Бесплатная электронная книга

УЧУСЬ PowerShell

Free unaffiliated eBook created from
Stack Overflow contributors.

#powershell

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: powershell

It is an unofficial and free PowerShell ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official PowerShell.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с PowerShell

замечания

Windows PowerShell - это компонент оболочки и сценариев в системе управления Windows, инфраструктура управления автоматизацией и конфигурацией, разработанная Microsoft, основанная на .NET Framework. PowerShell устанавливается по умолчанию во всех поддерживаемых версиях клиентских и серверных операционных систем Windows с Windows 7 / Windows Server 2008 R2. Powershell можно обновить в любое время, загрузив более позднюю версию Windows Management Framework (WMF). Версия «Alpha» PowerShell 6 является кросс-платформенной (Windows, Linux и OS X) и ее необходимо загрузить и установить с этой страницы выпуска .

Дополнительные ресурсы:

- Документация MSDN: https://msdn.microsoft.com/en-us/powershell/scripting/powershell-scripting
- TechNet: https://technet.microsoft.com/en-us/scriptcenter/dd742419.aspx
    - О страницах
- Галерея PowerShell: https://www.powershellgallery.com/
- Блог MSDN: https://blogs.msdn.microsoft.com/powershell/
- Github: https://github.com/powershell
- Сайт сообщества: http://powershell.com/cs/

Версии

Версия	Включено в Windows	Заметки	Дата выхода
1,0	XP / Server 2008		2006-11-01
2,0	7 / Server 2008 R2		2009-11-01
3.0	8 / Сервер 2012		2012-08-01
4,0	8.1 / Сервер 2012 R2		2013-11-01
5.0	10 / Технический обзор сервера 2016		2015-12-16
5,1	10 Юбилейная версия / Сервер 2016		2017-01-27

Examples

Установка или настройка

Windows

PowerShell входит в состав Windows Management Framework. Установка и настройка не требуются в современных версиях Windows.

Обновления PowerShell можно выполнить, установив новую версию Windows Management Framework.

Другие платформы

«Beta» версия PowerShell 6 может быть установлена на других платформах. Пакеты установки доступны здесь .

Например, PowerShell 6 для Ubuntu 16.04 публикуется для пакетных репозиториев для простой установки (и обновлений).

Для установки выполните следующие действия:

```
# Import the public repository GPG keys
curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -

# Register the Microsoft Ubuntu repository
curl https://packages.microsoft.com/config/ubuntu/16.04/prod.list | sudo tee
/etc/apt/sources.list.d/microsoft.list

# Update apt-get
sudo apt-get update

# Install PowerShell
sudo apt-get install -y powershell

# Start PowerShell
powershell
```

После регистрации репозитория Microsoft один раз в качестве суперпользователя, с этого момента вам просто нужно использовать `sudo apt-get upgrade powershell` update `sudo apt-get upgrade powershell` для его обновления. Затем просто запустите `powershell`

Разрешить скрипты, хранящиеся на вашем компьютере, запускать незаписанные

По соображениям безопасности PowerShell по умолчанию настроен только на то, чтобы разрешить выполнение скриптов. Выполнение следующей команды позволит вам запускать неподписанные скрипты (для этого необходимо запустить PowerShell в качестве администратора).

```
Set-ExecutionPolicy RemoteSigned
```

Другой способ запускать сценарии PowerShell - использовать `Bypass` как `ExecutionPolicy` :

```
powershell.exe -ExecutionPolicy Bypass -File "c:\MyScript.ps1"
```

Или из существующей консоли PowerShell или сеанса ISE, запустив:

```
 Set-ExecutionPolicy Bypass Process
```

Временное обходное решение для политики выполнения также может быть достигнуто путем запуска исполняемого файла Powershell и передачи любой допустимой политики в качестве параметра `-ExecutionPolicy` . Эта политика действует только во время жизни процесса, поэтому административный доступ к реестру не требуется.

```
C:\>powershell -ExecutionPolicy RemoteSigned
```

Существует несколько других политик, и сайты в Интернете часто рекомендуют вам использовать `Set-ExecutionPolicy Unrestricted` . Эта политика сохраняется до изменения и снижает уровень безопасности системы. Это не рекомендуется. Рекомендуется использовать `RemoteSigned` поскольку он позволяет локально сохраненный и написанный код и требует, чтобы удаленный код был подписан с сертификатом из доверенного корня.

Кроме того, остерегайтесь того, что политика выполнения может быть принудительно применена групповой политикой, так что даже если политика будет изменена на `Unrestricted` общесистемную, групповая политика может вернуть эту настройку в следующий интервал ее выполнения (обычно 15 минут). Вы можете увидеть политику выполнения, установленную в различных областях, используя `Get-ExecutionPolicy -List`

Документация TechNet:
Set-ExecutionPolicy
about_Execution_Policies

Псевдонимы и аналогичные функции

В PowerShell существует много способов добиться того же результата. Это можно хорошо иллюстрировать простым и знакомым примером `Hello World` :

Использование `Write-Host` :

```
Write-Host "Hello World"
```

Использование `Write-Output` :

```
Write-Output 'Hello world'
```

Стоит отметить, что хотя `Write-Output` & `Write-Host` и записываются на экран, есть тонкая разница. `Write-Host` записывает *только* в stdout (т. Е. Экран консоли), тогда как `Write-Output` записывает как в stdout *AND, так* и в выходной [success] поток, допускающий перенаправление . Перенаправление (и потоки в целом) позволяют выводить одну команду для ввода в качестве входа в другую, включая присвоение переменной.

```
> $message = Write-Output "Hello World"
> $message
"Hello World"
```

Эти подобные функции не являются псевдонимами, но могут давать одинаковые результаты, если вы хотите избежать «загрязнения» потока успеха.

`Write-Output` выдается с псевдонимом `Echo` или `Write`

```
Echo 'Hello world'
Write 'Hello world'
```

Или просто набрав «Hello world»!

```
'Hello world'
```

Все это приведет к ожидаемому выпуску консоли

```
Hello world
```

Другим примером псевдонимов в PowerShell является общее сопоставление команд старой команды командной строки и команд BASH командлетам PowerShell. Все следующие производят список каталогов текущего каталога.

```
C:\Windows> dir
C:\Windows> ls
C:\Windows> Get-ChildItem
```

Наконец, вы можете создать свой собственный псевдоним с помощью командлета Set-Alias! В качестве примера давайте рассмотрим `Test-NetConnection` , который по сути является PowerShell, эквивалентным команде ping командной строки, для «ping».

```
Set-Alias -Name ping -Value Test-NetConnection
```

Теперь вы можете использовать `ping` вместо `Test-NetConnection` ! Имейте в виду, что если псевдоним уже используется, вы перезапишете ассоциацию.

Псевдоним будет жив, пока сессия не будет активна. Как только вы закроете сеанс и попытаетесь запустить псевдоним, который вы создали на своем последнем сеансе, он не будет работать. Чтобы преодолеть эту проблему, вы можете импортировать все свои

псевдонимы из Excel в свою сессию один раз, прежде чем начинать свою работу.

Трубопровод - использование вывода командлета PowerShell

Один из первых вопросов, который возникает у людей, когда они начинают использовать PowerShell для скриптинга, - это то, как манипулировать выходом из командлета для выполнения другого действия.

Символ трубопровода `|` используется в конце командлета, чтобы взять данные, которые он экспортирует, и передать их следующему командлету. Простым примером является использование Select-Object для отображения свойства Name файла, показанного в Get-ChildItem:

```
Get-ChildItem | Select-Object Name
#This may be shortened to:
gci | Select Name
```

Более продвинутое использование конвейера позволяет нам выводить вывод командлета в цикл foreach:

```
Get-ChildItem | ForEach-Object {
    Copy-Item -Path $_.FullName -destination C:\NewDirectory\
}

#This may be shortened to:
gci | % { Copy $_.FullName C:\NewDirectory\ }
```

Обратите внимание, что в приведенном выше примере используется автоматическая переменная $ _. $ _ - это короткий псевдоним $ PSItem, который является автоматической переменной, которая содержит текущий элемент в конвейере.

Комментирование

Чтобы прокомментировать силовые скрипты, добавив строку, используя символ `#` (хэш)

```
# This is a comment in powershell
Get-ChildItem
```

Вы также можете иметь многострочные комментарии, используя `<#` и `#>` в начале и в конце комментария соответственно.

```
<#
This is a
multi-line
comment
#>
Get-ChildItem
```

Вызов методов библиотеки .Net

Статические методы библиотеки .Net можно вызывать из PowerShell, инкапсулируя полное имя класса в третью скобку и затем вызывая метод, используя `::`

```
#calling Path.GetFileName()
C:\> [System.IO.Path]::GetFileName('C:\Windows\explorer.exe')
explorer.exe
```

Статические методы можно вызывать из самого класса, но для вызова нестатических методов требуется экземпляр класса .Net (объект).

Например, метод AddHours не может быть вызван из самого класса System.DateTime. Для этого требуется экземпляр класса:

```
C:\> [System.DateTime]::AddHours(15)
Method invocation failed because [System.DateTime] does not contain a method named 'AddHours'.
At line:1 char:1
+ [System.DateTime]::AddHours(15)
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : MethodNotFound
```

В этом случае мы сначала создаем объект , например:

```
C:\> $Object = [System.DateTime]::Now
```

Затем мы можем использовать методы этого объекта, даже методы, которые нельзя вызвать непосредственно из класса System.DateTime, например метод AddHours:

```
C:\> $Object.AddHours(15)

Monday 12 September 2016 01:51:19
```

Создание объектов

Командлет `New-Object` используется для создания объекта.

```
# Create a DateTime object and stores the object in variable "$var"
$var = New-Object System.DateTime

# calling constructor with parameters
$sr = New-Object System.IO.StreamReader -ArgumentList "file path"
```

Во многих случаях будет создан новый объект для экспорта данных или передачи его другому командному элементу. Это можно сделать так:

```
$newObject = New-Object -TypeName PSObject -Property @{
    ComputerName = "SERVER1"
```

```
    Role = "Interface"
    Environment = "Production"
}
```

Существует много способов создания объекта. Следующий метод, вероятно, самый короткий и быстрый способ создания `PSCustomObject` :

```
$newObject = [PSCustomObject]@{
    ComputerName = 'SERVER1'
    Role         = 'Interface'
    Environment  = 'Production'
}
```

Если у вас уже есть объект, но вам нужно только одно или два дополнительных свойства, вы можете просто добавить это свойство с помощью `Select-Object` :

```
Get-ChildItem | Select-Object FullName, Name,
    @{Name='DateTime'; Expression={Get-Date}},
    @{Name='PropertieName'; Expression={'CustomValue'}}
```

Все объекты могут храниться в переменных или передаваться в конвейер. Вы также можете добавить эти объекты в коллекцию, а затем показать результаты в конце.

Коллекции объектов хорошо работают с Export-CSV (и Import-CSV). Каждая строка CSV - это объект, в каждом столбце - свойство.

Команды форматирования преобразуют объекты в текстовый поток для отображения. Избегайте использования команд Format- * до последнего этапа обработки данных, чтобы поддерживать удобство использования объектов.

Прочитайте Начало работы с PowerShell онлайн: https://riptutorial.com/ru/powershell/topic/822/начало-работы-с-powershell

глава 2: Cmdlet Naming

Вступление

CmdLets следует называть с помощью схемы именования `<verb>-<noun>` , чтобы улучшить открытость.

Examples

Глаголы

Глаголы, используемые для обозначения CmdLets, должны быть названы из глаголов из списка, представленного `Get-Verb`

Более подробную информацию о том, как использовать глаголы, можно найти в Утвержденных глаголах для Windows PowerShell

Существительные

Существительные всегда должны быть единственными.

Будьте в согласии с существительными. Например, `Find-Package` нуждается в провайдере, а существительным является `PackageProvider` не `ProviderPackage` .

Прочитайте Cmdlet Naming онлайн: https://riptutorial.com/ru/powershell/topic/8703/cmdlet-naming

глава 3: HashTables

Вступление

Хэш-таблица - это структура, которая отображает ключи к значениям. Подробнее см. В разделе « Хэш-таблица» .

замечания

Важной концепцией, основанной на таблицах Hash, является Splatting . Это очень полезно для создания большого количества вызовов с повторяющимися параметрами.

Examples

Создание таблицы хешей

Пример создания пустой HashTable:

```
$hashTable = @{}
```

Пример создания HashTable с данными:

```
$hashTable = @{
    Name1 = 'Value'
    Name2 = 'Value'
    Name3 = 'Value3'
}
```

Получите доступ к значению хэш-таблицы по ключу.

Пример определения хэш-таблицы и доступа к значению с помощью ключа

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
$hashTable.Key1
#output
Value1
```

Пример доступа к ключу с недопустимыми символами для имени свойства:

```
$hashTable = @{
    'Key 1' = 'Value3'
    Key2 = 'Value4'
```

```
}
$hashTable.'Key 1'
#Output
Value3
```

Зацикливание по хеш-таблице

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}

foreach($key in $hashTable.Keys)
{
    $value = $hashTable.$key
    Write-Output "$key : $value"
}
#Output
Key1 : Value1
Key2 : Value2
```

Добавить пару ключевых значений в существующую хеш-таблицу

Например, чтобы добавить ключ «Key2» со значением «Value2» в хэш-таблицу, используя оператор добавления:

```
$hashTable = @{
    Key1 = 'Value1'
}
$hashTable += @{Key2 = 'Value2'}
$hashTable

#Output

Name                           Value
----                           -----
Key1                           Value1
Key2                           Value2
```

Например, чтобы добавить ключ «Key2» со значением «Value2» в хэш-таблицу с помощью метода «Добавить»:

```
$hashTable = @{
    Key1 = 'Value1'
}
$hashTable.Add("Key2", "Value2")
$hashTable

#Output

Name                           Value
----                           -----
Key1                           Value1
Key2                           Value2
```

Перечисление через ключи и пары ключевых значений

Перечисление через ключи

```
foreach ($key in $var1.Keys) {
    $value = $var1[$key]
    # or
    $value = $var1.$key
}
```

Перечисление через пары ключевого значения

```
foreach ($keyvaluepair in $var1.GetEnumerator()) {
    $key1 = $_.Key1
    $val1 = $_.Val1
}
```

Удаление пары значений ключа из существующей хеш-таблицы

Например, чтобы удалить ключ «Key2» со значением «Value2» из хеш-таблицы, используя оператор remove:

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
$hashTable.Remove("Key2", "Value2")
$hashTable

#Output

Name                           Value
----                           -----
Key1                           Value1
```

Прочитайте HashTables онлайн: https://riptutorial.com/ru/powershell/topic/8083/hashtables

глава 4: Loops

Вступление

Цикл представляет собой последовательность команд (инструкций), которые непрерывно повторяются до тех пор, пока не будет достигнуто определенное условие. Возможность многократного выполнения вашей программы блока кода является одной из самых простых, но полезных задач в программировании. Цикл позволяет писать очень простой оператор для получения значительно большего результата просто повторением. Если условие достигнуто, следующая инструкция «проваливается» к следующей последовательной инструкции или ветвям вне цикла.

Синтаксис

- for (<Инициализация>; <Условие>; <Повторение>) {<Script_Block>}
- <Коллекция> | Foreach-Object {<Script_Block_with _ $ __ as_current_item>}
- foreach (<Item> in <Collection>) {<Script_Block>}
- while (<Условие>) {<Script_Block>}
- do {<Script_Block>}, a (<Условие>)
- do {<Script_Block>} до (<Условие>)
- <Collection> .foreach ({<Script_Block_with _ $ __ as_current_item>})

замечания

Для каждого

Существует несколько способов запуска цикла foreach в PowerShell, и все они приносят свои преимущества и недостатки:

Решение	преимущества	Недостатки
Предписание	Самый быстрый. Лучше всего работает со статическими коллекциями (хранится в переменной).	Отсутствие ввода или вывода конвейера
Метод ForEach	Такой же синтаксис скриптового блока,	Нет поддержки ввода

Решение	преимущества	Недостатки
()	что и `Foreach-Object` , но быстрее. Лучше всего работает со статическими коллекциями (хранится в переменной). Поддерживает вывод конвейера.	трубопровода. Требуется PowerShell 4.0 или выше
Foreach-Object (командлет)	Поддерживает ввод и вывод конвейера. Поддерживает начальные и конечные скрипты для инициализации и закрытия соединений и т. Д. Наиболее гибкое решение.	Slowest

Спектакль

```
$foreach = Measure-Command { foreach ($i in (1..1000000)) { $i * $i } }
$foreachmethod = Measure-Command { (1..1000000).ForEach{ $_ * $_ } }
$foreachobject = Measure-Command { (1..1000000) | ForEach-Object { $_ * $_ } }

"Foreach: $($foreach.TotalSeconds)"
"Foreach method: $($foreachmethod.TotalSeconds)"
"ForEach-Object: $($foreachobject.TotalSeconds)"

Example output:

Foreach: 1.9039875
Foreach method: 4.7559563
ForEach-Object: 10.7543821
```

В то время как `Foreach-Object` является самым медленным, поддержка конвейера может быть полезна, поскольку позволяет обрабатывать элементы по мере их поступления (при чтении файла, получении данных и т. Д.). Это может быть очень полезно при работе с большими данными и низкой памятью, так как вам не нужно загружать все данные в память перед обработкой.

Examples

За

```
for($i = 0; $i -le 5; $i++){
    "$i"
}
```

Типичным использованием цикла for является работа с подмножеством значений в массиве. В большинстве случаев, если вы хотите итерировать все значения в массиве, рассмотрите использование инструкции foreach.

Для каждого

`ForEach` имеет два разных значения в PowerShell. Одним из них является ключевое слово, а другое - псевдоним для командлета ForEach-Object . Здесь описывается первая.

Этот пример демонстрирует печать всех элементов в массиве на хосте консоли:

```
$Names = @('Amy', 'Bob', 'Celine', 'David')

ForEach ($Name in $Names)
{
    Write-Host "Hi, my name is $Name!"
}
```

Этот пример демонстрирует захват вывода цикла ForEach:

```
$Numbers = ForEach ($Number in 1..20) {
    $Number # Alternatively, Write-Output $Number
}
```

Как и в последнем примере, вместо этого этот пример демонстрирует создание массива до хранения цикла:

```
$Numbers = @()
ForEach ($Number in 1..20)
{
    $Numbers += $Number
}
```

В то время как

Цикл while будет оценивать условие, и если true выполнит действие. Пока условие оценивается как true, действие будет продолжаться.

```
while(condition){
  code_block
}
```

В следующем примере создается цикл, который будет отсчитывать от 10 до 0

```
$i = 10
while($i -ge 0){
    $i
    $i--
}
```

В отличие от цикла `Do` -While условие оценивается до первого выполнения действия. Действие не будет выполнено, если начальное условие будет равно false.

Примечание. При оценке условия PowerShell будет рассматривать существование возвращаемого объекта как истинного. Это можно использовать несколькими способами, но ниже приведен пример мониторинга процесса. В этом примере будет создан процесс создания блокнота, а затем скройте текущую оболочку, пока выполняется этот процесс. Когда вы вручную закрываете экземпляр блокнота, условие while будет терпеть неудачу, и цикл прерывается.

```
Start-Process notepad.exe
while(Get-Process notepad -ErrorAction SilentlyContinue){
  Start-Sleep -Milliseconds 500
}
```

ForEach-Object

Командлет `ForEach-Object` работает аналогично инструкции `foreach` , но берет свой вход из конвейера.

Основное использование

```
$object | ForEach-Object {
    code_block
}
```

Пример:

```
$names = @("Any","Bob","Celine","David")
$names | ForEach-Object {
    "Hi, my name is $_!"
}
```

`Foreach-Object` имеет два псевдонима по умолчанию, `foreach` и `%` (сокращенный синтаксис). Наиболее распространенным является `%` потому что `foreach` можно путать с оператором foreach . Примеры:

```
$names | % {
    "Hi, my name is $_!"
}

$names | foreach {
    "Hi, my name is $_!"
}
```

Расширенное использование

`Foreach-Object` выделяется из альтернативных решений `foreach` потому что это командлет, что означает, что он предназначен для использования конвейера. Из-за этого он поддерживает три скриптовых блока, как командлет или расширенную функцию:

- **Начать** : Выполняется один раз, прежде чем перебирать элементы, которые поступают из конвейера. Обычно используется для создания функций для использования в цикле, создания переменных, открытия соединений (база данных, веб +) и т. Д.
- **Процесс** : Выполняется один раз за элемент, полученный из конвейера. «Нормальный» кодовый блок foreach. Это значение по умолчанию используется в приведенных выше примерах, когда параметр не указан.
- **Конец** : Выполняется один раз после обработки всех элементов. Обычно используется для закрытия соединений, создания отчета и т. Д.

Пример:

```
"Any","Bob","Celine","David" | ForEach-Object -Begin {
    $results = @()
} -Process {
    #Create and store message
    $results += "Hi, my name is $_!"
} -End {
    #Count messages and output
    Write-Host "Total messages: $($results.Count)"
    $results
}
```

Делать

Do-Loops полезны, когда вы всегда хотите запускать блок кода хотя бы один раз. До-петля будет оценивать условие после выполнения кодаблока, в отличие от цикла while, который делает это перед выполнением кодаблока.

Вы можете использовать do-loops двумя способами:

- Петля, *пока* условие верно:

  ```
  Do {
      code_block
  } while (condition)
  ```

- Loop *до* тех *пор, пока* условие не будет истинным, другими словами, цикл, пока условие ложно:

  ```
  Do {
      code_block
  } until (condition)
  ```

Реальные примеры:

```
$i = 0

Do {
```

```
    $i++
    "Number $i"
} while ($i -ne 3)

Do {
    $i++
    "Number $i"
} until ($i -eq 3)
```

Do-While и Do-Until - антонимы. Если код внутри одного и того же, условие будет отменено. Приведенный выше пример иллюстрирует это поведение.

Метод ForEach ()

4,0

Вместо командлета `ForEach-Object` здесь также есть возможность использовать метод `ForEach` непосредственно на объектных массивах, например, так

```
(1..10).ForEach({$_ * $_})
```

или - при желании - круглые скобки вокруг блока сценария могут быть опущены

```
(1..10).ForEach{$_ * $_}
```

Оба результата приведут к выходу ниже

```
1
4
9
16
25
36
49
64
81
100
```

Продолжить

`Continue` оператор работает в `For` , `ForEach` , `While` и `Do` петлю. Он пропускает текущую итерацию цикла, прыгая на вершину самого внутреннего цикла.

```
$i =0
while ($i -lt 20) {
    $i++
    if ($i -eq 7) { continue }
    Write-Host $I
}
```

Вышеуказанное будет выводить от 1 до 20 на консоль, но пропустите номер 7.

Примечание . При использовании контура конвейера вы должны использовать `return` вместо `Continue` .

Перерыв

Оператор `break` немедленно выйдет из цикла программы. Его можно использовать в циклах `For` , `ForEach` , `While` и `Do` или в `Switch` .

```
$i = 0
while ($i -lt 15) {
    $i++
    if ($i -eq 7) {break}
    Write-Host $i
}
```

Вышеуказанное будет считаться до 15, но остановится, как только будет достигнуто 7.

Примечание . При использовании контура конвейера `break` будет вести себя как `continue` . Чтобы смоделировать `break` в конвейере контура, вам нужно включить дополнительную логику, командлет и т. Д. Легче придерживаться конвейерных конвейеров, если вам нужно использовать `break` .

Перерыв этикетки

Break также может вызывать метку, помещенную перед созданием цикла:

```
$i = 0
:mainLoop While ($i -lt 15) {
    Write-Host $i -ForegroundColor 'Cyan'
    $j = 0
    While ($j -lt 15) {
        Write-Host $j -ForegroundColor 'Magenta'
        $k = $i*$j
        Write-Host $k -ForegroundColor 'Green'
        if ($k -gt 100) {
            break mainLoop
        }
        $j++
    }
    $i++
}
```

Примечание. Этот код будет увеличивать `$i` до `8` и `$j` до `13` что приведет к тому, что значение `$k` равно `104` . Поскольку `$k` превышает `100` , код будет выходить из обоих циклов.

Прочитайте Loops онлайн: https://riptutorial.com/ru/powershell/topic/1067/loops

глава 5: MongoDB

замечания

Самая сложная часть состоит в том, чтобы прикрепить **поддокумент** в документ, который еще не создан, если нам нужно, чтобы поддокумент находился в ожидаемом поиске, нам нужно будет перебирать цикл for в массив и использовать `$doc2.add("Key", "Value")` вместо использования массива `foreach` current с индексом. Это сделает субдокумент в двух строках, как вы можете видеть в `"Tags" = [MongoDB.Bson.BsonDocument] $doc2` .

Examples

MongoDB с драйвером C # 1.7 с использованием PowerShell

Мне нужно запросить все детали с виртуальной машины и обновить ее в MongoDB.

```
Which require the output look like this.
{
    "_id" : ObjectId("5800509f23888a12bccf2347"),
    "ResourceGrp" : "XYZZ-MachineGrp",
    "ProcessTime" : ISODate("2016-10-14T03:27:16.586Z"),
    "SubscriptionName" : "GSS",
    "OS" : "Windows",
    "HostName" : "VM1",
    "IPAddress" : "192.168.22.11",
    "Tags" : {
        "costCenter" : "803344",
        "BusinessUNIT" : "WinEng",
        "MachineRole" : "App",
        "OwnerEmail" : "zteffer@somewhere.com",
        "appSupporter" : "Steve",
        "environment" : "Prod",
        "implementationOwner" : "xyzr@somewhere.com",
        "appSoftware" : "WebServer",
        "Code" : "Gx",
        "WholeOwner" : "zzzgg@somewhere.com"
    },
    "SubscriptionID" : "",
    "Status" : "running fine",
    "ResourceGroupName" : "XYZZ-MachineGrp",
    "LocalTime" : "14-10-2016-11:27"
}
```

У меня есть 3 набора массивов в Powershell

```
        $MachinesList  # Array
        $ResourceList # Array
        $MachineTags  # Array

    pseudo code
```

```
        $mongoDriverPath = 'C:\Program Files (x86)\MongoDB\CSharpDriver 1.7';
        Add-Type -Path "$($mongoDriverPath)\MongoDB.Bson.dll";
        Add-Type -Path "$($mongoDriverPath)\MongoDB.Driver.dll";

$db = [MongoDB.Driver.MongoDatabase]::Create('mongodb://127.0.0.1:2701/RGrpMachines');
   [System.Collections.ArrayList]$TagList = $vm.tags
     $A1 = $Taglist.key
     $A2 = $Taglist.value
   foreach ($Machine in $MachinesList)
   {
       foreach($Resource in $ResourceList)
       {
                    $doc2 = $null
                   [MongoDB.Bson.BsonDocument] $doc2 = @{}; #Create a Document here
                    for($i = 0; $i -lt $TagList.count; $i++)
                           {
                                $A1Key = $A1[$i].ToString()
                                $A2Value = $A2[$i].toString()
                                $doc2.add("$A1Key", "$A2Value")
                           }

                     [MongoDB.Bson.BsonDocument] $doc = @{
                        "_id"= [MongoDB.Bson.ObjectId]::GenerateNewId();
                        "ProcessTime"= [MongoDB.Bson.BsonDateTime] $ProcessTime;
                        "LocalTime" = "$LocalTime";
                        "Tags" = [MongoDB.Bson.BsonDocument] $doc2;
                        "ResourceGrp" = "$RGName";
                        "HostName"= "$VMName";
                        "Status"= "$VMStatus";
                        "IPAddress"= "$IPAddress";
                        "ResourceGroupName"= "$RGName";
                        "SubscriptionName"= "$CurSubName";
                        "SubscriptionID"= "$subid";
                        "OS"= "$OSType";
                    }; #doc loop close

                    $collection.Insert($doc);
           }
   }
```

Прочитайте MongoDB онлайн: https://riptutorial.com/ru/powershell/topic/7438/mongodb

глава 6: PowerShell «Потоки»; Debug, Verbose, Warning, Error, Output и Information

замечания

https://technet.microsoft.com/en-us/library/hh849921.aspx

Examples

Write-Output

`Write-Output` генерирует выходной сигнал. Этот вывод может перейти к следующей команде после конвейера или консоли, чтобы он просто отображался.

Командлет отправляет объекты по основному конвейеру, также известному как «выходной поток» или «конвейер успеха». Чтобы отправить объекты ошибок в конвейер ошибок, используйте Write-Error.

```
# 1.) Output to the next Cmdlet in the pipeline
Write-Output 'My text' | Out-File -FilePath "$env:TEMP\Test.txt"

Write-Output 'Bob' | ForEach-Object {
    "My name is $_"
}

# 2.) Output to the console since Write-Output is the last command in the pipeline
Write-Output 'Hello world'

# 3.) 'Write-Output' CmdLet missing, but the output is still considered to be 'Write-Output'
'Hello world'
```

1. Командлет Write-Output отправляет указанный объект по конвейеру в следующую команду.
2. Если команда является последней командой в конвейере, объект отображается в консоли.
3. Интерпретатор PowerShell рассматривает это как неявный Write-Output.

Поскольку поведение по умолчанию `Write-Output` - отображать объекты в конце конвейера, обычно нет необходимости использовать командлет. Например, `Get-Process | Write-Output` эквивалентен `Get-Process` .

Настройки записи

Сообщения могут быть написаны с помощью;

```
Write-Verbose "Detailed Message"
Write-Information "Information Message"
Write-Debug "Debug Message"
Write-Progress "Progress Message"
Write-Warning "Warning Message"
```

Каждая из них имеет переменную предпочтения;

```
$VerbosePreference = "SilentlyContinue"
$InformationPreference = "SilentlyContinue"
$DebugPreference = "SilentlyContinue"
$ProgressPreference = "Continue"
$WarningPreference = "Continue"
```

Переменная предпочтения контролирует, как обрабатывается сообщение и последующее выполнение скрипта;

```
$InformationPreference = "SilentlyContinue"
Write-Information "This message will not be shown and execution continues"

$InformationPreference = "Continue"
Write-Information "This message is shown and execution continues"

$InformationPreference = "Inquire"
Write-Information "This message is shown and execution will optionally continue"

$InformationPreference = "Stop"
Write-Information "This message is shown and execution terminates"
```

Цвет сообщений можно контролировать для `Write-Error` , установив;

```
$host.PrivateData.ErrorBackgroundColor = "Black"
$host.PrivateData.ErrorForegroundColor = "Red"
```

Аналогичные настройки доступны для `Write-Verbose` , `Write-Debug` и `Write-Warning` .

Прочитайте PowerShell «Потоки»; Debug, Verbose, Warning, Error, Output и Information онлайн: https://riptutorial.com/ru/powershell/topic/3255/powershell--потоки---debug--verbose--warning--error--output-и-information

глава 7: PSScriptAnalyzer - анализатор сценариев PowerShell

Вступление

PSScriptAnalyzer, https://github.com/PowerShell/PSScriptAnalyzer , представляет собой статическую проверку кода для модулей и сценариев Windows PowerShell. PSScriptAnalyzer проверяет качество кода Windows PowerShell, запуская набор правил, основанных на лучших практиках PowerShell, определенных командой и сообществом PowerShell. Он генерирует DiagnosticResults (ошибки и предупреждения), чтобы информировать пользователей о потенциальных дефектах кода и предлагает возможные решения для улучшения.

```
PS> Install-Module -Name PSScriptAnalyzer
```

Синтаксис

1. `Get-ScriptAnalyzerRule [-CustomizedRulePath <string[]>] [-Name <string[]>] [-Severity <string[]>] [<CommonParameters>]`
2. `Invoke-ScriptAnalyzer [-Path] <string> [-CustomizedRulePath <string[]>] [-ExcludeRule <string[]>] [-IncludeRule<string[]>] [-Severity <string[]>] [-Recurse] [-SuppressedOnly] [<CommonParameters>]`

Examples

Анализ скриптов со встроенными наборами предустановок

ScriptAnalyzer поставляется с наборами встроенных предустановленных правил, которые могут использоваться для анализа скриптов. К ним относятся: `PSGallery` , `DSC` и `CodeFormatting` . Они могут выполняться следующим образом:

Правила галереи PowerShell

Для выполнения правил галереи PowerShell Gallery используйте следующую команду:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings PSGallery -Recurse
```

Правила DSC

Для выполнения правил DSC используйте следующую команду:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings DSC -Recurse
```

Правила форматирования кода

Для выполнения правил форматирования кода используйте следующую команду:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings CodeFormatting -Recurse
```

Анализ скриптов по каждому встроенному правилу

Для запуска анализатора сценариев в одном файле сценария выполните:

```
Invoke-ScriptAnalyzer -Path myscript.ps1
```

Это проанализирует ваш скрипт против каждого встроенного правила. Если ваш скрипт достаточно велик, что может привести к большому количеству предупреждений и / или ошибок.

Чтобы запустить анализатор сценариев по всему каталогу, укажите папку, содержащую файлы сценария, модуля и DSC, которые вы хотите проанализировать. Укажите параметр Recurse, если вы также хотите, чтобы подкаталоги искали файлы для анализа.

```
Invoke-ScriptAnalyzer -Path . -Recurse
```

Список всех встроенных правил

Чтобы увидеть все встроенные правила, выполните:

```
Get-ScriptAnalyzerRule
```

Прочитайте PSScriptAnalyzer - анализатор сценариев PowerShell онлайн: https://riptutorial.com/ru/powershell/topic/9619/psscriptanalyzer---анализатор-сценариев-powershell

глава 8: Splatting

Вступление

Splatting - это метод передачи нескольких параметров команде как единому блоку. Это делается путем сохранения параметров и их значений в виде пар ключ-значение в хэш-таблице и разбиения их на командлет с использованием оператора splatting `@` .

Splatting может сделать команду более читаемой и позволяет повторно использовать параметры в командных вызовах mulitple.

замечания

Примечание. Оператор выражения Array или `@()` имеют совсем другое поведение, чем оператор Splatting `@` .

Подробнее о about_Splatting @ TechNet

Examples

Параметры разбивки

Splatting выполняется путем замены знака `$` оператором splatting `@` при использовании переменной, содержащей HashTable параметров и значений в командном вызове.

```
$MyParameters = @{
    Name = "iexplore"
    FileVersionInfo = $true
}

Get-Process @MyParameters
```

Без splatting:

```
Get-Process -Name "iexplore" -FileVersionInfo
```

Вы можете комбинировать нормальные параметры с разбитыми параметрами, чтобы легко добавлять общие параметры к вашим вызовам.

```
$MyParameters = @{
    ComputerName = "StackOverflow-PC"
}

Get-Process -Name "iexplore" @MyParameters
```

```
Invoke-Command -ScriptBlock { "Something to excute remotely" } @MyParameters
```

Передача параметра Switch с использованием Splatting

Чтобы использовать Splatting для вызова `Get-Process` с `-FileVersionInfo` переключателя `-FileVersionInfo` похожее на это:

```
Get-Process -FileVersionInfo
```

Это вызов с использованием splatting:

```
$MyParameters = @{
    FileVersionInfo = $true
}

Get-Process @MyParameters
```

Примечание. Это полезно, потому что вы можете создать набор параметров по умолчанию и сделать вызов много раз, как это

```
$MyParameters = @{
    FileVersionInfo = $true
}

Get-Process @MyParameters -Name WmiPrvSE
Get-Process @MyParameters -Name explorer
```

Трубопроводы и сращивание

Объявление splat полезно для повторного использования наборов параметров несколько раз или с небольшими вариациями:

```
$splat = @{
   Class = "Win32_SystemEnclosure"
   Property = "Manufacturer"
   ErrorAction = "Stop"
}

Get-WmiObject -ComputerName $env:COMPUTERNAME @splat
Get-WmiObject -ComputerName "Computer2" @splat
Get-WmiObject -ComputerName "Computer3" @splat
```

Однако, если знак не отстутнут для повторного использования, вы можете не захотеть его объявить. Вместо этого можно использовать каналы:

```
@{
   ComputerName = $env:COMPUTERNAME
   Class = "Win32_SystemEnclosure"
   Property = "Manufacturer"
   ErrorAction = "Stop"
```

```
} | % { Get-WmiObject @_ }
```

Splatting от функции верхнего уровня до серии внутренней функции

Без splatting очень громоздко пытаться передавать значения вниз через стек вызовов. Но если вы комбинируете splatting с силой **@PSBoundParameters,** вы можете передать коллекцию параметров верхнего уровня вниз через слои.

```
Function Outer-Method
{
    Param
    (
        [string]
        $First,

        [string]
        $Second
    )

    Write-Host ($First) -NoNewline

    Inner-Method @PSBoundParameters
}

Function Inner-Method
{
    Param
    (
        [string]
        $Second
    )

    Write-Host (" {0}!" -f $Second)
}

$parameters = @{
    First = "Hello"
    Second = "World"
}


Outer-Method @parameters
```

Прочитайте Splatting онлайн: https://riptutorial.com/ru/powershell/topic/5647/splatting

глава 9: sql-запросы powershell

Вступление

Просматривая этот документ, вы можете узнать, как использовать SQL-запросы с помощью powershell

параметры

Вещь	Описание
$ ServerInstance	Здесь мы должны упомянуть экземпляр, в котором присутствует база данных
$ Database	Здесь мы должны упомянуть базу данных, в которой присутствует таблица
$ Query	Здесь мы должны выполнить запрос, который вы хотите выполнить в SQ
$ Имя пользователя и пароль	UserName и Пароль, которые имеют доступ в базе данных

замечания

Вы можете использовать функцию ниже, если в случае, если вы не можете импортировать модуль SQLPS

```
function Import-Xls
{

    [CmdletBinding(SupportsShouldProcess=$true)]

    Param(
        [parameter(
            mandatory=$true,
            position=1,
            ValueFromPipeline=$true,
            ValueFromPipelineByPropertyName=$true)]
        [String[]]
        $Path,

        [parameter(mandatory=$false)]
        $Worksheet = 1,

        [parameter(mandatory=$false)]
        [switch]
```

```
    $Force
)

Begin
{
    function GetTempFileName($extension)
    {
        $temp = [io.path]::GetTempFileName();
        $params = @{
            Path = $temp;
            Destination = $temp + $extension;
            Confirm = $false;
            Verbose = $VerbosePreference;
        }
        Move-Item @params;
        $temp += $extension;
        return $temp;
    }

    # since an extension like .xls can have multiple formats, this
    # will need to be changed
    #
    $xlFileFormats = @{
        # single worksheet formats
        '.csv'  = 6;        # 6, 22, 23, 24
        '.dbf'  = 11;       # 7, 8, 11
        '.dif'  = 9;        #
        '.prn'  = 36;       #
        '.slk'  = 2;        # 2, 10
        '.wk1'  = 31;       # 5, 30, 31
        '.wk3'  = 32;       # 15, 32
        '.wk4'  = 38;       #
        '.wks'  = 4;        #
        '.xlw'  = 35;       #

        # multiple worksheet formats
        '.xls'  = -4143;    # -4143, 1, 16, 18, 29, 33, 39, 43
        '.xlsb' = 50;       #
        '.xlsm' = 52;       #
        '.xlsx' = 51;       #
        '.xml'  = 46;       #
        '.ods'  = 60;       #
    }

    $xl = New-Object -ComObject Excel.Application;
    $xl.DisplayAlerts = $false;
    $xl.Visible = $false;
}

Process
{
    $Path | ForEach-Object {

        if ($Force -or $psCmdlet.ShouldProcess($_)) {

            $fileExist = Test-Path $_

            if (-not $fileExist) {
                Write-Error "Error: $_ does not exist" -Category ResourceUnavailable;

            } else {
```

```
                    # create temporary .csv file from excel file and import .csv
                    #
                    $_ = (Resolve-Path $_).toString();
                    $wb = $xl.Workbooks.Add($_);
                    if ($?) {
                        $csvTemp = GetTempFileName(".csv");
                        $ws = $wb.Worksheets.Item($Worksheet);
                        $ws.SaveAs($csvTemp, $xlFileFormats[".csv"]);
                        $wb.Close($false);
                        Remove-Variable -Name ('ws', 'wb') -Confirm:$false;
                        Import-Csv $csvTemp;
                        Remove-Item $csvTemp -Confirm:$false -Verbose:$VerbosePreference;
                    }
                }
            }
        }
    }

    End
    {
        $xl.Quit();
        Remove-Variable -name xl -Confirm:$false;
        [gc]::Collect();
    }
}
```

Examples

SQLExample

Для запроса всех данных из таблицы *MachineName* мы можем использовать команду, как *показано* ниже.

$ Query = "Выбрать * из MachineName"

$ Inst = "ServerInstance"

$ DbName = "DatabaseName

$ UID = "ИД пользователя"

$ Password = "Пароль"

```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $DBName -query $Query -Username $UID -Password
$Password
```

SQLQuery

Для запроса всех данных из таблицы *MachineName* мы можем использовать команду, как *показано* ниже.

$ Query = "Выбрать * из MachineName"

$ Inst = "ServerInstance"

$ DbName = "DatabaseName

$ UID = "ИД пользователя"

$ Password = "Пароль"

```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $DBName -query $Query -Username $UID -Password
$Password
```

Прочитайте sql-запросы powershell онлайн: https://riptutorial.com/ru/powershell/topic/8217/sql-запросы-powershell

глава 10: WMI и CIM

замечания

CIM против WMI

С PowerShell 3.0 есть два способа работы с классами управления в PowerShell, WMI и CIM. PowerShell 1.0 и 2.0 поддерживают только WMI-модуль, который теперь заменяется новым и улучшенным CIM-модулем. В более поздней версии PowerShell командлеты WMI будут удалены.

Сравнение CIM и WMI-модулей:

CIM-Командлет	WMI-Командлет	Что оно делает
Get-CimInstance	Get-WmiObject	Получает CIM / WMI-объекты для класса
Invoke-CimMethod	Invoke-WmiMethod	Вызывает метод класса CIM / WMI
Регистр-CimIndicationEvent	Регистр-WmiEvent	Регистрирует событие для класса CIM / WMI
Remove-CimInstance	Remove-WmiObject	Удалить объект CIM / WMI
Set-CimInstance	Set-WmiInstance	Обновления / Сохраняет объект CIM / WMI
Get-CimAssociatedInstance	N / A	Получить связанные экземпляры (связанный объект / классы)
Get-CimClass	Get-WmiObject -List	Список классов CIM / WMI
New-CimInstance	N / A	Создать новый CIM-объект
Get-CimSession	N / A	Списки CIM-сессий
New-CimSession	N / A	Создать новую CIM-сессию

CIM-Командлет	WMI-Командлет	Что оно делает
New-CimSessionOption	N / A	Создает объект с параметрами сеанса; протокол, кодирование, отключение шифрования и т. д. (для использования с `New-CimSession`)
Remove-CimSession	N / A	Удаляет / останавливает сеанс CIM

Дополнительные ресурсы

Должен ли я использовать CIM или WMI с Windows PowerShell? @ Эй, сценарист! Блог

Examples

Запрос объектов

CIM / WMI чаще всего используется для запроса информации или конфигурации на устройстве. Thof класс, который представляет конфигурацию, процесс, пользователя и т. Д. В PowerShell существует несколько способов доступа к этим классам и экземплярам, но наиболее распространенными способами являются использование `Get-WmiObject` `Get-CimInstance` (CIM) или `Get-WmiObject` (WMI).

Список всех объектов для CIM-класса

Вы можете перечислить все экземпляры класса.

3.0

CIM:

```
> Get-CimInstance -ClassName Win32_Process

ProcessId Name                         HandleCount WorkingSetSize VirtualSize
--------- ----                         ----------- -------------- -----------
0         System Idle Process          0           4096           65536
4         System                       1459        32768          3563520
480       Secure System                0           3731456        0
484       smss.exe                     52          372736         2199029891072
....
....
```

WMI:

```
Get-WmiObject -Class Win32_Process
```

Использование фильтра

Вы можете применить фильтр только для получения конкретных экземпляров класса CIM / WMI. Фильтры записываются с использованием `WQL` (по умолчанию) или CQL (add `-QueryDialect CQL`). `-Filter` использует `WHERE` часть полного WQL / CQL-запроса.

3.0

CIM:

```
Get-CimInstance -ClassName Win32_Process -Filter "Name = 'powershell.exe'"

ProcessId Name           HandleCount WorkingSetSize VirtualSize
--------- ----           ----------- -------------- -----------
4800      powershell.exe 676         88305664       2199697199104
```

WMI:

```
Get-WmiObject -Class Win32_Process -Filter "Name = 'powershell.exe'"

...
Caption                     : powershell.exe
CommandLine                 : "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
CreationClassName           : Win32_Process
CreationDate                : 20160913184324.393887+120
CSCreationClassName         : Win32_ComputerSystem
CSName                      : STACKOVERFLOW-PC
Description                 : powershell.exe
ExecutablePath              : C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
ExecutionState              :
Handle                      : 4800
HandleCount                 : 673
....
```

Использование WQL-запроса:

Вы также можете использовать WQL / CQL-запрос для запросов и фильтров.

3.0

CIM:

```
Get-CimInstance -Query "SELECT * FROM Win32_Process WHERE Name = 'powershell.exe'"

ProcessId Name           HandleCount WorkingSetSize VirtualSize
--------- ----           ----------- -------------- -----------
4800      powershell.exe 673         88387584       2199696674816
```

Запрос объектов в другом пространстве имен:

3.0

CIM:

```
> Get-CimInstance -Namespace "root/SecurityCenter2" -ClassName AntiVirusProduct


displayName               : Windows Defender
instanceGuid              : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
pathToSignedProductExe    : %ProgramFiles%\Windows Defender\MSASCui.exe
pathToSignedReportingExe  : %ProgramFiles%\Windows Defender\MsMpeng.exe
productState              : 397568
timestamp                 : Fri, 09 Sep 2016 21:26:41 GMT
PSComputerName            :
```

WMI:

```
> Get-WmiObject -Namespace "root\SecurityCenter2" -Class AntiVirusProduct

__GENUS                   : 2
__CLASS                   : AntiVirusProduct
__SUPERCLASS              :
__DYNASTY                 : AntiVirusProduct
__RELPATH                 : AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-
DA132C1ACF46}"
__PROPERTY_COUNT          : 6
__DERIVATION              : {}
__SERVER                  : STACKOVERFLOW-PC
__NAMESPACE               : ROOT\SecurityCenter2
__PATH                    : \\STACKOVERFLOW-
PC\ROOT\SecurityCenter2:AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-DA132C1ACF46}"
displayName               : Windows Defender
instanceGuid              : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
pathToSignedProductExe    : %ProgramFiles%\Windows Defender\MSASCui.exe
pathToSignedReportingExe  : %ProgramFiles%\Windows Defender\MsMpeng.exe
productState              : 397568
timestamp                 : Fri, 09 Sep 2016 21:26:41 GMT
PSComputerName            : STACKOVERFLOW-PC
```

Классы и пространства имен

В CIM и WMI имеется много классов, которые разделены на несколько пространств имен. Наиболее распространенным (и по умолчанию) пространством имен в Windows является `root/cimv2` . Чтобы найти righ-класс, полезно перечислить все или выполнить поиск.

Список доступных классов

Вы можете перечислить все доступные классы в пространстве имен по умолчанию (`root/cimv2`) на компьютере.

3.0

CIM:

```
Get-CimClass
```

WMI:

```
Get-WmiObject -List
```

Поиск класса

Вы можете искать определенные классы с помощью подстановочных знаков. Пример: поиск классов, содержащих слово `process` .

3.0

CIM:

```
> Get-CimClass -ClassName "*Process*"

   NameSpace: ROOT/CIMV2

CimClassName                        CimClassMethods      CimClassProperties

------------                        ---------------      ------------------

Win32_ProcessTrace                  {}                   {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
Win32_ProcessStartTrace             {}                   {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
Win32_ProcessStopTrace              {}                   {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
CIM_Process                         {}                   {Caption, Description, InstallDate,
Name...}
Win32_Process                       {Create, Terminat... {Caption, Description, InstallDate,
Name...}
CIM_Processor                       {SetPowerState, R... {Caption, Description, InstallDate,
Name...}
Win32_Processor                     {SetPowerState, R... {Caption, Description, InstallDate,
Name...}
...
```

WMI:

```
Get-WmiObject -List -Class "*Process*"
```

Список классов в другом пространстве имен

Корневое пространство имен просто называется `root` . Вы можете перечислить классы в другом пространстве имен, используя параметр `-NameSpace` .

3.0

CIM:

```
> Get-CimClass -Namespace "root/SecurityCenter2"

   NameSpace: ROOT/SecurityCenter2

CimClassName                        CimClassMethods      CimClassProperties

------------                        ---------------      ------------------
....
AntiSpywareProduct                  {}                   {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
AntiVirusProduct                    {}                   {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
FirewallProduct                     {}                   {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
```

WMI:

```
Get-WmiObject -Class "__Namespace" -Namespace "root"
```

Список доступных пространств имен

Чтобы найти доступные дочерние пространства имен `root` (или другого пространства имен), запросите объекты в классе `__NAMESPACE` для этого пространства имен.

3.0

CIM:

```
> Get-CimInstance -Namespace "root" -ClassName "__Namespace"

Name            PSComputerName
----            --------------
subscription
DEFAULT
CIMV2
msdtc
Cli
SECURITY
HyperVCluster
SecurityCenter2
RSOP
PEH
StandardCimv2
WMI
directory
Policy
```

```
virtualization
Interop
Hardware
ServiceModel
SecurityCenter
Microsoft
aspnet
Appv
```

WMI:

```
Get-WmiObject -List -Namespace "root"
```

Прочитайте WMI и CIM онлайн: https://riptutorial.com/ru/powershell/topic/6808/wmi-и-cim

глава 11: Автоматизация инфраструктуры

Вступление

Автоматизация служб управления инфраструктурой приводит к сокращению FTE, а также к кумулятивному повышению рентабельности инвестиций с использованием нескольких инструментов, оркестров, оркестровки Engine, сценариев и простого пользовательского интерфейса

Examples

Простой скрипт для тестирования интеграции с черным ящиком консольных приложений

Это простой пример того, как вы можете автоматизировать тесты для консольного приложения, которые взаимодействуют со стандартным вводом и стандартным выходом.

Проверенное приложение считывает и суммирует каждую новую строку и будет предоставлять результат после предоставления одной белой строки. Сценарий оболочки питания записывает «pass» при совпадении вывода.

```
$process = New-Object System.Diagnostics.Process
$process.StartInfo.FileName = ".\ConsoleApp1.exe"
$process.StartInfo.UseShellExecute = $false
$process.StartInfo.RedirectStandardOutput = $true
$process.StartInfo.RedirectStandardInput = $true
if ( $process.Start() ) {
    # input
    $process.StandardInput.WriteLine("1");
    $process.StandardInput.WriteLine("2");
    $process.StandardInput.WriteLine("3");
    $process.StandardInput.WriteLine();
    $process.StandardInput.WriteLine();
    # output check
    $output = $process.StandardOutput.ReadToEnd()
    if ( $output ) {
        if ( $output.Contains("sum 6") ) {
            Write "pass"
        }
        else {
            Write-Error $output
        }
    }
    $process.WaitForExit()
}
```

Прочитайте Автоматизация инфраструктуры онлайн:
https://riptutorial.com/ru/powershell/topic/10909/автоматизация-инфраструктуры

глава 12: Автоматические переменные

Вступление

Автоматические переменные создаются и поддерживаются Windows PowerShell. У одного есть возможность вызвать переменную как раз о любом имени в книге; Единственными исключениями для этого являются переменные, которые уже управляются PowerShell. Эти переменные, без сомнения, будут самыми повторяющимися объектами, которые вы используете в PowerShell рядом с функциями (например, **$?** - указывает состояние успеха / отказа последней операции)

Синтаксис

- `$$` - Содержит последний токен в последней строке, полученной сеансом.
- `$^` - Содержит первый токен в последней строке, полученной сеансом.
- `$?` - Содержит статус выполнения последней операции.
- `$_` - Содержит текущий объект в конвейере

Examples

$ PID

Содержит идентификатор процесса текущего хостинга.

```
PS C:\> $pid
26080
```

Булевы значения

`$true` и `$false` - две переменные, которые представляют логические TRUE и FALSE.

Обратите внимание, что вы должны указать знак доллара в качестве первого символа (который отличается от C #).

```
$boolExpr = "abc".Length -eq 3 # length of "abc" is 3, hence $boolExpr will be True
if($boolExpr -eq $true){
    "Length is 3"
}
# result will be "Length is 3"
$boolExpr -ne $true
#result will be False
```

Обратите внимание, что когда вы используете логическое значение true / false в вашем коде, вы записываете `$true` или `$false` , но когда Powershell возвращает логическое

значение, оно выглядит как `True` или `False`

$ нулевой

`$null` используется для представления отсутствующего или неопределенного значения. `$null` может использоваться в качестве пустого заполнителя для пустого значения в массивах:

```
PS C:\> $array = 1, "string", $null
PS C:\> $array.Count
3
```

Когда мы используем тот же массив, что и источник `ForEach-Object` , он обрабатывает все три элемента (включая $ null):

```
PS C:\> $array | ForEach-Object {"Hello"}
Hello
Hello
Hello
```

Быть осторожен! Это означает, что `ForEach-Object` **WILL** обрабатывает даже `$null` сам по себе:

```
PS C:\> $null | ForEach-Object {"Hello"} # THIS WILL DO ONE ITERATION !!!
Hello
```

Это очень неожиданный результат, если сравнить его с классическим циклом `foreach` :

```
PS C:\> foreach($i in $null) {"Hello"} # THIS WILL DO NO ITERATION
PS C:\>
```

$ OFS

Переменная, называемая разделителем выходных полей, содержит строковое значение, которое используется при преобразовании массива в строку. По умолчанию `$OFS = " "` (*пробел*), но его можно изменить:

```
PS C:\> $array = 1,2,3
PS C:\> "$array" # default OFS will be used
1 2 3
PS C:\> $OFS = ",." # we change OFS to comma and dot
PS C:\> "$array"
1,.2,.3
```

$ _ / $ PSItem

Содержит объект / элемент, который в настоящее время обрабатывается конвейером.

```
PS C:\> 1..5 | % { Write-Host "The current item is $_" }
The current item is 1
The current item is 2
The current item is 3
The current item is 4
The current item is 5
```

`$PSItem` и `$_` идентичны и могут использоваться взаимозаменяемо, но `$_` на сегодняшний день наиболее часто используется.

$?

Содержит статус последней операции. Когда нет ошибки, она установлена в значение `True` :

```
PS C:\> Write-Host "Hello"
Hello
PS C:\> $?
True
```

Если есть некоторая ошибка, для нее установлено значение `False` :

```
PS C:\> wrt-host
wrt-host : The term 'wrt-host' is not recognized as the name of a cmdlet, function, script
file, or operable program.
Check the spelling of the name, or if a path was included, verify that the path is correct and
try again.
At line:1 char:1
+ wrt-host
+ ~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (wrt-host:String) [], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException

PS C:\> $?
False
```

$ ошибка

Массив из последних объектов ошибки. Первый в массиве - самый последний:

```
PS C:\> throw "Error" # resulting output will be in red font
Error
At line:1 char:1
+ throw "Error"
+ ~~~~~~~~~~~~~
    + CategoryInfo          : OperationStopped: (Error:String) [], RuntimeException
    + FullyQualifiedErrorId : Error

PS C:\> $error[0] # resulting output will be normal string (not red    )
Error
At line:1 char:1
+ throw "Error"
+ ~~~~~~~~~~~~~
    + CategoryInfo          : OperationStopped: (Error:String) [], RuntimeException
```

```
    + FullyQualifiedErrorId : Error
```

Советы по использованию: при использовании переменной `$error` в командлете формата (например, в списке форматов) вам следует использовать переключатель `-Force` . В противном случае командлет формата будет выводить содержимое `$error` в указанном выше порядке.

Записи ошибок можно удалить, например, `$Error.Remove($Error[0])` .

Прочитайте Автоматические переменные онлайн:
https://riptutorial.com/ru/powershell/topic/5353/автоматические-переменные

глава 13: Автоматические переменные - часть 2

Вступление

Тема «Автоматические переменные» уже содержит 7 примеров, и мы не можем добавить больше. В этом разделе будут продолжены автоматические переменные.

Автоматические переменные - это переменные, которые хранят информацию о состоянии для PowerShell. Эти переменные создаются и поддерживаются Windows PowerShell.

замечания

Не уверен, что это лучший способ обработки документирования автоматических переменных, но это лучше, чем ничего. Прокомментируйте, если вы найдете лучший способ:

Examples

$ PSVersionTable

Содержит хеш-таблицу только для чтения (Constant, AllScope), которая отображает сведения о версии PowerShell, которая работает в текущем сеансе.

```
$PSVersionTable    #this call results in this:
Name                           Value
----                           -----
PSVersion                      5.0.10586.117
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.10586.117
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
```

Самый быстрый способ запустить версию PowerShell:

```
$PSVersionTable.PSVersion
# result :
Major  Minor  Build  Revision
-----  -----  -----  --------
5      0      10586  117
```

Прочитайте Автоматические переменные - часть 2 онлайн:
https://riptutorial.com/ru/powershell/topic/8639/автоматические-переменные---часть-2

глава 14: Анонимный IP (v4 и v6) в текстовом файле с Powershell

Вступление

Манипулирование регулярным выражением для IPv4 и IPv6 и замена поддельным IP-адресом в файле журнала

Examples

Анонимный IP-адрес в текстовом файле

```
# Read a text file and replace the IPv4 and IPv6 by fake IP Address


# Describe all variables
$SourceFile = "C:\sourcefile.txt"
$IPv4File = "C:\IPV4.txt"
$DestFile = "C:\ANONYM.txt"
$Regex_v4 = "(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})"
$Anonym_v4 = "XXX.XXX.XXX.XXX"
$Regex_v6 = "((([0-9A-Fa-f]{1,4}:){7}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){6}:[0-9A-Fa-
f]{1,4})|(([0-9A-Fa-f]{1,4}:){5}:([0-9A-Fa-f]{1,4}:)?[0-9A-Fa-f]{1,4})|(([0-9A-Fa-
f]{1,4}:){4}:([0-9A-Fa-f]{1,4}:){0,2}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){3}:([0-9A-Fa-
f]{1,4}:){0,3}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){2}:([0-9A-Fa-f]{1,4}:){0,4}[0-9A-Fa-
f]{1,4})|(([0-9A-Fa-f]{1,4}:){6}((b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b).){3}(b((25[0-
5])|(1d{2})|(2[0-4]d)|(d{1,2}))b))|(([0-9A-Fa-f]{1,4}:){0,5}:((b((25[0-5])|(1d{2})|(2[0-
4]d)|(d{1,2}))b).){3}(b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b))|(::([0-9A-Fa-
f]{1,4}:){0,5}((b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b).){3}(b((25[0-5])|(1d{2})|(2[0-
4]d)|(d{1,2}))b))|([0-9A-Fa-f]{1,4}::([0-9A-Fa-f]{1,4}:){0,5}[0-9A-Fa-f]{1,4})|(::([0-9A-Fa-
f]{1,4}:){0,6}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){1,7}:))"
$Anonym_v6 = "YYYY:YYYY:YYYY:YYYY:YYYY:YYYY:YYYY:YYYY"
$SuffixName = "-ANONYM."
$AnonymFile = ($Parts[0] + $SuffixName + $Parts[1])

# Replace matching IPv4 from sourcefile and creating a temp file IPV4.txt
Get-Content $SourceFile | Foreach-Object {$_ -replace $Regex_v4, $Anonym_v4} | Set-Content
$IPv4File

# Replace matching IPv6 from IPV4.txt and creating a temp file ANONYM.txt
Get-Content $IPv4File | Foreach-Object {$_ -replace $Regex_v6, $Anonym_v6} | Set-Content
$DestFile

# Delete temp IPV4.txt file
Remove-Item $IPv4File

# Rename ANONYM.txt in sourcefile-ANONYM.txt
$Parts = $SourceFile.Split(".")
If (Test-Path $AnonymFile)
{
    Remove-Item $AnonymFile
    Rename-Item $DestFile -NewName $AnonymFile
```

```
    }
    Else
    {
    Rename-Item $DestFile -NewName $AnonymFile
}
```

Прочитайте Анонимный IP (v4 и v6) в текстовом файле с Powershell онлайн: https://riptutorial.com/ru/powershell/topic/9171/анонимный-ip--v4-и-v6--в-текстовом-файле-с-powershell

глава 15: Архивный модуль

Вступление

Модуль Archive `Microsoft.PowerShell.Archive` предоставляет функции для хранения файлов в ZIP-архивах (`Compress-Archive`) и их извлечения (`Expand-Archive`). Этот модуль доступен в PowerShell 5.0 и выше.

В более ранних версиях PowerShell можно было использовать расширения сообщества или .NET System.IO.Compression.FileSystem .

Синтаксис

- **Expand-Archive / Compress-Archive**
- **-Дорожка**
  - путь файла (ов) для сжатия (Compress-Archive) или путь к архиву для извлечения формы файла (Expand-Archive)
  - есть несколько других параметров, связанных с Path, см. ниже.
- **-DestinationPath** (необязательно)
  - если вы не предоставите этот путь, архив будет создан в текущем рабочем каталоге (Compress-Archive) или содержимое архива будет извлечено в текущий рабочий каталог (Expand-Archive)

параметры

параметр	подробности
CompressionLevel	*(Только* для компрессорного *архива)* Установите уровень сжатия как для `Fastest` , `Optimal` и для `NoCompression`
подтвердить	Подсказки для подтверждения перед запуском
сила	Заставляет команду запускаться без подтверждения
LiteralPath	Путь, который используется буквально, *без подстановочных знаков* , использовать , чтобы указать несколько путей
Дорожка	Путь, который может содержать подстановочные знаки, использовать , чтобы указать несколько путей
Обновить	*(Только сжимать-архив)* Обновление существующего архива
Что, если	Имитировать команду

замечания

См. MSDN Microsoft.PowerShell.Archive (5.1) для дальнейшей справки

Examples

Сжатие-архив с шаблоном

```
Compress-Archive -Path C:\Documents\* -CompressionLevel Optimal -DestinationPath
C:\Archives\Documents.zip
```

Эта команда:

- Сжатие всех файлов в `C:\Documents`
- Использует `Optimal` сжатие
- Сохраните полученный архив в `C:\Archives\Documents.zip`
    - `-DestinationPath` добавит `.zip` если нет.
    - `-LiteralPath` можно использовать, если вам требуется называть его без `.zip` .

Обновление существующего ZIP с помощью Compress-Archive

```
Compress-Archive -Path C:\Documents\* -Update -DestinationPath C:\Archives\Documents.zip
```

- это добавит или заменит все файлы `Documents.zip` новыми из `C:\Documents`

Извлечение ZIP с расширением-архивом

```
Expand-Archive -Path C:\Archives\Documents.zip -DestinationPath C:\Documents
```

- это извлечет все файлы из `Documents.zip` в папку `C:\Documents`

Прочитайте Архивный модуль онлайн: https://riptutorial.com/ru/powershell/topic/9896/архивный-модуль

глава 16: Безопасность и криптография

Examples

Вычисление хэш-кодов строки через .Net Cryptography

Использование пространства имен .Net `System.Security.Cryptography.HashAlgorithm` для генерации хеш-кода сообщения с поддерживаемыми алгоритмами.

```
$example="Nobody expects the Spanish Inquisition."

#calculate
$hash=[System.Security.Cryptography.HashAlgorithm]::Create("sha256").ComputeHash(
[System.Text.Encoding]::UTF8.GetBytes($example))

#convert to hex
[System.BitConverter]::ToString($hash)

#2E-DF-DA-DA-56-52-5B-12-90-FF-16-FB-17-44-CF-B4-82-DD-29-14-FF-BC-B6-49-79-0C-0E-58-9E-46-2D-
3D
```

Часть `"sha256"` была использованным алгоритмом хеширования.

- можно удалить или изменить на нижний регистр

```
#convert to lower case hex without '-'
[System.BitConverter]::ToString($hash).Replace("-","").ToLower()

#2edfdada56525b1290ff16fb1744cfb482dd2914ffbcb649790c0e589e462d3d
```

Если формат base64 был предпочтительным, использование конвертера base64 для вывода

```
#convert to base64
[Convert]::ToBase64String($hash)

#Lt/a2lZSWxKQ/xb7F0TPtILdKRT/vLZJeQwOWJ5GLT0=
```

Прочитайте Безопасность и криптография онлайн:
https://riptutorial.com/ru/powershell/topic/5683/безопасность-и-криптография

глава 17: Введение в Pester

замечания

Pester - это тестовая среда для PowerShell, которая позволяет запускать тестовые примеры для вашего кода PowerShell. Его можно использовать для запуска ex. чтобы проверить, что ваши модули, скрипты и т. д. работают по назначению.

Что такое Pester и почему я должен ухаживать?

Examples

Начало работы с Pester

Чтобы начать работу с модульным тестированием кода PowerShell с помощью модуля Pester, вам необходимо ознакомиться с тремя ключевыми словами / командами:

- **Опишите** : Определяет группу тестов. Для всех тестовых файлов Pester требуется хотя бы один блок описаний.
- **Он** : Определяет индивидуальный тест. Вы можете иметь несколько блоков It внутри блока Describe.
- **Должен** : команда verify / test. Он используется для определения результата, который следует считать успешным.

Образец:

```
Import-Module Pester

#Sample function to run tests against
function Add-Numbers{
    param($a, $b)
    return [int]$a + [int]$b
}

#Group of tests
Describe "Validate Add-Numbers" {

        #Individual test cases
        It "Should add 2 + 2 to equal 4" {
            Add-Numbers 2 2 | Should Be 4
        }

        It "Should handle strings" {
            Add-Numbers "2" "2" | Should Be 4
        }

        It "Should return an integer"{
            Add-Numbers 2.3 2 | Should BeOfType Int32
        }
```

```
}
```

Выход:

```
Describing Validate Add-Numbers
 [+] Should add 2 + 2 to equal 4 33ms
 [+] Should handle strings 19ms
 [+] Should return an integer 23ms
```

Прочитайте Введение в Pester онлайн: https://riptutorial.com/ru/powershell/topic/5753/введение-в-pester

глава 18: Введение в Psake

Синтаксис

- Задача - основная функция для выполнения шага сценария сборки
- Зависит - свойство, определяющее, от чего зависит текущий шаг
- default - всегда должна быть задана по умолчанию задача, которая будет выполняться, если не задана начальная задача
- FormatTaskName - указывает, как каждый шаг отображается в окне результатов.

замечания

psake - это инструмент автоматизации сборки, написанный в PowerShell, и вдохновлен Rake (Ruby make) и Bake (Boo make). Он используется для создания сборок с использованием шаблона зависимостей. Документация доступна здесь

Examples

Основной план

```
Task Rebuild -Depends Clean, Build  {
   "Rebuild"
 }

Task Build {
   "Build"
 }

Task Clean {
   "Clean"
 }

Task default -Depends Build
```

Пример FormatTaskName

```
# Will display task as:
# -------- Rebuild --------
# -------- Build --------
FormatTaskName "-------- {0} --------"

# will display tasks in yellow colour:
# Running Rebuild
FormatTaskName {
    param($taskName)
    "Running $taskName" - foregroundcolor yellow
}
```

```
Task Rebuild -Depends Clean, Build  {
   "Rebuild"
 }

Task Build {
   "Build"
 }

Task Clean {
   "Clean"
 }

Task default -Depends Build
```

Запустить задачу условно

```
propreties {
    $isOk = $false
}

# By default the Build task won't run, unless there is a param $true
Task Build -precondition { return $isOk } {
   "Build"
 }

Task Clean {
   "Clean"
 }

Task default -Depends Build
```

ContinueOnError

```
Task Build -depends Clean {
   "Build"
 }

Task Clean -ContinueOnError {
   "Clean"
    throw "throw on purpose, but the task will continue to run"
 }

Task default -Depends Build
```

Прочитайте Введение в Psake онлайн: https://riptutorial.com/ru/powershell/topic/5019/введение-в-psake

глава 19: Вложение управляемого кода (C # | VB)

Вступление

В этом разделе кратко описывается, как C # или VB .NET Managed code может быть сценарием и использоваться в сценарии PowerShell. В этом разделе не рассматриваются все грани командлета Add-Type.

Дополнительные сведения о командлете Add-Type см. В документации MSDN (для версии 5.1): https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.utility/add-тип

параметры

параметр	подробности
-TypeDefinition <String_>	Принимает код как строку
-Language <String_>	Указывает язык управляемого кода. Принятые значения: CSharp, CSharpVersion3, CSharpVersion2, VisualBasic, JScript

замечания

Удаление добавленных типов

В более поздних версиях PowerShell Remove-TypeData была добавлена в библиотеки командлетов PowerShell, которые могут позволить удалить тип в сеансе. Дополнительные сведения об этом командлете см. Здесь: https://msdn.microsoft.com/en-us/powershell/reference/4.0/microsoft.powershell.utility/remove-typedata.

Синтаксис CSharp и .NET

Для тех, кто имеет опыт работы с .NET, само собой разумеется, что разные версии C # могут быть совершенно радикальными в своем уровне поддержки определенного синтаксиса.

Если использовать Powershell 1.0 и / или -Language CSharp, управляемый код будет использовать .NET 2.0, который не обладает рядом функций, которые разработчики C #

обычно используют без второй мысли в наши дни, такие как Generics, Linq и Lambda. Кроме того, это формальный полиморфизм, который обрабатывается с дефолтными параметрами в более поздних версиях C # /. NET.

Examples

Пример C

В этом примере показано, как встроить базовый C # в сценарий PowerShell, добавить его в рабочую область / сеанс и использовать код в синтаксисе PowerShell.

```
$code = "
using System;

namespace MyNameSpace
{
    public class Responder
    {
        public static void StaticRespond()
        {
            Console.WriteLine("Static Response");
        }

        public void Respond()
        {
            Console.WriteLine("Instance Respond");
        }
    }
}
"@

# Check the type has not been previously added within the session, otherwise an exception is
raised
if (-not ([System.Management.Automation.PSTypeName]'MyNameSpace.Responder').Type)
{
    Add-Type -TypeDefinition $code -Language CSharp;
}

[MyNameSpace.Responder]::StaticRespond();

$instance = New-Object MyNameSpace.Responder;
$instance.Respond();
```

Пример VB.NET

В этом примере показано, как встроить базовый C # в сценарий PowerShell, добавить его в рабочую область / сеанс и использовать код в синтаксисе PowerShell.

```
$code = @"
Imports System

Namespace MyNameSpace
    Public Class Responder
        Public Shared Sub StaticRespond()
```

```
            Console.WriteLine("Static Response")
        End Sub

        Public Sub Respond()
            Console.WriteLine("Instance Respond")
        End Sub
    End Class
End Namespace
"@

# Check the type has not been previously added within the session, otherwise an exception is
raised
if (-not ([System.Management.Automation.PSTypeName]'MyNameSpace.Responder').Type)
{
    Add-Type -TypeDefinition $code -Language VisualBasic;
}

[MyNameSpace.Responder]::StaticRespond();

$instance = New-Object MyNameSpace.Responder;
$instance.Respond();
```

Прочитайте Вложение управляемого кода (C # | VB) онлайн:
https://riptutorial.com/ru/powershell/topic/9823/вложение-управляемого-кода--c-sharp---vb-

глава 20: Возвратное поведение в PowerShell

Вступление

Его можно использовать для выхода из текущей области, которая может быть функцией, сценарием или сценарием. В PowerShell результат каждого оператора возвращается как результат, даже без явного ключевого слова Return или для указания того, что конец области был достигнут.

замечания

Вы можете больше узнать о семантике возврата на странице about_Return на TechNet или вызвать `get-help return` из приглашения PowerShell.

Известные вопросы и ответы с большим количеством примеров / объяснений:

- Возвращаемое значение функции в PowerShell
- PowerShell: функция не имеет правильного возвращаемого значения

---

about_return на MSDN объясняет это кратко:

> Ключевое слово Return выходит из функции, скрипта или скриптового блока. Его можно использовать для выхода из области в определенной точке, для возврата значения или для указания того, что конец области был достигнут.
>
> Пользователи, знакомые с такими языками, как C или C #, могут захотеть использовать ключевое слово Return, чтобы логика оставила явную область видимости.
>
> В Windows PowerShell результаты каждого оператора возвращаются как выходные данные, даже без оператора, содержащего ключевое слово Return. Языки, такие как C или C #, возвращают только значение или значения, заданные ключевым словом Return.

Examples

Ранний выход

```
function earlyexit {
```

```
    "Hello"
    return
    "World"
}
```

«Привет» будет помещен в выходной конвейер, «Мир» не будет

Попался! Возвращение в трубопровод

```
get-childitem | foreach-object { if ($_.IsReadOnly) { return } }
```

Командлеты трубопроводов (например: `ForEach-Object` , `Where-Object` и т. Д.) Работают при закрытии. Возврат здесь будет двигаться только к следующему элементу на конвейере, а не к завершению обработки. Вы можете использовать **break** вместо **возврата,** если хотите выйти из обработки.

```
get-childitem | foreach-object { if ($_.IsReadOnly) { break } }
```

Попался! Игнорирование нежелательного вывода

Вдохновленный

- PowerShell: функция не имеет правильного возвращаемого значения

```
function bar {
 [System.Collections.ArrayList]$MyVariable = @()
 $MyVariable.Add("a") | Out-Null
 $MyVariable.Add("b") | Out-Null
 $MyVariable
}
```

`Out-Null` необходим, потому что метод .NET `ArrayList.Add` возвращает количество элементов в коллекции после добавления. Если опустить, трубопровод будет содержать `1, 2, "a", "b"`

Существует несколько способов опустить нежелательный вывод:

```
function bar
{
    # New-Item cmdlet returns information about newly created file/folder
    New-Item "test1.txt" | out-null
    New-Item "test2.txt" > $null
    [void](New-Item "test3.txt")
    $tmp = New-Item "test4.txt"
}
```

Примечание. Чтобы узнать больше о том, почему предпочитаете `> $null` , см. [Тема еще не создана].

Возврат со значением

(перефразированный от about_return)

Следующие методы будут иметь те же значения на конвейере

```
function foo {
    $a = "Hello"
    return $a
}

function bar {
    $a = "Hello"
    $a
    return
}

function quux {
    $a = "Hello"
    $a
}
```

Как работать с функциями возвращается

Функция возвращает все, что не захвачено чем-то другим.
Если u использует ключевое слово **return** , каждый оператор после строки возврата не будет выполнен!

Как это:

```
Function Test-Function
{
    Param
    (
        [switch]$ExceptionalReturn
    )
    "Start"
    if($ExceptionalReturn){Return "Damn, it didn't work!"}
    New-ItemProperty -Path "HKCU:\" -Name "test" -Value "TestValue" -Type "String"
    Return "Yes, it worked!"
}
```

Тест-Function
Вернусь:

- Начните
- Недавно созданный раздел реестра (это потому, что есть некоторые операторы, которые создают вывод, которого вы не можете ожидать)
- Да, это сработало!

Test-Function -ExceptionalReturn Вернет:

- Начните
- Черт, это не сработало!

Если вы сделаете это так:

```
Function Test-Function
{
    Param
    (
        [switch]$ExceptionalReturn
    )
    . {
       "Start"
        if($ExceptionalReturn)
        {
            $Return = "Damn, it didn't work!"
            Return
        }
        New-ItemProperty -Path "HKCU:\" -Name "test" -Value "TestValue" -Type "String"
        $Return = "Yes, it worked!"
        Return
    } | Out-Null
    Return $Return
}
```

Тест-Function
Вернусь:

- Да, это сработало!

Test-Function -ExceptionalReturn Вернет:

- Черт, это не сработало!

С помощью этого трюка вы можете контролировать возвращенный вывод, даже если вы не знаете, что будет вызывать каждое утверждение.

Он работает так

```
.{<Statements>} | Out-Null
```

. делает следующий скриптовый блок включен в код
{} отмечает блок сценария
| Out-Null передает любой неожиданный вывод в Out-Null (так что он ушел!)
Поскольку включен скриптовый блок, он получает ту же область действия, что и остальная часть функции.
Таким образом, вы можете получить доступ к переменным, которые были сделаны внутри скриптового блока.

Прочитайте Возвратное поведение в PowerShell онлайн:
https://riptutorial.com/ru/powershell/topic/4781/возвратное-поведение-в-powershell

глава 21: Встроенные переменные

Вступление

PowerShell предлагает множество полезных «автоматических» (встроенных) переменных. Некоторые автоматические переменные заполняются только в особых обстоятельствах, а другие доступны по всему миру.

Examples

$ PSScriptRoot

```
Get-ChildItem -Path $PSScriptRoot
```

В этом примере извлекается список дочерних элементов (каталогов и файлов) из папки, в которой находится файл сценария.

Автоматическая переменная `$PSScriptRoot` равна `$null` если используется вне файла кода PowerShell. Если он используется *внутри* сценария PowerShell, он автоматически определяет полный путь файловой системы к каталогу, содержащему файл сценария.

В Windows PowerShell 2.0 эта переменная действительна только в модулях сценариев (. psm1). Начиная с Windows PowerShell 3.0, он действителен во всех сценариях.

$ Args

```
$Args
```

Содержит массив необъявленных параметров и / или значений параметров, которые передаются в блок функций, сценариев или сценариев. Когда вы создаете функцию, вы можете объявить параметры с помощью ключевого слова param или добавив список параметров, разделенных запятыми, в круглые скобки после имени функции.

В действии события переменная $ Args содержит объекты, которые представляют аргументы события обрабатываемого события. Эта переменная заполняется только в блоке Action команды регистрации событий. Значение этой переменной также можно найти в свойстве SourceArgs объекта PSEventArgs (System.Management.Automation.PSEventArgs), возвращаемого Get-Event.

$ PSItem

```
Get-Process | ForEach-Object -Process {
```

```
  $PSItem.Name
}
```

То же, что и `$_` . Содержит текущий объект в объекте конвейера. Вы можете использовать эту переменную в командах, которые выполняют действие для каждого объекта или для выбранных объектов в конвейере.

$?

```
Get-Process -Name doesnotexist
Write-Host -Object "Was the last operation successful? $?"
```

Содержит статус выполнения последней операции. Он содержит TRUE, если последняя операция выполнена успешно, и FALSE, если она не удалась.

$ ошибка

```
Get-Process -Name doesnotexist
Write-Host -Object ('The last error that occurred was: {0}' -f $error[0].Exception.Message)
```

Содержит массив объектов ошибок, которые представляют самые последние ошибки. Самая последняя ошибка - первый объект ошибки в массиве ($ Error [0]).

Чтобы предотвратить добавление ошибки в массив $ Error, используйте общий параметр ErrorAction со значением Ignore. Дополнительные сведения см. В разделе about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216) .

Прочитайте Встроенные переменные онлайн: https://riptutorial.com/ru/powershell/topic/8732/встроенные-переменные

глава 22: Выполнение исполняемых файлов

Examples

Консольные приложения

```
PS> console_app.exe
PS> & console_app.exe
PS> Start-Process console_app.exe
```

Приложения для графического интерфейса пользователя

```
PS> gui_app.exe (1)
PS> & gui_app.exe (2)
PS> & gui_app.exe | Out-Null (3)
PS> Start-Process gui_app.exe (4)
PS> Start-Process gui_app.exe -Wait (5)
```

Приложения GUI запускаются в другом процессе и немедленно возвращают управление хосту PowerShell. Иногда вам необходимо, чтобы приложение завершило обработку до следующего выполнения инструкции PowerShell. Это может быть достигнуто путем подключения вывода приложения к $ null (3) или с помощью Start-Process с помощью переключателя -Wait (5).

Консольные потоки

```
PS> $ErrorActionPreference = "Continue" (1)
PS> & console_app.exe *>&1 | % { $_ } (2)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.ErrorRecord] } (3)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.WarningRecord] } (4)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.VerboseRecord] } (5)
PS> & console_app.exe *>&1 (6)
PS> & console_app.exe 2>&1 (7)
```

Поток 2 содержит объекты System.Management.Automation.ErrorRecord. Обратите внимание, что некоторые приложения, такие как git.exe, используют «поток ошибок» для информационных целей, которые не обязательно являются ошибками. В этом случае лучше всего посмотреть код выхода, чтобы определить, следует ли интерпретировать поток ошибок как ошибки.

PowerShell понимает эти потоки: вывод, ошибка, предупреждение, многословность, отладка, прогресс. Нативные приложения обычно используют только эти потоки: вывод, ошибка, предупреждение.

В PowerShell 5 все потоки могут быть перенаправлены на стандартный поток вывода / успеха (6).

В более ранних версиях PowerShell только конкретные потоки могут быть перенаправлены на стандартный поток вывода / успеха (7). В этом примере «поток ошибок» будет перенаправлен на выходной поток.

Коды выхода

```
PS> $LastExitCode
PS> $?
PS> $Error[0]
```

Это встроенные переменные PowerShell, которые предоставляют дополнительную информацию о самой последней ошибке. `$LastExitCode` - это окончательный код выхода последнего встроенного приложения, которое было выполнено. `$?` и `$Error[0]` - последняя запись об ошибке, которая была создана PowerShell.

Прочитайте Выполнение исполняемых файлов онлайн:
https://riptutorial.com/ru/powershell/topic/7707/выполнение-исполняемых-файлов

глава 23: Графический интерфейс в Powershell

Examples

WPF GUI для командлета Get-Service

```
Add-Type -AssemblyName PresentationFramework

[xml]$XAMLWindow = '
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Height="Auto"
    SizeToContent="WidthAndHeight"
    Title="Get-Service">
    <ScrollViewer Padding="10,10,10,0" ScrollViewer.VerticalScrollBarVisibility="Disabled">
        <StackPanel>
            <StackPanel Orientation="Horizontal">
                <Label Margin="10,10,0,10">ComputerName:</Label>
                <TextBox Name="Input" Margin="10" Width="250px"></TextBox>
            </StackPanel>
            <DockPanel>
                <Button Name="ButtonGetService" Content="Get-Service" Margin="10"
Width="150px" IsEnabled="false"/>
                <Button Name="ButtonClose" Content="Close" HorizontalAlignment="Right"
Margin="10" Width="50px"/>
            </DockPanel>
        </StackPanel>
    </ScrollViewer >
</Window>
'

# Create the Window Object
$Reader=(New-Object System.Xml.XmlNodeReader $XAMLWindow)
$Window=[Windows.Markup.XamlReader]::Load( $Reader )

# TextChanged Event Handler for Input
$TextboxInput = $Window.FindName("Input")
$TextboxInput.add_TextChanged.Invoke({
    $ComputerName = $TextboxInput.Text
    $ButtonGetService.IsEnabled = $ComputerName -ne ''
})

# Click Event Handler for ButtonClose
$ButtonClose = $Window.FindName("ButtonClose")
$ButtonClose.add_Click.Invoke({
    $Window.Close();
})

# Click Event Handler for ButtonGetService
$ButtonGetService = $Window.FindName("ButtonGetService")
$ButtonGetService.add_Click.Invoke({
    $ComputerName = $TextboxInput.text.Trim()
    try{
```

```
        Get-Service -ComputerName $computerName | Out-GridView -Title "Get-Service on
$ComputerName"
    }catch{

[System.Windows.MessageBox]::Show($_.exception.message,"Error",[System.Windows.MessageBoxButton]::OK,[S

    }
})

# Open the Window
$Window.ShowDialog() | Out-Null
```

Это создает диалоговое окно, которое позволяет пользователю выбрать имя компьютера, затем отобразит таблицу служб и их статусы на этом компьютере.
В этом примере используется WPF, а не Windows Forms.

Прочитайте Графический интерфейс в Powershell онлайн:
https://riptutorial.com/ru/powershell/topic/7141/графический-интерфейс-в-powershell

глава 24: Динамические параметры PowerShell

Examples

«Простой» динамический параметр

Этот пример добавляет новый параметр в MyTestFunction, если `$SomeUsefulNumber` больше 5.

```
function MyTestFunction
{
    [CmdletBinding(DefaultParameterSetName='DefaultConfiguration')]
    Param
    (
        [Parameter(Mandatory=$true)][int]$SomeUsefulNumber
    )

    DynamicParam
    {
        $paramDictionary = New-Object -Type
System.Management.Automation.RuntimeDefinedParameterDictionary
        $attributes = New-Object System.Management.Automation.ParameterAttribute
        $attributes.ParameterSetName = "__AllParameterSets"
        $attributes.Mandatory = $true
        $attributeCollection = New-Object -Type
System.Collections.ObjectModel.Collection[System.Attribute]
        $attributeCollection.Add($attributes)
        # If "SomeUsefulNumber" is greater than 5, then add the "MandatoryParam1" parameter
        if($SomeUsefulNumber -gt 5)
        {
            # Create a mandatory string parameter called "MandatoryParam1"
            $dynParam1 = New-Object -Type
System.Management.Automation.RuntimeDefinedParameter("MandatoryParam1", [String],
$attributeCollection)
            # Add the new parameter to the dictionary
            $paramDictionary.Add("MandatoryParam1", $dynParam1)
        }
        return $paramDictionary
    }

    process
    {
        Write-Host "SomeUsefulNumber = $SomeUsefulNumber"
        # Notice that dynamic parameters need a specific syntax
        Write-Host ("MandatoryParam1 = {0}" -f $PSBoundParameters.MandatoryParam1)
    }

}
```

Использование:

```
PS >  MyTestFunction -SomeUsefulNumber 3
SomeUsefulNumber = 3
```

```
MandatoryParam1 =

PS >  MyTestFunction -SomeUsefulNumber 6
cmdlet MyTestFunction at command pipeline position 1
Supply values for the following parameters:
MandatoryParam1:

PS >MyTestFunction -SomeUsefulNumber 6 -MandatoryParam1 test
SomeUsefulNumber = 6
MandatoryParam1 = test
```

Во втором примере использования вы можете четко видеть, что параметр отсутствует.

Динамические параметры также учитываются при автозавершении.
Вот что произойдет, если вы нажмете Ctrl + пробел в конце строки:

```
PS >MyTestFunction -SomeUsefulNumber 3 -<ctrl+space>
Verbose              WarningAction        WarningVariable      OutBuffer
Debug                InformationAction    InformationVariable  PipelineVariable
ErrorAction          ErrorVariable        OutVariable

PS >MyTestFunction -SomeUsefulNumber 6 -<ctrl+space>
MandatoryParam1      ErrorAction          ErrorVariable        OutVariable
Verbose              WarningAction        WarningVariable      OutBuffer
Debug                InformationAction    InformationVariable  PipelineVariable
```

Прочитайте Динамические параметры PowerShell онлайн:
https://riptutorial.com/ru/powershell/topic/6704/динамические-параметры-powershell

глава 25: Использование ShouldProcess

Синтаксис

- $ PSCmdlet.ShouldProcess ("Target")
- $ PSCmdlet.ShouldProcess («Цель», «Действие»)

параметры

параметр	подробности
цель	Изменен ресурс.
действие	Выполняется операция. По умолчанию используется имя командлета.

замечания

`$PSCmdlet.ShouldProcess()` также автоматически записывает сообщение в подробный вывод.

```
PS> Invoke-MyCmdlet -Verbose
VERBOSE: Performing the operation "Invoke-MyCmdlet" on target "Target of action"
```

Examples

Добавление поддержки -WhatIf и -Confirm для вашего командлета

```
function Invoke-MyCmdlet {
    [CmdletBinding(SupportsShouldProcess = $true)]
    param()
    # ...
}
```

Использование метода ShouldProcess () с одним аргументом

```
if ($PSCmdlet.ShouldProcess("Target of action")) {
    # Do the thing
}
```

При использовании `-WhatIf` :

```
What if: Performing the action "Invoke-MyCmdlet" on target "Target of action"
```

При использовании `-Confirm` :

```
Are you sure you want to perform this action?
Performing operation "Invoke-MyCmdlet" on target "Target of action"
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"):
```

Пример полного использования

Другие примеры не могли объяснить мне, как вызвать условную логику.

Этот пример также показывает, что базовые команды также будут слушать флаг -Confirm!

```
<#
Restart-Win32Computer
#>

function Restart-Win32Computer
{
    [CmdletBinding(SupportsShouldProcess=$true,ConfirmImpact="High")]
    param (
    [parameter(Mandatory=$true,ValueFromPipeline=$true,ValueFromPipelineByPropertyName=$true)]
    [string[]]$computerName,
    [parameter(Mandatory=$true)]
    [string][ValidateSet("Restart","LogOff","Shutdown","PowerOff")] $action,
    [boolean]$force = $false
)
BEGIN {
# translate action to numeric value required by the method
switch($action) {
    "Restart"
    {
        $_action = 2
        break
    }
    "LogOff"
    {
        $_action = 0
        break
    }
    "Shutdown"
    {
        $_action = 2
        break
    }
    "PowerOff"
    {
        $_action = 8
        break
    }
}
# to force, add 4 to the value
if($force)
{
    $_action += 4
}
write-verbose "Action set to $action"
}
PROCESS {
    write-verbose "Attempting to connect to $computername"
    # this is how we support -whatif and -confirm
```

```
    # which are enabled by the SupportsShouldProcess
    # parameter in the cmdlet bindnig
    if($pscmdlet.ShouldProcess($computername)) {
        get-wmiobject win32_operatingsystem -computername $computername | invoke-wmimethod -
name Win32Shutdown -argumentlist $_action
    }
}
}
#Usage:
#This will only output a description of the actions that this command would execute if -WhatIf
is removed.
'localhost','server1'| Restart-Win32Computer -action LogOff -whatif

#This will request the permission of the caller to continue with this item.
#Attention: in this example you will get two confirmation request because all cmdlets called
by this cmdlet that also support ShouldProcess, will ask for their own confirmations...
'localhost','server1'| Restart-Win32Computer -action LogOff -Confirm
```

Прочитайте Использование ShouldProcess онлайн:
https://riptutorial.com/ru/powershell/topic/1145/использование-shouldprocess

глава 26: Использование индикатора выполнения

Вступление

Индикатор выполнения может использоваться, чтобы показать, что что-то находится в процессе. Это функция экономии времени и слайка, которую нужно иметь. Полосы прогресса невероятно полезны при отладке, чтобы выяснить, какая часть скрипта выполняется, и они удовлетворяют для людей, выполняющих скрипты, отслеживать, что происходит. Обычно для отображения какого-то прогресса, когда сценарий занимает много времени для завершения. Когда пользователь запускает скрипт, и ничего не происходит, возникает вопрос, правильно ли запущен скрипт.

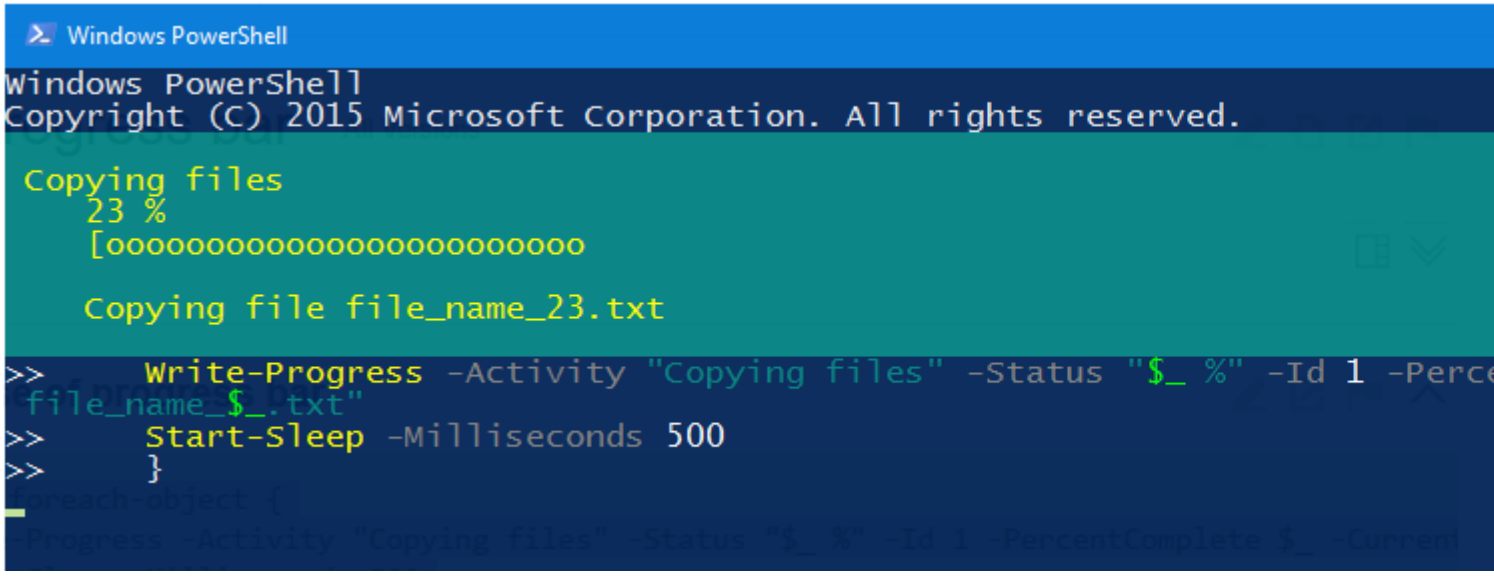
Examples

Простое использование индикатора выполнения

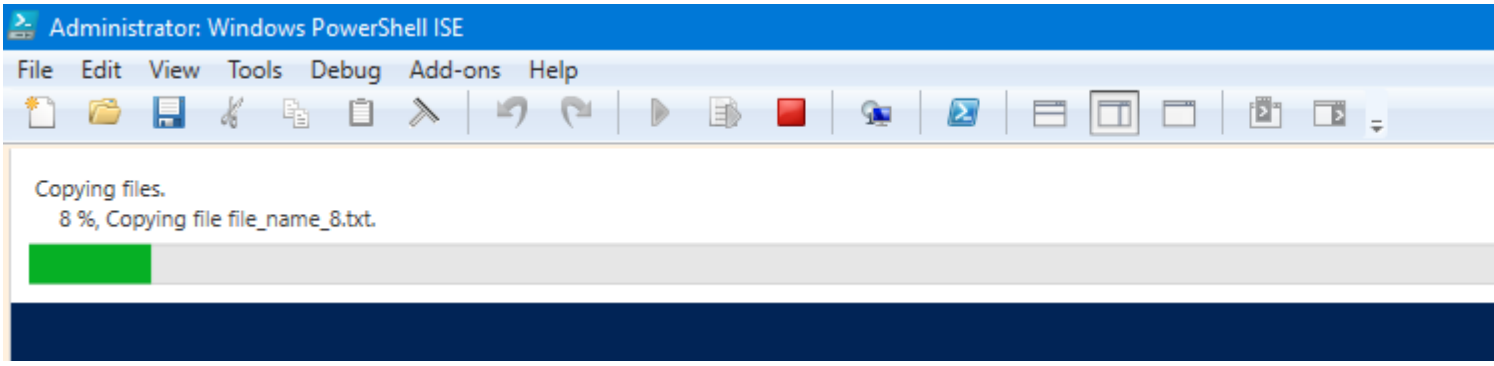
```
1..100 | ForEach-Object {
        Write-Progress -Activity "Copying files" -Status "$_ %" -Id 1 -PercentComplete $_ -
CurrentOperation "Copying file file_name_$_.txt"
        Start-Sleep -Milliseconds 500    # sleep simulates working code, replace this line
with your executive code (i.e. file copying)
    }
```

Обратите внимание, что для краткости этот пример не содержит исполнительного кода (моделируется с помощью `Start-Sleep`). Однако можно запускать его напрямую, как есть, а затем модифицировать и играть с ним.

Вот как выглядит результат в консоли PS:



Вот как выглядит результат в PS ISE:

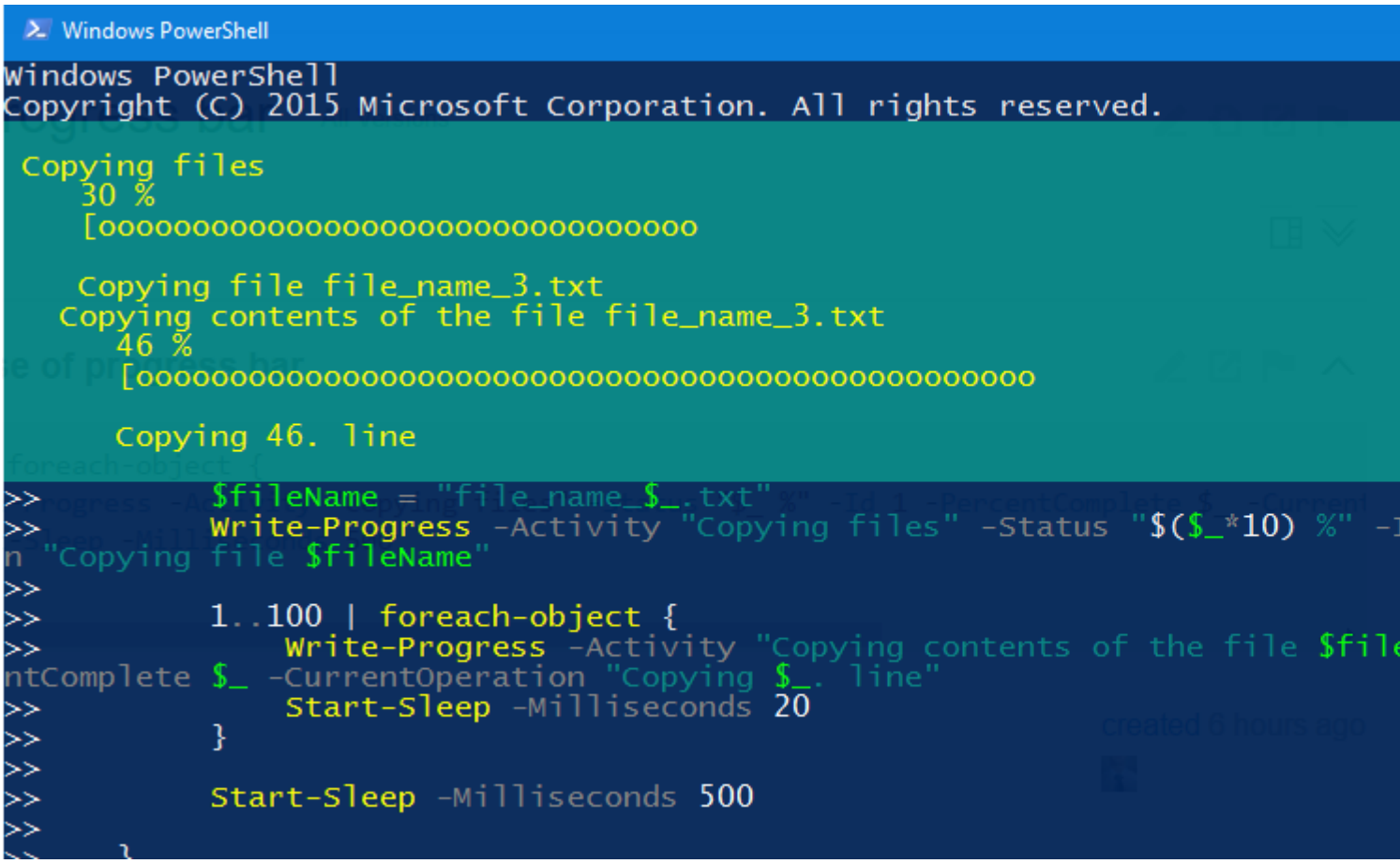


Использование внутреннего индикатора выполнения

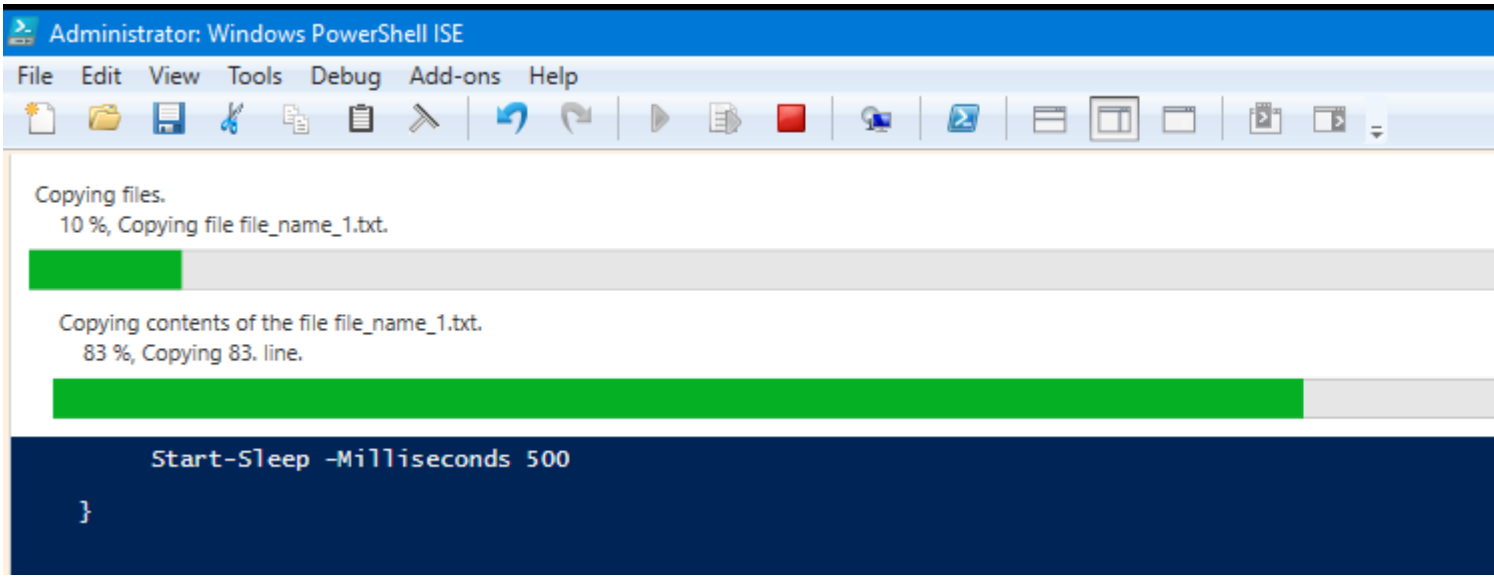
```
1..10 | foreach-object {
        $fileName = "file_name_$_.txt"
        Write-Progress -Activity "Copying files" -Status "$($_*10) %" -Id 1 -PercentComplete
($_*10) -CurrentOperation "Copying file $fileName"

        1..100 | foreach-object {
            Write-Progress -Activity "Copying contents of the file $fileName" -Status "$_ %" -
Id 2 -ParentId 1 -PercentComplete $_ -CurrentOperation "Copying $_. line"

            Start-Sleep -Milliseconds 20 # sleep simulates working code, replace this line
with your executive code (i.e. file copying)
        }

        Start-Sleep -Milliseconds 500 # sleep simulates working code, replace this line with
your executive code (i.e. file search)

   }
```

Обратите внимание, что для краткости этот пример не содержит исполнительного кода (моделируется с помощью `Start-Sleep`). Однако можно запускать его напрямую, как есть, а затем модифицировать и играть с ним.

Вот как выглядит результат в консоли PS:



Вот как выглядит результат в PS ISE:



Прочитайте Использование индикатора выполнения онлайн:
https://riptutorial.com/ru/powershell/topic/5020/использование-индикатора-выполнения

глава 27: Использование справочной системы

замечания

`Get-Help` - это командлет для чтения разделов справки в PowerShell.

Подробнее TechNet

Examples

Обновление справочной системы

3.0

Начиная с PowerShell 3.0, вы можете загрузить и обновить автономную справочную документацию с помощью одного командлета.

```
Update-Help
```

Обновление справки на нескольких компьютерах (или компьютерах, не подключенных к Интернету).

Выполните следующее на компьютере с файлами справки

```
Save-Help -DestinationPath \\Server01\Share\PSHelp -Credential $Cred
```

Для запуска на многих компьютерах удаленно

```
Invoke-Command -ComputerName (Get-Content Servers.txt) -ScriptBlock {Update-Help -SourcePath
\\Server01\Share\Help -Credential $cred}
```

Использование Get-Help

`Get-Help` можно использовать для просмотра справки в PowerShell. Вы можете искать командлеты, функции, провайдеры или другие темы.

Чтобы просмотреть справочную документацию о работе, используйте:

```
Get-Help about_Jobs
```

Вы можете искать темы с помощью подстановочных знаков. Если вы хотите просмотреть доступные разделы справки с заголовком, начинающимся с `about_` , попробуйте:

```
Get-Help about_*
```

Если вам нужна помощь по `Select-Object` , вы должны использовать:

```
Get-Help Select-Object
```

Вы также можете использовать `help` псевдонимов или `man` .

Просмотр онлайн-версии справки

Вы можете получить доступ к онлайновой справочной документации, используя:

```
Get-Help Get-Command -Online
```

Примеры просмотра

Показать примеры использования для конкретного командлета.

```
Get-Help Get-Command -Examples
```

Просмотр страницы полной справки

Просмотрите полную документацию по этой теме.

```
Get-Help Get-Command -Full
```

Просмотр справки по определенному параметру

Вы можете просмотреть справку по определенному параметру, используя:

```
Get-Help Get-Content -Parameter Path
```

Прочитайте Использование справочной системы онлайн:
https://riptutorial.com/ru/powershell/topic/5644/использование-справочной-системы

глава 28: Использование существующих статических классов

Вступление

Эти классы являются справочными библиотеками методов и свойств, которые не изменяют состояние, одним словом, неизменяемым. Вам не нужно создавать их, вы просто используете их. Классы и методы, такие как они называются статическими классами, потому что они не создаются, не разрушаются или не изменяются. Вы можете ссылаться на статический класс, окружая имя класса квадратными скобками.

Examples

Создание нового GUID мгновенно

Используйте существующие классы .NET немедленно с помощью PowerShell с помощью [class] :: Method (args):

```
PS C:\> [guid]::NewGuid()

Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

Аналогично, в PowerShell 5+ вы можете использовать командлет `New-Guid` :

```
PS C:\> New-Guid

Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

Чтобы получить GUID как `[String]` , ссылается на свойство `.Guid` :

```
[guid]::NewGuid().Guid
```

Использование .Net Math Class

Вы можете использовать класс .Net Math для выполнения вычислений ([System.Math])

Если вы хотите узнать, какие методы доступны, вы можете использовать:

```
[System.Math] | Get-Member -Static -MemberType Methods
```

Вот несколько примеров использования класса Math:

```
PS C:\> [System.Math]::Floor(9.42)
9
PS C:\> [System.Math]::Ceiling(9.42)
10
PS C:\> [System.Math]::Pow(4,3)
64
PS C:\> [System.Math]::Sqrt(49)
7
```

Добавление типов

По имени сборки добавьте библиотеку

```
Add-Type -AssemblyName "System.Math"
```

или по пути к файлу:

```
Add-Type -Path "D:\Libs\CustomMath.dll"
```

Использовать добавленный тип:

```
[CustomMath.NameSpace]::Method(param1, $variableParam, [int]castMeAsIntParam)
```

Прочитайте Использование существующих статических классов онлайн: https://riptutorial.com/ru/powershell/topic/1522/использование-существующих-статических-классов

глава 29: Как загрузить последний артефакт из Artifactory с использованием сценария Powershell (v2.0 или ниже)?

Вступление

Эта документация объясняет и предоставляет инструкции по загрузке новейшего артефакта из репозитория JFrog Artifactory с использованием сценария Powershell (версия 2.0 или ниже).

Examples

Скрипт Powershell для скачивания последнего artifcat

```
$username = 'user'
$password= 'password'
$DESTINATION = "D:\test\latest.tar.gz"
$client = New-Object System.Net.WebClient
$client.Credentials = new-object System.Net.NetworkCredential($username, $password)
$lastModifiedResponse =
$client.DownloadString('https://domain.org.com/artifactory/api/storage/FOLDER/repo/?lastModified')

[System.Reflection.Assembly]::LoadWithPartialName("System.Web.Extensions")
$serializer = New-Object System.Web.Script.Serialization.JavaScriptSerializer
$getLatestModifiedResponse = $serializer.DeserializeObject($lastModifiedResponse)
$downloaUriResponse = $getLatestModifiedResponse.uri
Write-Host $json.uri
$latestArtifcatUrlResponse=$client.DownloadString($downloaUriResponse)
[System.Reflection.Assembly]::LoadWithPartialName("System.Web.Extensions")
$serializer = New-Object System.Web.Script.Serialization.JavaScriptSerializer
$getLatestArtifact = $serializer.DeserializeObject($latestArtifcatUrlResponse)
Write-Host $getLatestArtifact.downloadUri
$SOURCE=$getLatestArtifact.downloadUri
$client.DownloadFile($SOURCE,$DESTINATION)
```

Прочитайте Как загрузить последний артефакт из Artifactory с использованием сценария Powershell (v2.0 или ниже)? онлайн: https://riptutorial.com/ru/powershell/topic/8883/как-загрузить-последний-артефакт-из-artifactory-с-использованием-сценария-powershell--v2-0-или-ниже--

глава 30: Классы PowerShell

Вступление

Класс - это расширяемый программно-кодовый шаблон для создания объектов, предоставляющий начальные значения для состояния (переменные-члены) и реализации поведения (функции-члены или методы). Класс является планом для объекта. Он используется в качестве модели для определения структуры объектов. Объект содержит данные, к которым мы обращаемся через свойства, и что мы можем работать над использованием методов. PowerShell 5.0 добавила возможность создавать свои собственные классы.

Examples

Методы и свойства

```
class Person {
    [string] $FirstName
    [string] $LastName
    [string] Greeting() {
        return "Greetings, {0} {1}!" -f $this.FirstName, $this.LastName
    }
}

$x = [Person]::new()
$x.FirstName = "Jane"
$x.LastName = "Doe"
$greeting = $x.Greeting() # "Greetings, Jane Doe!"
```

Список доступных конструкторов для класса

5.0

В PowerShell 5.0+ вы можете просмотреть доступные конструкторы, вызвав статический `new` метод без круглых скобок.

```
PS> [DateTime]::new

OverloadDefinitions
-------------------
datetime new(long ticks)
datetime new(long ticks, System.DateTimeKind kind)
datetime new(int year, int month, int day)
datetime new(int year, int month, int day, System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second)
datetime new(int year, int month, int day, int hour, int minute, int second,
System.DateTimeKind kind)
datetime new(int year, int month, int day, int hour, int minute, int second,
```

```
System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.DateTimeKind kind)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.Globalization.Calendar calendar, System.DateTimeKind kind)
```

Это тот же метод, который вы можете использовать для перечисления определений перегрузки для любого метода

```
> 'abc'.CompareTo

OverloadDefinitions
-------------------
int CompareTo(System.Object value)
int CompareTo(string strB)
int IComparable.CompareTo(System.Object obj)
int IComparable[string].CompareTo(string other)
```

Для более ранних версий вы можете создать свою собственную функцию для отображения доступных конструкторов:

```
function Get-Constructor {
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline=$true)]
        [type]$type
    )

    Process {
        $type.GetConstructors() |
        Format-Table -Wrap @{
            n="$($type.Name) Constructors"
            e={ ($_.GetParameters() | % { $_.ToString() }) -Join ", " }
        }
    }
}
```

Использование:

```
Get-Constructor System.DateTime
#Or [datetime] | Get-Constructor

DateTime Constructors
---------------------
Int64 ticks
Int64 ticks, System.DateTimeKind kind
Int32 year, Int32 month, Int32 day
Int32 year, Int32 month, Int32 day, System.Globalization.Calendar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second,
System.DateTimeKind kind
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second,
System.Globalization.Calendar calendar
```

```
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.DateTimeKind kind
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.Globalization.Cal
endar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.Globalization.Cal
endar calendar, System.DateTimeKind kind
```

Перегрузка конструктора

```
class Person {
    [string] $Name
    [int] $Age

    Person([string] $Name) {
        $this.Name = $Name
    }

    Person([string] $Name, [int]$Age) {
        $this.Name = $Name
        $this.Age = $Age
    }
}
```

Получить всех участников экземпляра

```
PS > Get-Member -InputObject $anObjectInstance
```

Это вернет всех членов экземпляра типа. Вот часть примера вывода для экземпляра String

```
   TypeName: System.String

Name             MemberType            Definition
----             ----------            ----------
Clone            Method                System.Object Clone(), System.Object ICloneable.Clone()
CompareTo        Method                int CompareTo(System.Object value), int
CompareTo(string strB), i...
Contains         Method                bool Contains(string value)
CopyTo           Method                void CopyTo(int sourceIndex, char[] destination, int
destinationI...
EndsWith         Method                bool EndsWith(string value), bool EndsWith(string
value, System.S...
Equals           Method                bool Equals(System.Object obj), bool Equals(string
value), bool E...
GetEnumerator    Method                System.CharEnumerator GetEnumerator(),
System.Collections.Generic...
GetHashCode      Method                int GetHashCode()
GetType          Method                type GetType()
...
```

Шаблон базового класса

```
# Define a class
class TypeName
{
   # Property with validate set
   [ValidateSet("val1", "Val2")]
   [string] $P1

   # Static property
   static [hashtable] $P2

   # Hidden property does not show as result of Get-Member
   hidden [int] $P3

   # Constructor
   TypeName ([string] $s)
   {
       $this.P1 = $s
   }

   # Static method
   static [void] MemberMethod1([hashtable] $h)
   {
       [TypeName]::P2 = $h
   }

   # Instance method
   [int] MemberMethod2([int] $i)
   {
       $this.P3 = $i
       return $this.P3
   }
}
```

Наследование от родительского класса к классу детей

```
class ParentClass
{
    [string] $Message = "Its under the Parent Class"

    [string] GetMessage()
    {
        return ("Message: {0}" -f $this.Message)
    }
}

# Bar extends Foo and inherits its members
class ChildClass : ParentClass
{

}
$Inherit = [ChildClass]::new()
```

SO, **$ Inherit.Message** даст вам

> «Его под родительским классом»

Прочитайте Классы PowerShell онлайн: https://riptutorial.com/ru/powershell/topic/1146/классы-

powershell

глава 31: Кодирование / декодирование URL

замечания

Регулярное выражение, используемое в примерах примеров *декодирования,* было взято из RFC 2396, Приложение B: Анализ ссылки на URI с регулярным выражением ; для потомков, вот цитата:

> Следующая строка представляет собой регулярное выражение для разбиения ссылки на URI на его компоненты.
>
> ```
> ^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?
>  12            3  4          5       6  7        8 9
> ```
>
> Цифры во второй строке выше предназначены только для удобства чтения; они указывают контрольные точки для каждого подвыражения (т. е. каждая спаренная скобка). Мы ссылаемся на значение, сопоставляемое для подвыражения как $. Например, сопоставление указанного выражения с
>
> ```
> http://www.ics.uci.edu/pub/ietf/uri/#Related
> ```
>
> приводит к следующим подвыражениям:
>
> ```
> $1 = http:
> $2 = http
> $3 = //www.ics.uci.edu
> $4 = www.ics.uci.edu
> $5 = /pub/ietf/uri/
> $6 = <undefined>
> $7 = <undefined>
> $8 = #Related
> $9 = Related
> ```

Examples

Быстрый старт: кодирование

```
$url1 = [uri]::EscapeDataString("http://test.com?test=my value")
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value

$url2 = [uri]::EscapeUriString("http://test.com?test=my value")
# url2: http://test.com?test=my%20value

# HttpUtility requires at least .NET 1.1 to be installed.
$url3 = [System.Web.HttpUtility]::UrlEncode("http://test.com?test=my value")
```

```
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
```

Примечание. Дополнительная информация о HTTPUtility .

Быстрый старт: декодирование

Примечание. В этих примерах используются переменные, созданные в разделе « *Быстрый старт: кодирование* » выше.

```
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value
[uri]::UnescapeDataString($url1)
# Returns: http://test.com?test=my value

# url2: http://test.com?test=my%20value
[uri]::UnescapeDataString($url2)
# Returns: http://test.com?test=my value

# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
[uri]::UnescapeDataString($url3)
# Returns: http://test.com?test=my+value

# Note: There is no `[uri]::UnescapeUriString()`;
#       which makes sense since the `[uri]::UnescapeDataString()`
#       function handles everything it would handle plus more.

# HttpUtility requires at least .NET 1.1 to be installed.
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value
[System.Web.HttpUtility]::UrlDecode($url1)
# Returns: http://test.com?test=my value

# HttpUtility requires at least .NET 1.1 to be installed.
# url2: http://test.com?test=my%20value
[System.Web.HttpUtility]::UrlDecode($url2)
# Returns: http://test.com?test=my value

# HttpUtility requires at least .NET 1.1 to be installed.
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
[System.Web.HttpUtility]::UrlDecode($url3)
# Returns: http://test.com?test=my value
```

Примечание. Дополнительная информация о HTTPUtility .

Кодировать строку запроса с помощью `[uri] :: EscapeDataString ()`

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qs_data = @{
    'foo1'='bar1';
    'foo2'= 'complex;/?:@&=+$, bar''"';
    'complex;/?:@&=+$, foo''"'='bar2';
}

[System.Collections.ArrayList] $qs_array = @()
foreach ($qs in $qs_data.GetEnumerator()) {
    $qs_key = [uri]::EscapeDataString($qs.Name)
```

```
    $qs_value = [uri]::EscapeDataString($qs.Value)
    $qs_array.Add("${qs_key}=${qs_value}") | Out-Null
}

$url = $url_format -f @([uri]::"UriScheme${scheme}", ($qs_array -join '&'))
```

С помощью `[uri]::EscapeDataString()` вы заметите, что апостроф (') не был закодирован:

> https://example.vertigion.com/foos ? foo2 = сложный% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20bar '% 22 & complex% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20foo '% 22 = bar2 & foo1 = BAR1

Кодировать строку запроса с помощью `[System.Web.HttpUtility] :: UrlEncode ()`

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qs_data = @{
    'foo1'='bar1';
    'foo2'= 'complex;/?:@&=+$, bar''"';
    'complex;/?:@&=+$, foo''"'='bar2';
}

[System.Collections.ArrayList] $qs_array = @()
foreach ($qs in $qs_data.GetEnumerator()) {
    $qs_key = [System.Web.HttpUtility]::UrlEncode($qs.Name)
    $qs_value = [System.Web.HttpUtility]::UrlEncode($qs.Value)
    $qs_array.Add("${qs_key}=${qs_value}") | Out-Null
}

$url = $url_format -f @([uri]::"UriScheme${scheme}", ($qs_array -join '&'))
```

С помощью `[System.Web.HttpUtility]::UrlEncode()` вы заметите, что пробелы превращаются в знаки плюса (`+`) вместо `%20` :

> https://example.vertigion.com/foos ? foo2 = комплекс% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + bar% 27% 22 и комплекс% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + Foo% 27% 22 = bar2 & foo1 = BAR1

Декодировать URL с помощью `[uri] :: UnescapeDataString ()`

Закодировано с помощью `[uri]::EscapeDataString()`

Сначала мы расшифруем URL и строку запроса, закодированную с помощью `[uri]::EscapeDataString()` в приведенном выше примере:

> https://example.vertigion.com/foos ? foo2 = сложный% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20bar '% 22 & complex% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20foo '% 22 = bar2 & foo1 = BAR1

```
$url =
```

```
'https://example.vertigion.com/foos?foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar''%22&complex%3B%2F

$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [uri]::UnescapeDataString($qs_key),
            [uri]::UnescapeDataString($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

Это возвращает вас `[hashtable]$url_parts` ; который равен (**Примечание:** *пробелы* в сложных частях являются *пробелами*):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=ba

QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1
```

Закодировано с помощью `[System.Web.HttpUtility]::UrlEncode()`

Теперь мы расшифруем URL и строку запроса, закодированную с помощью `[System.Web.HttpUtility]::UrlEncode()` в приведенном выше примере:

> https://example.vertigion.com/foos ? foo2 = комплекс% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + bar% 27% 22 и комплекс% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + Foo% 27% 22 = bar2 & foo1 = BAR1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%

$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [uri]::UnescapeDataString($qs_key),
            [uri]::UnescapeDataString($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

Это дает вам `[hashtable]$url_parts` , что равно (**Примечание:** *пробелы* в сложных частях являются *плюсовыми знаками* (`+`) в первой части и *пробелами* во второй части):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=ba

QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1
```

Декодировать URL с помощью `[System.Web.HttpUtility] :: UrlDecode ()`

Закодировано с помощью `[uri]::EscapeDataString()`

Сначала мы расшифруем URL и строку запроса, закодированную с помощью `[uri]::EscapeDataString()` в приведенном выше примере:

https://example.vertigion.com/foos ? foo2 = сложный% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20bar '% 22 & complex% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20foo '% 22 = bar2 & foo1 = BAR1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar''%22&complex%3B%2

$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [System.Web.HttpUtility]::UrlDecode($qs_key),
            [System.Web.HttpUtility]::UrlDecode($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

Это возвращает вас `[hashtable]$url_parts` ; который равен (**Примечание:** *пробелы* в сложных частях являются *пробелами*):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=ba

QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1
```

Закодировано с помощью `[System.Web.HttpUtility]::UrlEncode()`

Теперь мы расшифруем URL и строку запроса, закодированную с помощью

`[System.Web.HttpUtility]::UrlEncode()` в приведенном выше примере:

> https://example.vertigion.com/foos ? foo2 = комплекс% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + bar% 27% 22 и комплекс% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + Foo% 27% 22 = bar2 & foo1 = BAR1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%

$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [System.Web.HttpUtility]::UrlDecode($qs_key),
            [System.Web.HttpUtility]::UrlDecode($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

Это возвращает вас `[hashtable]$url_parts` ; который равен (**Примечание:** *пробелы* в сложных частях являются *пробелами*):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=ba

QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1
```

Прочитайте Кодирование / декодирование URL онлайн:

https://riptutorial.com/ru/powershell/topic/7352/кодирование---декодирование-url

глава 32: Командная строка PowerShell.exe

параметры

параметр	Описание
-Помощь | -? | /?	Показывает помощь
-File <FilePath> [<Args>]	Путь к скрипт-файлу, который должен быть выполнен, и аргументы (необязательно)
-Command {- | <script-block> [-args <arg-array>] | <string> [<CommandParameters>]}	Команды, которые должны выполняться, а затем аргументы
-EncodedCommand <Base64EncodedCommand>	Закодированные команды Base64
-ExecutionPolicy <ExecutionPolicy>	Устанавливает политику выполнения только для этого процесса
-InputFormat {Текст | XML}	Устанавливает формат ввода данных, отправленных для обработки. Текст (строки) или XML (сериализованный CLIXML)
-Mta	PowerShell 3.0+: запускает PowerShell в многопоточной квартире (STA по умолчанию)
-sta	PowerShell 2.0: запускает PowerShell в однопоточной квартире (по умолчанию MTA)
-Нет выхода	Удерживает консоль PowerShell после выполнения скрипта / команды
-nologo	Скрывает авторский знак на старте
-NonInteractive	Скрывает консоль от пользователя
-NoProfile	Избегайте загрузки профилей PowerShell для машины или пользователя
-OutputFormat {Текст | XML}	Устанавливает формат вывода для данных,

параметр	Описание
	возвращаемых с PowerShell. Текст (строки) или XML (сериализованный CLIXML)
-PSConsoleFile <FilePath>	Загружает предварительно созданный файл консоли, который настраивает среду (созданную с помощью `Export-Console`)
-Version <версия Windows PowerShell>	Укажите версию PowerShell для запуска. В основном используется с `2.0`
-WindowStyle <style>	Указывает, следует ли запускать процесс PowerShell как `normal` , `hidden` , `minimized` или `maximized` окно.

Examples

Выполнение команды

Параметр `-Command` используется для указания команд, которые будут выполняться при запуске. Он поддерживает несколько входов данных.

-Command <string>

Вы можете указать команды для запуска в качестве строки. Несколько точек с запятой `;` - могут быть выполнены отдельные отчеты.

```
>PowerShell.exe -Command "(Get-Date).ToShortDateString()"
10.09.2016

>PowerShell.exe -Command "(Get-Date).ToShortDateString(); 'PowerShell is fun!'"
10.09.2016
PowerShell is fun!
```

-Command {scriptblock}

Параметр `-Command` также поддерживает ввод скриптового блока (одно или несколько операторов, заключенных в фигурные скобки `{ #code }` . Это работает только при вызове `PowerShell.exe` из другого сеанса Windows PowerShell.

```
PS > powershell.exe -Command {
"This can be useful, sometimes..."
(Get-Date).ToShortDateString()
```

```
}
This can be useful, sometimes...
10.09.2016
```

-Command - (стандартный ввод)

Вы можете передавать команды со стандартного ввода с помощью `-Command -` . Стандартный ввод может поступать от `echo` , чтения файла, устаревшего консольного приложения и т. Д.

```
>echo "Hello World";"Greetings from PowerShell" | PowerShell.exe -NoProfile -Command -
Hello World
Greetings from PowerShell
```

Выполнение файла сценария

Вы можете указать файл для `ps1` скрипта для запуска его содержимого при запуске с использованием параметра `-File` .

Основной скрипт

MyScript.ps1

```
(Get-Date).ToShortDateString()
"Hello World"
```

Выход:

```
>PowerShell.exe -File Desktop\MyScript.ps1
10.09.2016
Hello World
```

Использование параметров и аргументов

Вы можете добавлять параметры и / или аргументы после пути к файлу, чтобы использовать их в скрипте. Аргументы будут использоваться как значения для неопределенных / доступных параметров сценария, остальные будут доступны в `$args` - array

MyScript.ps1

```
param($Name)
```

```
"Hello $Name! Today's date it $((Get-Date).ToShortDateString())"
"First arg: $($args[0])"
```

Выход:

```
>PowerShell.exe -File Desktop\MyScript.ps1 -Name StackOverflow foo
Hello StackOverflow! Today's date it 10.09.2016
First arg: foo
```

Прочитайте Командная строка PowerShell.exe онлайн:
https://riptutorial.com/ru/powershell/topic/5839/командная-строка-powershell-exe

глава 33: Конфигурация желаемого состояния

Examples

Простой пример - включение WindowsFeature

```
configuration EnableIISFeature
{
    node localhost
    {
        WindowsFeature IIS
        {
            Ensure = “Present”
            Name = “Web-Server”
        }
    }
}
```

Если вы запустите эту конфигурацию в Powershell (EnableIISFeature), она создаст файл localhost.mof. Это «скомпилированная» конфигурация, которую вы можете запустить на машине.

Чтобы проверить конфигурацию DSC на вашем локальном хосте, вы можете просто вызвать следующее:

```
Start-DscConfiguration -ComputerName localhost -Wait
```

Запуск DSC (mof) на удаленной машине

Запуск DSC на удаленной машине почти так же просто. Предполагая, что вы уже настроили удаленную систему Powershell (или включили WSMAN).

```
$remoteComputer = "myserver.somedomain.com"
$cred = (Get-Credential)
Start-DSCConfiguration -ServerName $remoteComputer -Credential $cred -Verbose
```

Nb: Предполагая, что вы собрали конфигурацию для своего узла на своем локальном компьютере (и что файл myserver.somedomain.com.mof присутствует до начала настройки)

Импорт psd1 (файл данных) в локальную переменную

Иногда бывает полезно проверить файлы данных Powershell и выполнить итерацию через узлы и серверы.

Powershell 5 (WMF5) добавил эту аккуратную небольшую функцию для этого, так называемый Import-PowerShellDataFile.

Пример:

```
$data = Import-PowerShellDataFile -path .\MydataFile.psd1
$data.AllNodes
```

Список доступных ресурсов DSC

Чтобы просмотреть доступные ресурсы DSC на вашем авторизованном узле:

```
 Get-DscResource
```

Это отобразит все ресурсы для всех установленных модулей (находящихся в вашем PSModulePath) на вашем авторизованном узле.

Чтобы просмотреть все доступные ресурсы DSC, которые можно найти в онлайн-источниках (PSGallery ++) на WMF 5:

```
Find-DSCResource
```

Импорт ресурсов для использования в DSC

Прежде чем вы сможете использовать ресурс в конфигурации, вы должны явно импортировать его. Просто установив его на свой компьютер, вы не сможете использовать этот ресурс неявно.

Импортируйте ресурс с помощью Import-DscResource.

Пример, показывающий, как импортировать ресурс PSDesiredStateConfiguration и ресурс файла.

```
Configuration InstallPreReqs
{
   param(); # params to DSC goes here.

   Import-DscResource PSDesiredStateConfiguration

   File CheckForTmpFolder {
        Type = 'Directory'
        DestinationPath = 'C:\Tmp'
        Ensure = "Present"
    }
 }
```

Примечание . Чтобы ресурсы DSC работали, вы должны установить модули на целевых компьютерах при запуске конфигурации. Если вы их не установили, конфигурация

завершится неудачно.

Прочитайте Конфигурация желаемого состояния онлайн:
https://riptutorial.com/ru/powershell/topic/5662/конфигурация-желаемого-состояния

глава 34: Модули Powershell

Вступление

Начиная с версии PowerShell версии 2.0, разработчики могут создавать модули PowerShell. Модули PowerShell инкапсулируют набор общих функций. Например, существуют модули PowerShell, зависящие от поставщика, которые управляют различными облачными службами. Существуют также общие модули PowerShell, которые взаимодействуют со службами социальных сетей и выполняют общие задачи программирования, такие как кодирование Base64, работа с именами каналов и т. Д.

Модули могут выдавать псевдонимы команд, функции, переменные, классы и т. Д.

Examples

Создать манифест модуля

```
@{
  RootModule = 'MyCoolModule.psm1'
  ModuleVersion = '1.0'
  CompatiblePSEditions = @('Core')
  GUID = '6b42c995-67da-4139-be79-597a328056cc'
  Author = 'Bob Schmob'
  CompanyName = 'My Company'
  Copyright = '(c) 2017 Administrator. All rights reserved.'
  Description = 'It does cool stuff.'
  FunctionsToExport = @()
  CmdletsToExport = @()
  VariablesToExport = @()
  AliasesToExport = @()
  DscResourcesToExport = @()
}
```

Каждый хороший модуль PowerShell имеет манифест модуля. Модуль манифеста просто содержит метаданные о модуле PowerShell и не определяет фактическое содержимое модуля.

Файл манифеста - это файл сценария PowerShell с `.psd1` файла `.psd1` , содержащим HashTable. HashTable в манифесте должен содержать определенные ключи, чтобы PowerShell корректно интерпретировал его как файл модуля PowerShell.

В приведенном выше примере представлен список основных ключей HashTable, которые составляют манифест модуля, но есть много других. Команда `New-ModuleManifest` помогает создать новый скелет манифеста модуля.

Пример простого модуля

```
function Add {
  [CmdletBinding()]
  param (
    [int] $x
  , [int] $y
  )

  return $x + $y
}

Export-ModuleMember -Function Add
```

Это простой пример того, как может выглядеть файл модуля сценария PowerShell. Этот файл будет называться `MyCoolModule.psm1` и ссылается на файл манифеста модуля (.psd1). Вы заметите, что команда `Export-ModuleMember` позволяет нам указать, какие функции в модуле мы хотим «экспортировать» или выставить пользователю модуля. Некоторые функции будут доступны только для внутреннего использования и не должны отображаться, поэтому они будут исключены из вызова `Export-ModuleMember` .

Экспорт переменной из модуля

```
$FirstName = 'Bob'
Export-ModuleMember -Variable FirstName
```

Чтобы экспортировать переменную из модуля, вы используете команду `Export-ModuleMember` с параметром `-Variable` . Помните, однако, что если переменная также явно не экспортируется в файл манифеста модуля (.psd1), то переменная не будет видна потребителю модуля. Подумайте о манифесте модуля как о «привратнике». Если функция или переменная не разрешена в манифесте модуля, она не будет видна потребителю модуля.

Примечание. Экспорт переменной аналогичен созданию поля в открытом классе. Это нецелесообразно. Было бы лучше разоблачить функцию, чтобы получить поле и функцию для установки поля.

Структурирование модулей PowerShell

Вместо того, чтобы определять все ваши функции в одном `.psm1` сценария PowerShell `.psm1` PowerShell, вы можете разбить свою функцию на отдельные файлы. Затем вы можете передать эти файлы из вашего файла модуля скрипта, что, по сути, относится к ним так, как если бы они были частью самого файла `.psm1` .

Рассмотрим эту структуру каталога модуля:

```
\MyCoolModule
  \Functions
    Function1.ps1
    Function2.ps1
```

```
    Function3.ps1
MyCoolModule.psd1
MyCoolModule.psm1
```

Внутри файла `MyCoolModule.psm1` вы можете вставить следующий код:

```
Get-ChildItem -Path $PSScriptRoot\Functions |
  ForEach-Object -Process { . $PSItem.FullName }
```

Это будет точечный источник отдельных файлов функций в `.psm1` модуля `.psm1` .

Расположение модулей

PowerShell ищет модули в каталогах, перечисленных в пути $ Env: PSModule.

Модуль, называемый *foo* , в папке с именем *foo* будет найден с помощью `Import-Module foo`

В этой папке PowerShell будет искать манифест модуля (foo.psd1), файл модуля (foo.psm1), DLL (foo.dll).

Видимость члена модуля

По умолчанию только функции, определенные в модуле, видны вне модуля. Другими словами, если вы определяете переменные и псевдонимы в модуле, они не будут доступны, кроме кода модуля.

Чтобы переопределить это поведение, вы можете использовать командлет `Export-ModuleMember` . Он имеет параметры, называемые `-Function` , `-Variable` и `-Alias` которые позволяют вам точно указать, какие элементы экспортируются.

Важно отметить, что если вы используете `Export-ModuleMember` , будут видны **только** указанные вами элементы.

Прочитайте Модули Powershell онлайн: https://riptutorial.com/ru/powershell/topic/8734/модули-powershell

глава 35: Модули, скрипты и функции

Вступление

Модули PowerShell расширяют возможности системного администратора, администратора баз данных и разработчика. Является ли это просто методом совместного использования функций и скриптов.

Функции Powershell должны избегать повторных кодов. См. [Функции PS] [1] [1]: Функции PowerShell

Сценарии PowerShell используются для автоматизации административных задач, которые состоят из командной строки оболочки и связанных командлетов, построенных поверх . NET Framework.

Examples

функция

Функция представляет собой именованный блок кода, который используется для определения многоразового кода, который должен быть простым в использовании. Он обычно включается внутри скрипта, чтобы помочь повторно использовать код (чтобы избежать дублирования кода) или распространяться как часть модуля, чтобы сделать его полезным для других в нескольких сценариях.

Сценарии, в которых функция может быть полезной:

- Вычислить среднее значение для группы чисел
- Создание отчета для запуска процессов
- Напишите функцию, которая проверяет компьютер, является «здоровой», пинговая компьютер и доступ к `c$` -share

Функции создаются с использованием ключевого слова `function` , за которым следует однословное имя и блок сценария, содержащий код для выполнения при вызове имени функции.

```
function NameOfFunction {
    Your code
}
```

демонстрация

```
function HelloWorld {
```

```
    Write-Host "Greetings from PowerShell!"
}
```

Использование:

```
> HelloWorld
Greetings from PowerShell!
```

скрипт

Сценарий представляет собой текстовый файл с расширением файла `.ps1` который содержит команды PowerShell, которые будут выполняться при вызове сценария. Поскольку скрипты являются сохраненными файлами, их легко переносить между компьютерами.

Сценарии часто записываются для решения конкретной проблемы, например:

- Запуск еженедельной задачи обслуживания
- Чтобы установить и настроить решение / приложение на компьютере

демонстрация

MyFirstScript.ps1:

```
Write-Host "Hello World!"
2+2
```

Вы можете запустить скрипт, введя путь к файлу, используя:

- Абсолютный путь, например. `c:\MyFirstScript.ps1`
- Относительный путь, например `.\MyFirstScript.ps1` если текущий каталог вашей консоли PowerShell был `C:\`

Использование:

```
> .\MyFirstScript.ps1
Hello World!
4
```

Сценарий также может импортировать модули, определять его собственные функции и т. Д.

MySecondScript.ps1:

```
function HelloWorld {
    Write-Host "Greetings from PowerShell!"
}
```

```
HelloWorld
Write-Host "Let's get started!"
2+2
HelloWorld
```

Использование:

```
> .\MySecondScript.ps1
Greetings from PowerShell!
Let's get started!
4
Greetings from PowerShell!
```

модуль

Модуль представляет собой набор связанных функций многократного использования (или командлетов), которые могут быть легко распространены среди других пользователей PowerShell и использованы в нескольких сценариях или непосредственно в консоли. Модуль обычно сохраняется в его собственном каталоге и состоит из:

- Один или несколько файлов кода с `.psm1` файла `.psm1` содержащим функции или бинарные сборки (`.dll`), содержащие командлеты
- Модуль manifest `.psd1` описывающий имя, версию, автор, описание, функции / командлеты, которые он предоставляет и т. Д.
- Другие требования к его работе, в т.ч. зависимостей, скриптов и т. д.

Примеры модулей:

- Модуль, содержащий функции / командлеты, которые выполняют статистику по набору данных
- Модуль для запросов и настройки баз данных

Чтобы PowerShell легко находил и импортировал модуль, он часто помещается в один из известных модулей модулей PowerShell, определенных в `$env:PSModulePath` .

демонстрация

Список модулей, которые установлены в одном из известных модулей:

```
Get-Module -ListAvailable
```

Импортируйте модуль, например. Модуль `Hyper-V` :

```
Import-Module Hyper-V
```

Список доступных команд в модуле, например. `Microsoft.PowerShell.Archive` -модуль

```
> Import-Module Microsoft.PowerShell.Archive
> Get-Command -Module Microsoft.PowerShell.Archive

CommandType Name             Version Source
----------- ----             ------- ------
Function    Compress-Archive 1.0.1.0 Microsoft.PowerShell.Archive
Function    Expand-Archive   1.0.1.0 Microsoft.PowerShell.Archive
```

Расширенные функции

Расширенные функции ведут себя так же, как командлеты. PowerShell ISE включает в себя два скелета расширенных функций. Получите доступ к ним через меню, отредактируйте фрагменты кода или Ctrl + J. (Начиная с версии PS 3.0, более поздние версии могут отличаться)

Ключевые вещи, которые включают в себя расширенные функции,

- встроенная, настраиваемая справка для функции, доступная через `Get-Help`
- может использовать [CmdletBinding ()], что заставляет функцию действовать как командлет
- расширенные параметры параметров

Простая версия:

```
<#
.Synopsis
   Short description
.DESCRIPTION
   Long description
.EXAMPLE
   Example of how to use this cmdlet
.EXAMPLE
   Another example of how to use this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding()]
    [OutputType([int])]
    Param
    (
        # Param1 help description
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        $Param1,

        # Param2 help description
        [int]
        $Param2
    )

    Begin
    {
    }
    Process
```

```
    {
    }
    End
    {
    }
}
```

Полная версия:

```
<#
.Synopsis
   Short description
.DESCRIPTION
   Long description
.EXAMPLE
   Example of how to use this cmdlet
.EXAMPLE
   Another example of how to use this cmdlet
.INPUTS
   Inputs to this cmdlet (if any)
.OUTPUTS
   Output from this cmdlet (if any)
.NOTES
   General notes
.COMPONENT
   The component this cmdlet belongs to
.ROLE
   The role this cmdlet belongs to
.FUNCTIONALITY
   The functionality that best describes this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding(DefaultParameterSetName='Parameter Set 1',
                   SupportsShouldProcess=$true,
                   PositionalBinding=$false,
                   HelpUri = 'http://www.microsoft.com/',
                   ConfirmImpact='Medium')]
    [OutputType([String])]
    Param
    (
        # Param1 help description
        [Parameter(Mandatory=$true,
                    ValueFromPipeline=$true,
                    ValueFromPipelineByPropertyName=$true,
                    ValueFromRemainingArguments=$false,
                    Position=0,
                    ParameterSetName='Parameter Set 1')]
        [ValidateNotNull()]
        [ValidateNotNullOrEmpty()]
        [ValidateCount(0,5)]
        [ValidateSet("sun", "moon", "earth")]
        [Alias("p1")]
        $Param1,

        # Param2 help description
        [Parameter(ParameterSetName='Parameter Set 1')]
        [AllowNull()]
        [AllowEmptyCollection()]
        [AllowEmptyString()]
```

```
        [ValidateScript({$true})]
        [ValidateRange(0,5)]
        [int]
        $Param2,

        # Param3 help description
        [Parameter(ParameterSetName='Another Parameter Set')]
        [ValidatePattern("[a-z]*")]
        [ValidateLength(0,15)]
        [String]
        $Param3
    )

    Begin
    {
    }
    Process
    {
        if ($pscmdlet.ShouldProcess("Target", "Operation"))
        {
        }
    }
    End
    {
    }
}
```

Прочитайте Модули, скрипты и функции онлайн:
https://riptutorial.com/ru/powershell/topic/5755/модули--скрипты-и-функции

глава 36: Модуль ActiveDirectory

Вступление

В этом разделе вы познакомитесь с некоторыми из основных командлетов, используемых в модуле Active Directory для PowerShell, для управления пользователями, группами, компьютерами и объектами.

замечания

Помните, что система помощи PowerShell является одним из лучших ресурсов, которые вы можете использовать.

```
Get-Help Get-ADUser -Full
Get-Help Get-ADGroup -Full
Get-Help Get-ADComputer -Full
Get-Help Get-ADObject -Full
```

Вся справочная документация предоставит примеры, синтаксис и помощь по параметрам.

Examples

модуль

```
#Add the ActiveDirectory Module to current PowerShell Session
Import-Module ActiveDirectory
```

пользователей

Получить пользователя Active Directory

```
Get-ADUser -Identity JohnSmith
```

Получить все свойства, связанные с пользователем

```
Get-ADUser -Identity JohnSmith -Properties *
```

Получить выбранные свойства для пользователя

```
Get-ADUser -Identity JohnSmith -Properties * | Select-Object -Property sAMAccountName, Name,
Mail
```

Новый пользователь AD

```
New-ADUser -Name "MarySmith" -GivenName "Mary" -Surname "Smith" -DisplayName "MarySmith" -Path
"CN=Users,DC=Domain,DC=Local"
```

группы

Получить группу Active Directory

```
Get-ADGroup -Identity "My-First-Group" #Ensure if group name has space quotes are used
```

Получить все свойства, связанные с группой

```
Get-ADGroup -Identity "My-First-Group" -Properties *
```

Получить всех членов группы

```
Get-ADGroupMember -Identity "My-First-Group" | Select-Object -Property sAMAccountName
Get-ADgroup "MY-First-Group" -Properties Members | Select -ExpandProperty Members
```

Добавить пользователя AD в группу AD

```
Add-ADGroupMember -Identity "My-First-Group" -Members "JohnSmith"
```

Новая группа AD

```
New-ADGroup -GroupScope Universal -Name "My-Second-Group"
```

компьютеры

Восстановить компьютер AD

```
Get-ADComputer -Identity "JohnLaptop"
```

Получить все свойства, связанные с компьютером

```
Get-ADComputer -Identity "JohnLaptop" -Properties *
```

Получить выбор свойств компьютера

```
Get-ADComputer -Identity "JohnLaptop" -Properties * | Select-Object -Property Name, Enabled
```

Объекты

Получить объект Active Directory

```
#Identity can be ObjectGUID, Distinguished Name or many more
```

```
Get-ADObject -Identity "ObjectGUID07898"
```

Переместить объект Active Directory

```
Move-ADObject -Identity "CN=JohnSmith,OU=Users,DC=Domain,DC=Local" -TargetPath
"OU=SuperUser,DC=Domain,DC=Local"
```

Изменение объекта Active Directory

```
Set-ADObject -Identity "CN=My-First-Group,OU=Groups,DC=Domain,DC=local" -Description "This is
My First Object Modification"
```

Прочитайте Модуль ActiveDirectory онлайн: https://riptutorial.com/ru/powershell/topic/8213/модуль-activedirectory

глава 37: Модуль ISE

Вступление

Интегрированная среда сценариев Windows PowerShell (ISE) - это хост-приложение, которое позволяет писать, запускать и тестировать скрипты и модули в графической и интуитивно понятной среде. Основные возможности Windows PowerShell ISE включают синтаксическую раскраску, завершение табуляции, Intellisense, визуальную отладку, соответствие Unicode и контекстно-зависимую справку и обеспечивают богатый опыт работы с сценариями.

Examples

Тестовые скрипты

Простым, но мощным использованием ISE является, например, написание кода в верхней части (с интуитивной синтаксической раскраской) и запуск кода, просто его маркировка и нажатие клавиши F8.

```
function Get-Sum
{
    foreach ($i in $Input)
    {$Sum += $i}
    $Sum

1..10 | Get-Sum

#output
55
```

Прочитайте Модуль ISE онлайн: https://riptutorial.com/ru/powershell/topic/10954/модуль-ise

глава 38: Модуль SharePoint

Examples

Загрузка оснастки SharePoint

Загрузка Snapin SharePoint может быть выполнена с помощью следующего:

```
Add-PSSnapin "Microsoft.SharePoint.PowerShell"
```

Это работает только в 64-битной версии PowerShell. Если в окне указано «Windows PowerShell (x86)» в заголовке, вы используете неправильную версию.

Если Snap-In уже загружен, приведенный выше код вызовет ошибку. Использование следующего будет загружаться только при необходимости, которое может использоваться в Командлетах / функциях:

```
if ((Get-PSSnapin "Microsoft.SharePoint.PowerShell" -ErrorAction SilentlyContinue) -eq $null)
{
    Add-PSSnapin "Microsoft.SharePoint.PowerShell"
}
```

Кроме того, если вы запустите оболочку управления SharePoint, она автоматически включит оснастку.

Чтобы получить список всех доступных командлетов SharePoint, выполните следующие действия:

```
Get-Command -Module Microsoft.SharePoint.PowerShell
```

Итерация по всем спискам семейства сайтов

Распечатайте все имена списков и количество элементов.

```
$site = Get-SPSite -Identity https://mysharepointsite/sites/test
foreach ($web in $site.AllWebs)
{
    foreach ($list in $web.Lists)
    {
        # Prints list title and item count
        Write-Output "$($list.Title), Items: $($list.ItemCount)"
    }
}
$site.Dispose()
```

Получить все установленные функции в семействе сайтов

```
Get-SPFeature -Site https://mysharepointsite/sites/test
```

Get-SPFeature также можно запускать в веб-области (`-Web <WebUrl>`), области фермы (`-Farm`) и области веб-приложений (`-WebApplication <WebAppUrl>`).

Получить все сироты в семействе сайтов

Другое использование Get-SPFeature может состоять в том, чтобы найти все функции, которые не имеют области видимости:

```
Get-SPFeature -Site https://mysharepointsite/sites/test |? { $_.Scope -eq $null )
```

Прочитайте Модуль SharePoint онлайн: https://riptutorial.com/ru/powershell/topic/5147/модуль-sharepoint

глава 39: Модуль запланированных заданий

Вступление

Примеры использования модуля «Запланированные задания», доступного в Windows 8 / Server 2012 и далее.

Examples

Запустить сценарий PowerShell в запланированной задаче

Создает запланированную задачу, которая выполняется немедленно, а затем при запуске запускает `C:\myscript.ps1` как `SYSTEM`

```
$ScheduledTaskPrincipal = New-ScheduledTaskPrincipal -UserId "SYSTEM" -LogonType
ServiceAccount
$ScheduledTaskTrigger1 = New-ScheduledTaskTrigger -AtStartup
$ScheduledTaskTrigger2 = New-ScheduledTaskTrigger -Once -At $(Get-Date) -RepetitionInterval
"00:01:00" -RepetitionDuration $([timeSpan] "24855.03:14:07")
$ScheduledTaskActionParams = @{
    Execute = "PowerShell.exe"
    Argument = '-executionpolicy Bypass -NonInteractive -c C:\myscript.ps1 -verbose >>
C:\output.log 2>&1"'
}
$ScheduledTaskAction = New-ScheduledTaskAction @ScheduledTaskActionParams
Register-ScheduledTask -Principal $ScheduledTaskPrincipal -Trigger
@($ScheduledTaskTrigger1,$ScheduledTaskTrigger2) -TaskName "Example Task" -Action
$ScheduledTaskAction
```

Прочитайте Модуль запланированных заданий онлайн:
https://riptutorial.com/ru/powershell/topic/10940/модуль-запланированных-заданий

глава 40: Наборы параметров

Вступление

Наборы параметров используются для ограничения возможной комбинации параметров или для принудительного использования параметров при выборе 1 или более параметров.

Примеры объяснят использование и причину набора параметров.

Examples

Простые наборы параметров

```
function myFunction
{
    param(
        # If parameter 'a' is used, then 'c' is mandatory
        # If parameter 'b' is used, then 'c' is optional, but allowed
        # You can use parameter 'c' in combination with either 'a' or 'b'
        # 'a' and 'b' cannot be used together

        [parameter(ParameterSetName="AandC", mandatory=$true)]
        [switch]$a,
        [parameter(ParameterSetName="BandC", mandatory=$true)]
        [switch]$b,
        [parameter(ParameterSetName="AandC", mandatory=$true)]
        [parameter(ParameterSetName="BandC", mandatory=$false)]
        [switch]$c
    )
    # $PSCmdlet.ParameterSetName can be used to check which parameter set was used
    Write-Host $PSCmdlet.ParameterSetName
}

# Valid syntaxes
myFunction -a -c
# => "Parameter set : AandC"
myFunction -b -c
# => "Parameter set : BandC"
myFunction -b
# => "Parameter set : BandC"

# Invalid syntaxes
myFunction -a -b
# => "Parameter set cannot be resolved using the specified named parameters."
myFunction -a
# => "Supply values for the following parameters:
#    c:"
```

Параметр позволяет принудительно использовать парметер, если выбрано другое.

Если вы хотите, например, принудительно использовать параметр Password, если предоставлен параметр User. (и наоборот)

```
Function Do-Something
{
    Param
    (
        [Parameter(Mandatory=$true)]
        [String]$SomeThingToDo,
        [Parameter(ParameterSetName="Credentials", mandatory=$false)]
        [String]$Computername = "LocalHost",
        [Parameter(ParameterSetName="Credentials", mandatory=$true)]
        [String]$User,
        [Parameter(ParameterSetName="Credentials", mandatory=$true)]
        [SecureString]$Password
    )

    #Do something
}

# This will not work he will ask for user and password
Do-Something -SomeThingToDo 'get-help about_Functions_Advanced' -ComputerName

# This will not work he will ask for password
Do-Something -SomeThingToDo 'get-help about_Functions_Advanced' -User
```

Параметр, заданный для ограничения комбинации пармеров

```
Function Do-Something
{
    Param
    (
        [Parameter(Mandatory=$true)]
        [String]$SomeThingToDo,
        [Parameter(ParameterSetName="Silently", mandatory=$false)]
        [Switch]$Silently,
        [Parameter(ParameterSetName="Loudly", mandatory=$false)]
        [Switch]$Loudly
    )

    #Do something
}

# This will not work because you can not use the combination Silently and Loudly
Do-Something -SomeThingToDo 'get-help about_Functions_Advanced' -Silently -Loudly
```

Прочитайте Наборы параметров онлайн: https://riptutorial.com/ru/powershell/topic/6598/наборы-параметров

глава 41: Обработка ошибок

Вступление

В этом разделе обсуждаются типы ошибок и обработка ошибок в PowerShell.

Examples

Типы ошибок

Ошибка является ошибкой, можно задаться вопросом, как могут быть типы в ней. Ну, с powershell ошибка в целом относится к двум критериям,

- Ошибка завершения
- Ошибка без прерывания

Как сказано в названии, Terminating errors завершает выполнение, а ошибки, не связанные с завершением, позволяют продолжить выполнение следующего оператора.

> Это верно, если предположить, что значение **$ ErrorActionPreference** по умолчанию (Continue). $ ErrorActionPreference - это переменная Prefrence, которая сообщает powershell, что делать в случае ошибки «Non-Terminating».

Ошибка завершения

Завершающая ошибка может быть обработана с типичным уловкой try, как показано ниже

```
Try
{
    Write-Host "Attempting Divide By Zero"
    1/0
}
Catch
{
    Write-Host "A Terminating Error: Divide by Zero Caught!"
}
```

Вышеприведенный фрагмент будет выполнен, и ошибка будет обнаружена через блок catch.

Ошибка при закрытии

В противном случае ошибка останова с другой стороны не будет поймана в блоке catch по умолчанию. Причиной этого является ошибка, не связанная с завершением, не считается критической ошибкой.

```
Try
{
    Stop-Process -Id 123456
}
Catch
{
    Write-Host "Non-Terminating Error: Invalid Process ID"
}
```

Если вы выполните вышеприведенную строку, вы не получите вывод из блока catch, так как ошибка не считается критической, и выполнение будет просто продолжено из следующей команды. Однако ошибка будет отображаться в консоли. Чтобы справиться с ошибкой без прерывания, вам просто нужно изменить предпочтение ошибки.

```
Try
{
    Stop-Process -Id 123456 -ErrorAction Stop
}
Catch
{
    "Non-Terminating Error: Invalid Process ID"
}
```

Теперь, с обновленной опцией «Ошибка», эта ошибка будет считаться завершающей ошибкой и будет поймана в блоке catch.

Вызов завершающих и не завершающих ошибок:

Командлет **Write-Error** просто записывает ошибку в вызывающую хост-программу. Это не останавливает выполнение. Если в качестве **броска** вы получите завершающую ошибку и прекратите выполнение

```
Write-host "Going to try a non terminating Error "
Write-Error "Non terminating"
Write-host "Going to try a terminating Error "
throw "Terminating Error "
Write-host "This Line wont be displayed"
```

Прочитайте Обработка ошибок онлайн: https://riptutorial.com/ru/powershell/topic/8075/обработка-ошибок

глава 42: Обработка секретов и учетных данных

Вступление

В Powershell, чтобы избежать сохранения пароля в *ясном тексте,* мы используем разные методы шифрования и сохраняем его как защищенную строку. Если вы не укажете ключ или ключ безопасности, это будет работать только для одного и того же пользователя на одном компьютере, и вы сможете расшифровать зашифрованную строку, если вы не используете Keys / SecureKeys. Любой процесс, который работает под той же учетной записью пользователя, сможет расшифровать эту зашифрованную строку на том же компьютере.

Examples

Запрос учетных данных

Чтобы запросить учетные данные, вы должны почти всегда использовать командлет `Get-Credential` :

```
$credential = Get-Credential
```

Предварительно заполненное имя пользователя:

```
$credential = Get-Credential -UserName 'myUser'
```

Добавить пользовательское приглашение:

```
$credential = Get-Credential -Message 'Please enter your company email address and password.'
```

Доступ к паролю Plaintext

Пароль в объекте учетных данных является зашифрованным `[SecureString]` . Самый простой способ - получить `[NetworkCredential]` который не хранит пароль в зашифрованном виде:

```
$credential = Get-Credential
$plainPass = $credential.GetNetworkCredential().Password
```

Вспомогательный метод (`.GetNetworkCredential()`) существует только на `[PSCredential]` . Чтобы напрямую иметь дело с `[SecureString]` , используйте методы .NET:

```
$bstr = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR($secStr)
$plainPass = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto($bstr)
```

Работа со хранимыми учетными данными

Чтобы легко хранить и извлекать зашифрованные учетные данные, используйте встроенную XML-сериализацию PowerShell (Clixml):

```
$credential = Get-Credential

$credential | Export-CliXml -Path 'C:\My\Path\cred.xml'
```

Чтобы повторно импортировать:

```
$credential = Import-CliXml -Path 'C:\My\Path\cred.xml'
```

Важно помнить, что по умолчанию это использует API защиты данных Windows, а ключ, используемый для шифрования пароля, специфичен как для *пользователя, так и для машины* , на которой работает этот код.

В результате зашифрованные учетные данные не могут быть импортированы другим пользователем или одним и тем же пользователем на другом компьютере.

Зашифровав несколько версий одних и тех же учетных данных с разными пользователями и на разных компьютерах, вы можете иметь один и тот же секрет для нескольких пользователей.

Поместив имя пользователя и компьютера в имя файла, вы можете хранить все зашифрованные секреты таким образом, чтобы один и тот же код мог использовать их без жесткого кодирования:

Encrypter

```
# run as each user, and on each computer

$credential = Get-Credential

$credential | Export-CliXml -Path
"C:\My\Secrets\myCred_${env:USERNAME}_${env:COMPUTERNAME}.xml"
```

Код, который использует сохраненные учетные данные:

```
$credential = Import-CliXml -Path
"C:\My\Secrets\myCred_${env:USERNAME}_${env:COMPUTERNAME}.xml"
```

Правильная версия файла для текущего пользователя будет загружена автоматически (или он не будет работать, потому что файл не существует).

Сохранение учетных данных в зашифрованном виде и передача его в качестве параметра, когда требуется

```
$username = "user1@domain.com"
$pwdTxt = Get-Content "C:\temp\Stored_Password.txt"
$securePwd = $pwdTxt | ConvertTo-SecureString
$credObject = New-Object System.Management.Automation.PSCredential -ArgumentList $username,
$securePwd
# Now, $credObject is having the credentials stored and you can pass it wherever you want.


## Import Password with AES

$username = "user1@domain.com"
$AESKey = Get-Content $AESKeyFilePath
$pwdTxt = Get-Content $SecurePwdFilePath
$securePwd = $pwdTxt | ConvertTo-SecureString -Key $AESKey
$credObject = New-Object System.Management.Automation.PSCredential -ArgumentList $username,
$securePwd

# Now, $credObject is having the credentials stored with AES Key and you can pass it wherever
you want.
```

Прочитайте Обработка секретов и учетных данных онлайн:
https://riptutorial.com/ru/powershell/topic/2917/обработка-секретов-и-учетных-данных

глава 43: Общие параметры

замечания

Общие параметры могут использоваться с любым командлетом (это означает, что как только вы помечаете свою функцию как командлет [см. `CmdletBinding()`], вы получаете все эти параметры бесплатно).

Вот список всех общих параметров (псевдоним в скобках после соответствующего параметра):

```
-Debug (db)
-ErrorAction (ea)
-ErrorVariable (ev)
-InformationAction (ia) # introduced in v5
-InformationVariable (iv) # introduced in v5
-OutVariable (ov)
-OutBuffer (ob)
-PipelineVariable (pv)
-Verbose (vb)
-WarningAction (wa)
-WarningVariable (wv)
-WhatIf (wi)
-Confirm (cf)
```

Examples

Параметр ErrorAction

Возможные значения: `Continue | Ignore | Inquire | SilentlyContinue | Stop | Suspend` .

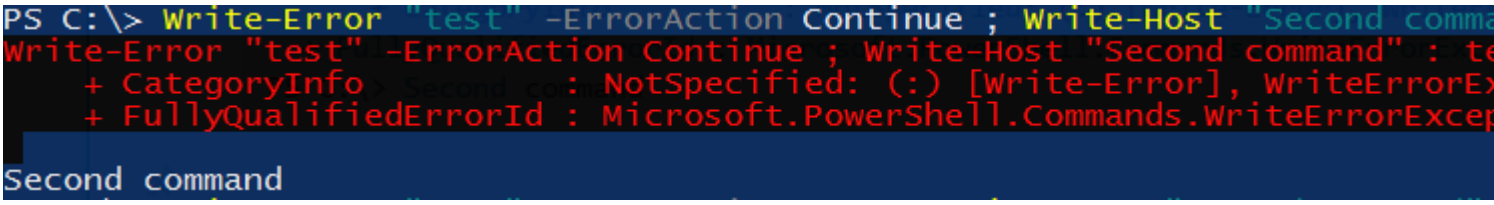
Значение этого параметра будет определять, как командлет будет обрабатывать ошибки, не связанные с завершением (например, созданные из Write-Error, чтобы узнать больше об обработке ошибок см. [*Тема еще не создана*]).

Значение по умолчанию (если этот параметр опущен) `Continue` .

-ErrorAction Продолжить

Эта опция выдаст сообщение об ошибке и продолжит выполнение.

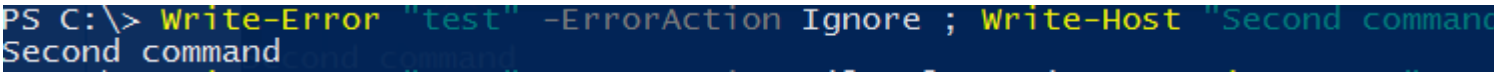
```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second command"
```



-ErrorAction Игнорировать

Этот параметр не будет вызывать сообщения об ошибке и продолжит выполнение. Также к переменной `$Error` не будет добавлена `$Error` .
Этот вариант был введен в версии 3.

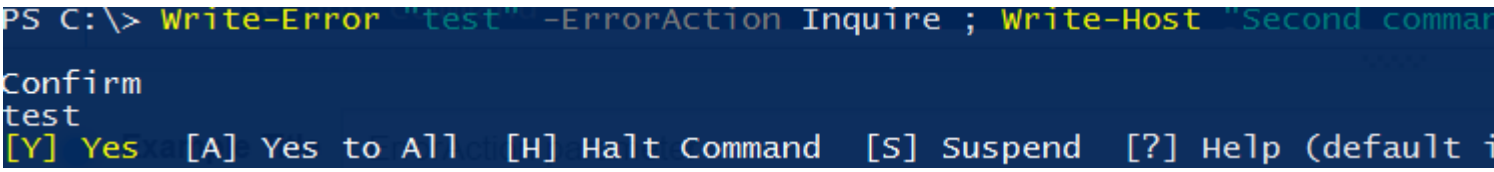
```
PS C:\> Write-Error "test" -ErrorAction Ignore ; Write-Host "Second command"
```



-ErrorAction запрашивать

Эта опция выдаст сообщение об ошибке и предложит пользователю выбрать действие.

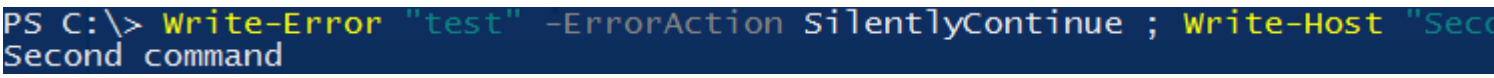
```
PS C:\> Write-Error "test" -ErrorAction Inquire ; Write-Host "Second command"
```



-ErrorAction SilentlyContinue

Эта опция не выдаст сообщение об ошибке и продолжит выполнение. Все ошибки будут добавлены в автоматическую переменную `$Error` .

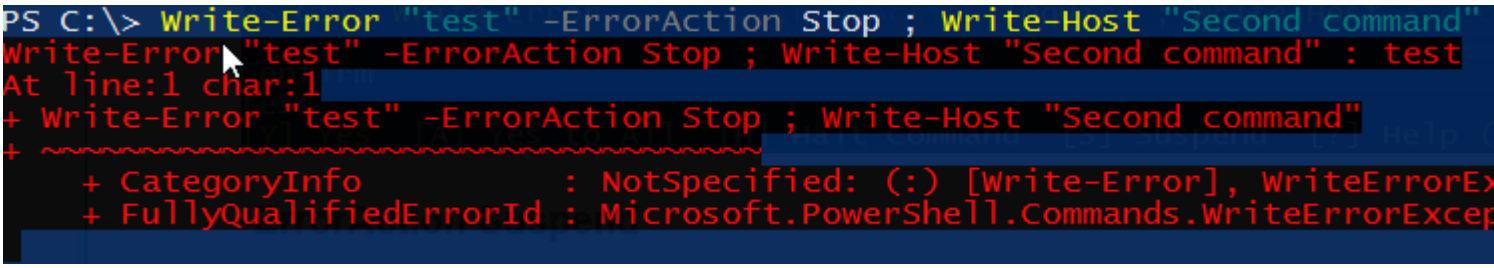
```
PS C:\> Write-Error "test" -ErrorAction SilentlyContinue ; Write-Host "Second command"
```



-ErrorAction Stop

Эта опция выдаст сообщение об ошибке и не продолжит выполнение.

```
PS C:\> Write-Error "test" -ErrorAction Stop ; Write-Host "Second command"
```



-ErrorAction Suspend

Доступно только в рабочих процессах Powershell. При использовании, если команда запускается с ошибкой, рабочий процесс приостанавливается. Это позволяет исследовать такую ошибку и дает возможность возобновить рабочий процесс. Подробнее о системе Workflow см. В разделе [тема еще не создана].

Прочитайте Общие параметры онлайн: https://riptutorial.com/ru/powershell/topic/5951/общие-параметры

глава 44: Оператор switch

Вступление

Оператор switch позволяет проверять переменную для равенства по отношению к списку значений. Каждое значение называется *случаем* , а переменная, *включенная,* проверяется для каждого случая коммутатора. Это позволяет вам писать сценарий, который может выбирать из серии опций, но не требует, чтобы вы писали длинную серию утверждений if.

замечания

В этом разделе документируется ***оператор switch,*** используемый для ветвления потока скрипта. Не путайте его с ***параметрами переключателя,*** которые используются в функциях в виде логических флагов.

Examples

Простой коммутатор

Операторы switch сравнивают одно тестовое значение с несколькими условиями и выполняют любые связанные действия для успешных сравнений. Это может привести к нескольким совпадениям / действиям.

Учитывая следующий переключатель ...

```
switch($myValue)
{
    'First Condition'    { 'First Action' }
    'Second Condition'   { 'Second Action' }
}
```

`'First Action'` будет выводиться, если `$myValue` задано как `'First Condition'` .

`'Section Action'` будет выводиться, если `$myValue` задано как `'Second Condition'` .

Ничего не будет `$myValue` если `$myValue` не соответствует ни условиям.

Заявление о переключении с параметром Regex

Параметр `-Regex` позволяет операторам операторов выполнять регулярное выражение, сопоставляющее условия.

Пример:

```
switch -Regex ('Condition')
{
  'Con\D+ion'    {'One or more non-digits'}
  'Conditio*$'   {'Zero or more "o"'}
  'C.ndition'    {'Any single char.'}
  '^C\w+ition$'  {'Anchors and one or more word chars.'}
  'Test'         {'No match'}
}
```

Выход:

```
One or more non-digits
Any single char.
Anchors and one or more word chars.
```

Простой коммутатор с перерывом

Ключевое слово `break` может использоваться в операторах switch для выхода из инструкции перед оценкой всех условий.

Пример:

```
switch('Condition')
{
  'Condition'
  {
    'First Action'
  }
  'Condition'
  {
    'Second Action'
    break
  }
  'Condition'
  {
    'Third Action'
  }
}
```

Выход:

```
First Action
Second Action
```

Из-за ключевого слова `break` во втором действии третье условие не оценивается.

Заявление переключателя с параметром подстановочного знака

Параметр `-Wildcard` позволяет операторам-переключателям выполнять подстановочные знаки, соответствующие условиям.

Пример:

```
switch -Wildcard ('Condition')
{
    'Condition'           {'Normal match'}
    'Condit*'             {'Zero or more wildcard chars.'}
    'C[aoc]ndit[f-l]on'   {'Range and set of chars.'}
    'C?ndition'           {'Single char. wildcard'}
    'Test*'               {'No match'}
}
```

Выход:

```
Normal match
Zero or more wildcard chars.
Range and set of chars.
Single char. wildcard
```

Заявление о выводе с точным параметром

Параметр `-Exact` обеспечивает выполнение операторов switch для выполнения точного, нечувствительного к регистру сопоставления с строковыми условиями.

Пример:

```
switch -Exact ('Condition')
{
  'condition'   {'First Action'}
  'Condition'   {'Second Action'}
  'conditioN'   {'Third Action'}
  '^*ondition$' {'Fourth Action'}
  'Conditio*'   {'Fifth Action'}
}
```

Выход:

```
First Action
Second Action
Third Action
```

Выполняются первые-третье действия, потому что соответствующие им условия соответствуют входу. Строки регулярных выражений и подстановочных знаков в четвертом и пятом состояниях не совпадают.

Обратите внимание, что четвертое условие также будет соответствовать входной строке, если выполняется сопоставление регулярных выражений, но в этом случае игнорируется, потому что это не так.

Заявление о переключении с параметром CaseSensitive

Параметр `-CaseSensitive` вводит операторы switch для выполнения точного, `-CaseSensitive` регистр соответствия условий.

Пример:

```
switch -CaseSensitive ('Condition')
{
  'condition'   {'First Action'}
  'Condition'   {'Second Action'}
  'conditioN'   {'Third Action'}
}
```

Выход:

```
Second Action
```

Второе действие - единственное действие, выполняемое, потому что оно является единственным условием, которое точно соответствует строке `'Condition'` при учете чувствительности к регистру.

Выписка переключателя с параметром файла

Параметр `-file` позволяет оператору switch получать входные данные из файла. Каждая строка файла оценивается оператором switch.

Пример файла `input.txt` :

```
condition
test
```

Пример оператора switch:

```
switch -file input.txt
{
  'condition' {'First Action'}
  'test'      {'Second Action'}
  'fail'      {'Third Action'}
}
```

Выход:

```
First Action
Second Action
```

Простой коммутатор с условием по умолчанию

Ключевое слово `Default` используется для выполнения действия, когда никакие другие условия не соответствуют входному значению.

Пример:

```
switch('Condition')
{
  'Skip Condition'
  {
    'First Action'
  }
  'Skip This Condition Too'
  {
    'Second Action'
  }
  Default
  {
    'Default Action'
  }
}
```

Выход:

```
Default Action
```

Заявление о выводе с выражениями

Условиями могут быть также выражения:

```
$myInput = 0

switch($myInput) {
    # because the result of the expression, 4,
    # does not equal our input this block should not be run.
    (2+2)  { 'True. 2 +2 = 4' }

    # because the result of the expression, 0,
    # does equal our input this block should be run.
    (2-2) { 'True. 2-2 = 0' }

    # because our input is greater than -1 and is less than 1
    # the expression evaluates to true and the block should be run.
    { $_ -gt -1 -and $_ -lt 1 } { 'True. Value is 0' }
}

#Output
True. 2-2 = 0
True. Value is 0
```

Прочитайте Оператор switch онлайн: https://riptutorial.com/ru/powershell/topic/1174/оператор-switch

глава 45: операторы

Вступление

Оператор - это символ, который представляет действие. Он сообщает компилятору / интерпретатору выполнить определенную математическую, реляционную или логическую операцию и дать конечный результат. PowerShell интерпретируется определенным образом и категоризирует соответственно, так как арифметические операторы выполняют операции в основном по номерам, но также влияют на строки и другие типы данных. Наряду с основными операторами PowerShell имеет ряд операторов, которые экономят время и усилия по кодированию (например: -like, -match, -replace и т. Д.).

Examples

Арифметические операторы

```
1 + 2      # Addition
1 - 2      # Subtraction
-1         # Set negative value
1 * 2      # Multiplication
1 / 2      # Division
1 % 2      # Modulus
100 -shl 2 # Bitwise Shift-left
100 -shr 1 # Bitwise Shift-right
```

Логические операторы

```
-and # Logical and
-or  # Logical or
-xor # Logical exclusive or
-not # Logical not
!    # Logical not
```

Операторы присваивания

Простая арифметика:

```
$var = 1      # Assignment. Sets the value of a variable to the specified value
$var += 2     # Addition. Increases the value of a variable by the specified value
$var -= 1     # Subtraction. Decreases the value of a variable by the specified value
$var *= 2     # Multiplication. Multiplies the value of a variable by the specified value
$var /= 2     # Division. Divides the value of a variable by the specified value
$var %= 2     # Modulus. Divides the value of a variable by the specified value and then
              # assigns the remainder (modulus) to the variable
```

Приращение и декремент:

```
$var++   # Increases the value of a variable, assignable property, or array element by 1
$var--   # Decreases the value of a variable, assignable property, or array element by 1
```

Операторы сравнения

Операторы сравнения PowerShell состоят из ведущего тире (`-`), за которым следует имя (`eq` для `equal` , `gt` для `greater than` и т. Д.).

Именам могут предшествовать специальные символы для изменения поведения оператора:

```
i # Case-Insensitive Explicit (-ieq)
c # Case-Sensitive Explicit (-ceq)
```

Нечувствительность к регистру - это значение по умолчанию, если не указано, («a» -eq «A») такое же, как («a» -ieq «A»).

Простые операторы сравнения:

```
2 -eq 2    # Equal to (==)
2 -ne 4    # Not equal to (!=)
5 -gt 2    # Greater-than (>)
5 -ge 5    # Greater-than or equal to (>=)
5 -lt 10   # Less-than (<)
5 -le 5    # Less-than or equal to (<=)
```

Операторы сравнения строк:

```
"MyString" -like "*String"            # Match using the wildcard character (*)
"MyString" -notlike "Other*"          # Does not match using the wildcard character (*)
"MyString" -match "$String^"          # Matches a string using regular expressions
"MyString" -notmatch "$Other^"        # Does not match a string using regular expressions
```

Операторы сравнения коллекции:

```
"abc", "def" -contains "def"            # Returns true when the value (right) is present
                                        # in the array (left)
"abc", "def" -notcontains "123"         # Returns true when the value (right) is not present
                                        # in the array (left)
"def" -in "abc", "def"                  # Returns true when the value (left) is present
                                        # in the array (right)
"123" -notin "abc", "def"               # Returns true when the value (left) is not present
                                        # in the array (right)
```

Операторы перенаправления

Выходной поток успеха:

```
cmdlet > file     # Send success output to file, overwriting existing content
cmdlet >> file    # Send success output to file, appending to existing content
cmdlet 1>&2       # Send success and error output to error stream
```

Выходной поток ошибки:

```
cmdlet 2> file    # Send error output to file, overwriting existing content
cmdlet 2>> file   # Send error output to file, appending to existing content
cmdlet 2>&1       # Send success and error output to success output stream
```

Выходной поток предупреждения: (PowerShell 3.0+)

```
cmdlet 3> file    # Send warning output to file, overwriting existing content
cmdlet 3>> file   # Send warning output to file, appending to existing content
cmdlet 3>&1       # Send success and warning output to success output stream
```

Подробный выходной поток: (PowerShell 3.0+)

```
cmdlet 4> file    # Send verbose output to file, overwriting existing content
cmdlet 4>> file   # Send verbose output to file, appending to existing content
cmdlet 4>&1       # Send success and verbose output to success output stream
```

Отладочный выходной поток: (PowerShell 3.0+)

```
cmdlet 5> file    # Send debug output to file, overwriting existing content
cmdlet 5>> file   # Send debug output to file, appending to existing content
cmdlet 5>&1       # Send success and debug output to success output stream
```

Выходной поток информации: (PowerShell 5.0+)

```
cmdlet 6> file    # Send information output to file, overwriting existing content
cmdlet 6>> file   # Send information output to file, appending to existing content
cmdlet 6>&1       # Send success and information output to success output stream
```

Все выходные потоки:

```
cmdlet *> file    # Send all output streams to file, overwriting existing content
cmdlet *>> file   # Send all output streams to file, appending to existing content
cmdlet *>&1       # Send all output streams to success output stream
```

Отличия от оператора трубы (|)

Операторы перенаправления перенаправляют потоки в файлы или потоки в потоки. Оператор трубы передает объект вниз по трубопроводу в командлет или на выход. Как работает трубопровод, в целом отличается от того, как работает перенаправление, и его можно прочитать в разделе «Работа с конвейером PowerShell»

Смешивание типов операндов: тип левого операнда диктует поведение.

Для добавления

```
"4" + 2         # Gives "42"
4 + "2"         # Gives 6
```

```
1,2,3 + "Hello" # Gives 1,2,3,"Hello"
"Hello" + 1,2,3 # Gives "Hello1 2 3"
```

Для умножения

```
"3" * 2   # Gives "33"
2 * "3"   # Gives 6
1,2,3 * 2 # Gives 1,2,3,1,2,3
2 * 1,2,3 # Gives an error op_Multiply is missing
```

Воздействие может иметь скрытые последствия для операторов сравнения:

```
$a = Read-Host "Enter a number"
Enter a number : 33
$a -gt 5
False
```

Операторы манипуляции строками

Заменить оператора:

Оператор `-replace` заменяет шаблон во входном значении с использованием регулярного выражения. Этот оператор использует два аргумента (разделенных запятой): шаблон регулярного выражения и его значение замены (которое по умолчанию является необязательным и пустая строка).

```
"The rain in Seattle" -replace 'rain','hail'        #Returns: The hail in Seattle
"kenmyer@contoso.com" -replace '^[\w]+@(.+)', '$1'  #Returns: contoso.com
```

Операторы разделения и присоединения:

Оператор `-split` разбивает строку на массив подстрок.

```
"A B C" -split " "      #Returns an array string collection object containing A,B and C.
```

Оператор `-join` объединяет массив строк в одну строку.

```
"E","F","G" -join ":"   #Returns a single string: E:F:G
```

Прочитайте операторы онлайн: https://riptutorial.com/ru/powershell/topic/1071/операторы

глава 46: Операции с базовым набором

Вступление

Набор представляет собой набор элементов, которые могут быть любыми. Независимо от того, какой оператор нам нужно работать над этими наборами, коротким *операторы множеств* и операция также известна как *заданная операция* . Базовая установка включает в себя Union, Intersection, а также сложение, вычитание и т. Д.

Синтаксис

- Group-Object
- Group-Object -Property <propertyName>
- Group-Object -Property <propertyName>, <propertyName2>
- Group-Object -Property <propertyName> -CaseSensitive
- Group-Object -Property <propertyName> -Culture <культура>
- Group-Object -Property <ScriptBlock>
- Сортировка-объект
- Sort-Object -Property <propertyName>
- Sort-Object -Property <ScriptBlock>
- Sort-Object -Property <propertyName>, <propertyName2>
- Sort-Object -Property <свойствоObject> -CaseSensitive
- Sort-Object -Property <propertyObject> -Descending
- Sort-Object -Property <свойствоObject> -Unique
- Sort-Object -Property <свойствоObject> -Culture <культура>

Examples

Фильтрация: Где-Объект / где /?

Фильтрация перечисления с использованием условного выражения

Синонимы:

```
Where-Object
where
?
```

Пример:

```
$names = @( "Aaron", "Albert", "Alphonse","Bernie", "Charlie", "Danny", "Ernie", "Frank")

$names | Where-Object { $_ -like "A*" }
$names | where { $_ -like "A*" }
$names | ? { $_ -like "A*" }
```

Возвращает:

Аарон
Альберт
Альфонс

Заказ: Сортировка-Объект / сортировка

Сортировка перечисления в порядке возрастания или убывания

Синонимы:

```
Sort-Object
sort
```

Предполагая, что:

```
$names = @( "Aaron", "Aaron", "Bernie", "Charlie", "Danny" )
```

По возрастанию сортировка по умолчанию:

```
$names | Sort-Object
$names | sort
```

Аарон
Аарон
Bernie
Чарли
Дэнни

Чтобы запросить убывающий порядок:

```
$names | Sort-Object -Descending
$names | sort -Descending
```

Дэнни

Чарли
Bernie
Аарон
Аарон

Вы можете сортировать, используя выражение.

```
$names | Sort-Object { $_.length }
```

Аарон
Аарон
Дэнни
Bernie
Чарли

Группировка: группа-объект / группа

Вы можете группировать перечисление, основанное на выражении.

Синонимы:

```
Group-Object
group
```

Примеры:

```
$names = @( "Aaron", "Albert", "Alphonse","Bernie", "Charlie", "Danny", "Ernie", "Frank")

$names | Group-Object -Property Length
$names | group -Property Length
```

Отклик:

подсчитывать	название	группа
4	5	{Аарон, Дэнни, Эрни, Франк}
2	6	{Альберт, Берни}
1	8	{Альфонс}
1	7	{Чарли}

Проектирование: Select-Object / select

Проецирование перечисления позволяет вам извлекать определенные элементы каждого

объекта, извлекать все детали или вычислять значения для каждого объекта

Синонимы:

```
Select-Object
select
```

Выбор подмножества свойств:

```
$dir = dir "C:\MyFolder"

$dir | Select-Object Name, FullName, Attributes
$dir | select Name, FullName, Attributes
```

название	ФИО	Атрибуты
Изображений	C: \ MyFolder \ Images	каталог
data.txt	C: \ MyFolder \ data.txt	Архив
source.c	C: \ MyFolder \ source.c	Архив

Выбор первого элемента и отображение всех его свойств:

```
$d | select -first 1 *
```

PSPath
PSParentPath
PSChildName
PSDrive
PSProvider
PsIsContainer
BaseName
Режим
название
родитель
Существует

корень
ФИО
расширение
CreationTime
CreationTimeUtc
LastAccessTime
LastAccessTimeUtc
LastWriteTime
LastWriteTimeUtc
Атрибуты

Прочитайте Операции с базовым набором онлайн:
https://riptutorial.com/ru/powershell/topic/1557/операции-с-базовым-набором

глава 47: Отправка электронной почты

Вступление

Полезным методом для администраторов Exchange Server является возможность отправлять сообщения электронной почты через SMTP из PowerShell. В зависимости от версии PowerShell, установленной на вашем компьютере или сервере, существует несколько способов отправки писем через powershell. Существует встроенная опция командлета, которая проста и проста в использовании. Он использует командлет ***Send-MailMessage*** .

параметры

параметр	подробности
Вложения <String []>	Имена файлов путей и файлов для прикрепления к сообщению. Пути и имена файлов могут быть отправлены в Send-MailMessage.
Bcc <String []>	Адреса электронной почты, которые получают копию сообщения электронной почты, но не отображаются в качестве получателя в сообщении. Введите имена (необязательно) и адрес электронной почты (обязательно), например имя someone@example.com или someone@example.com.
Тело <String_>	Содержание сообщения электронной почты.
BodyAsHtml	Он указывает, что содержимое находится в формате HTML.
Cc <String []>	Адреса электронной почты, которые получают копию сообщения электронной почты. Введите имена (необязательно) и адрес электронной почты (обязательно), например имя someone@example.com или someone@example.com.
мандат	Указывает учетную запись пользователя, у которой есть разрешение на отправку сообщения с указанного адреса электронной почты. По умолчанию используется текущий пользователь. Введите имя, например User или Domain \ User, или введите объект PSCredential.
DeliveryNotificationOption	Определяет параметры уведомления о доставке для

параметр	подробности
	сообщения электронной почты. Можно указать несколько значений. Уведомления о доставке отправляются в сообщении по адресу, указанному в параметре To. Допустимые значения: Нет, OnSuccess, OnFailure, Delay, Never.
кодирование	Кодирование тела и объекта. Допустимые значения: ASCII, UTF8, UTF7, UTF32, Unicode, BigEndianUnicode, Default, OEM.
От	Адреса электронной почты, с которых отправляется почта. Введите имена (необязательно) и адрес электронной почты (требуется), например Name someone@example.com или someone@example.com.
порт	Альтернативный порт на SMTP-сервере. Значение по умолчанию - 25. Доступно в Windows PowerShell 3.0.
приоритет	Приоритет сообщения электронной почты. Допустимые значения: нормальный, высокий, низкий.
SmtpServer	Имя SMTP-сервера, отправляющего сообщение электронной почты. Значение по умолчанию - значение переменной $ PSEmailServer.
Предмет	Тема сообщения электронной почты.
к	Адреса электронной почты, на которые отправляется почта. Введите имена (необязательно) и адрес электронной почты (обязательно), например имя someone@example.com или someone@example.com
UseSSL	Использует протокол Secure Sockets Layer (SSL) для установления соединения с удаленным компьютером для отправки почты

Examples

Простой Send-MailMessage

```
Send-MailMessage -From sender@bar.com -Subject "Email Subject" -To receiver@bar.com -
SmtpServer smtp.com
```

Send-MailMessage с предопределенными параметрами

```
$parameters = @{
    From = 'from@bar.com'
    To = 'to@bar.com'
    Subject = 'Email Subject'
    Attachments =  @('C:\files\samplefile1.txt','C:\files\samplefile2.txt')
    BCC = 'bcc@bar.com'
    Body = 'Email body'
    BodyAsHTML = $False
    CC = 'cc@bar.com'
    Credential = Get-Credential
    DeliveryNotificationOption = 'onSuccess'
    Encoding = 'UTF8'
    Port = '25'
    Priority = 'High'
    SmtpServer = 'smtp.com'
    UseSSL = $True
}

# Notice: Splatting requires @ instead of $ in front of variable name
Send-MailMessage @parameters
```

SMTPClient - отправить сообщение с .txt в сообщении body

```
# Define the txt which will be in the email body
$Txt_File = "c:\file.txt"

function Send_mail {
    #Define Email settings
    $EmailFrom = "source@domain.com"
    $EmailTo = "destination@domain.com"
    $Txt_Body = Get-Content $Txt_File -RAW
    $Body = $Body_Custom + $Txt_Body
    $Subject = "Email Subject"
    $SMTPServer = "smtpserver.domain.com"
    $SMTPClient = New-Object Net.Mail.SmtpClient($SmtpServer, 25)
    $SMTPClient.EnableSsl = $false
    $SMTPClient.Send($EmailFrom, $EmailTo, $Subject, $Body)

}

$Body_Custom = "This is what contain file.txt : "

Send_mail
```

Прочитайте Отправка электронной почты онлайн:
https://riptutorial.com/ru/powershell/topic/2040/отправка-электронной-почты

глава 48: Переменные в PowerShell

Вступление

Переменные используются для хранения значений. Пусть значение будет любого типа, мы должны его хранить где-нибудь, чтобы мы могли использовать его во всей консоли / скрипте. Имена переменных в PowerShell начинаются с **$** , как в *$ Variable1* , а значения присваиваются с помощью **=** , например **$ Variable1 = "Value 1"** .PowerShell поддерживает огромное количество типов переменных; такие как текстовые строки, целые числа, десятичные знаки, массивы и даже расширенные типы, такие как номера версий или IP-адреса.

Examples

Простая переменная

Все переменные в powershell начинаются с знака доллара США (`$`). Самый простой пример:

```
$foo = "bar"
```

Этот оператор выделяет переменную с именем `foo` со строковым значением «bar».

Удаление переменной

Чтобы удалить переменную из памяти, можно использовать командлет `Remove-Item` . Примечание. Имя переменной НЕ включает `$` .

```
Remove-Item Variable:\foo
```

`Variable` есть провайдер, позволяющий большинству командлетов * -item работать так же, как файловые системы.

Другим методом удаления переменной является использование командлета Remove-Variable и его псевдоним rv

```
$var = "Some Variable" #Define variable 'var' containing the string 'Some Variable'
$var #For test and show string 'Some Variable' on the console

Remove-Variable -Name var
$var

#also can use alias 'rv'
rv var
```

Объем

По умолчанию сфера для переменного является вшит контейнером. Если внешний сценарий или другой контейнер, то область действия - `Global` . Чтобы указать область действия , она имеет префикс имени переменной `$scope:varname` :

```
$foo = "Global Scope"
function myFunc {
    $foo = "Function (local) scope"
    Write-Host $global:foo
    Write-Host $local:foo
    Write-Host $foo
}
myFunc
Write-Host $local:foo
Write-Host $foo
```

Выход:

```
    Global Scope
    Function (local) scope
    Function (local) scope
    Global Scope
    Global Scope
```

Чтение вывода CmdLet

По умолчанию, powershell вернет результат в вызывающий Entity. Рассмотрим ниже пример,

```
Get-Process -Name excel
```

Это просто возвращает возвращаемый процесс, который соответствует имени excel, вызывающему объекту. В этом случае узел PowerShell. Он печатает что-то вроде,

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  SI ProcessName

-------  ------    -----      ----- -----   ------     --  -- -----------

   1037      54    67632      62544   617     5.23   4544   1 EXCEL
```

Теперь, если вы назначаете выход переменной, она просто ничего не печатает. И, конечно, переменная содержит выход. (Будь он строкой, Объект - Любой тип, если на то пошло)

```
$allExcel = Get-Process -Name excel
```

Итак, скажем, у вас есть сценарий, в котором вы хотите назначить переменную динамическим именем, вы можете использовать параметр **`-OutVariable`**

```
Get-Process -Name excel -OutVariable AllRunningExcel
```

Обратите внимание, что здесь отсутствует «$». Основное различие между этими двумя назначениями заключается в том, что он также выводит результат отдельно от назначения его переменной AllRunningExcel. Вы также можете назначить его другой переменной.

```
$VarOne = Get-Process -Name excel -OutVariable VarTwo
```

Хотя, приведенный выше сценарий очень редок, обе переменные $ VarOne & $ VarTwo будут иметь одинаковое значение.

Теперь рассмотрим это,

```
Get-Process -Name EXCEL -OutVariable MSOFFICE
Get-Process -Name WINWORD -OutVariable +MSOFFICE
```

Первое утверждение просто получило бы процесс excel и присвоило бы его переменной MSOFFICE, а затем выполнило бы процессы ms word и «добавило» это к существующему значению MSOFFICE. Это будет выглядеть примерно так,

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  SI ProcessName

-------  ------    -----      ----- -----   ------     --  -- -----------

   1047      54    67720      64448   618     5.70   4544   1 EXCEL

   1172      70    50052      81780   584     1.83  14968   1 WINWORD
```

Перечисление списков нескольких переменных

Powershell допускает множественное назначение переменных и обрабатывает почти все, как массив или список. Это означает, что вместо того, чтобы делать что-то вроде этого:

```
$input = "foo.bar.baz"
$parts = $input.Split(".")
$foo = $parts[0]
$bar = $parts[1]
$baz = $parts[2]
```

Вы можете просто сделать это:

```
$foo, $bar, $baz = $input.Split(".")
```

Так как Powershell рассматривает присвоения таким образом, как списки, если в списке больше значений, чем элементов в списке переменных для их назначения, последняя переменная становится массивом оставшихся значений. Это означает, что вы также можете делать такие вещи:

```
$foo, $leftover = $input.Split(".") #Sets $foo = "foo", $leftover = ["bar","baz"]
$bar = $leftover[0] # $bar = "bar"
```

```
$baz = $leftover[1] # $baz = "baz"
```

Массивы

Объявление Array в Powershell практически не отличается от экземпляра любой другой переменной, т. Е. Используется синтаксис `$name =` . Элементы массива объявляются путем разделения их запятыми (`,`):

```
$myArrayOfInts = 1,2,3,4
$myArrayOfStrings = "1","2","3","4"
```

Добавление в arry

Добавление в массив так же просто, как использование оператора `+` :

```
$myArrayOfInts = $myArrayOfInts + 5
//now contains 1,2,3,4 & 5!
```

Объединение массивов вместе

Опять же, это так же просто, как использование оператора `+`

```
$myArrayOfInts = 1,2,3,4
$myOtherArrayOfInts = 5,6,7
$myArrayOfInts = $myArrayOfInts + $myOtherArrayOfInts
//now 1,2,3,4,5,6,7
```

Прочитайте Переменные в PowerShell онлайн: https://riptutorial.com/ru/powershell/topic/3457/переменные-в-powershell

глава 49: Переменные среды

Examples

Переменные среды Windows видны как привод PS, называемый Env:

Вы можете просмотреть список со всеми переменными окружения:
Get-Childitem env:

Мгновенный вызов переменных среды с помощью $ env:

```
$env:COMPUTERNAME
```

Прочитайте Переменные среды онлайн: https://riptutorial.com/ru/powershell/topic/5635/переменные-среды

глава 50: Повторное признание Amazon Web Services (AWS)

Вступление

Amazon Rekognition - это сервис, который упрощает добавление анализа изображений в ваши приложения. С Rekognition вы можете обнаруживать объекты, сцены и лица в изображениях. Вы также можете искать и сравнивать лица. API-интерфейс Rekognition позволяет вам быстро добавлять в ваши приложения сложный визуальный поиск и классификацию изображений на основе глубокого обучения.

Examples

Обнаруживать метки изображений с помощью распознавания AWS

```
$BucketName = 'trevorrekognition'
$FileName = 'kitchen.jpg'

New-S3Bucket -BucketName $BucketName
Write-S3Object -BucketName $BucketName -File $FileName
$REKResult = Find-REKLabel -Region us-east-1 -ImageBucket $BucketName -ImageName $FileName

$REKResult.Labels
```

После запуска сценария выше вы должны напечатать результаты на своем хосте PowerShell, которые выглядят примерно так:

```
RESULTS:

Confidence Name
---------- ----
86.87605   Indoors
86.87605   Interior Design
86.87605   Room
77.4853    Kitchen
77.25354   Housing
77.25354   Loft
66.77325   Appliance
66.77325   Oven
```

Используя модуль AWS PowerShell в сочетании с услугой ReSognition AWS, вы можете обнаруживать метки в изображении, такие как идентификация объектов в комнате, атрибуты о сделанных вами фотографиях и соответствующий уровень достоверности, который AWS Rekognition имеет для каждого из этих атрибутов.

Команда `Find-REKLabel` - это та, которая позволяет вам вызывать поиск этих атрибутов /

меток. Хотя вы можете предоставить контент изображения в виде байтового массива во время вызова API, лучшим способом является загрузка ваших файлов изображений в ведро AWS S3, а затем укажите службу Rekognition на объекты S3, которые вы хотите проанализировать. Пример выше показывает, как это сделать.

Сравнить сходство с лицом с Rexognition AWS

```
$BucketName = 'trevorrekognition'

### Create a new AWS S3 Bucket
New-S3Bucket -BucketName $BucketName

### Upload two different photos of myself to AWS S3 Bucket
Write-S3Object -BucketName $BucketName -File myphoto1.jpg
Write-S3Object -BucketName $BucketName -File myphoto2.jpg

### Perform a facial comparison between the two photos with AWS Rekognition
$Comparison = @{
    SourceImageBucket = $BucketName
    TargetImageBucket = $BucketName
    SourceImageName = 'myphoto1.jpg'
    TargetImageName = 'myphoto2.jpg'
    Region = 'us-east-1'
}
$Result = Compare-REKFace @Comparison
$Result.FaceMatches
```

Пример сценария, приведенный выше, должен дать вам результаты, похожие на следующие:

```
Face                                Similarity
----                                ----------
Amazon.Rekognition.Model.ComparedFace 90
```

Услуга AWS Rekognition позволяет выполнять сравнение лиц между двумя фотографиями. Использование этой услуги довольно просто. Просто загрузите два файла изображений, которые вы хотите сравнить, в ведро AWS S3. Затем вызовите команду `Compare-REKFace` , аналогичную приведенному выше примеру. Конечно, вам нужно будет предоставить свое собственное, глобально уникальное имя и имена файлов S3 Bucket.

Прочитайте Повторное признание Amazon Web Services (AWS) онлайн: https://riptutorial.com/ru/powershell/topic/9581/повторное-признание-amazon-web-services--aws-

глава 51: Поддержка с комментариями

Вступление

В PowerShell есть механизм документирования, основанный на комментариях. Он позволяет документировать сценарии и функции с комментариями кодов. Основанная на комментариях помощь в большинстве случаев написана в блоках комментариев, содержащих несколько ключевых слов справки. Ключевые слова справки начинаются с точек и определяют разделы справки, которые будут отображаться при запуске командлета `Get-Help` .

Examples

Поддержка функций на основе комментариев

```
<#

.SYNOPSIS
    Gets the content of an INI file.

.DESCRIPTION
    Gets the content of an INI file and returns it as a hashtable.

.INPUTS
    System.String

.OUTPUTS
    System.Collections.Hashtable

.PARAMETER FilePath
    Specifies the path to the input INI file.

.EXAMPLE
    C:\PS>$IniContent = Get-IniContent -FilePath file.ini
    C:\PS>$IniContent['Section1'].Key1
    Gets the content of file.ini and access Key1 from Section1.

.LINK
    Out-IniFile

#>
function Get-IniContent
{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory=$true,ValueFromPipeline=$true)]
        [ValidateNotNullOrEmpty()]
        [ValidateScript({(Test-Path $_) -and ((Get-Item $_).Extension -eq ".ini")})]
        [System.String]$FilePath
    )
```

```
    # Initialize output hash table.
    $ini = @{}
    switch -regex -file $FilePath
    {
        "^\[(.+)\]$" # Section
        {
            $section = $matches[1]
            $ini[$section] = @{}
            $CommentCount = 0
        }
        "^(;.*)$" # Comment
        {
            if( !($section) )
            {
                $section = "No-Section"
                $ini[$section] = @{}
            }
            $value = $matches[1]
            $CommentCount = $CommentCount + 1
            $name = "Comment" + $CommentCount
            $ini[$section][$name] = $value
        }
        "(.+?)\s*=\s*(.*)" # Key
        {
            if( !($section) )
            {
                $section = "No-Section"
                $ini[$section] = @{}
            }
            $name,$value = $matches[1..2]
            $ini[$section][$name] = $value
        }
    }

    return $ini
}
```

Вышеупомянутую функциональную документацию можно отобразить, запустив `Get-Help -Name Get-IniContent -Full` :

```
PS C:\Scripts> Get-Help -Name Get-IniContent -Full

NAME
    Get-IniContent

SYNOPSIS
    Gets the content of an INI file.


SYNTAX
    Get-IniContent [-FilePath] <String> [<CommonParameters>]


DESCRIPTION
    Gets the content of an INI file and returns it as a hashtable.


PARAMETERS
    -FilePath <String>
        Specifies the path to the input INI file.

        Required?                    true
        Position?                    1
        Default value
        Accept pipeline input?       true (ByValue)
        Accept wildcard characters?  false

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    System.String


OUTPUTS
    System.Collections.Hashtable


    -------------------------- EXAMPLE 1 --------------------------

    C:\PS>$IniContent = Get-IniContent -FilePath file.ini

    C:\PS>$IniContent['Section1'].Key1
    Gets the content of file.ini and access Key1 from Section1.




RELATED LINKS
    Out-IniFile



PS C:\Scripts>
```

Обратите внимание, что ключевые слова, основанные на комментариях, начинаются с . соответствуют разделам результатов `Get-Help` .

Сценарий, основанный на комментариях

```
<#

.SYNOPSIS
    Reads a CSV file and filters it.
```

```
.DESCRIPTION
    The ReadUsersCsv.ps1 script reads a CSV file and filters it on the 'UserName' column.

.PARAMETER Path
    Specifies the path of the CSV input file.

.INPUTS
    None. You cannot pipe objects to ReadUsersCsv.ps1.

.OUTPUTS
    None. ReadUsersCsv.ps1 does not generate any output.

.EXAMPLE
    C:\PS> .\ReadUsersCsv.ps1 -Path C:\Temp\Users.csv -UserName j.doe

#>
Param
(
    [Parameter(Mandatory=$true,ValueFromPipeline=$false)]
    [System.String]
    $Path,
    [Parameter(Mandatory=$true,ValueFromPipeline=$false)]
    [System.String]
    $UserName
)

Import-Csv -Path $Path | Where-Object -FilterScript {$_.UserName -eq $UserName}
```

Вышеупомянутую документацию сценария можно отобразить, запустив `Get-Help -Name ReadUsersCsv.ps1 -Full` :

```
PS C:\Scripts> Get-Help -Name .\ReadUsersCsv.ps1 -Full

NAME
    C:\Scripts\ReadUsersCsv.ps1

SYNOPSIS
    Reads a CSV file and filters it.


SYNTAX
    C:\Scripts\ReadUsersCsv.ps1 [-Path] <String> [-UserName] <String> [<CommonParameters>]


DESCRIPTION
    The ReadUsersCsv.ps1 script reads a CSV file and filters it on the 'UserName' column.


PARAMETERS
    -Path <String>
        Specifies the path of the CSV input file.

        Required?                    true
        Position?                    1
        Default value
        Accept pipeline input?       false
        Accept wildcard characters?  false

    -UserName <String>
        Specifies the user name that will be used to filter the CSV file.

        Required?                    true
        Position?                    2
        Default value
        Accept pipeline input?       false
        Accept wildcard characters?  false

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    None. You cannot pipe objects to ReadUsersCsv.ps1.


OUTPUTS
    None. ReadUsersCsv.ps1 does not generate any output.


    -------------------------- EXAMPLE 1 --------------------------

    C:\PS>.\ReadUsersCsv.ps1 -Path C:\Temp\Users.csv -UserName j.doe




RELATED LINKS



PS C:\Scripts>
```

Прочитайте Поддержка с комментариями онлайн:
https://riptutorial.com/ru/powershell/topic/9530/поддержка-с-комментариями

глава 52: Принудительное выполнение сценариев

Синтаксис

- #Requires -Version <N> [. <N>]
- #Requires -PSSnapin <PSSnapin-Name> [-Version <N> [. <N>]]
- #Requires -Modules {<Module-Name> | <Hashtable>}
- #Requires -ShellId <ShellId>
- #Requires -RunAsAdministrator

замечания

`#requires` может быть помещен в любую строку в скрипте (он не должен быть первой строкой), но он должен быть первым оператором в этой строке.

Несколько операторов `#requires` могут использоваться в одном скрипте.

Для получения дополнительной информации обратитесь к официальной документации по Technet - about_about_Requires .

Examples

Обеспечить минимальную версию узла powershell

```
#requires -version 4
```

После попытки запуска этого скрипта в более низкой версии вы увидите это сообщение об ошибке

> . \ script.ps1: сценарий «script.ps1» не может быть запущен, потому что в нем содержится оператор «#requires» в строке 1 для Windows PowerShell версии 5.0. Версия, требуемая скриптом, не соответствует текущей версии Windows PowerShell версии 2.0.

Принудительно запустить скрипт в качестве администратора

4,0

```
#requires -RunAsAdministrator
```

После попытки запуска этого скрипта без прав администратора вы увидите это сообщение

об ошибке

> . \ script.ps1: сценарий «script.ps1» не может быть запущен, потому что он содержит инструкцию «#requires» для работы в качестве администратора. Текущий сеанс Windows PowerShell не работает как администратор. Запустите Windows PowerShell с помощью параметра «Запуск от имени администратора» и повторите попытку запуска сценария.

Прочитайте Принудительное выполнение сценариев онлайн:
https://riptutorial.com/ru/powershell/topic/5637/принудительное-выполнение-сценариев

глава 53: Простая служба хранения (S3) Amazon Web Services (AWS)

Вступление

В этом разделе документации основное внимание уделяется разработке простейшей службы хранения данных Amazon Web Services (AWS) (S3). S3 - это действительно простой сервис для взаимодействия. Вы создаете S3 «ведра», которые могут содержать ноль или более «объектов». Создав ведро, вы можете загружать файлы или произвольные данные в ведро S3 как «объект». Вы ссылаетесь на объекты S3 внутри ведра на «ключ» (имя) объекта.

параметры

параметр	подробности
BucketName	Название ведро AWS S3, над которым вы работаете.
CannedACLName	Имя встроенного (предварительно определенного) списка контроля доступа (ACL), который будет связан с ведром S3.
файл	Имя файла в локальной файловой системе, которое будет загружено в ведомость AWS S3.

Examples

Создайте новый ведро S3

```
New-S3Bucket -BucketName trevor
```

Имя ведра Simple Storage Service (S3) должно быть глобально уникальным. Это означает, что если кто-то еще использовал имя ведра, которое вы хотите использовать, вы должны принять решение о новом имени.

Загрузите локальный файл в ведро S3

```
Set-Content -Path myfile.txt -Value 'PowerShell Rocks'
Write-S3Object -BucketName powershell -File myfile.txt
```

Загрузка файлов из локальной файловой системы в AWS S3 осуществляется легко,

используя команду `Write-S3Object` . В самой базовой форме вам нужно только указать параметр `-BucketName` , чтобы указать, какой ведро S3 вы хотите загрузить в файл, и параметр `-File` , который указывает относительный или абсолютный путь к локальному файлу, который вы хотите загрузите в ведро S3.

Удалить ведро S3

```
Get-S3Object -BucketName powershell | Remove-S3Object -Force
Remove-S3Bucket -BucketName powershell -Force
```

Чтобы удалить ведро S3, вы должны сначала удалить все объекты S3, которые хранятся внутри ведра, при условии, что у вас есть разрешение на это. В приведенном выше примере мы извлекаем список всех объектов внутри ведра, а затем `Remove-S3Object` их в команду `Remove-S3Object` чтобы удалить их. Когда все объекты будут удалены, мы можем использовать команду `Remove-S3Bucket` для удаления ведра.

Прочитайте Простая служба хранения (S3) Amazon Web Services (AWS) онлайн: https://riptutorial.com/ru/powershell/topic/9579/простая-служба-хранения--s3--amazon-web-services--aws-

глава 54: Профили Powershell

замечания

Файл профиля - это сценарий powershell, который будет запускаться во время запуска консоли powershell. Таким образом, мы можем подготовить нашу среду к нам каждый раз, когда мы начинаем новую сессию PowerShell.

Типичные вещи, которые мы хотим сделать в начале PowerShell:

- часто используемые модули импорта (ActiveDirectory, Exchange, некоторая определенная DLL)
- протоколирование
- изменение приглашения
- диагностика

Существует несколько профильных файлов и мест, которые имеют разные виды использования, а также иерархию начального порядка:

хозяин	пользователь	Дорожка	Начать заказ	переменная
Все	Все	% WINDIR% \ System32 \ WindowsPowerShell \ v1.0 \ profile.ps1	1	$ profile.AllUsersAllHos
Все	Текущий	% USERPROFILE% \ Documents \ WindowsPowerShell \ profile.ps1	3	$ profile.CurrentUserAll
Приставка	Все	% WINDIR% \ System32 \ WindowsPowerShell \ v1.0 \ Microsoft.PowerShell_profile.ps1	2	$ profile.AllUsersCurren
Приставка	Текущий	% USERPROFILE% \ Documents \ WindowsPowerShell \ Microsoft.PowerShell_profile.ps1	4	$ profile.CurrentUserCurr
ISE	Все	% WINDIR% \ System32 \ WindowsPowerShell \ v1.0 \ Microsoft.PowerShellISE_profile.ps1	2	$ profile.AllUsersCurren
ISE	Текущий	% USERPROFILE% \ Documents \ WindowsPowerShell \ Microsoft.PowerShellISE_profile.ps1	4	$ profile.CurrentUserCurr

Examples

Создание базового профиля

Профиль PowerShell используется для автоматической загрузки пользовательских переменных и функций.

Профили PowerShell не создаются автоматически для пользователей.

Чтобы создать профиль PowerShell `C:>New-Item -ItemType File $profile` .

Если вы находитесь в ISE, вы можете использовать встроенный редактор `C:>psEdit $profile`

Легкий способ начать свой личный профиль для текущего хоста - сохранить некоторый текст в пути, хранящемся в `$profile` -variable

```
"#Current host, current user" > $profile
```

Дальнейшая модификация профиля может быть выполнена с использованием PowerShell ISE, блокнота, кода Visual Studio или любого другого редактора.

`$profile` Profile -variable возвращает текущий профиль пользователя для текущего хоста по умолчанию, но вы можете получить доступ к пути к политике машины (всем пользователям) и / или к профилю для всех хостов (консоль, ISE, сторонняя сторона), используя это свойства.

```
PS> $PROFILE | Format-List -Force

AllUsersAllHosts       : C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
AllUsersCurrentHost    :
C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts    : C:\Users\user\Documents\WindowsPowerShell\profile.ps1
CurrentUserCurrentHost :
C:\Users\user\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
Length                 : 75

PS> $PROFILE.AllUsersAllHosts
C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
```

Прочитайте Профили Powershell онлайн: https://riptutorial.com/ru/powershell/topic/5636/профили-powershell

глава 55: Псевдонимы

замечания

Система именования Powershell имеет довольно строгие правила назначения командлетов (шаблон Verb-Noun, см. [Тема еще не создана] для получения дополнительной информации). Но на самом деле не очень удобно писать `Get-ChildItems` каждый раз, когда вы хотите перечислить файлы в каталоге в интерактивном режиме.
Поэтому Powershell позволяет использовать ярлыки - псевдонимы - вместо имен командлетов.

Вы можете написать `ls` , `dir` или `gci` вместо `Get-ChildItem` и получить тот же результат. Псевдоним эквивалентен его командлету.

Некоторые общие псевдонимы:

псевдоним	командлет
%, для каждого	For-EachObject
?, где	Where-Object
cat, gc, type	Get-Content
cd, chdir, sl	Set-Location
cls, clear	Clear-Host
cp, copy, cpi	Copy-Item
реж / LS / GCI	Get-ChildItem
эхо, писать	Write-Output
Флорида	Format-List
фут	Format-Table
ФВ	Format-Wide
gc, pwd	Get-Location
грамм	Get-Member
IEX	Invoke-Expression

псевдоним	командлет
б	Invoke-Item
mv, переместить	Move-Item
rm, rmdir, del, erase, rd, ri	Удалить объект
спать	Start-Sleep
старт, соки	Запуск процесса

В приведенной выше таблице вы можете увидеть, как псевдонимы позволяли имитировать команды, известные из других сред (cmd, bash), следовательно, увеличивали возможности обнаружения.

Examples

Get-Alias

Чтобы перечислить все псевдонимы и их функции:

```
Get-Alias
```

Чтобы получить все псевдонимы для конкретного командлета:

```
PS C:\> get-alias -Definition Get-ChildItem

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           dir -> Get-ChildItem
Alias           gci -> Get-ChildItem
Alias           ls -> Get-ChildItem
```

Чтобы найти псевдонимы путем сопоставления:

```
PS C:\> get-alias -Name p*

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           popd -> Pop-Location
Alias           proc -> Get-Process
Alias           ps -> Get-Process
Alias           pushd -> Push-Location
Alias           pwd -> Get-Location
```

Set-Alias

Этот командлет позволяет создавать новые альтернативные имена для выхода из

командлетов

```
PS C:\> Set-Alias -Name proc -Value Get-Process
PS C:\> proc

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  SI ProcessName
-------  ------    -----      ----- -----   ------     --  -- -----------
    292      17    13052      20444 ...19     7.94    620   1 ApplicationFrameHost
....
```

Имейте в виду, что любой псевдоним, который вы создаете, будет сохраняться только в текущем сеансе. Когда вы начинаете новый сеанс, вам нужно снова создать свои псевдонимы. Профили Powershell (см. [Тема еще не создана]) отлично подходят для этих целей.

Прочитайте Псевдонимы онлайн: https://riptutorial.com/ru/powershell/topic/5287/псевдонимы

глава 56: Работа с конвейером PowerShell

Вступление

PowerShell представляет модель конвейеризации объекта, которая позволяет отправлять целые объекты вниз по конвейеру до потребления команд или (по крайней мере) вывода. В отличие от классической конвейерной обработки на основе строк, информация в конвейерных объектах не обязательно должна находиться на определенных позициях. Командлеты могут объявлять для взаимодействия с объектами из конвейера в качестве входных данных, а возвращаемые значения автоматически отправляются в конвейер.

Синтаксис

- BEGIN Первый блок. Выполняется один раз в начале. Ввод конвейера здесь равен $ null, поскольку он не был установлен.
- ПРОЦЕСС Второй блок. Выполняется для каждого элемента конвейера. Параметр конвейера равен текущему обрабатываемому элементу.
- END Последний блок. Выполняется один раз в конце. Параметр конвейера равен последнему элементу ввода, поскольку он не был изменен с момента его установки.

замечания

В большинстве случаев ввод конвейера будет представлять собой массив объектов. Хотя поведение блока `PROCESS{}` может показаться похожим на блок `foreach{}` , пропуская элемент в массиве, требуется другой процесс.

Если, как и в `foreach{}` , вы использовали `continue` внутри блока `PROCESS{}` , он сломал бы конвейер, пропустив все следующие утверждения, включая блок `END{}` . Вместо этого используйте `return` - он только завершит блок `PROCESS{}` для текущего элемента и переместится к следующему.

В некоторых случаях возникает необходимость выводить результат функций с различным кодированием. Кодирование вывода CmdLets управляется переменной `$OutputEncoding` . Когда вывод предназначен для ввода в конвейер для родных приложений, может быть хорошей идеей исправить кодировку в соответствии с целевым значением `$OutputEncoding = [Console]::OutputEncoding`

Дополнительные ссылки:

Статья в блоге с более `$OutputEncoding` описанием `$OutputEncoding`
https://blogs.msdn.microsoft.com/powershell/2006/12/11/outputencoding-to-the-rescue/

Examples

Запись функций с расширенным жизненным циклом

В этом примере показано, как функция может принимать конвейерные входные данные и эффективно выполнять итерацию.

Обратите внимание, что `begin` и `end` структуры функции являются необязательными при конвейерной обработке, но этот `process` требуется при использовании `ValueFromPipeline` или `ValueFromPipelineByPropertyName` .

```
function Write-FromPipeline{
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline)]
        $myInput
    )
    begin {
        Write-Verbose -Message "Beginning Write-FromPipeline"
    }
    process {
        Write-Output -InputObject $myInput
    }
    end {
        Write-Verbose -Message "Ending Write-FromPipeline"
    }
}

$foo = 'hello','world',1,2,3

$foo | Write-FromPipeline -Verbose
```

Выход:

```
VERBOSE: Beginning Write-FromPipeline
hello
world
1
2
3
VERBOSE: Ending Write-FromPipeline
```

Базовая поддержка трубопроводов в функциях

Это пример функции с простейшей возможной поддержкой конвейерной обработки. Любая функция с поддержкой конвейера должна иметь по крайней мере один параметр с параметром ParameterAttribute `ValueFromPipeline` или `ValueFromPipelineByPropertyName` , как показано ниже.

```
function Write-FromPipeline {
    param(
```

```
        [Parameter(ValueFromPipeline)]  # This sets the ParameterAttribute
        [String]$Input
    )
    Write-Host $Input
}

$foo = 'Hello World!'

$foo | Write-FromPipeline
```

Выход:

```
Hello World!
```

Примечание. В PowerShell 3.0 и выше поддерживаются значения по умолчанию для ParameterAttributes. В более ранних версиях вы должны указать `ValueFromPipeline=$true` .

Рабочая концепция трубопровода

В серии конвейеров каждая функция выполняется параллельно другим, подобно параллельным потокам. Первый обработанный объект передается на следующий конвейер, и следующая обработка немедленно выполняется в другом потоке. Это объясняет высокую скорость усиления по сравнению со стандартным `ForEach`

```
@( bigFile_1, bigFile_2, ..., bigFile_n) | Copy-File | Encrypt-File | Get-Md5
```

1. step - скопировать первый файл (в Thread `Copy-file`)
2. step - копировать второй файл (в Thread `Copy-file`) и одновременно шифровать первый (в `Encrypt-File`)
3. step - копировать третий файл (в `Copy-file` Thread) и одновременно шифровать второй файл (в `Encrypt-File`) и одновременно `get-Md5` первого (в `Get-Md5`)

Прочитайте Работа с конвейером PowerShell онлайн:
https://riptutorial.com/ru/powershell/topic/3937/работа-с-конвейером-powershell

глава 57: Работа с объектами

Examples

Обновление объектов

Добавление свойств

Если вы хотите добавить свойства к существующему объекту, вы можете использовать командлет Add-Member. С помощью PSObjects значения хранятся в виде «Свойства примечания»,

```
$object = New-Object -TypeName PSObject -Property @{
        Name = $env:username
        ID = 12
        Address = $null
    }

Add-Member -InputObject $object -Name "SomeNewProp" -Value "A value" -MemberType NoteProperty

# Returns
PS> $Object
Name ID Address SomeNewProp
---- -- ------- -----------
nem  12         A value
```

Вы также можете добавлять свойства с помощью Select-Object Cmdlet (так называемые рассчитанные свойства):

```
$newObject = $Object | Select-Object *, @{label='SomeOtherProp'; expression={'Another value'}}

# Returns
PS> $newObject
Name ID Address SomeNewProp SomeOtherProp
---- -- ------- ----------- -------------
nem  12         A value     Another value
```

Вышеуказанная команда может быть сокращена до следующего:

```
$newObject = $Object | Select *,@{l='SomeOtherProp';e={'Another value'}}
```

Удаление свойств

Командлет Select-Object можно использовать для удаления свойств объекта:

```
$object = $newObject | Select-Object * -ExcludeProperty ID, Address

# Returns
PS> $object
Name SomeNewProp SomeOtherProp
---- ----------- -------------
nem  A value     Another value
```

Создание нового объекта

PowerShell, в отличие от некоторых других языков сценариев, отправляет объекты по конвейеру. Это означает, что при отправке данных из одной команды в другую важно иметь возможность создавать, изменять и собирать объекты.

Создание объекта просто. Большинство объектов, которые вы создаете, будут настраиваемыми объектами в PowerShell, а типом для этого является PSObject. PowerShell также позволит вам создать любой объект, который вы могли бы создать в .NET.

Вот пример создания новых объектов с несколькими свойствами:

Вариант 1: Новый объект

```
$newObject = New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}

# Returns
PS> $newObject
Name ID Address
---- -- -------
nem  12
```

Вы можете сохранить объект в переменной, предварительно `$newObject =` команду с помощью `$newObject =`

Вам также может потребоваться хранить коллекции объектов. Это можно сделать, создав пустую коллекционную переменную и добавив объекты в коллекцию, например:

```
$newCollection = @()
$newCollection += New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}
```

Затем вы можете запрограммировать этот объект коллекции по объекту. Для этого найдите раздел Loop в документации.

Вариант 2: Select-Object

Менее распространенным способом создания объектов, которые вы все равно найдете в Интернете, является следующее:

```
$newObject = 'unuseddummy' | Select-Object -Property Name, ID, Address
$newObject.Name = $env:username
$newObject.ID = 12

# Returns
PS> $newObject
Name ID Address
---- -- -------
nem  12
```

Вариант 3: ускоритель типа pscustomobject (требуется PSv3 +)

Ускоритель упорядоченного типа заставляет PowerShell сохранять наши свойства в том порядке, в котором мы их определяли. Вам не нужен ускоритель упорядоченного типа для использования `[PSCustomObject]` :

```
$newObject = [PSCustomObject][Ordered]@{
    Name = $env:Username
    ID = 12
    Address = $null
}

# Returns
PS> $newObject
Name ID Address
---- -- -------
nem  12
```

Изучение объекта

Теперь, когда у вас есть объект, может быть полезно выяснить, что это такое. Командлет Get-Member можно использовать, чтобы узнать, что такое объект и что он содержит:

```
Get-Item c:\windows | Get-Member
```

Это дает:

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

Далее следует список свойств и методов, которыми обладает объект.

Другой способ получить тип объекта - использовать метод GetType, например:

```
C:\> $Object = Get-Item C:\Windows
C:\> $Object.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     DirectoryInfo                            System.IO.FileSystemInfo
```

Чтобы просмотреть список свойств, которые имеет объект, вместе с их значениями, вы можете использовать командлет Format-List с его параметром Property, установленным на: * (что означает все).

Вот пример, с результатом:

```
C:\> Get-Item C:\Windows | Format-List -Property *


PSPath            : Microsoft.PowerShell.Core\FileSystem::C:\Windows
PSParentPath      : Microsoft.PowerShell.Core\FileSystem::C:\
PSChildName       : Windows
PSDrive           : C
PSProvider        : Microsoft.PowerShell.Core\FileSystem
PSIsContainer     : True
Mode              : d-----
BaseName          : Windows
Target            : {}
LinkType          :
Name              : Windows
Parent            :
Exists            : True
Root              : C:\
FullName          : C:\Windows
Extension         :
CreationTime      : 30/10/2015 06:28:30
CreationTimeUtc   : 30/10/2015 06:28:30
LastAccessTime    : 16/08/2016 17:32:04
LastAccessTimeUtc : 16/08/2016 16:32:04
LastWriteTime     : 16/08/2016 17:32:04
LastWriteTimeUtc  : 16/08/2016 16:32:04
Attributes        : Directory
```

Создание экземпляров общих классов

Примечание: примеры, написанные для PowerShell 5.1. Вы можете создавать экземпляры общих классов

```
#Nullable System.DateTime
[Nullable[datetime]]$nullableDate = Get-Date -Year 2012
$nullableDate
$nullableDate.GetType().FullName
$nullableDate = $null
$nullableDate

#Normal System.DateTime
```

```
[datetime]$aDate = Get-Date -Year 2013
$aDate
$aDate.GetType().FullName
$aDate = $null #Throws exception when PowerShell attempts to convert null to
```

Выдает вывод:

```
Saturday, 4 August 2012 08:53:02
System.DateTime
Sunday, 4 August 2013 08:53:02
System.DateTime
Cannot convert null to type "System.DateTime".
At line:14 char:1
+ $aDate = $null
+ ~~~~~~~~~~~~~~
    + CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException
    + FullyQualifiedErrorId : RuntimeException
```

Также возможны общие коллекции

```
[System.Collections.Generic.SortedDictionary[int, String]]$dict =
[System.Collections.Generic.SortedDictionary[int, String]]::new()
$dict.GetType().FullName

$dict.Add(1, 'a')
$dict.Add(2, 'b')
$dict.Add(3, 'c')


$dict.Add('4', 'd') #powershell auto converts '4' to 4
$dict.Add('5.1', 'c') #powershell auto converts '5.1' to 5

$dict

$dict.Add('z', 'z') #powershell can't convert 'z' to System.Int32 so it throws an error
```

Выдает вывод:

```
System.Collections.Generic.SortedDictionary`2[[System.Int32, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089],[System.String, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089]]

Key Value
--- -----
  1 a
  2 b
  3 c
  4 d
  5 c
Cannot convert argument "key", with value: "z", for "Add" to type "System.Int32": "Cannot
convert value "z" to type "System.Int32". Error: "Input string was not in a correct format.""
At line:15 char:1
+ $dict.Add('z', 'z') #powershell can't convert 'z' to System.Int32 so  ...
+ ~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], MethodException
    + FullyQualifiedErrorId : MethodArgumentConversionInvalidCastArgument
```

Прочитайте Работа с объектами онлайн: https://riptutorial.com/ru/powershell/topic/1328/работа-с-объектами

глава 58: Работа с файлами XML

Examples

Доступ к файлу XML

```
<!-- file.xml -->
<people>
    <person id="101">
        <name>Jon Lajoie</name>
        <age>22</age>
    </person>
    <person id="102">
        <name>Lord Gaben</name>
        <age>65</age>
    </person>
    <person id="103">
        <name>Gordon Freeman</name>
        <age>29</age>
    </person>
</people>
```

Загрузка файла XML

Чтобы загрузить файл XML, вы можете использовать любой из них:

```
# First Method
$xdoc = New-Object System.Xml.XmlDocument
$file = Resolve-Path(".\file.xml")
$xdoc.load($file)

# Second Method
[xml] $xdoc = Get-Content ".\file.xml"

# Third Method
$xdoc = [xml] (Get-Content ".\file.xml")
```

Доступ к XML как объектам

```
PS C:\> $xml = [xml](Get-Content file.xml)
PS C:\> $xml

PS C:\> $xml.people

person
--------
{Jon Lajoie, Lord Gaben, Gordon Freeman}

PS C:\> $xml.people.person

id                                      name                                    age
```

```
--                                   ----                                 ---
101                                  Jon Lajoie                           22
102                                  Lord Gaben                           65
103                                  Gordon Freeman                       29

PS C:\> $xml.people.person[0].name
Jon Lajoie

PS C:\> $xml.people.person[1].age
65

PS C:\> $xml.people.person[2].id
103
```

Доступ к XML с помощью XPath

```
PS C:\> $xml = [xml](Get-Content file.xml)
PS C:\> $xml

PS C:\> $xml.SelectNodes("//people")

person
--------
{Jon Lajoie, Lord Gaben, Gordon Freeman}

PS C:\> $xml.SelectNodes("//people//person")

id                                   name                                 age
--                                   ----                                 ---
101                                  Jon Lajoie                           22
102                                  Lord Gaben                           65
103                                  Gordon Freeman                       29

PS C:\> $xml.SelectSingleNode("people//person[1]//name")
Jon Lajoie

PS C:\> $xml.SelectSingleNode("people//person[2]//age")
65

PS C:\> $xml.SelectSingleNode("people//person[3]//@id")
103
```

Доступ к XML, содержащему пространства имен с XPath

```
PS C:\> [xml]$xml = @"
<ns:people xmlns:ns="http://schemas.xmlsoap.org/soap/envelope/">
    <ns:person id="101">
        <ns:name>Jon Lajoie</ns:name>
    </ns:person>
    <ns:person id="102">
        <ns:name>Lord Gaben</ns:name>
    </ns:person>
    <ns:person id="103">
        <ns:name>Gordon Freeman</ns:name>
    </ns:person>
</ns:people>
"@
```

```
PS C:\> $ns = new-object Xml.XmlNamespaceManager $xml.NameTable
PS C:\> $ns.AddNamespace("ns", $xml.DocumentElement.NamespaceURI)
PS C:\> $xml.SelectNodes("//ns:people/ns:person", $ns)

id                                      name
--                                      ----
101                                     Jon Lajoie
102                                     Lord Gaben
103                                     Gordon Freeman
```

Создание XML-документа с помощью XmlWriter ()

```
# Set The Formatting
$xmlsettings = New-Object System.Xml.XmlWriterSettings
$xmlsettings.Indent = $true
$xmlsettings.IndentChars = "    "

# Set the File Name Create The Document
$XmlWriter = [System.XML.XmlWriter]::Create("C:\YourXML.xml", $xmlsettings)

# Write the XML Decleration and set the XSL
$xmlWriter.WriteStartDocument()
$xmlWriter.WriteProcessingInstruction("xml-stylesheet", "type='text/xsl' href='style.xsl'")

# Start the Root Element
$xmlWriter.WriteStartElement("Root")

    $xmlWriter.WriteStartElement("Object") # <-- Start <Object>

        $xmlWriter.WriteElementString("Property1","Value 1")
        $xmlWriter.WriteElementString("Property2","Value 2")

        $xmlWriter.WriteStartElement("SubObject") # <-- Start <SubObject>
            $xmlWriter.WriteElementString("Property3","Value 3")
        $xmlWriter.WriteEndElement() # <-- End <SubObject>

    $xmlWriter.WriteEndElement() # <-- End <Object>

$xmlWriter.WriteEndElement() # <-- End <Root>

# End, Finalize and close the XML Document
$xmlWriter.WriteEndDocument()
$xmlWriter.Flush()
$xmlWriter.Close()
```

Выходной файл XML

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type='text/xsl' href='style.xsl'?>
<Root>
    <Object>
        <Property1>Value 1</Property1>
        <Property2>Value 2</Property2>
        <SubObject>
            <Property3>Value 3</Property3>
        </SubObject>
    </Object>
</Root>
```

Пример данных

Документ XML

Сначала давайте определим образец XML-документа с именем « **books.xml** » в нашем текущем каталоге:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book>
        <title>Of Mice And Men</title>
        <author>John Steinbeck</author>
        <pageCount>187</pageCount>
        <publishers>
            <publisher>
                <isbn>978-88-58702-15-4</isbn>
                <name>Pascal Covici</name>
                <year>1937</year>
                <binding>Hardcover</binding>
                <first>true</first>
            </publisher>
            <publisher>
                <isbn>978-05-82461-46-8</isbn>
                <name>Longman</name>
                <year>2009</year>
                <binding>Hardcover</binding>
            </publisher>
        </publishers>
        <characters>
            <character name="Lennie Small" />
            <character name="Curley's Wife" />
            <character name="George Milton" />
            <character name="Curley" />
        </characters>
        <film>True</film>
    </book>
    <book>
        <title>The Hunt for Red October</title>
        <author>Tom Clancy</author>
        <pageCount>387</pageCount>
        <publishers>
            <publisher>
                <isbn>978-08-70212-85-7</isbn>
                <name>Naval Institute Press</name>
                <year>1984</year>
                <binding>Hardcover</binding>
                <first>true</first>
            </publisher>
            <publisher>
                <isbn>978-04-25083-83-3</isbn>
                <name>Berkley</name>
                <year>1986</year>
                <binding>Paperback</binding>
```

```
            </publisher>
            <publisher>
                <isbn>978-08-08587-35-4</isbn>
                <name>Penguin Putnam</name>
                <year>2010</year>
                <binding>Paperback</binding>
            </publisher>
        </publishers>
        <characters>
            <character name="Marko Alexadrovich Ramius" />
            <character name="Jack Ryan" />
            <character name="Admiral Greer" />
            <character name="Bart Mancuso" />
            <character name="Vasily Borodin" />
        </characters>
        <film>True</film>
    </book>
</books>
```

Новые данные

Что мы хотим сделать, так это добавить несколько новых книг в этот документ, скажем, «*Игры патриотов* » Тома Клэнси (да, я поклонник работ Клэнси ^ __ ^) и любителя Sci-Fi: *«Путеводитель автостопа по галактике»* Дугласом Адамсом в основном потому, что Зафоду Библброксу просто интересно читать.

Так или иначе мы приобрели данные для новых книг и сохранили их как список PSCustomObjects:

```
$newBooks = @(
    [PSCustomObject] @{
        "Title" = "Patriot Games";
        "Author" = "Tom Clancy";
        "PageCount" = 540;
        "Publishers" = @(
            [PSCustomObject] @{
                "ISBN" = "978-0-39-913241-4";
                "Year" = "1987";
                "First" = $True;
                "Name" = "Putnam";
                "Binding" = "Hardcover";
            }
        );
        "Characters" = @(
            "Jack Ryan", "Prince of Wales", "Princess of Wales",
            "Robby Jackson", "Cathy Ryan", "Sean Patrick Miller"
        );
        "film" = $True;
    },
    [PSCustomObject] @{
        "Title" = "The Hitchhiker's Guide to the Galaxy";
        "Author" = "Douglas Adams";
        "PageCount" = 216;
        "Publishers" = @(
            [PSCustomObject] @{
                "ISBN" = "978-0-33-025864-7";
```

```
                "Year" = "1979";
                "First" = $True;
                "Name" = "Pan Books";
                "Binding" = "Hardcover";
            }
        );
        "Characters" = @(
            "Arthur Dent", "Marvin", "Zaphod Beeblebrox", "Ford Prefect",
            "Trillian", "Slartibartfast", "Dirk Gently"
        );
        "film" = $True;
    }
);
```

Шаблоны

Теперь нам нужно определить несколько структурных элементов скелета для наших новых данных. В принципе, вы хотите создать скелет / шаблон для каждого списка данных. В нашем примере это означает, что нам нужен шаблон для книги, персонажей и издателей. Мы также можем использовать это для определения нескольких значений по умолчанию, таких как значение для тега `film` .

```
$t_book = [xml] @'
<book>
    <title />
    <author />
    <pageCount />
    <publishers />
    <characters />
    <film>False</film>
</book>
'@;

$t_publisher = [xml] @'
<publisher>
    <isbn/>
    <name/>
    <year/>
    <binding/>
    <first>false</first>
</publisher>
'@;

$t_character = [xml] @'
<character name="" />
'@;
```

Мы закончили настройку.

Добавление новых данных

Теперь, когда мы настроены вместе с нашими примерными данными, добавим

пользовательские объекты в объект документа XML.

```
# Read the xml document
$xml = [xml] Get-Content .\books.xml;

# Let's show a list of titles to see what we've got currently:
$xml.books.book | Select Title, Author, @{N="ISBN";E={If ( $_.Publishers.Publisher.Count ) {
$_.Publishers.publisher[0].ISBN} Else { $_.Publishers.publisher.isbn}}};;

# Outputs:
# title                              author         ISBN
# -----                              ------         ----
# Of Mice And Men                    John Steinbeck 978-88-58702-15-4
# The Hunt for Red October           Tom Clancy     978-08-70212-85-7

# Let's show our new books as well:
$newBooks | Select Title, Author, @{N="ISBN";E={$_.Publishers[0].ISBN}};

# Outputs:
# Title                              Author        ISBN
# -----                              ------        ----
# Patriot Games                      Tom Clancy    978-0-39-913241-4
# The Hitchhiker's Guide to the Galaxy Douglas Adams 978-0-33-025864-7

# Now to merge the two:

ForEach ( $book in $newBooks ) {
    $root = $xml.SelectSingleNode("/books");

    # Add the template for a book as a new node to the root element
    [void]$root.AppendChild($xml.ImportNode($t_book.book, $true));

    # Select the new child element
    $newElement = $root.SelectSingleNode("book[last()]");

    # Update the parameters of that new element to match our current new book data
    $newElement.title     = [String]$book.Title;
    $newElement.author    = [String]$book.Author;
    $newElement.pageCount = [String]$book.PageCount;
    $newElement.film      = [String]$book.Film;

    # Iterate through the properties that are Children of our new Element:
    ForEach ( $publisher in $book.Publishers ) {
        # Create the new child publisher element
        # Note the use of "SelectSingleNode" here, this allows the use of the "AppendChild"
method as it returns
        # a XmlElement type object instead of the $Null data that is currently stored in that
leaf of the
        # XML document tree

[void]$newElement.SelectSingleNode("publishers").AppendChild($xml.ImportNode($t_publisher.publisher,
$true));

        # Update the attribute and text values of our new XML Element to match our new data
        $newPublisherElement = $newElement.SelectSingleNode("publishers/publisher[last()]");
        $newPublisherElement.year = [String]$publisher.Year;
        $newPublisherElement.name = [String]$publisher.Name;
        $newPublisherElement.binding = [String]$publisher.Binding;
        $newPublisherElement.isbn = [String]$publisher.ISBN;
        If ( $publisher.first ) {
            $newPublisherElement.first = "True";
```

```
        }
    }

    ForEach ( $character in $book.Characters ) {
        # Select the characters xml element
        $charactersElement = $newElement.SelectSingleNode("characters");

        # Add a new character child element
        [void]$charactersElement.AppendChild($xml.ImportNode($t_character.character, $true));

        # Select the new characters/character element
        $characterElement = $charactersElement.SelectSingleNode("character[last()]");

        # Update the attribute and text values to match our new data
        $characterElement.name = [String]$character;
    }
}

# Check out the new XML:
$xml.books.book | Select Title, Author, @{N="ISBN";E={If ( $_.Publishers.Publisher.Count ) {
$_.Publishers.publisher[0].ISBN} Else { $_.Publishers.publisher.isbn}}};

# Outputs:
# title                              author         ISBN
# -----                              ------         ----
# Of Mice And Men                    John Steinbeck 978-88-58702-15-4
# The Hunt for Red October           Tom Clancy     978-08-70212-85-7
# Patriot Games                      Tom Clancy     978-0-39-913241-4
# The Hitchhiker's Guide to the Galaxy Douglas Adams  978-0-33-025864-7
```

Теперь мы можем написать наш XML на диск, или экран, или в Интернете, или где угодно!

прибыль

Хотя это может и не быть процедурой для всех, кого я нашел, чтобы помочь избежать целой группы

`[void]$xml.SelectSingleNode("/complicated/xpath/goes[here]").AppendChild($xml.CreateElement("newElementN`

за которым следует `$xml.SelectSingleNode("/complicated/xpath/goes/here/newElementName") = $textValue`

Я думаю, что метод, описанный в этом примере, более чист и легче разбирается для нормальных людей.

улучшения

Возможно, можно изменить шаблон, чтобы включить элементы с дочерними элементами, а не разбивать каждый раздел как отдельный шаблон. Вам просто нужно позаботиться о том, чтобы клонировать предыдущий элемент, когда вы просматриваете список.

Прочитайте Работа с файлами XML онлайн: https://riptutorial.com/ru/powershell/topic/4882/

работа-с-файлами-xml

глава 59: Рабочие процессы PowerShell

Вступление

PowerShell Workflow - это функция, которая была введена с версии PowerShell версии 3.0. Определения рабочего процесса очень похожи на определения функций PowerShell, однако они выполняются в среде Windows Workflow Foundation, а не непосредственно в движке PowerShell.

Несколько уникальных функций «вне коробки» включены в механизм Workflow, в первую очередь, настойчивость работы.

замечания

Функция PowerShell Workflow поддерживается исключительно на платформе Microsoft Windows под PowerShell Desktop Edition. PowerShell Core Edition, поддерживаемый в Linux, Mac и Windows, не поддерживает функцию PowerShell Workflow.

При создании рабочего процесса PowerShell имейте в виду, что рабочие процессы вызывают действия, а не командлеты. Вы все равно можете вызывать командлеты из рабочего процесса PowerShell, но Workflow Engine неявно `InlineScript` вызов cmdlet в `InlineScript` . Вы также можете явно `InlineScript` код внутри действия `InlineScript` , который выполняет код PowerShell; по умолчанию действие `InlineScript` выполняется в отдельном процессе и возвращает результат в вызывающий Workflow.

Examples

Пример простого рабочего процесса

```
workflow DoSomeWork {
  Get-Process -Name notepad | Stop-Process
}
```

Это базовый пример определения рабочего процесса PowerShell.

Рабочий процесс с входными параметрами

Подобно функциям PowerShell, рабочие процессы могут принимать входной параметр. Параметры ввода могут быть необязательно привязаны к определенному типу данных, таким как строка, целое число и т. Д. Используйте ключевое слово стандартного `param` для определения блока входных параметров непосредственно после объявления рабочего процесса.

```
workflow DoSomeWork {
  param (
    [string[]] $ComputerName
  )
  Get-Process -ComputerName $ComputerName
}

DoSomeWork -ComputerName server01, server02, server03
```

Запуск рабочего процесса в качестве фонового задания

Рабочие процессы PowerShell по своей сути оснащены возможностью запуска в фоновом режиме. Чтобы вызвать рабочий процесс в качестве фонового задания PowerShell, при вызове рабочего процесса используйте параметр `-AsJob` .

```
workflow DoSomeWork {
  Get-Process -ComputerName server01
  Get-Process -ComputerName server02
  Get-Process -ComputerName server03
}

DoSomeWork -AsJob
```

Добавление параллельного блока в рабочий процесс

```
workflow DoSomeWork {
  parallel {
    Get-Process -ComputerName server01
    Get-Process -ComputerName server02
    Get-Process -ComputerName server03
  }
}
```

Одной из уникальных возможностей PowerShell Workflow является возможность определять блок действий как параллельный. Чтобы использовать эту функцию, используйте ключевое слово `parallel` внутри вашего Workflow.

Вызов параллельных действий рабочего процесса может помочь повысить производительность вашего рабочего процесса.

Прочитайте Рабочие процессы PowerShell онлайн:
https://riptutorial.com/ru/powershell/topic/8745/рабочие-процессы-powershell

глава 60: Разбор CSV

Examples

Основное использование Import-Csv

Учитывая следующий CSV-файл

```
String,DateTime,Integer
First,2016-12-01T12:00:00,30
Second,2015-12-01T12:00:00,20
Third,2015-12-01T12:00:00,20
```

Можно импортировать CSV-строки в объекты PowerShell с помощью команды `Import-Csv`

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows

String DateTime            Integer
------ --------            -------
First  2016-12-01T12:00:00 30
Second 2015-11-03T13:00:00 20
Third  2015-12-05T14:00:00 20

> Write-Host $row[0].String1
Third
```

Импорт из CSV и свойств литья для исправления типа

По умолчанию `Import-CSV` импортирует все значения в виде строк, поэтому для получения объектов DateTime- и integer-объектов нам нужно их отбирать или анализировать.

Использование `Foreach-Object` :

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows | ForEach-Object {
    #Cast properties
    $_.DateTime = [datetime]$_.DateTime
    $_.Integer = [int]$_.Integer

    #Output object
    $_
}
```

Использование рассчитанных свойств:

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows | Select-Object String,
    @{name="DateTime";expression={ [datetime]$_.DateTime }},
    @{name="Integer";expression={ [int]$_.Integer }}
```

Выход:

```
String DateTime            Integer
------ --------            -------
First  01.12.2016 12:00:00      30
Second 03.11.2015 13:00:00      20
Third  05.12.2015 14:00:00      20
```

Прочитайте Разбор CSV онлайн: https://riptutorial.com/ru/powershell/topic/5691/разбор-csv

глава 61: Рассеивание Powershell

замечания

- about_Remote
- about_RemoteFAQ
- about_RemoteTroubleshooting

Examples

Включение PowerShell Remoting

Перенаправление PowerShell необходимо сначала включить на сервере, к которому вы хотите удаленно подключиться.

```
Enable-PSRemoting -Force
```

Эта команда выполняет следующие действия:

- Выполняет командлет Set-WSManQuickConfig, который выполняет следующие задачи:
- Запуск службы WinRM.
- Устанавливает тип запуска в службе WinRM для Automatic.
- Создает прослушиватель для приема запросов на любой IP-адрес, если он еще не существует.
- Включает исключение брандмауэра для связи WS-Management.
- Регистрирует конфигурации сеанса Microsoft.PowerShell и Microsoft.PowerShell.Workflow, если они еще не зарегистрированы.
- Регистрирует конфигурацию сеанса Microsoft.PowerShell32 на 64-разрядных компьютерах, если она еще не зарегистрирована.
- Включает все конфигурации сеанса.
- Изменяет дескриптор безопасности всех конфигураций сеанса, чтобы разрешить удаленный доступ.
- Перезагружает службу WinRM, чтобы сделать предыдущие изменения эффективными.

Только для не-доменных сред

Для серверов в домене AD аутентификация удаленного доступа PS выполняется через Kerberos («Default») или NTLM («Negotiate»). Если вы хотите разрешить удаленный доступ к серверу, отличному от домена, у вас есть два варианта.

Либо настройте WSMan-связь через HTTPS (для чего требуется генерация сертификата), либо включите базовую аутентификацию, которая отправляет ваши учетные данные через проводную base64-кодировку (это в основном то же самое, что и обычный текст, поэтому будьте осторожны с этим).

В любом случае вам придется добавить удаленные системы в список надежных хостов WSM.

Включение базовой проверки подлинности

```
Set-Item WSMan:\localhost\Service\AllowUnencrypted $true
```

Затем на компьютере вы хотите подключиться, вы обязательно должны сказать ему доверять компьютеру вы подключая *к.*

```
Set-Item WSMan:\localhost\Client\TrustedHosts '192.168.1.1,192.168.1.2'
```

```
Set-Item WSMan:\localhost\Client\TrustedHosts *.contoso.com
```

```
Set-Item WSMan:\localhost\Client\TrustedHosts *
```

Важно : вы должны сообщить своему клиенту, чтобы он доверял компьютеру, адресованному так, как вы хотите подключиться (например, если вы подключаетесь через IP-адрес, он должен доверять IP-адресу, а не имени хоста)

Подключение к удаленному серверу через PowerShell

Использование учетных данных с вашего локального компьютера:

```
Enter-PSSession 192.168.1.1
```

Запрос учетных данных на удаленном компьютере

```
Enter-PSSession 192.168.1.1 -Credential $(Get-Credential)
```

Выполнять команды на удаленном компьютере

После включения удаленного доступа Powershell (Enable-PSRemoting) Вы можете запускать команды на удаленном компьютере следующим образом:

```
Invoke-Command -ComputerName "RemoteComputerName" -ScriptBlock {
    Write host "Remote Computer Name: $ENV:ComputerName"
}
```

Вышеупомянутый метод создает временный сеанс и закрывает его сразу после завершения

команды или скриптового блока.

Чтобы оставить сеанс открытым и запустить в нем другую команду позже, сначала необходимо создать удаленный сеанс:

```
$Session = New-PSSession -ComputerName "RemoteComputerName"
```

Затем вы можете использовать этот сеанс каждый раз, когда вы вызываете команды на удаленном компьютере:

```
Invoke-Command -Session $Session -ScriptBlock {
    Write host "Remote Computer Name: $ENV:ComputerName"
}

Invoke-Command -Session $Session -ScriptBlock {
    Get-Date
}
```

Если вам нужно использовать разные учетные данные, вы можете добавить их с `-Credential` параметра `-Credential` Parameter:

```
$Cred = Get-Credential
Invoke-Command -Session $Session -Credential $Cred -ScriptBlock {...}
```

Удаление предупреждения о сериализации

> Замечания:
>
> Важно знать, что удаленная передача сериализует объекты PowerShell в удаленной системе и десериализует их в конце сеанса удаленного доступа, то есть они преобразуются в XML во время транспортировки и теряют все их методы.

```
$output = Invoke-Command -Session $Session -ScriptBlock {
    Get-WmiObject -Class win32_printer
}

$output | Get-Member -MemberType Method

   TypeName: Deserialized.System.Management.ManagementObject#root\cimv2\Win32_Printer

Name     MemberType Definition
----     ---------- ----------
GetType  Method     type GetType()
ToString Method     string ToString(), string ToString(string format, System.IFormatProvi...
```

Если у вас есть методы на обычном объекте PS:

```
Get-WmiObject -Class win32_printer | Get-Member -MemberType Method

 TypeName: System.Management.ManagementObject#root\cimv2\Win32_Printer

Name                  MemberType Definition

----                  ---------- ----------

CancelAllJobs         Method     System.Management.ManagementBaseObject CancelAllJobs()

GetSecurityDescriptor Method     System.Management.ManagementBaseObject
GetSecurityDescriptor()
Pause                 Method     System.Management.ManagementBaseObject Pause()

PrintTestPage         Method     System.Management.ManagementBaseObject PrintTestPage()

RenamePrinter         Method     System.Management.ManagementBaseObject
RenamePrinter(System.String NewPrinterName)
Reset                 Method     System.Management.ManagementBaseObject Reset()

Resume                Method     System.Management.ManagementBaseObject Resume()

SetDefaultPrinter     Method     System.Management.ManagementBaseObject SetDefaultPrinter()

SetPowerState         Method     System.Management.ManagementBaseObject
SetPowerState(System.UInt16 PowerState, System.String Time)
SetSecurityDescriptor Method     System.Management.ManagementBaseObject
SetSecurityDescriptor(System.Management.ManagementObject#Win32_SecurityDescriptor Descriptor)
```

Использование аргумента

Чтобы использовать аргументы в качестве параметров для удаленного сценария, можно использовать параметр `ArgumentList` для `Invoke-Command` или использовать синтаксис `$Using`:

Использование `ArgumentList` с неназванными параметрами (то есть в том порядке, в котором они передаются скриптблоку):

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Write-Host "Calling script block remotely with $($Args.Count)"
    Get-Service -Name $args[0]
    Remove-Item -Path $args[1] -ErrorAction SilentlyContinue -Force
}
```

Использование `ArgumentList` с именованными параметрами:

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Param($serviceToShowInRemoteSession,$fileToDelete)
```

```
    Write-Host "Calling script block remotely with $($Args.Count)"
    Get-Service -Name $serviceToShowInRemoteSession
    Remove-Item -Path $fileToDelete -ErrorAction SilentlyContinue -Force
}
```

Использование `$Using:` syntax:

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ScriptBlock {
    Get-Service $Using:servicesToShow
    Remove-Item -Path $fileName -ErrorAction SilentlyContinue -Force
}
```

Лучшая практика для автоматической очистки PSSessions

Когда удаленный сеанс создается с помощью командлета `New-PSsession` , PSSession сохраняется до окончания текущего сеанса PowerShell. Это означает, что по умолчанию `PSSession` и все связанные ресурсы будут по-прежнему использоваться до окончания текущего сеанса PowerShell.

Несколько активных `PSSessions` могут стать нагрузкой на ресурсы, особенно для длинных или взаимосвязанных сценариев, которые создают сотни `PSSessions` в одном сеансе PowerShell.

Лучше всего явно удалить каждую `PSSession` после ее окончания. [1]

Следующий шаблон кода использует `try-catch-finally` для достижения вышеуказанного, объединяя обработку ошибок с надежным способом, чтобы все созданные `PSSessions` были удалены, когда они были использованы:

```
try
{
    $session = New-PSsession -Computername "RemoteMachineName"
    Invoke-Command -Session $session -ScriptBlock {write-host "This is running on
$ENV:ComputerName"}
}
catch
{
    Write-Output "ERROR: $_"
}
finally
{
    if ($session)
    {
        Remove-PSSession $session
    }
}
```

Ссылки: [1] https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/new-pssession

Прочитайте Рассеивание Powershell онлайн: https://riptutorial.com/ru/powershell/topic/3087/рассеивание-powershell

глава 62: Расчетные свойства

Вступление

Вычисляемые свойства в Powershell являются собственными производными (расчетными) свойствами. Он позволяет пользователю форматировать определенное свойство так, как он хочет. Вычисление (выражение) может быть вполне возможно.

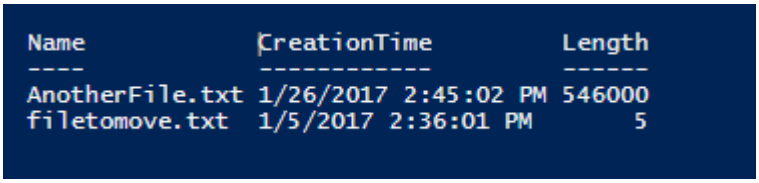
Examples

Отобразить размер файла в KB - Расчетные свойства

Давайте рассмотрим нижеприведенный фрагмент,

```
Get-ChildItem -Path C:\MyFolder | Select-Object Name, CreationTime, Length
```

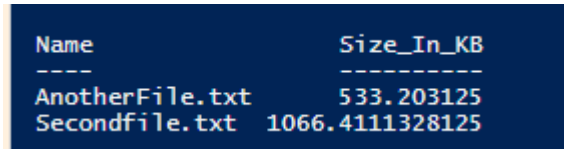
Он просто выводит содержимое папки с выбранными свойствами. Что-то вроде,



Что делать, если я хочу отобразить размер файла в КБ? В этом случае полезны исчисляемые свойства.

```
Get-ChildItem C:\MyFolder | Select-Object Name, @{Name="Size_In_KB";Expression={$_.Length /
1Kb}}
```

Что производит,



`Expression` - это то, что содержит расчет для рассчитанного свойства. И да, это может быть что угодно!

Прочитайте Расчетные свойства онлайн: https://riptutorial.com/ru/powershell/topic/8913/расчетные-свойства

глава 63: Регулярные выражения

Синтаксис

- 'text' -match 'RegExPattern'
- 'text' -replace 'RegExPattern', 'newvalue'
- [regex] :: Матч («текст», «шаблон») # Совместный матч
- [regex] :: Матчи ("текст", "шаблон") # Количество совпадений
- [Регулярное выражение] :: Заменить ("текст", "образец", "NewValue")
- [regex] :: Replace («текст», «шаблон», {param ($ m)}) #MatchEvaluator
- [regex] :: Escape ("input") # Специальные символы Escape

Examples

Единый матч

Вы можете быстро определить, содержит ли текст определенный шаблон с использованием Regex. Существует несколько способов работы с Regex в PowerShell.

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Content wrapped in ()
$pattern = '\(.*?\)'
```

Использование оператора -Match

Чтобы определить, соответствует ли строка шаблону с помощью встроенного `-matches` оператора, используйте синтаксис `'input' -match 'pattern'` . Это вернет `true` или `false` зависимости от результата поиска. Если было совпадение, вы можете просмотреть совпадение и группы (если они определены в шаблоне), обратившись к `$Matches` -variable.

```
> $text -match $pattern
True

> $Matches

Name Value
---- -----
0    (a)
```

Вы также можете использовать `-match` для фильтрации через массив строк и только возвращать строки, содержащие совпадение.

```
> $textarray = @"
This is (a) sample
text, this is
a (sample text)
"@ -split "`n"

> $textarray -match $pattern
This is (a) sample
a (sample text)
```

2,0

Использование Select-String

PowerShell 2.0 представил новый командлет для поиска по тексту с помощью регулярного выражения. Он возвращает объект `MatchInfo` каждого текстового ввода, который содержит совпадение. Вы можете получить доступ к его свойствам, чтобы найти соответствующие группы и т. Д.

```
> $m = Select-String -InputObject $text -Pattern $pattern

> $m

This is (a) sample
text, this is
a (sample text)

> $m | Format-List *

IgnoreCase : True
LineNumber : 1
Line       : This is (a) sample
             text, this is
             a (sample text)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(a)}
```

Подобно `-match` , `Select-String` также может использоваться для фильтрации через массив строк путем соединения массива с ним. Он создает `MatchInfo` -объект для каждой строки, которая содержит совпадение.

```
> $textarray | Select-String -Pattern $pattern

This is (a) sample
a (sample text)
```

```
#You can also access the matches, groups etc.
> $textarray | Select-String -Pattern $pattern | fl *


IgnoreCase : True
LineNumber : 1
Line       : This is (a) sample
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(a)}

IgnoreCase : True
LineNumber : 3
Line       : a (sample text)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(sample text)}
```

`Select-String` также может выполнять поиск с использованием обычного текстового шаблона (без регулярного выражения), добавляя переключатель `-SimpleMatch` .

Использование [RegEx] :: Match ()

Вы также можете использовать метод static `Match()` доступный в классе .NET `[RegEx]` .

```
> [regex]::Match($text,$pattern)

Groups   : {(a)}
Success  : True
Captures : {(a)}
Index    : 8
Length   : 3
Value    : (a)

> [regex]::Match($text,$pattern) | Select-Object -ExpandProperty Value
(a)
```

замещать

Общей задачей для регулярного выражения является замена текста, соответствующего шаблону с новым значением.

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@
```

```
#Sample pattern: Text wrapped in ()
$pattern = '\(.*?\)'

#Replace matches with:
$newvalue = 'test'
```

Использование оператора -Replace

Оператор `-replace` в PowerShell может использоваться для замены текста, соответствующего шаблону, новым значением с использованием синтаксиса `'input' -replace 'pattern', 'newvalue'` .

```
> $text -replace $pattern, $newvalue
This is test sample
text, this is
a test
```

Использование метода [RegEx] :: Replace ()

Замена совпадений также может быть выполнена с помощью метода `Replace()` в классе `[RegEx]` .NET.

```
[regex]::Replace($text, $pattern, 'test')
This is test sample
text, this is
a test
```

Заменить текст динамическим значением с помощью MatchEvalutor

Иногда вам нужно заменить значение, соответствующее шаблону, новым значением, основанным на этом конкретном совпадении, что делает невозможным предсказать новое значение. Для этих типов сценариев `MatchEvaluator` может быть очень полезен.

В PowerShell `MatchEvaluator` так же прост, как скриптовый блок с одним параметром, который содержит объект `Match` для текущего совпадения. Результатом действия будет новое значение для этого конкретного соответствия. `MatchEvalutor` может использоваться с статическим методом `[Regex]::Replace()` .

Пример : Замена текста внутри `()` его длиной

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@
```

```
#Sample pattern: Content wrapped in ()
$pattern = '(?<=\().*?(?=\))'

$MatchEvalutor = {
    param($match)

    #Replace content with length of content
    $match.Value.Length

}
```

Выход:

```
> [regex]::Replace($text, $pattern, $MatchEvalutor)

This is 1 sample
text, this is
a 11
```

Пример: Сделайте `sample` верхнем регистре

```
#Sample pattern: "Sample"
$pattern = 'sample'

$MatchEvalutor = {
    param($match)

    #Return match in upper-case
    $match.Value.ToUpper()

}
```

Выход:

```
> [regex]::Replace($text, $pattern, $MatchEvalutor)

This is (a) SAMPLE
text, this is
a (SAMPLE text)
```

Escape специальные символы

Регулярный шаблон использует множество специальных символов для описания шаблона. Пример `.` означает «любой символ», `+` «один или несколько» и т. д.

Чтобы использовать эти символы, выполните следующие действия `.` , `+` и т. д., в шаблоне вам нужно избегать их, чтобы удалить их особое значение. Это делается с помощью escape-символа, который является обратным слэшем `\` в регулярном выражении. Пример. Чтобы выполнить поиск `+` , вы должны использовать шаблон `\+` .

Трудно запомнить все специальные символы в регулярном выражении, поэтому, чтобы

избежать каждого специального символа в строке, которую вы хотите найти, вы можете использовать метод `[RegEx]::Escape("input")` .

```
> [regex]::Escape("(foo)")
\(foo\)

> [regex]::Escape("1+1.2=2.2")
1\+1\.2=2\.2
```

Несколько совпадений

Существует несколько способов найти все совпадения для шаблона в тексте.

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Content wrapped in ()
$pattern = '\(.*?\)'
```

Использование Select-String

Вы можете найти все совпадения (глобальное соответствие), добавив переключатель `-AllMatches` **в** `Select-String` .

```
> $m = Select-String -InputObject $text -Pattern $pattern -AllMatches

> $m | Format-List *

IgnoreCase : True
LineNumber : 1
Line       : This is (a) sample
             text, this is
             a (sample text)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(a), (sample text)}

#List all matches
> $m.Matches

Groups   : {(a)}
Success  : True
Captures : {(a)}
Index    : 8
Length   : 3
Value    : (a)
```

```
Groups   : {(sample text)}
Success  : True
Captures : {(sample text)}
Index    : 37
Length   : 13
Value    : (sample text)

#Get matched text
> $m.Matches | Select-Object -ExpandProperty Value
(a)
(sample text)
```

Использование [RegEx] :: Матчи ()

Метод `Matches()` в .NET `[regex] -class также может использоваться для глобального поиска нескольких совпадений.

```
> [regex]::Matches($text,$pattern)

Groups   : {(a)}
Success  : True
Captures : {(a)}
Index    : 8
Length   : 3
Value    : (a)

Groups   : {(sample text)}
Success  : True
Captures : {(sample text)}
Index    : 37
Length   : 13
Value    : (sample text)

> [regex]::Matches($text,$pattern) | Select-Object -ExpandProperty Value

(a)
(sample text)
```

Прочитайте Регулярные выражения онлайн: https://riptutorial.com/ru/powershell/topic/6674/регулярные-выражения

глава 64: Связь TCP с PowerShell

Examples

Прослушиватель TCP

```
Function Receive-TCPMessage {
    Param (
        [Parameter(Mandatory=$true, Position=0)]
        [ValidateNotNullOrEmpty()]
        [int] $Port
    )
    Process {
        Try {
            # Set up endpoint and start listening
            $endpoint = new-object System.Net.IPEndPoint([ipaddress]::any,$port)
            $listener = new-object System.Net.Sockets.TcpListener $EndPoint
            $listener.start()

            # Wait for an incoming connection
            $data = $listener.AcceptTcpClient()

            # Stream setup
            $stream = $data.GetStream()
            $bytes = New-Object System.Byte[] 1024

            # Read data from stream and write it to host
            while (($i = $stream.Read($bytes,0,$bytes.Length)) -ne 0){
                $EncodedText = New-Object System.Text.ASCIIEncoding
                $data = $EncodedText.GetString($bytes,0, $i)
                Write-Output $data
            }

            # Close TCP connection and stop listening
            $stream.close()
            $listener.stop()
        }
        Catch {
            "Receive Message failed with: `n" + $Error[0]
        }
    }
}
```

Начните слушать со следующего и захватите любое сообщение в переменной `$msg` :

```
$msg = Receive-TCPMessage -Port 29800
```

TCP-отправитель

```
Function Send-TCPMessage {
    Param (
            [Parameter(Mandatory=$true, Position=0)]
            [ValidateNotNullOrEmpty()]
```

```
            [string]
            $EndPoint
        ,
            [Parameter(Mandatory=$true, Position=1)]
            [int]
            $Port
        ,
            [Parameter(Mandatory=$true, Position=2)]
            [string]
            $Message
    )
    Process {
        # Setup connection
        $IP = [System.Net.Dns]::GetHostAddresses($EndPoint)
        $Address = [System.Net.IPAddress]::Parse($IP)
        $Socket = New-Object System.Net.Sockets.TCPClient($Address,$Port)

        # Setup stream wrtier
        $Stream = $Socket.GetStream()
        $Writer = New-Object System.IO.StreamWriter($Stream)

        # Write message to stream
        $Message | % {
            $Writer.WriteLine($_)
            $Writer.Flush()
        }

        # Close connection and stream
        $Stream.Close()
        $Socket.Close()
    }
}
```

Отправить сообщение с:

```
Send-TCPMessage -Port 29800 -Endpoint 192.168.0.1 -message "My first TCP message !"
```

Примечание . Сообщения TCP могут быть заблокированы вашим программным брандмауэром или любыми внешними брандмауэрами, которые вы пытаетесь пройти. Убедитесь, что TCP-порт, который вы установили в приведенной выше команде, открыт и что вы настроили прослушиватель на том же порту.

Прочитайте Связь TCP с PowerShell онлайн: https://riptutorial.com/ru/powershell/topic/5125/связь-tcp-с-powershell

глава 65: Связь с API-интерфейсом RESTful

Вступление

REST означает передачу репрезентативного состояния (иногда пишется «ReST»). Он основан на протоколе связи без кэширования, клиент-сервер и кэшировании, и в основном используется протокол HTTP. Он в основном используется для создания веб-сервисов, которые являются легкими, обслуживаемыми и масштабируемыми. Служба, основанная на REST, называется службой RESTful, а API, которые используются для нее, являются API RESTful. В PowerShell `Invoke-RestMethod` используется для борьбы с ними.

Examples

Использовать вложенные веб-камеры Slack.com

Определите свою полезную нагрузку для отправки для получения более сложных данных

```
$Payload = @{ text="test string"; username="testuser" }
```

Используйте командлет `ConvertTo-Json` и `Invoke-RestMethod` для выполнения вызова

```
Invoke-RestMethod -Uri "https://hooks.slack.com/services/yourwebhookstring" -Method Post -Body
(ConvertTo-Json $Payload)
```

Сообщение для hipChat

```
$params = @{
    Uri = "https://your.hipchat.com/v2/room/934419/notification?auth_token=???"
    Method = "POST"
    Body = @{
        color = 'yellow'
        message = "This is a test message!"
        notify = $false
        message_format = "text"
    } | ConvertTo-Json
    ContentType = 'application/json'
}

Invoke-RestMethod @params
```

Использование REST с объектами PowerShell для получения и размещения отдельных данных

Получите данные REST и сохраните их в объекте PowerShell:

```
$Post = Invoke-RestMethod -Uri "http://jsonplaceholder.typicode.com/posts/1"
```

Измените свои данные:

```
$Post.title = "New Title"
```

ОТМЕТЬТЕ данные REST

```
$Json = $Post | ConvertTo-Json
Invoke-RestMethod -Method Put -Uri "http://jsonplaceholder.typicode.com/posts/1" -Body $Json -
ContentType 'application/json'
```

Использование REST с объектами PowerShell для получения и POST многих элементов

Получите данные REST и сохраните их в объекте PowerShell:

```
$Users = Invoke-RestMethod -Uri "http://jsonplaceholder.typicode.com/users"
```

Измените многие элементы в ваших данных:

```
$Users[0].name = "John Smith"
$Users[0].email = "John.Smith@example.com"
$Users[1].name = "Jane Smith"
$Users[1].email = "Jane.Smith@example.com"
```

ОТПРАВИТЬ все данные REST назад:

```
$Json = $Users | ConvertTo-Json
Invoke-RestMethod -Method Post -Uri "http://jsonplaceholder.typicode.com/users" -Body $Json -
ContentType 'application/json'
```

Использование REST с PowerShell для удаления элементов

Определите элемент, который необходимо удалить, и удалите его:

```
Invoke-RestMethod -Method Delete -Uri "http://jsonplaceholder.typicode.com/posts/1"
```

Прочитайте Связь с API-интерфейсом RESTful онлайн:
https://riptutorial.com/ru/powershell/topic/3869/связь-с-api-интерфейсом-restful

глава 66: Соглашения об именах

Examples

функции

```
Get-User()
```

- Используйте шаблон *Verb-Noun* при назначении функции.
- Глагол подразумевает действие, например `Get` , `Set` , `New` , `Read` , `Write` и многие другие. См. Утвержденные глаголы .
- Существительное должно быть сингулярным, даже если оно действует на несколько элементов. `Get-User()` может возвращать одного или нескольких пользователей.
- Используйте случай Паскаля для глагола и существительного. Например, `Get-UserLogin()`

Прочитайте Соглашения об именах онлайн: https://riptutorial.com/ru/powershell/topic/9714/соглашения-об-именах

глава 67: Создание ресурсов класса DSC

Вступление

Начиная с версии PowerShell версии 5.0 вы можете использовать определения класса PowerShell для создания ресурсов конфигурации требуемого состояния (DSC).

Чтобы помочь в создании ресурса DSC, существует `[DscResource()]` который применяется к определению класса, и `[DscProperty()]` для назначения свойств, которые настраиваются пользователем ресурса DSC.

замечания

Ресурс DSC на основе классов должен:

- `[DscResource()]` атрибут `[DscResource()]`
- Определите метод `Test()` который возвращает `[bool]`
- Определите метод `Get()` который возвращает свой собственный тип объекта (например, `[Ticket]`)
- Определите метод `Set()` который возвращает `[void]`
- По меньшей мере одно `Key` свойство DSC

После создания ресурса DSC PowerShell на основе класса он должен быть «экспортирован» из модуля с использованием файла манифеста модуля (.psd1). В манифесте модуля ключ `DscResourcesToExport` используется для объявления массива ресурсов DSC (имена классов) для «экспорта» из модуля. Это позволяет потребителям модуля DSC «видеть» ресурсы класса в модуле.

Examples

Создание класса скелета ресурсов DSC

```
[DscResource()]
class File {
}
```

В этом примере показано, как построить внешний раздел класса PowerShell, который объявляет ресурс DSC. Вам все равно нужно заполнить содержание определения класса.

Скелет ресурсов DSC с ключевым свойством

```
[DscResource()]
```

```
class Ticket {
  [DscProperty(Key)]
  [string] $TicketId
}
```

Ресурс DSC должен объявить хотя бы одно свойство ключа. Ключевое свойство - это то, что уникально идентифицирует ресурс из других ресурсов. Например, предположим, что вы создаете ресурс DSC, который представляет билет в системе продажи билетов. Каждый билет будет уникально представлен идентификатором билета.

Каждое свойство, которое будет отображаться для *пользователя* ресурса DSC, должно быть украшено `[DscProperty()]` . Эти атрибуты принимают `key` параметр, указывающий, что это свойство является ключевым атрибутом для ресурса DSC.

Ресурс DSC с обязательным имуществом

```
[DscResource()]
class Ticket {
  [DscProperty(Key)]
  [string] $TicketId

  [DscProperty(Mandatory)]
  [string] $Subject
}
```

При создании ресурса DSC вы часто обнаружите, что не каждое свойство должно быть обязательным. Тем не менее, есть некоторые основные свойства, которые вы хотите обеспечить, настраиваются пользователем ресурса DSC. Вы используете `Mandatory` параметр `[DscResource()]` для объявления свойства, как того требует пользователь ресурса DSC.

В приведенном выше примере мы добавили свойство `Subject` к ресурсу `Ticket` , который представляет собой уникальный билет в системе продажи билетов и обозначил его как `Mandatory` свойство.

Ресурс DSC с необходимыми методами

```
[DscResource()]
class Ticket {
  [DscProperty(Key)]
  [string] $TicketId

  # The subject line of the ticket
  [DscProperty(Mandatory)]
  [string] $Subject

  # Get / Set if ticket should be open or closed
  [DscProperty(Mandatory)]
  [string] $TicketState

  [void] Set() {
```

```
    # Create or update the resource
  }

  [Ticket] Get() {
    # Return the resource's current state as an object
    $TicketState = [Ticket]::new()
    return $TicketState
  }

  [bool] Test() {
    # Return $true if desired state is met
    # Return $false if desired state is not met
    return $false
  }
}
```

Это полный ресурс DSC, который демонстрирует все основные требования к созданию допустимого ресурса. Реализации метода не являются полными, но предоставляются с целью показать базовую структуру.

Прочитайте Создание ресурсов класса DSC онлайн:
https://riptutorial.com/ru/powershell/topic/8733/создание-ресурсов-класса-dsc

глава 68: Специальные операторы

Examples

Оператор выражения массива

Возвращает выражение как массив.

```
@(Get-ChildItem $env:windir\System32\ntdll.dll)
```

Вернет массив с одним элементом

```
@(Get-ChildItem $env:windir\System32)
```

Вернет массив со всеми элементами в папке (что не является изменением поведения из внутреннего выражения.

Операция вызова

```
$command = 'Get-ChildItem'
& $Command
```

Запустит `Get-ChildItem`

Оператор точного поиска

, . \ MyScript.ps1

запускается `.\myScript.ps1` в текущей области, создавая любые функции и переменную, доступные в текущей области.

Прочитайте Специальные операторы онлайн: https://riptutorial.com/ru/powershell/topic/8981/специальные-операторы

глава 69: Струны

Синтаксис

- "(Double-quoted) Строка"
- 'Литеральная строка'
- @»
  Здесь строка
  «@
- @»
  Литературная здесь-строка
  @

замечания

Строки - это объекты, представляющие текст.

Examples

Создание базовой строки

строка

Строки создаются путем обертывания текста двойными кавычками. Строки с двойными кавычками могут вычислять переменные и специальные символы.

```
$myString = "Some basic text"
$mySecondString = "String with a $variable"
```

Чтобы использовать двойную кавычку внутри строки, ее нужно экранировать с помощью escape-символа, backtick (`). Одинарные кавычки могут использоваться внутри строки с двумя кавычками.

```
$myString = "A `"double quoted`" string which also has 'single quotes'."
```

Литеральная строка

Литеральные строки - это строки, которые не оценивают переменные и специальные символы. Он создается с использованием одинарных кавычек.

```
$myLiteralString = 'Simple text including special characters (`n) and a $variable-reference'
```

Чтобы использовать одинарные кавычки внутри литеральной строки, используйте двойные одинарные кавычки или буквенную строку здесь. Двойные qutoes можно безопасно использовать внутри строки

```
$myLiteralString = 'Simple string with ''single quotes'' and "double quotes".'
```

Строка формата

```
$hash = @{ city = 'Berlin' }

$result = 'You should really visit {0}' -f $hash.city
Write-Host $result #prints "You should really visit Berlin"
```

Строки формата могут использоваться с оператором `-f` или статическим методом `[String]::Format(string format, args)` .NET.

Многострочная строка

Существует несколько способов создания многострочной строки в PowerShell:

- Вы можете использовать специальные символы для возврата каретки и / или новой строки вручную или использовать переменную `NewLine` -environment для вставки значения «новая строка» в системах)

  ```
  "Hello`r`nWorld"
  "Hello{0}World" -f [environment]::NewLine
  ```

- Создайте linebreak при определении строки (до закрытия цитаты)

  ```
  "Hello
  World"
  ```

- Использование строки здесь. *Это самый распространенный метод.*

  ```
  @"
  Hello
  World
  "@
  ```

Здесь строка

Здесь строки очень полезны при создании многострочных строк. Одним из самых больших

преимуществ по сравнению с другими многострочными строками является то, что вы можете использовать кавычки, не избегая их с помощью обратного хода.

Здесь строка

Здесь строки начинаются с `@"` и строки и заканчиваются на `"@` на собственной строке (`"@` **должны быть первыми символами на линии, даже не пробелы** / **вкладки**).

```
@"
Simple
    Multiline string
with "quotes"
"@
```

Литературная здесь-строка

Вы также можете создать литерал здесь-string, используя одинарные кавычки, когда вы не хотите, чтобы какие-либо выражения расширялись точно так же, как строка обычного литерала.

```
@'
The following line won't be expanded
$(Get-Date)
because this is a literal here-string
'@
```

Конкатенация строк

Использование переменных в строке

Вы можете конкатенировать строки, используя переменные внутри строки с двумя кавычками. Это не работает со свойствами.

```
$string1 = "Power"
$string2 = "Shell"
"Greetings from $string1$string2"
```

Используя оператор `+`

Вы также можете присоединить строки, используя оператор `+` .

```
$string1 = "Greetings from"
```

```
$string2 = "PowerShell"
$string1 + " " + $string2
```

Это также работает со свойствами объектов.

```
"The title of this console is '" + $host.Name + "'"
```

Использование подвыражений

Результат / результат подвыражений `$()` может использоваться в строке. Это полезно при доступе к пропозиции объекта или выполнении сложного выражения. Подвыражения могут содержать несколько операторов, разделенных точкой с запятой `;`

```
"Tomorrow is $((Get-Date).AddDays(1).DayOfWeek)"
```

Специальные символы

При использовании внутри строки с двойными кавычками escape-символ (backtick `` ` ``) представляет особый символ.

```
`0    #Null
`a    #Alert/Beep
`b    #Backspace
`f    #Form feed (used for printer output)
`n    #New line
`r    #Carriage return
`t    #Horizontal tab
`v    #Vertical tab (used for printer output)
```

Пример:

```
> "This`tuses`ttab`r`nThis is on a second line"
This    uses    tab
This is on a second line
```

Вы также можете избежать специальных символов со специальными значениями:

```
`#    #Comment-operator
`$    #Variable operator
``    #Escape character
`'    #Single quote
`"    #Double quote
```

Прочитайте Струны онлайн: https://riptutorial.com/ru/powershell/topic/5124/струны

глава 70: Сценарии подписания

замечания

Подписание сценария заставит ваши скрипты соответствовать всем политикам exeuction в PowerShell и обеспечить целостность скрипта. Подписанные сценарии не будут выполняться, если они были изменены после подписания.

Для подписания скриптов требуется сертификат подписи кода. Рекомендации:

- Персональные скрипты / тестирование (не общие): Сертификат от доверенного органа сертификации (внутреннего или стороннего) **или** самозаверяющего сертификата.
- Общая внутренняя организация: сертификат от доверенного органа сертификации (внутреннего или стороннего)
- Общая внешняя организация: сертификат доверенного лица, уполномоченного третьей стороны

Подробнее о about_Signing @ TechNet

Политика выполнения

У PowerShell есть настраиваемые политики выполнения, которые определяют, какие условия необходимы для сценария или конфигурации, которые будут выполнены. Политика исключения может быть установлена для нескольких областей; компьютер, текущий пользователь и текущий процесс. **Политики выполнения можно легко обойти и не предназначать для ограничения пользователей, а скорее защищать их от непреднамеренного нарушения правил подписи.**

Доступные политики:

настройка	Описание
ограниченный	Никаких сценариев не разрешено
AllSigned	Все скрипты должны быть подписаны
RemoteSigned	Разрешены все локальные скрипты; только подписанные удаленные сценарии
неограниченный	Нет требований. Все сценарии разрешены, но будут предупреждать, прежде чем запускать скрипты, загруженные из Интернета

настройка	Описание
байпас	Все сценарии разрешены и не отображаются предупреждения
Неопределенный	Удалите текущую политику выполнения для текущей области. Использует родительскую политику. Если все политики не определены, будут использоваться ограничения.

Вы можете изменить текущие политики выполнения с помощью `Set-ExecutionPolicy` -cmdlet, групповой политики или параметра `-ExecutionPolicy` при запуске процесса `powershell.exe` .

Подробнее о about_Execution_Policies @ TechNet

Examples

Подписание сценария

Подписание сценария выполняется с помощью `Set-AuthenticodeSignature` -cmdlet и сертификата подписи кода.

```
#Get the first available personal code-signing certificate for the logged on user
$cert = @(Get-ChildItem -Path Cert:\CurrentUser\My -CodeSigningCert)[0]

#Sign script using certificate
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1
```

Вы также можете прочитать сертификат из `.pfx` файла, используя:

```
$cert = Get-PfxCertificate -FilePath "C:\MyCodeSigningCert.pfx"
```

Сценарий будет действителен до истечения срока действия cetificate. Если во время подписания используется timestamp-сервер, сценарий будет продолжать действовать после истечения срока действия сертификата. Также полезно добавить цепочку доверия для сертификата (включая полномочия root), чтобы помочь большинству компьютеров доверять сертифицированному, используемому для подписи скрипта.

```
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1 -IncludeChain All -
TimeStampServer "http://timestamp.verisign.com/scripts/timstamp.dll"
```

Рекомендуется использовать timestamp-сервер у доверенного поставщика сертификатов, такого как Verisign, Comodo, Thawte и т. Д.

Изменение политики выполнения с помощью Set-ExecutionPolicy

Чтобы изменить политику выполнения для области по умолчанию (LocalMachine), используйте:

```
Set-ExecutionPolicy AllSigned
```

Чтобы изменить политику для определенной области, используйте:

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy AllSigned
```

Вы можете подавить подсказки, добавив переключатель `-Force` .

Обход политики выполнения для одного скрипта

Часто вам может потребоваться выполнить неподписанный скрипт, который не соответствует текущей политике выполнения. Легкий способ сделать это - обходить политику выполнения для этого единственного процесса. Пример:

```
powershell.exe -ExecutionPolicy Bypass -File C:\MyUnsignedScript.ps1
```

Или вы можете использовать стенографию:

```
powershell -ep Bypass C:\MyUnsignedScript.ps1
```

Другие правила выполнения:

политика	Описание
`AllSigned`	Можно запускать только скрипты, подписанные доверенным издателем.
`Bypass`	Нет ограничений; все сценарии Windows PowerShell могут быть запущены.
`Default`	Обычно `RemoteSigned` , но управляется через ActiveDirectory
`RemoteSigned`	Загруженные скрипты должны быть подписаны доверенным издателем, прежде чем они могут быть запущены.
`Restricted`	Никакие сценарии не могут быть запущены. Windows PowerShell можно использовать только в интерактивном режиме.
`Undefined`	Не Доступно
`Unrestricted*`	Подобно `bypass`

`Unrestricted*` ***Caveat:*** *если вы запускаете неподписанный скрипт, загруженный из*

Интернета, вам предлагается получить разрешение до его запуска.

Дополнительная информация доступна здесь .

Получить текущую политику выполнения

Получение эффективной политики выполнения для текущего сеанса:

```
PS> Get-ExecutionPolicy
RemoteSigned
```

Список всех эффективных политик выполнения для текущего сеанса:

```
PS> Get-ExecutionPolicy -List

        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy       Undefined
   UserPolicy       Undefined
      Process       Undefined
  CurrentUser       Undefined
 LocalMachine    RemoteSigned
```

Перечислите политику выполнения для определенной области, например. процесс:

```
PS> Get-ExecutionPolicy -Scope Process
Undefined
```

Получение подписи из подписанного скрипта

Получите информацию о сигнатуре Authenticode из подписанного скрипта с помощью `Get-AuthenticodeSignature` -cmdlet:

```
Get-AuthenticodeSignature .\MyScript.ps1 | Format-List *
```

Создание самоподписанного сертификата подписи кода для тестирования

При подписании личных сценариев или при тестировании подписи кода может оказаться полезным создать самоподписанный сертификат подписи кода.

5.0

Начиная с PowerShell 5.0 вы можете создать самозаверяющий сертификат подписи кода, используя `New-SelfSignedCertificate` -cmdlet:

```
New-SelfSignedCertificate -FriendlyName "StackOverflow Example Code Signing" -
```

```
CertStoreLocation Cert:\CurrentUser\My -Subject "SO User" -Type CodeSigningCert
```

В более ранних версиях вы можете создать самозаверяющий сертификат, используя средство `makecert.exe` , обнаруженное в SDK .NET Framework и Windows SDK.

Самозаверяющемуся ceriticate будет доверять только компьютеры, установившие сертификат. Для сценариев, которые будут использоваться совместно, рекомендуется использовать сертификат из доверенного центра сертификации (внутреннего или доверенного стороннего производителя).

Прочитайте Сценарии подписания онлайн: https://riptutorial.com/ru/powershell/topic/5670/сценарии-подписания

глава 71: Управление пакетами

Вступление

PowerShell Package Management позволяет вам находить, устанавливать, обновлять и удалять модули PowerShell и другие пакеты.

PowerShellGallery.com является источником по умолчанию для модулей PowerShell. Вы также можете просмотреть сайт для доступных пакетов, выполнить команду и просмотреть код.

Examples

Найдите модуль PowerShell с использованием шаблона

Чтобы найти модуль, который заканчивается `DSC`

```
Find-Module -Name *DSC
```

Создайте резервную копию модуля PowerShell по умолчанию

Если по какой-то причине удаляется репозиторий `PSGallery` модуля PowerShell по умолчанию. Вам нужно будет создать его. Это команда.

```
Register-PSRepository -Default
```

Найти модуль по названию

```
Find-Module -Name <Name>
```

Установка модуля по имени

```
Install-Module -Name <name>
```

Удалите модуль моего имени и версии

```
Uninstall-Module -Name <Name> -RequiredVersion <Version>
```

Обновление модуля по имени

```
Update-Module -Name <Name>
```

Прочитайте Управление пакетами онлайн: https://riptutorial.com/ru/powershell/topic/8698/управление-пакетами

глава 72: Условная логика

Синтаксис

- если (выражение) {}
- если (выражение) {} еще {}
- если (выражение) {} ElseIf (выражение) {}
- если (выражение) {} ElseIf (выражение) {} еще {}

замечания

См. Также Операторы сравнения , которые можно использовать в условных выражениях.

Examples

if, else и else if

Powershell поддерживает стандартные операторы условной логики, как и многие языки программирования. Они позволяют выполнять определенные функции или команды при определенных обстоятельствах.

Если а, `if` команды внутри скобок (`{}`) выполняются только в том случае, если выполнены условия внутри if (`()`)

```
$test = "test"
if ($test -eq "test"){
    Write-Host "if condition met"
}
```

Вы также можете сделать что- `else` . Здесь команды `else` выполняются, `if` условия `if` **не** выполняются:

```
$test = "test"
if ($test -eq "test2"){
    Write-Host "if condition met"
}
else{
    Write-Host "if condition not met"
}
```

или `elseif` . Команда else if запускает команды, если условия `if` не выполняются и выполняются условия `elseif` :

```
$test = "test"
if ($test -eq "test2"){
```

```
    Write-Host "if condition met"
}
elseif ($test -eq "test"){
    Write-Host "ifelse condition met"
}
```

Обратите внимание на приведенное выше использование `-eq` (равенство) CmdLet и not `=` or `==` как это делают многие другие языки для эквализации.

Отрицание

Вы можете отказаться от логического значения, т. Е. Ввести оператор `if` если условие ложно, а не true. Это может быть сделано с помощью `-Not` CMDlet

```
$test = "test"
if (-Not $test -eq "test2"){
    Write-Host "if condition not met"
}
```

Вы также можете использовать `!` :

```
$test = "test"
if (!($test -eq "test2")){
    Write-Host "if condition not met"
}
```

существует также оператор `-ne` (не равный):

```
$test = "test"
if ($test -ne "test2"){
    Write-Host "variable test is not equal to 'test2'"
}
```

Если условная стенография

Если вы хотите использовать стенограмму, вы можете использовать условную логику со следующей стенографией. Только строка «false» будет оцениваться как true (2.0).

```
#Done in Powershell 2.0
$boolean = $false;
$string = "false";
$emptyString = "";

If($boolean){
    # this does not run because $boolean is false
    Write-Host "Shorthand If conditions can be nice, just make sure they are always boolean."
}

If($string){
    # This does run because the string is non-zero length
    Write-Host "If the variable is not strictly null or Boolean false, it will evaluate to
true as it is an object or string with length greater than 0."
```

```
}

If($emptyString){
    # This does not run because the string is zero-length
    Write-Host "Checking empty strings can be useful as well."
}

If($null){
    # This does not run because the condition is null
    Write-Host "Checking Nulls will not print this statement."
}
```

Прочитайте Условная логика онлайн: https://riptutorial.com/ru/powershell/topic/7208/условная-логика

глава 73: Фоновые задания PowerShell

Вступление

Рабочие места были представлены в PowerShell 2.0 и помогли решить проблему, присущую инструментам командной строки. В двух словах, если вы запустите длительную задачу, ваше приглашение недоступно, пока задача не завершится. В качестве примера долговременной задачи подумайте об этой простой команде PowerShell:

Get-ChildItem -Path c: \ -Recurse

Для получения полного списка каталогов вашего диска C: потребуется некоторое время. Если вы запустите его как «Job», консоль вернет управление, и вы сможете позже зафиксировать результат.

замечания

Работа PowerShell выполняется в новом процессе. У этого есть плюсы и минусы, которые связаны между собой.

Плюсы:

1. Работа выполняется в чистом процессе, включая среду.
2. Задание может выполняться асинхронно с вашим основным процессом PowerShell

Минусы:

1. Изменения в рабочей среде не будут присутствовать в задании.
2. Параметры передаются и возвращаемые результаты сериализуются.
    - Это означает, что если вы измените объект параметра во время выполнения задания, он не будет отражен в задании.
    - Это также означает, что если объект не может быть сериализован, вы не можете его передать или вернуть (хотя PowerShell может копировать любые параметры и передавать / возвращать PSObject.)

Examples

Основное создание рабочих мест

Запустите блок сценариев в качестве фонового задания:

```
$job = Start-Job -ScriptBlock {Get-Process}
```

Запустите сценарий в качестве фонового задания:

```
$job = Start-Job -FilePath "C:\YourFolder\Script.ps1"
```

Запустите задание с помощью `Invoke-Command` на удаленном компьютере:

```
$job = Invoke-Command -ComputerName "ComputerName" -ScriptBlock {Get-Service winrm} -JobName
"WinRM" -ThrottleLimit 16 -AsJob
```

Начать работу как другой пользователь (запрашивает пароль):

```
Start-Job -ScriptBlock {Get-Process} -Credential "Domain\Username"
```

Или же

```
Start-Job -ScriptBlock {Get-Process} -Credential (Get-Credential)
```

Начать работу как другой пользователь (нет приглашения):

```
$username = "Domain\Username"
$password = "password"
$secPassword = ConvertTo-SecureString -String $password -AsPlainText -Force
$credentials = New-Object System.Management.Automation.PSCredential -ArgumentList @($username,
$secPassword)
Start-Job -ScriptBlock {Get-Process} -Credential $credentials
```

Управление базой данных

Получить список всех заданий в текущем сеансе:

```
Get-Job
```

Ожидание выполнения задания до получения результата:

```
$job | Wait-job | Receive-Job
```

Тайм-аут задания, если он работает слишком долго (10 секунд в этом примере)

```
$job | Wait-job -Timeout 10
```

Остановка задания (завершает все задачи, которые ожидаются в очереди заданий до окончания):

```
$job | Stop-Job
```

Удалите задание из списка текущих заданий текущего сеанса:

```
$job | Remove-Job
```

Примечание . Следующие действия будут работать только в `Workflow` Jobs.

Приостановить работу `Workflow` (пауза):

```
$job | Suspend-Job
```

Возобновление `Workflow` :

```
$job | Resume-Job
```

Прочитайте Фоновые задания PowerShell онлайн:
https://riptutorial.com/ru/powershell/topic/3970/фоновые-задания-powershell

глава 74: Функции PowerShell

Вступление

Функция в основном является именованным блоком кода. Когда вы вызываете имя функции, выполняется блок сценария внутри этой функции. Это список операторов PowerShell, который имеет имя, которое вы назначаете. Когда вы запускаете функцию, вы вводите имя функции. Это метод экономии времени при решении повторяющихся задач. Форматы PowerShell в трех частях: ключевое слово «Функция», за которым следует имя, наконец, полезная нагрузка, содержащая блок сценария, который заключен в фигурные скобки / скобки для скобок.

Examples

Простая функция без параметров

Это пример функции, которая возвращает строку. В этом примере функция вызывается в инструкции, присваивающей значение переменной. Значение в этом случае является возвращаемым значением функции.

```
function Get-Greeting{
    "Hello World"
}

# Invoking the function
$greeting = Get-Greeting

# demonstrate output
$greeting
Get-Greeting
```

`function` объявляет следующий код как функцию.

`Get-Greeting` - это имя функции. В любое время, когда эта функция должна использоваться в скрипте, эту функцию можно вызвать путем вызова ее по имени.

`{ ... }` - это блок сценария, выполняемый функцией.

Если вышеуказанный код выполняется в ISE, результаты будут выглядеть примерно так:

```
Hello World
Hello World
```

Основные параметры

Функция может быть определена с помощью параметров с помощью блока param:

```
function Write-Greeting {
    param(
        [Parameter(Mandatory,Position=0)]
        [String]$name,
        [Parameter(Mandatory,Position=1)]
        [Int]$age
    )
    "Hello $name, you are $age years old."
}
```

Или используя простой синтаксис функции:

```
function Write-Greeting ($name, $age) {
    "Hello $name, you are $age years old."
}
```

Примечание. Параметры каста не требуются ни в одном определении параметра.

Простой синтаксис функций (SFS) имеет очень ограниченные возможности по сравнению с блоком param.
Хотя вы можете определить параметры, которые должны быть выставлены внутри функции, вы не можете указывать атрибуты параметров , использовать `[CmdletBinding()]` параметров , включать `[CmdletBinding()]` , с SFS (и это не исчерпывающий список).

Функции могут вызываться с помощью упорядоченных или именованных параметров.

Порядок параметров в вызове сопоставляется с порядком объявления в заголовке функции (по умолчанию) или может быть задан с использованием атрибута параметра `Position` (как показано в примере расширенной функции выше).

```
$greeting = Write-Greeting "Jim" 82
```

Альтернативно, эта функция может быть вызвана с именованными параметрами

```
$greeting = Write-Greeting -name "Bob" -age 82
```

Обязательные параметры

Параметры функции могут быть отмечены как обязательные

```
function Get-Greeting{
    param
    (
        [Parameter(Mandatory=$true)]$name
    )
    "Hello World $name"
}
```

Если функция вызывается без значения, в командной строке будет указано значение:

```
$greeting = Get-Greeting

cmdlet Get-Greeting at command pipeline position 1
Supply values for the following parameters:
name:
```

Расширенные функции

Это копия расширенного функционального фрагмента из PowerShell ISE. В основном это шаблон для многих вещей, которые вы можете использовать с расширенными функциями в Powershell. Ключевые моменты:

- интеграция get-help - начало функции содержит блок комментариев, который настроен для чтения командлетом get-help. Функциональный блок может быть расположен в конце, если это необходимо.
- cmdletbinding - функция будет вести себя как командлет
- параметры
- наборы параметров

```
<#
.Synopsis
   Short description
.DESCRIPTION
   Long description
.EXAMPLE
   Example of how to use this cmdlet
.EXAMPLE
   Another example of how to use this cmdlet
.INPUTS
   Inputs to this cmdlet (if any)
.OUTPUTS
   Output from this cmdlet (if any)
.NOTES
   General notes
.COMPONENT
   The component this cmdlet belongs to
.ROLE
   The role this cmdlet belongs to
.FUNCTIONALITY
   The functionality that best describes this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding(DefaultParameterSetName='Parameter Set 1',
                   SupportsShouldProcess=$true,
                   PositionalBinding=$false,
                   HelpUri = 'http://www.microsoft.com/',
                   ConfirmImpact='Medium')]
    [Alias()]
    [OutputType([String])]
    Param
    (
        # Param1 help description
```

```
        [Parameter(Mandatory=$true,
                   ValueFromPipeline=$true,
                   ValueFromPipelineByPropertyName=$true,
                   ValueFromRemainingArguments=$false,
                   Position=0,
                   ParameterSetName='Parameter Set 1')]
        [ValidateNotNull()]
        [ValidateNotNullOrEmpty()]
        [ValidateCount(0,5)]
        [ValidateSet("sun", "moon", "earth")]
        [Alias("p1")]
        $Param1,

        # Param2 help description
        [Parameter(ParameterSetName='Parameter Set 1')]
        [AllowNull()]
        [AllowEmptyCollection()]
        [AllowEmptyString()]
        [ValidateScript({$true})]
        [ValidateRange(0,5)]
        [int]
        $Param2,

        # Param3 help description
        [Parameter(ParameterSetName='Another Parameter Set')]
        [ValidatePattern("[a-z]*")]
        [ValidateLength(0,15)]
        [String]
        $Param3
    )

    Begin
    {
    }
    Process
    {
        if ($pscmdlet.ShouldProcess("Target", "Operation"))
        {
        }
    }
    End
    {
    }
}
```

Проверка параметров

Существует множество способов проверки ввода параметров в PowerShell.

Вместо того, чтобы писать код внутри функций или скриптов для проверки значений параметров, эти ParameterAttributes будут выдавать, если передаются недопустимые значения.

ValidateSet

Иногда нам нужно ограничить возможные значения, которые может принимать параметр.

Предположим, мы хотим разрешить только красный, зеленый и синий для параметра `$Color` в скрипте или функции.

Мы можем использовать атрибут параметра `ValidateSet` для ограничения этого. У этого есть дополнительное преимущество, позволяющее завершить вкладку при настройке этого аргумента (в некоторых средах).

```
param(
    [ValidateSet('red','green','blue',IgnoreCase)]
    [string]$Color
)
```

Вы также можете указать `IgnoreCase` для отключения чувствительности к регистру.

ValidateRange

Этот метод проверки параметров принимает минимальное и максимальное значение Int32 и требует, чтобы параметр находился в этом диапазоне.

```
param(
    [ValidateRange(0,120)]
    [Int]$Age
)
```

ValidatePattern

Этот метод проверки параметров принимает параметры, соответствующие указанному шаблону регулярного выражения.

```
param(
    [ValidatePattern("\w{4-6}\d{2}")]
    [string]$UserName
)
```

ValidateLength

Этот метод проверки параметров проверяет длину переданной строки.

```
param(
    [ValidateLength(0,15)]
    [String]$PhoneNumber
)
```

ValidateCount

Этот метод проверки параметров проверяет количество аргументов, переданных,

например, массивом строк.

```
param(
    [ValidateCount(1,5)]
    [String[]]$ComputerName
)
```

ValidateScript

Наконец, метод ValidateScript необычайно гибкий, принимая скриптовый блок и оценивая его с помощью $ _ для представления переданного аргумента. Затем он передает аргумент, если результат равен $ true (включая любой вывод как действительный).

Это можно использовать для проверки наличия файла:

```
param(
    [ValidateScript({Test-Path $_})]
    [IO.FileInfo]$Path
)
```

Чтобы проверить, существует ли пользователь в AD:

```
param(
    [ValidateScript({Get-ADUser $_})]
    [String]$UserName
)
```

И почти все, что вы можете написать (как это не ограничивается oneliners):

```
param(
    [ValidateScript({
        $AnHourAgo = (Get-Date).AddHours(-1)
        if ($_ -lt $AnHourAgo.AddMinutes(5) -and $_ -gt $AnHourAgo.AddMinutes(-5)) {
            $true
        } else {
            throw "That's not within five minutes. Try again."
        }
    })]
    [String]$TimeAboutAnHourAgo
)
```

Прочитайте Функции PowerShell онлайн: https://riptutorial.com/ru/powershell/topic/1673/функции-powershell

кредиты

S. No	Главы	Contributors
1	Начало работы с PowerShell	4444, autosvet, Brant Bobby, Chris N, Clijsters, Community, DarkLite1, DAXaholic, Eitan, FoxDeploy, Gordon Bell, Greg Bray, Ian Miller, It-Z, JNYRanger, Jonas, Luboš Turek, Mark Wragg, Mathieu Buisson, Mrk, Nacimota, oɔɯəɹ, Poorkenny, Sam Martin, th1rdey3, TheIncorrigible1, Tim, tjrobinson, TravisEz13, vonPryz, Xalorous
2	Cmdlet Naming	TravisEz13
3	HashTables	Florian Meyer, Ranadip Dutta, TravisEz13
4	Loops	Blockhead, Christopher G. Lewis, Clijsters, CmdrTchort, DAXaholic, Eris, Frode F., Gomibushi, Gordon Bell, Jay Bazuzi, Jon, jumbo, mákos, Poorkenny, Ranadip Dutta, Richard, Roman , SeeuD1, Shawn Esterman, StephenP, TessellatingHeckler, TheIncorrigible1, VertigoRay
5	MongoDB	Thomas Gerot, Zteffer
6	PowerShell « Потоки»; Debug, Verbose, Warning, Error, Output и Information	DarkLite1, Dave Anderson, megamorf
7	PSScriptAnalyzer - анализатор сценариев PowerShell	Mark Wragg, mattnicola
8	Splatting	autosvet, Frode F., Moerwald, Petru Zaharia, Poorkenny, RamenChef, Ranadip Dutta, TravisEz13, xXhRQ8sD2L7Z
9	sql-запросы powershell	Venkatakrishnan
10	WMI и CIM	Frode F.
11	Автоматизация инфраструктуры	Giulio Caccin, Ranadip Dutta

12	Автоматические переменные	Brant Bobby, jumbo, Mateusz Piotrowski, Moerwald, Ranadip Dutta, Roman
13	Автоматические переменные - часть 2	Roman
14	Анонимный IP (v4 и v6) в текстовом файле с Powershell	NooJ
15	Архивный модуль	James Ruskin, RapidCoder
16	Безопасность и криптография	YChi Lu
17	Введение в Pester	Frode F., Sam Martin
18	Введение в Psake	Roman
19	Вложение управляемого кода (C # | VB)	ajb101
20	Возвратное поведение в PowerShell	Bert Levrau, camilohe, Eris, jumbo, Ranadip Dutta, Thomas Gerot
21	Встроенные переменные	Trevor Sullivan
22	Выполнение исполняемых файлов	RamenChef, W1M0R
23	Графический интерфейс в Powershell	Sam Martin
24	Динамические параметры PowerShell	Poorkenny
25	Использование ShouldProcess	Brant Bobby, Charlie Joynt, Schwarzie2478
26	Использование	Clijsters, jumbo, Ranadip Dutta

	индикатора выполнения	
27	Использование справочной системы	Frode F., Madniz, mattnicola, RamenChef
28	Использование существующих статических классов	Austin T French, briantist, motcke, Ranadip Dutta, Xenophane
29	Как загрузить последний артефакт из Artifactory с использованием сценария Powershell (v2.0 или ниже)?	ANIL
30	Классы PowerShell	boeprox, Brant Bobby, Frode F., Jaqueline Vanek, Mert Gülsoy, Ranadip Dutta, xvorsx
31	Кодирование / декодирование URL	VertigoRay
32	Командная строка PowerShell.exe	Frode F.
33	Конфигурация желаемого состояния	autosvet, CmdrTchort, Frode F., RamenChef
34	Модули Powershell	autosvet, Mike Shepard, TravisEz13, Trevor Sullivan
35	Модули, скрипты и функции	Frode F., Ranadip Dutta, Xalorous
36	Модуль ActiveDirectory	Lachie White
37	Модуль ISE	Florian Meyer
38	Модуль SharePoint	Raziel

39	Модуль запланированных заданий	Sam Martin
40	Наборы параметров	Bert Levrau, Poorkenny
41	Обработка ошибок	Prageeth Saravanan
42	Обработка секретов и учетных данных	4444, briantist, Ranadip Dutta, TravisEz13
43	Общие параметры	autosvet, jumbo, RamenChef
44	Оператор switch	Anthony Neace, Frode F., jumbo, oɔɯəɹ, Ranadip Dutta, TravisEz13
45	операторы	Anthony Neace, Bevo, Clijsters, Gordon Bell, JPBlanc, Mark Wragg, Ranadip Dutta
46	Операции с базовым набором	Euro Micelli, Ranadip Dutta, TravisEz13
47	Отправка электронной почты	Adam M., jimmyb, megamorf, NooJ, Ranadip Dutta, void, Yusuke Arakawa
48	Переменные в PowerShell	autosvet, Eris, Liam, Prageeth Saravanan, Ranadip Dutta, restless1987, Steve K
49	Переменные среды	autosvet
50	Повторное признание Amazon Web Services (AWS)	Trevor Sullivan
51	Поддержка с комментариями	Christophe
52	Принудительное выполнение сценариев	autosvet, Frode F., jumbo, RamenChef
53	Простая служба хранения (S3) Amazon Web Services (AWS)	Trevor Sullivan
54	Профили	Frode F., Kolob Canyon

	Powershell	
55	Псевдонимы	jumbo
56	Работа с конвейером PowerShell	Alban, Atsch, Clijsters, Deptor, James Ruskin, Keith, oɔɯəɹ, Sam Martin
57	Работа с объектами	Chris N, djwork, Mathieu Buisson, megamorf
58	Работа с файлами XML	autosvet, Frode F., Giorgio Gambino, Lieven Keersmaekers, RamenChef, Richard, Rowshi
59	Рабочие процессы PowerShell	Trevor Sullivan
60	Разбор CSV	Andrei Epure, Frode F.
61	Рассеивание Powershell	Avshalom, megamorf, Moerwald, Sam Martin, ShaneC
62	Расчетные свойства	Prageeth Saravanan
63	Регулярные выражения	Frode F.
64	Связь TCP с PowerShell	autosvet, RamenChef, Richard
65	Связь с API-интерфейсом RESTful	autosvet, Clijsters, HAL9256, kdtong, RamenChef, Ranadip Dutta, Sam Martin, YChi Lu
66	Соглашения об именах	niksofteng
67	Создание ресурсов класса DSC	Trevor Sullivan
68	Специальные операторы	TravisEz13
69	Струны	Frode F., restless1987, void
70	Сценарии подписания	AP., Frode F.

71	Управление пакетами	TravisEz13
72	Условная логика	Liam, lloyd, miken32, TravisEz13
73	Фоновые задания PowerShell	Clijsters, mattnicola, Ranadip Dutta, Richard, TravisEz13
74	Функции PowerShell	Bert Levrau, Eris, James Ruskin, Luke Ryan, niksofteng, Ranadip Dutta, Richard, TessellatingHeckler, TravisEz13, Xalorous