GroupBy <...>]
Cmdlet	Format-Table	Format-Table [[-Property] <Object[]>] [-AutoSize...]
Cmdlet	Format-Wide	Format-Wide [[-Property] <Object>] [-AutoSize] [...]

Вы уже знакомы с *Format-Table*. Он отображает данные в формате таблицы. Этот формат используется по умолчанию для многих объектов, включая *System.Diagnostics.Process*. *Format-Wide* также довольно понятен. PowerShell выводит одно свойство (как правило, наиболее информативное) в несколько колонок. *Format-Custom* интересен, но вряд ли вы будете им часто пользоваться - он будет вызываться для типов .NET, имеющих пользовательский формат, наподобие *System.DateTime*:

```
<View>
  <Name>DateTime</Name>
  <ViewSelectedBy>
    <TypeName>System.DateTime</TypeName>
  </ViewSelectedBy>
  <CustomControl>
    <CustomEntries>
      <CustomEntry>
        <CustomItem>
          <ExpressionBinding>
            <PropertyName>DateTime</PropertyName>
          </ExpressionBinding>
        </CustomItem>
      </CustomEntry>
    </CustomEntries>
  </CustomControl>
</View>
```

*DateTime* имеет тип *ScriptProperty*, который PowerShell определяет так:

```
PS> Get-Date | Get-Member -Name DateTime
```

```
TypeName: System.DateTime
```

Name	MemberType	Definition
-----	-----	-----
DateTime	ScriptProperty	System.Object DateTime {get=if (\$this.DisplayHint -i

Так мы подходим к моему любимому методу форматирования для исследования вывода PowerShell. Вы наверняка заметили, что содержание колонки *Definition* обрезается. Если необходимо вывести её полное содержание, используйте командлет *Format-List*. Этот командлет выводит различные значения свойств в отдельных строках, так что данные редко сокращаются. Например:

```
PS> Get-Date | Get-Member -Name DateTime | Format-List
```

```
TypeName    : System.DateTime
Name        : DateTime
MemberType  : ScriptProperty
Definition  : System.Object DateTime {get=if ($this.DisplayHint -ieq "Date")
            {
                "{0}" -f $this.ToLongDateString()
            }
            elseif ($this.DisplayHint -ieq "Time")
            {
                "{0}" -f $this.ToLongTimeString()
            }
            else
            {
                "{0} {1}" -f $this.ToLongDateString(),
                $this.ToLongTimeString()
            }
            };}
```

Теперь мы видим полное определение свойств *DateTime*. Замечание: часто PowerShell выводит сокращённый набор значений свойств с помощью командлета *Format-List*. Это делается для того, чтобы отсекал обычно ненужную информацию. Однако, если вам необходим самый подробный вывод всех деталей, используйте этот командлет так: "*format-list \**".

Смотрите, вот стандартный вывод для объекта *Process*:

```
PS> (Get-Process)[0] | Format-List
```

```
Id        : 3388
Handles   : 230
CPU       : 33,328125
Name      : AcroRd32
```

А здесь мы запрашиваем вывод всех свойств:

```
PS> (Get-Process)[0] | Format-List *
```

```
__NounName      : Process
Name            : AcroRd32
Handles        : 230
VM              : 186585088
WS              : 73781248
PM              : 71942144
NPM            : 8760
Path           : C:\Program Files\Adobe\AcroRd32.exe
Company        : Adobe Systems Incorporated
CPU            : 33,59375
FileVersion    : 8.1.0.2007051100
ProductVersion : 8.1.0.2007051100
Description    : Adobe Reader 8.1
```

Product : Adobe Reader

Id : 3388

...

Понимаете, о чём я? Посмотрите, как много информации может быть не выведено, если вы символом звёздочки не укажете, что хотите видеть все свойства объекта.

## Часть 7: Режимы синтаксического разбора PowerShell

Способ, которым PowerShell осуществляет разбор строк, может удивить тех, кто ранее использовал оболочки с упрощенным парсингом, такие как *CMD.EXE*. Парсинг в PowerShell значительно отличается, так как он используется и как интерактивная оболочка командной строки, и как язык сценариев. Сложность заключалась в том, что разработчикам необходимо было:

1. Позволить выполнение команд и программ с аргументами в командной строке. Следствие: аргументы (имена файлов, пути) не должны требовать кавычек, если в аргументе отсутствуют пробелы.
2. Позволять выполнение сценариев, содержащих выражения, которые аналогичны выражениям в большинстве языков скриптов/программирования. Следствие: Сценарий PowerShell должен верно определять выражения вида  $2 + 2$  и  $\$date.Second$ , а также строки, использующие кавычки типа *"del -r \* is being executed"*.
3. Предоставить возможность копирования интерактивно введенного кода и вставки его в сценарий для последующего применения. Следствие: эти две области применения - интерактивная и скриптовая - должны "уживаться".

Частью мощного языка сценариев должна являться поддержка не только данных строкового типа. Фактически PowerShell поддерживает большинство типов .NET, включая *String*, *Int8*, *Int16*, *Int32*, *Decimal*, *Single*, *Double*, *Boolean*, *Array*, *ArrayList*, *StringBuilder* и многие другие .NET-типы. Звучит неплохо, но всё же - что там с режимами разбора? Давайте подумаем, как язык может представлять строчный литерал? Ну, большинство, как правило, ожидают эту строку: *"Hello World"*. Фактически, PowerShell распознаёт её как строку:

```
PS> "Hello World".GetType().Name
String
PS> "Hello World"
Hello World
```

Если вы напечатаете строку сразу за приглашением и нажмёте клавишу *Enter*, PowerShell, благодаря среде REPL(Read-eval-print-loop), отобразит её в консоли, как показано выше. А если необходимо указать аргумент для команды в кавычках?

```
PS> del "foo.txt", "bar.txt", "baz.txt"
```

Вы сразу ощутите отличия от других оболочек. Хуже того, эти кавычки довольно быстро начинают надоедать. Я думаю, что разработчики PowerShell вовремя решили, что им понадобятся два различных режима синтаксического анализа строк. Во-первых, необходимо разбирать строки так, как это делают традиционные оболочки, в которых имена файлов и процессов, путей не обрамляются кавычками. Во-вторых, необходимо иметь возможность воспринимать строковое выражение так, как это принято в языках программирования - заключая его в кавычки. В PowerShell первый режим называется режим разбора команд, а второй - режим разбора выражений. Важно понимать, в каком режиме вы находитесь в данный момент, и ещё важнее - как переключаться между ними.

Давайте взглянем на пример. Очевидно, хотим ввести команду для удаления этих файлов:

```
PS> del foo.txt, bar.txt, baz.txt
```

Так даже лучше - для имён файлов кавычки не требуются. PowerShell обрабатывает эти имена как строковые выражения даже без кавычек в режиме команд. А что случится, если путь будет содержать пробелы? Вы можете сами попробовать это:

```
PS> del 'C:\Documents and Settings\Keith\_lessht'
```

И это работает так, как и ожидалось. А если необходимо выполнить программу с пробелами в пути?

```
PS> 'C:\Program Files\Windows NT\Accessories\wordpad.exe'  
C:\Program Files\Windows NT\Accessories\wordpad.exe
```

Такая конструкция не работает. PowerShell решил, что мы ввели строку и он просто отобразил её на экране. Это произошло потому, что PowerShell находится в режиме выражений. Нам необходимо сообщить ему, что эту строку необходимо анализировать в режиме команд. Для этого используется оператор '&':

```
PS> & 'C:\Program Files\Windows NT\Accessories\wordpad.exe'
```

*Подсказка: для автоматического завершения части пути используйте Tab и Shift-Tab. Если путь содержит пробелы, PowerShell также вставит оператор вызова и автоматически поставит кавычки вокруг пути.*

Этот пример показывает, что PowerShell анализирует первый не пустой символ в строке для определения, в каком режиме строка будет обработана. Любой из нижеприведённых символов переводит PowerShell в режим команд:

```
[_aA-zZ]  
&  
.  
\  
/
```

У этого правила есть одно исключение - если строка начинается с ключевого слова (*if, do, while, foreach* и т. д.), PowerShell переходит в режим выражений и будет ожидать оставшуюся часть выражения, связанную с этим ключевым словом.

Преимущества командного режима:

- Строка не нуждается в обрамлении кавычками, если она не содержит пробелы
- Числа обрабатываются как числа, все другие аргументы - как строки, за исключением тех, которые начинаются с @, \$, (, ' или ". Числа интерпретируются как *Int32, Int64, Double* или *Decimal* в зависимости от их написания и величины, необходимой для хранения числа, например, *12, 30GB, 1E-3, 100.01d*.

Хорошо, а для чего нужен режим выражений? Ну, как я уже говорил, необходимо иметь возможность оценки выражений, таких, как это:

```
PS> 64-2  
62
```

Не удивительно, что некоторые оболочки могут интерпретировать это выражение, например, попытавшись выполнить команду с именем '64-2'. Как же PowerShell определяет, что строку необходимо анализировать в режиме выражений? Если строка начинается с цифры, или одного из этих знаков - @, \$, (, ' или " - строка разбирается в режиме выражений. Преимущества этого режима:

- Устранение возможности неверного толкования команд как строк, например, `del -recurse *` - это команда, а `"del -recurse *"` - это строка.
- Прямое указание арифметических операций и сопоставления выражений, например `64-2 (62)` и `$array.count -gt 100`. В режиме команд `-gt` интерпретировалось бы как параметр, если предыдущая лексема соответствует верной команде.

Одним из следствий правил режима анализа строк является то, что для запуска исполняемых файлов или сценариев с именами, начинающимися с цифры, необходимо заключать их в кавычки и использовать оператор вызова:

```
PS> & '64E1'
```

Если вы попытаетесь выполнить `64E1` без оператора вызова, PowerShell не сможет понять, хотите вы ввести число `64E1(640)` или выполнить исполняемый файл с именем `64E1.exe` или `64E1.ps1`. В этом случае зависит от вас, в какой режим анализа перейдет PowerShell.

*Примечание: я заметил, что в случае указания полного имени (например, `64E1.ps1` или `64E1.exe`), нет необходимости заключать команду в кавычки.*

А что делать, если в одной строке необходимо сочетать различные режимы? Легко. Просто используйте выражения группировки `()`, подвыражение `$()` или подвыражение массива `@()`. Это заставит парсер производить повторную оценку режима синтаксического анализа, который будет произведён по первому символу внутри скобок.

Чем различаются выражения группировки `()`, подвыражения `$()` и подвыражения массива `@()`? Выражение группировки может содержать лишь простое выражение или простой конвейер. Подвыражение может содержать несколько операторов, разделённых точкой с запятой. Вывод каждого оператора передаётся на вывод подвыражения, который может быть скаляром, коллекцией или пустым множеством. Массив подвыражений ведёт себя точно так же, как подвыражение, за исключением того, что он гарантирует, что выводом будет массив. Два случая, имеющих различия:

1. Когда массив не имеет вывода, подвыражение вернёт пустой массив
2. Если результатом является скалярное значение, будет выведен массив с одним элементом, содержащим скалярную величину.

Если выводом уже является массив, то использование оператора массива не действует, то есть массив не будет ещё раз "обёрнут" в другой массив.

В следующем примере я внедряю команду `"Get-ChildItem C:\Windows"` в строку, разбор которой начинается в режиме выражений. Когда будет встречено выражение группировки `(get-childitem c:\windows)`, начнётся новая оценка режима анализа. Обнаружив, что первым символом является `'g'`, парсер перейдёт в режим команд до конца текста внутри выражения группировки. Замечу, что `".Length"` анализируется в режиме выражений, поскольку не входит в выражение группировки, и PowerShell вернется обратно к предыдущему режиму парсинга. `".Length"` заставляет PowerShell получить свойство `Length` для объекта, выведенного выражением группировки. В моём случае, это массив объектов `FileInfo` и `DirectoryInfo`. Свойство `Length` сообщит количество элементов в этом массиве.

```
PS> 10 + (get-childitem c:\windows).Length
115
```

Мы можем сделать наоборот. То есть, вставить выражение в строку, разбор которой начинается в командном режиме. В примере ниже мы используем выражение, вычисляющее количество объектов, выбранных из последовательности.

```
PS> Get-Process | Select -first (1.5 * 2)
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
239	9	64840	65644	189	34,59	224	AcroRd32
36	2	328	1184	14	0,06	496	ADCDLicSvc
107	6	1136	3500	33	0,06	2440	alg

Используя возможность начать новый режим, можно вложить команды в команды. Это очень эффективная возможность. В следующем примере PowerShell успешно анализирует строку в режиме команд, пока не встретит выражение '@/'. В этом месте начнётся новое определение режима разбора, но будет обнаружена следующая команда. Эта команда будет получать имя файла для переименования из его первой строки. Я использую подвыражение массива для того, чтобы он гарантированно вернул массив строк, даже если файл содержит только одну строку. Если вместо него использовать простое подвыражение и файл будет содержать только одну строку, то PowerShell требование вернуть элемент с индексом [0] расценит так - "вернуть первый символ, встреченный в строке". Тогда это будет соответствовать символу "f" в нижеприведённом примере.

```
PS> Get-ChildItem [a-z].txt |
>>Foreach{Rename-Item $_ -NewName @(Get-Content $_)[0] -WhatIf}
What if: Performing operation "Rename File" on Target
"Item: C:\a.txt Destination: C:\file_a.txt".
What if: Performing operation "Rename File" on Target
"Item: C:\b.txt Destination: C:\file_b.txt".
```

Последняя тонкость, на которую я бы хотел обратить ваше внимание - различие между использованием оператора исполнения & и вызовом команды с помощью точки. Рассмотрим вызов простого скрипта, который определяет переменную `$foo = 'PowerShell Rocks!'`. Давайте выполним этот сценарий, используя оператор вызова и посмотрим его воздействие на глобальную сессию:

```
PS> $foo
PS> & .\script.ps1
PS> $foo
```

Как видим, оператор вызова запускает команду в дочерней области видимости, которая будет утеряна по окончании работы команды (сценария, функции и т. д.). То есть, такой способ не оказывает влияния на значение переменной `$foo` в глобальной области видимости. Теперь попробуем с точкой:

```
PS> $foo
PS> . C:\Users\Keith\script.ps1
PS> $foo
PowerShell Rocks!
```

Если мы ставим в начале строки точку, сценарий будет выполнен в текущей области видимости. В результате переменная `$foo` из скрипта `script.ps1` становится ссылкой на глобальную переменную `$foo` (если сценарий вызван из командной строки с точкой), и сценарий успешно изменяет значение глобальной переменной `$foo`. Такое поведение не несёт сюрпризов - оно сходно с поведением в других командных оболочках. Эти же правила применяются к вызову функций. Однако для внешних `exe`-файлов неважно, производится их вызов точкой или оператором вызова. Выполнение `exe`-файлов происходит в отдельном потоке и не оказывает влияния на текущую область видимости.

Вот полезная таблица, которая поможет вам запомнить правила, определяющие режимы синтаксического анализа PowerShell

Первый непробельный символ	Режим
[_aA-zZ], &, . или \	Режим команд
[0-9], ', ", \$, (, @ и любой другой символ, не являющийся символом режима команд	Режим выражений

Поняв тонкости режимов анализа, вы сможете избежать сюрпризы, которые получают начинающие - например, как выполнить *exe*-файл, в пути которого содержатся пробелы.

## Часть 8: Параметры привязки элементов конвейера `ByPropertyName` (по имени)

Нам всем нравится решать проблемы эффективными способами. Кульминацией эффективности в PowerShell являются команды в одну строку. В целях обучения, я считаю, гораздо лучше расширить эту лаконичность, составляя команды в несколько строк. Вместе с тем нельзя отрицать, что когда вам необходимо что-то быстро набрать в консоли PowerShell с её возможностями редактирования, менее утомительно использовать однострочные команды. Это не вина PowerShell - он использует антикварную консольную подсистему Windows, которая не менялась со времён выхода NT в 1993 году.

Одна из возможностей, позволяющих сократить длину вводимых команд - использование привязки параметров конвейера. Я часто вижу людей, которые пишут команды вроде этой:

```
PS> Get-ChildItem . *.cs -r | Foreach { Get-Content $_.fullname } | ...
```

Это вполне работает, но использование *Foreach-Object* не является здесь технически необходимым. Многие командлеты PowerShell связывают свой "первичный" параметр с тем, что передаётся по конвейеру. Это явно указывается в справке для *Get-Content*:

### ПАРАМЕТРЫ

`-path <string[]>`

Определяет путь к элементу. Командлет *Get-Content* извлекает содержимое элемента. Подстановочные знаки разрешены. Имя параметра ("`-Path`" или "`-FilePath`") можно не указывать.

Требуется?	true
Позиция?	1
Значение по умолчанию	Невозможно - должен быть указан путь
Принимать входные данные конвейера?	true (ByPropertyName)
Принимать подстановочные знаки?	true

...

`-literalPath <string[]>`

Определяет путь к элементу. В отличие от значения `Path`, значение `LiteralPath` используется точно так, как оно введено. Никакие знаки не интерпретируются как подстановочные знаки. Если путь включает знаки управляющих последовательностей, его нужно заключить в одинарные кавычки. Одинарные кавычки указывают оболочке Windows PowerShell, что никакие знаки не следует интерпретировать как знаки управляющих последовательностей.

Требуется?	true
Позиция?	1
Значение по умолчанию	
Принимать входные данные конвейера?	true (ByPropertyName)
Принимать подстановочные знаки?	false

*Примечание: для такой подробной детализации используйте `Get-Help` с параметром `-Full`.*

Из четырёх параметров *Get-Content*, которые принимают входные данные конвейера (*ByPropertyName*), я привёл лишь два. Ещё два - *ReadCount* и *TotalCount*. Квалификатор *ByPropertyName* просто означает, что если объект, поступающий на вход элемента конвейера, имеет свойство с тем же именем, оно "привязывается" как входной параметр. Это действует в том случае, если типы совпадают или могут быть преобразованы.

Например, вышеприведённую команду можно записать и так - убрав командлет *Foreach-Object*:

```
PS> Get-ChildItem *.cs -r | Get-Content | ...
```

Интуитивно понятно, что *Get-Content* вполне способен оперировать с объектами *System.IO.FileInfo*, которые выведет *Get-ChildItem*. Тем не менее, это не явное использование только что упомянутого правила *ByPropertyName*. Почему? Объект *FileInfo*, выводимый командлетом *Get-ChildItem* не имеет свойств *Path* или *LiteralPath* похожих на *PSPath*. Каким же образом *Get-Content* определяет путь файла в этом сценарии? Есть два способа узнать это. Первый, наиболее простой - использование командлета *Trace-Command*, показывающего, как привязываются параметры. Второй - с помощью сборки *Red Gate .NET Reflector*. Давайте сначала попробуем *Trace-Command*.

*Trace-Command* - это встроенное средство, показывающее часть внутренних процессов, производимых PowerShell. Я хочу предупредить, что этот командлет выводит достаточно много данных. Одним из полезных применений его может быть трассировка привязки параметров. Например, для команды, указанной выше, это будет выглядеть так:

```
PS> Trace-Command -Name ParameterBinding -PSHost -Expression {  
    Get-ChildItem log.txt | Get-Content }  
}
```

Он выводит значительное количество текста, и, к сожалению, выводит его в отладочный поток, который не так легко найти или перенаправить в файл. Ну да ладно. Нам более всего интересна вот эта часть:

```
BIND PIPELINE object to parameters: [Get-Content]  
PIPELINE object TYPE = [System.IO.FileInfo]  
RESTORING pipeline parameter's original values  
Parameter [ReadCount] PIPELINE INPUT ValueFromPipelineByPropertyName NO COERCION  
Parameter [TotalCount] PIPELINE INPUT ValueFromPipelineByPropertyName NO COERCION  
Parameter [Path] PIPELINE INPUT ValueFromPipelineByPropertyName NO COERCION  
Parameter [Credential] PIPELINE INPUT ValueFromPipelineByPropertyName NO COERCION  
Parameter [ReadCount] PIPELINE INPUT ValueFromPipelineByPropertyName NO COERCION  
Parameter [TotalCount] PIPELINE INPUT ValueFromPipelineByPropertyName NO COERCION  
Parameter [LiteralPath] PIPELINE INPUT ValueFromPipelineByPropertyName NO COERCION  
BIND arg [Microsoft.PowerShell.Core\FileSystem::C:\Users\Keith\log.txt] to parameter  
[LiteralPath]
```

Здесь я привожу лишь часть той информации, которую выводит *Trace-Command*. Из неё следует, что PowerShell пытался связать объект *FileInfo*, с параметрами командлета *Get-Content*. Попытки преобразования были безуспешными (NO COERCION), за исключением параметра *LiteralPath*. Это показывает, как *Get-Content* получает путь к файлу, однако никак не проясняет самой ситуации. Объект *FileInfo* не имеет свойства *LiteralPath* или расширенных свойств с именем *LiteralPath*.

Можно использовать второй способ - *.NET Reflector* и просмотреть исходный код PowerShell. После запуска *.NET Reflector* и загрузки сборки *Microsoft.PowerShell.Commands.Management.dll* необходимо найти *GetContentCommand* и посмотреть параметр *LiteralPath*:

```
[Alias(new string[] { "PSPath" })]  
[Parameter(Position = 0, ParameterSetName = "LiteralPath", Mandatory = true,  
    ValueFromPipeline = false, ValueFromPipelineByPropertyName = true)]  
public string[] LiteralPath { }
```

Обратите внимание на атрибут *Alias* этого параметра. Он создаёт ещё одно верное имя для параметра *LiteralPath* - *PSPath*. Оно соответствует расширенному свойству *PSPath*, которое PowerShell добавляет к объектам *FileInfo*. Именно это и позволяет передать *ByPropertyName* по конвейеру. Свойство *FileInfo* для *PSPath* соответствует параметру *LiteralPath*, хотя и является псевдонимом.

Очень часто объект может быть прямо передан по конвейеру следующему командлету, потому что PowerShell ищет наиболее подходящий параметр передаваемого объекта для привязки в качестве входного параметра.

Ещё один пример конвейеризации без использования командлета *Foreach-Object*:

```
PS> Get-ChildItem *.txt | Rename-Item -NewName {$_.name + '.bak'}
```

Теперь вы знаете, как PowerShell осуществляет привязку свойств входного объекта конвейера к параметрам командлета. А благодаря .NET Reflector мы знаем, что некоторые параметры имеют псевдонимы, наподобие *PSPath* для упрощения процесса привязки.

Мы говорили о передаче объекта по конвейеру с использованием *ByPropertyName*. Существует ещё один тип передачи, с использованием не имени объекта, а его значения - *ByValue*, и сейчас мы поговорим и о нём.

## Часть 9: Параметры привязки элементов конвейера ByValue (по значению)

Параметр привязки ByValue получает сам объект, а не его свойства, и пытается привязать его тип (и преобразуя его, если необходимо) к параметрам, которые отмечены как ByValue. Например, большинство командлетов \*-Object связывают ByValue с любым объектом, который передаётся им по конвейеру. Справка по Where-Object показывает это:

```
-inputObject <psobject>
```

Указывает объекты, которые необходимо отфильтровать. Если сохранить вывод команды в переменной, то можно использовать параметр InputObject, чтобы передать ее командлету Where-Object. Однако в большинстве случаев параметр InputObject не указывается в этой команде. Вместо этого при передаче объекта по конвейеру оболочка Windows PowerShell связывает его с параметром InputObject.

Требуется?	false
Позиция?	named
Значение по умолчанию	
Принимать входные данные конвейера?	true (ByValue)
Принимать подстановочные знаки?	false

Выглядит менее понятно, чем ByPropertyName. Спрашивается, как выполняется такая инструкция? Это одна из вещей, которые мне очень нравятся в PowerShell. Объект предоставляет много метаданных о самом себе, содержит много описывающей его информации. Вы можете легко просмотреть каждый параметр любого командлета, который в настоящее время загружен в PowerShell. Во-первых, давайте взглянем на информацию, доступную для параметров:

```
PS> Get-Command -CommandType cmdlet | Select -Expand ParameterSets |  
>>Select -Expand Parameters -First 1 | Get-Member
```

```
TypeName: System.Management.Automation.CommandParameterInfo
```

Name	MemberType	Definition
-----	-----	-----
...		
Aliases	Property	System.Collections.ObjectModel.Rea...
Attributes	Property	System.Collections.ObjectModel.Rea...
HelpMessage	Property	System.String HelpMessage {get;}
IsDynamic	Property	System.Boolean IsDynamic {get;}
IsMandatory	Property	System.Boolean IsMandatory {get;}
Name	Property	System.String Name {get;}
ParameterType	Property	System.Type ParameterType {get;}
Position	Property	System.Int32 Position {get;}
ValueFromPipeline	Property	System.Boolean ValueFromPipeline {...
ValueFromPipelineByPropertyName	Property	System.Boolean ValueFromPipelineBy...
ValueFromRemainingArguments	Property	System.Boolean ValueFromRemainingA...

нас интересуют здесь свойства Name и все ValueFromPipeline. Основываясь на этой информации, легко подсчитать количество для каждого типа:

```

PS> (Get-Command -CommandType cmdlet | Select -Expand ParameterSets |
>> Select -Expand Parameters |
>> Where {$_ .ValueFromPipeline -and !$_ .ValueFromPipelineByPropertyName} |
>> Measure-Object).Count
>>
55
PS> (Get-Command -CommandType cmdlet | Select -Expand ParameterSets |
>> Select -Expand Parameters |
>> Where {!$_ .ValueFromPipeline -and $_ .ValueFromPipelineByPropertyName} |
>> Measure-Object).Count
>>
196
PS> (Get-Command -CommandType cmdlet | Select -Expand ParameterSets |
>> Select -Expand Parameters |
>> Where {$_ .ValueFromPipeline -and $_ .ValueFromPipelineByPropertyName} |
>> Measure-Object).Count
>>
66

```

Вот результат:

Тип привязки в конвейере	Количество
ValueFromPipeline (по значению, ByValue)	55
ValueFromPipelineByPropertyName (по имени)	196
Оба типа	66

Как видно, привязка по имени используется более часто. Привязка по значению в конвейере применяется в основном с командлетами, обрабатывающими объекты общим способом — наподобие фильтрации и сортировки. В запросе, приведённом ниже, видно, что параметр *InputObject* используется наиболее часто в конвейере при привязке по значению:

```

PS> Get-Command -CommandType cmdlet | Select -Expand ParameterSets |
>> Select -Expand Parameters |
>> Where {$_ .ValueFromPipeline -and !$_ .ValueFromPipelineByPropertyName} |
>> Group Name -NoElement | Sort Count -Desc
>>

```

Count Name

-----

```

40 InputObject
 4 Message
 3 String
 2 SecureString
 1 ExecutionPolicy
 1 Object
 1 AclObject

```

## 1 DifferenceObject

### 1 Id 1 Command

Дальнейшие изыскания позволяют увидеть командлеты, которые используют привязку по значению к параметру `InputObject`.

*Примечание: Один параметр может встречаться в разных наборах параметров одного командлета. Это объясняет, почему здесь показано лишь 36 командлетов, при 40 упоминаниях `InputObject`.*

```
PS> $CmdletName = @{{Name='CmdletName';Expression={$_.Name}}}
PS> Get-Command -CommandType cmdlet | Select $CmdletName -Expand ParameterSets |
>> Select CmdletName -Expand Parameters |
>> Where {$_.ValueFromPipeline -and !$_.ValueFromPipelineByPropertyName} |
>> Group Name | Sort Count -Desc | Select -First 1 | Foreach {$_.Group} |
>> Sort CmdletName -Unique | Format-Wide CmdletName -AutoSize
>>
Add-History      Add-Member      ConvertTo-Html  Export-Clixml   Export-Csv      ForEach-Object
Format-Custom   Format-List     Format-Table     Format-Wide     Get-Member      Get-Process
Get-Service     Get-Unique     Group-Object    Measure-Command Measure-Object  Out-Default
Out-File        Out-Host       Out-Null        Out-Printer     Out-String      Restart-Service
Resume-Service  Select-Object  Select-String   Sort-Object     Start-Service   Stop-Process
Stop-Service    Suspend-Service Tee-Object      Trace-Command   Where-Object    Write-Output
```

Как можно увидеть, большинство из этих командлетов предназначается для работы с объектами в целом, без привязки к каким то конкретным типам.

*Примечание: Разработчикам командлетов следует помнить о том, что именно с помощью привязки параметров ваш командлет получает объекты из конвейера. Когда вы создаёте командлет на C# вам не доступен эквивалент переменной `$_`. Если вы хотите чтобы ваш командлет работал с конвейером, в атрибутах параметра (`ParameterAttribute`), свойство `ValueFromPipeline` и/или `ValueFromPipelineByPropertyName` должно быть установлено в `True` как минимум на одном из параметров командлета.*

Как уже сказано выше, большинство атрибутов `ByValue` приходится на параметр `InputObject` (тип `PsObject` или `PsObject[]`), так что они принимают практически всё. Однако не все командлеты работают таким способом. Параметр `-Id` (тип `long[]`) у командлета `Get-History` привязывается к объектам из конвейера по значению. Вывод команды `Trace-Command` показывает, как тяжело приходится потрудиться PowerShell'у, чтобы конвертировать передаваемый объект в нужный тип. В данном примере он превращает скалярное строковое значение `'1'` в массив `Int64`.

```
PS> Trace-Command -Name ParameterBinding -PSHost -Expression {'1' | Get-History}
BIND NAMED cmd line args [Get-History]
BIND POSITIONAL cmd line args [Get-History]
MANDATORY PARAMETER CHECK on cmdlet [Get-History]
CALLING BeginProcessing
BIND PIPELINE object to parameters: [Get-History]
PIPELINE object TYPE = [System.String]
RESTORING pipeline parameter's original values
Parameter [Id] PIPELINE INPUT ValueFromPipeline NO COERCION
BIND arg [1] to parameter [Id]
```

```
Binding collection parameter Id: argument type [String], parameter type
[System.Int64[]], collection type Array, element type [System.Int64],
no coerceElementType
```

```
Creating array with element type [System.Int64] and 1 elements
```

```
Argument type String is not IList, treating this as scalar
```

```
BIND arg [1] to param [Id] SKIPPED
```

```
Parameter [Id] PIPELINE INPUT ValueFromPipeline WITH COERCION
```

```
BIND arg [1] to parameter [Id]
```

```
COERCE arg type [System.Management.Automation.PSObject] to [System.Int64[]]
```

```
ENCODING arg into collection
```

```
Binding collection parameter Id: argument type [PSObject], parameter type
System.Int64[], collection type Array, element type [System.Int64],
coerceElementType
```

```
Creating array with element type [System.Int64] and 1 elements
```

```
Argument type PSObject is not IList, treating this as scalar
```

```
COERCE arg type [System.Management.Automation.PSObject] to [System.Int64]
```

```
CONVERT arg type to param type using LanguagePrimitives.ConvertTo
```

```
CONVERT SUCCESSFUL using LanguagePrimitives.ConvertTo: [1]
```

```
Adding scalar element of type Int64 to array position 0
```

```
Executing VALIDATION metadata:
```

```
[System.Management.Automation.ValidateRangeAttribute]
```

```
BIND arg [System.Int64[]] to param [Id] SUCCESSFUL
```

```
MANDATORY PARAMETER CHECK on cmdlet [Get-History]
```

```
CALLING ProcessRecord
```

```
CALLING EndProcessing
```

Заметьте, что при первой попытке PowerShell пытается конвертировать строку в массив *Int64*, но ему это не удаётся. Затем он пытается обработать исходные данные как *PSObject*. Он передаёт этот *PSObject* функции вспомогательного класса *LanguagePrimitives.ConvertTo()*, которая успешно преобразовывает '1' в массив *Int64*, содержащий элемент 1.

Когда параметр помечен и как *ByValue* и как *ByPropertyName* одновременно, PowerShell пытается связать его с аргументами в следующем порядке:

1. Привязать по значению, без преобразования типов
2. Привязать по имени свойства без преобразования типов
3. Привязать по значению, с преобразованием типов
4. Привязать по имени свойства без преобразования типов

Кроме того, еще используется дополнительная логика при нахождении лучшего совпадения среди нескольких наборов параметров.

Последнее замечание, касающееся параметров. Файлы справки PowerShell создаются не полностью автоматически, и, как результат - они не всегда корректны. Например, посмотрите справку к *Get-Content*, и попробуйте найти упоминание параметра *-Wait*. Вы не найдете его. Однако метаданные командлета всегда содержат полную и точную информацию, например:

```
PS> Get-Command Get-Content -Syntax
```

```
Get-Content [-Path] <String[]> [-ReadCount <Int64>] [-TotalCount <Int64>] [-Filter <String>]
```

```
[-Include <String[]>] [-Exclude <String[]>] [-Force] [-Credential <PSCredential>] [-Verbose]
```

```
[-Debug] [-ErrorAction <ActionPreference>] [-ErrorVariable <String>] [-OutVariable <String>]
```

```
[-OutBuffer <Int32>] [-Delimiter <String>] [-Wait] [-Encoding <FileSystemCmdletProviderEncoding>]
```

```
Get-Content [-LiteralPath] <String[]> [-ReadCount <Int64>] [-TotalCount <Int64>] [-Filter <String>]
```

```
[-Include <String[]>] [-Exclude <String[]>] [-Force] [-Credential <PSCredential>] [-Verbose]
```

`[-Debug] [-ErrorAction <ActionPreference>] [-ErrorVariable <String>] [-OutVariable <String>]  
[-OutBuffer <Int32>] [-Delimiter <String>] [-Wait] [-Encoding <FileSystemCmdletProviderEncoding>]`

Надеюсь, эта глава добавила вам немного понимания о том, как работают параметры привязывающиеся по значению (*ByValue*), и не только. В итоге, практически, вам не нужно много знать о привязке параметров, потому что в большинстве случаев она работает интуитивно. Просто старайтесь обращать внимание на те параметры, которые привязываются по имени (*ByPropertyName*). Их привязка в конвейере не всегда является очевидной.

<http://vam.in.ua/index.php/it/25-ms-powershell.html>

## Часть 10: Регулярные выражения – один из мощнейших инструментов PowerShell

Windows PowerShell основан на .NET Framework. То есть, он построен с использованием .NET Framework и предоставляет возможности .NET Framework пользователю. Одна из наиболее удобных возможностей в .NET Framework - класс *Regex* в пространстве имён *System.Text.RegularExpressions*. Это очень мощный механизм регулярных выражений. PowerShell использует его в ряде сценариев с помощью:

- оператора *-match*
- оператора *-notmatch*
- параметра *-Pattern* командлета *Select-String*

Конечно, чтобы использовать эти операторы и *Select-String* максимально эффективно, необходимо хорошо понимать применение регулярных выражений. Справочная система PowerShell содержит раздел с именем "about\_Regular\_Expression", который вы можете вызвать так:

```
PS> help about_reg*
```

Данный раздел представляет собой лишь краткое справочное руководство по различным метасимволам, с его помощью вы не научитесь создавать мощные регулярные выражения. Чтобы узнать, как получить максимальную отдачу от регулярных выражений, и, следовательно, PowerShell, я очень рекомендую прочитать книгу Jeffrey Friedl *Mastering Regular Expressions* (Джеффри Фридл, *Регулярные выражения*).

В механизме поддержки регулярных выражений PowerShell существует один недостаток, который необходимо знать. Большинство других скриптовых языков поддерживают синтаксис, позволяющий обнаружить все вхождения шаблона в строке. Например, в Perl я могу сделать так:

```
$_ = "paul xjohny xgeorgey xringoy stu pete brian"; # PERL script
($first, $second, $third) = /x(.+?)y/g;
```

К сожалению, командлет *Select-String* не поддерживает эту возможность в версии 1.0. Тем не менее, вы можете обойти это ограничение, используя класс *System.Text.RegularExpressions.Regex* напрямую. Нет необходимости указывать имя класса полностью, PowerShell имеет для него псевдоним [regex]. Например:

```
PS> $str = "paul xjohny xgeorgey xringoy stu pete brian"
PS> $first,$second,$third = ([regex]'x(.+?)y').Matches($str) |
>>Foreach {$_.Groups[1].Value}
PS> $first
john
PS> $second
george
PS> $third
ringo
```

Одна из вещей, за которой необходимо тщательно следить - написание регулярного выражения для поиска во всей строке. Например, вы решили использовать *Get-Content* для получения содержимого файла и применения к нему регулярного выражения. В этом случае необходимо помнить, что *Get-Content* считывает файл частями - строка за строкой. Для применения регулярного выражения ко всему содержимому файла, необходимо будет преобразовать это содержимое в одну строку. Я могу сделать это в PowerShell 1.0, например, так:

```
PS> $regex = [regex]'(?<CMultilineComment>/\*[^]*\*(?:[/][^]*\*)*/)'
PS> Get-Content foo.c | Join-String -Newline | Foreach {$regex.Matches($_)} |
>> Foreach {$_.Groups["CMultilineComment"].Value}
>>
```

Здесь я использовал командлет Join-String, разработанный PowerShell Community Extensions, который принимает отдельные строки, выводимые Get-Content, и создаёт из них одну строку. Также в примере использован именованный захват CMultilineComment. Этот пример демонстрирует, что при отсутствии возможностей в самом PowerShell, использование среды .NET может стать замечательным "запасным выходом".

## Дополнение для PowerShell 2.0

В PowerShell 2.0 появились новые возможности, помогающие выполнять поиск в вышеописанном случае. Во-первых, добавлен оператор объединения, с помощью которого можно соединять многострочный текст в одну строку. Во-вторых, в Select-String появились новые параметры, такие как -Context, -NotMatch и -AllMatches. Параметр AllMatches как раз подходит для решения предыдущей задачи. Вот как можно выполнить поиск комментариев в PowerShell 2.0:

```
$pattern = '(?<CMultilineComment>/\*[^]*\*(?:[/][^]*\*)*/)'
PS> (get-content .\foo.c) -join "`n" | Select-String $pattern -all |
>>Foreach {$_.Matches} | Foreach {$_.Value}
```

Использование регулярных выражений - очень мощный инструмент PowerShell. Научитесь его использовать, и он откроет вам много возможностей поиска и обработки текстовых данных.

## Часть 11: Сравнение массивов

В PowerShell используется несколько полезных операторов вроде `-contains`, которые проверяют, содержит ли массив заданный элемент. Но насколько я могу сказать, в PowerShell отсутствует простой способ проверить, является ли содержимое двух массивов идентичным. Такая потребность возникает довольно часто, и меня слегка удивило отсутствие такой возможности.

Я столкнулся с такой необходимостью при ответе на вопрос в группе новостей *microsoft.public.windows.powershell*. Его автор спрашивал о поиске файлов с кодировкой UTF-8 при помощи проверки метки порядка байтов (англ. *Byte Order Mark*, BOM - способ определения формата представления Юникода в текстовом файле; для обозначения формата UTF-8 в заголовке файла используется последовательность EF BB BF - прим. переводчика). Одним из простых способов решения такой задачи может быть таким:

```
PS> $preamble = [System.Text.Encoding]::UTF8.GetPreamble()
PS> $preamble | foreach {"0x{0:X2}" -f $_}
0xEF
0xBB
0xBF
PS> $fileHeader = Get-Content Utf8File.txt -Enc byte -Total 3
PS> $fileheader | foreach {"0x{0:X2}" -f $_}
0xEF
0xBB
0xBF
```

Визуально, конечно, легко проверить, совпадают значения или нет. Но, к сожалению, визуальная проверка не работает в сценариях. Можно также проверять каждый отдельный элемент, что годится для трех элементов массива, но когда вы столкнётесь, скажем, с 10 элементами, этот подход начинает выглядеть утомительным. Вы думаете, что мы могли бы просто сравнить эти два массива непосредственно примерно так:

```
PS> $preamble -eq $fileHeader | Get-TypeName
WARNING: Get-TypeName did not receive any input. The input may be an empty
collection. You can either prepend the collection expression with the comma
operator e.g. ",$collection | gtn" or you can pass the variable or expression
to Get-TypeName as an argument e.g. "gtn $collection".
PS> $preamble -eq 0xbb
187
```

*ЗАМЕЧАНИЕ: Get-TypeName - это функция, написанная сообществом PowerShell Community Extensions.*

Сравнение массивов с помощью оператора `-eq` на самом деле не сравнивает содержимое этих двух массивов. Как показывает код в первой строке, эта конструкция ничего не возвращает. Когда слева от оператора `-eq` расположен массив, PowerShell возвращает *элементы массива, значение которых совпадает со значением, заданным справа от оператора*. Последние две строки в примере выше это поясняют - при сравнении с `'0xbb'` возвращен элемент со значением 187.

Похоже, нам придётся создать свой собственный механизм сравнения массивов. Вот один из способов:

```

function AreArraysEqual($a1, $a2) {
    if ($a1 -isnot [array] -or $a2 -isnot [array]) {
        throw "Both inputs must be an array"
    }
    if ($a1.Rank -ne $a2.Rank) {
        return $false
    }
    if ([System.Object]::ReferenceEquals($a1, $a2)) {
        return $true
    }
    for ($r = 0; $r -lt $a1.Rank; $r++) {
        if ($a1.GetLength($r) -ne $a2.GetLength($r)) {
            return $false
        }
    }

    $enum1 = $a1.GetEnumerator()
    $enum2 = $a2.GetEnumerator()
    while ($enum1.MoveNext() -and $enum2.MoveNext()) {
        if ($enum1.Current -ne $enum2.Current) {
            return $false
        }
    }
    return $true
}

```

Он работает так, как и ожидалось:

```

PS> AreArraysEqual $preamble $fileHeader
True

```

Однако существует способ сделать сравнение и штатными средствами PowerShell, но он не совсем очевиден. По крайней мере, для меня.

```

PS> @(Compare-Object $preamble $fileHeader -sync 0).Length -eq 0
True

```

*Compare-Object* сравнивает массивы, и если различий не будет, на выход ничего передано не будет. Если мы "обернём" вывод *Compare-Object* в массив с помощью оператора *@()*, то получим массив, содержащий 0 или более элементов, в зависимости от результата. Таким образом, если длина этого массива будет равна 0, это будет означать, что сравниваемые массивы равны.

*Compare-Object* сравнивает два объекта с точки зрения одинакового набора элементов. Обычно при этом не учитывается последовательность расположения элементов. Посмотрите:

```

PS> $a1 = 1,1,2
PS> $a2 = 1,2,1
PS> @(Compare-Object $a1 $a2).length -eq 0
True

```

Скажем так, это не совсем то, когда мы говорим о сравнении на эквивалентность. К счастью, можно использовать параметр *SyncWindow*, установив ему значение 0. Это заставит *Compare-Object* сравнивать объекты, соблюдая последовательность расположения элементов в них.

Давайте замерим производительность этих двух методов:

```
PS> $a1 = 1..10000
PS> $a2 = 1..10000
PS> (Measure-Command { AreArraysEqual $a1 $a2 }).TotalSeconds
1.236252
PS> (Measure-Command { @(Compare-Object $a1 $a2 -sync 0).Length -eq 0 }).TotalSeconds
0.3259954
```

Как видим, *Compare-Object* "уверенно побеждает" мою функцию *AreArraysEqual*, что, впрочем, и не удивительно\*.

*\* - за исключением случая, когда происходит сравнение объектов, ссылающихся на один и тот же массив. В этом случае функция оказывается на два порядка быстрее. Видимо, Compare-Object не использует проверку System.Object.ReferenceEquals. Следует признать, что это частный случай сценария.*

Всё-таки, *Compare-Object* использует скомпилированный код, а функция - интерпретируется. Если вам необходим быстрый способ сравнить массивы, просто запомните, что массивы - это тоже объекты, а для сравнения объектов лучше всего использовать *Compare-Object*.  
<http://vam.in.ua/index.php/it/25-ms-powershell.html>

## Часть 12: Старайтесь использовать `Set-PSDebug -Strict` в своих сценариях

Windows PowerShell, как и многие другие динамические языки, позволяет использовать переменные без объявления типа и без присваивания им начальных значений. Это удобно для интерактивного использования, и вы можете делать что-то наподобие этого:

```
PS> Get-ChildItem | Foreach -Process {$sum += $_.Name.Length} -End {$sum}
```

Здесь переменная `$sum` не объявлена, но мы добавляем к ней значение и присваиваем его. PowerShell просто берёт значение `$null` и преобразует его в `0` в этом случае. Попробуйте сами ввести:

```
PS> $xyzy -eq $null
True
```

Это не означает, что ранее эта переменная была объявлена. Конечно, мы можем убедиться в том, что она не определена, обратившись к диску переменных:

```
PS> Test-Path Variable:\xyzy
False
```

Так почему мы должны стараться использовать `Set-PSDebug -Strict` в сценариях? Дело в том, что однажды вы можете ошибиться, допустив, например, досадную опечатку в коде. На обнаружение такой ошибки и её устранение вам придётся тратить время. Возможно, вы захотите избегать повторения таких ошибок в будущем. Возьмём для примера такой скрипт:

```
$succeeded = test-path C:\ProjectX\Src\BuiltComponents\Release\app.exe

if ($succeeded) {
    ... <archive bits, label build, etc>
}
else {
    ... <email team that build failed, etc>
}
```

Этот скрипт содержит ошибку, о которой вам не сообщит PowerShell. Он с радостью покажет *build failed* даже в случае существования файла *app.exe*. Потому что в названии переменной, которой присваивается результат проверки пути файла, допущена небольшая опечатка. Здесь, в маленьком фрагменте, вероятно, обнаружить опечатку не столь сложно, но если речь будет идти о сценарии в несколько сотен строк?

Вы можете предотвратить проблемы такого рода, поместив команду `Set-PSDebug -Strict` в начале файла сценария, сразу после инструкции `param()` (если она имеется). Например, этот же сценарий в виде *Foo.ps1*:

```
Set-PSDebug -Strict
$succeeded = test-path C:\ProjectX\Src\BuiltComponents\Release\app.exe
if ($succeeded) {
    "yeah"
}
else {
    "doh"
}
```

```
PS C:\Temp> .\foo.ps1
```

```
The variable $succeeded cannot be retrieved because it has not been set yet.
```

```
At C:\Temp\foo.ps1:6 char:14
```

```
+ if ($Succeeded) <<<< {
```

Что бы произошло если бы мы не написали *Set-PSDebug -Strict*? Сценарий будет всегда выводить "doh".

В некоторых случаях избежать подобные ошибки помогает инициализация переменных. Возможно, название этой части звучит и слишком "перестраховочно", и вы можете не использовать *Set-PSDebug -Strict* при написании сценариев. Как всегда - решать вам.

### Примечание для PowerShell 2.0

В PowerShell 2.0, следует использовать новый командлет *Set-StrictMode*:

```
param(...)
```

```
Set-StrictMode -version Latest
```

```
<rest of your script>
```

*Set-StrictMode* проверяет больше, чем использование только инициализированных переменных. Он также осуществляет проверку ссылок на несуществующие свойства, вызовы функций и методов .NET и неименованные переменные типа *\${}*.

<http://vam.in.ua/index.php/it/25-ms-powershell.html>

## Часть 13: Комментирование строк в файле сценария

Windows PowerShell 1.0 не позволяет комментировать несколько строк. В версии 2.0 этот недостаток устранён, но я расскажу об этом в конце этого раздела. Если вы используете исключительно PowerShell 2.0, я бы всё равно рекомендовал ознакомиться с этим разделом, потому что в нём описаны некоторые особенности использования многострочных текстовых переменных (*here-string*). Многострочные комментарии полезны, когда необходимо закомментировать сразу несколько строк в сценарии. Существует и другой допустимый приём - использование многострочных текстовых переменных. Они позволяют ввести несколько строк кода, предотвращая их немедленную интерпретацию. Однако то, как будет расценивать содержимое строки интерпретатор PowerShell, зависит от типа используемой строки. Например, в двойных кавычках переменные будут заменены их значениями, а выражения - выполняться. Пример строки с двойными кавычками, в которой выполнено выражение :

```
PS> @"
>> $(get-process)
>> "@
>>
System.Diagnostics.Process (audiodg) System.Diagnostics.Process (csrss)
...
```

А в строке с одинарными кавычками такого не произойдёт:

```
PS> '@'
>> $(get-process)
>> '@
>>
$(get-process)
```

Используйте многострочный текст с одинарными кавычками для комментирования строк сценария, так как в них никакие выражения не выполняются. Необходимо лишь помнить, что строка - это выражение, и если больше ничего не делать, её содержимое будет выведено в консоль. Обычно бывает нежелательно отображать закомментированные строки на экране. Чтобы избежать вывода этой информации, можно привести строку к типу *[void]* или перенаправить её в *\$null*:

```
[void]@'
"Получаем информацию о процессах"
get-process | select Name, Id
"Останавливаем процессы с именем vd*"
stop-process -name vd*
'@
```

Это позволяет эффективно комментировать строки скрипта. У этого метода существует несколько особенностей. После начальных символов последовательности '@' не должно быть пробелов. Если после этой последовательности будет стоять хотя бы один пробел, вы получите вот такую загадочную ошибку:

```
Unrecognized token in source text.
At C:\Temp\foo.ps1:1 char:1
+ @ <<<< '
```

Далее, закрывающие символы '@' должны начинаться с начала строки, иначе вы получите ещё одну загадку:

```
Encountered end of line while processing a string token.  
At C:\Temp\foo.ps1:1 char:3  
+ '@' <<<<
```

Ещё одна особенность - в PowerShell 1.0 нельзя вложить одну автономную строку в другую строку того же типа. Это означает, что вы не можете использовать строку с одинарными кавычками для окружения комментарием части сценария, если внутри этой части уже содержится строка с одинарными кавычками.

## Дополнение для PowerShell 2.0

В PowerShell 2.0 введена корректная поддержка многострочных комментариев. Используются они так:

```
<# Это многострочный  
комментарий  
в PowerShell 2.0  
#>
```

И наконец-то, многострочный текст в PowerShell 2.0 может быть вложен друг в друга:

```
@"  
<Processes>  
  $(Get-Process | Foreach {  
@"  
  <Process name="$($_.name)" id="$($_.id)" workingSet="$($_.ws)">`r`n  
"@  
  })  
  </Processes>  
"@
```

## Дополнительные материалы

Оригинал документа на английском языке вы можете [загрузить с блога Keith Hill](#). Перевод выполнен Сергеем Вальковским для [WindowsFAQ.ru](#). Дополнительные материалы и информацию о Windows PowerShell на русском языке можно найти в блогах Василия Гусева <http://xaegr.wordpress.com>, Андрея Бешкова <http://blogs.technet.com/abeshkov/default.aspx>, Дмитрия Сотникова <http://www.itcommunity.ru/blogs/dmitrysotnikov/default.aspx>, и на крупнейшем собрании видеоуроков по PowerShell на русском языке - <http://www.techdays.ru/Category.aspx?Tag=PowerShell>.  
<http://vam.in.ua/index.php/it/25-ms-powershell.html>