PowerShell Workflows

https://sergeyvasin.net/tag/workflow/

Оглавление

[PowerShell Workflows: Основы 2](#_Toc58848050)

[PowerShell Workflows: Ограничения 9](#_Toc58848051)

[**Неподдерживаемые активности Windows PowerShell** 10](#_Toc58848052)

[**Области действия переменных** 13](#_Toc58848053)

[**Десериализация объектов** 15](#_Toc58848054)

[**Командлеты, для которых не было создано активностей** 16](#_Toc58848055)

[PowerShell Workflows: Вложения 19](#_Toc58848056)

[**Заключение** 23](#_Toc58848057)

[PowerShell Workflows: Механизм заданий 24](#_Toc58848058)

[Запускаем рабочий процесс в качестве задания 24](#_Toc58848059)

[Останавливаем и запускаем рабочие процессы 25](#_Toc58848060)

[Приостанавливаем задание рабочего процесса 27](#_Toc58848061)

[Контрольные точки 29](#_Toc58848062)

[Резюме 33](#_Toc58848063)

[PowerShell Workflows: Перезагрузка компьютера 33](#_Toc58848064)

[**Перезагрузка удаленного компьютера** 33](#_Toc58848065)

[**Перезагрузка компьютера, на котором запущен рабочий процесс** 35](#_Toc58848066)

[**Перезапуск рабочего процесса вручную** 36](#_Toc58848067)

[**Автоматический перезапуск рабочего процесса** 39](#_Toc58848068)

[**Gotchas** 41](#_Toc58848069)

[**Conclusion** 41](#_Toc58848070)

[PowerShell Workflows: Используем параметры 42](#_Toc58848071)

[Рабочие процессы и удаленный доступ 44](#_Toc58848072)

[Командлеты рабочих процессов 49](#_Toc58848073)

[Резюме 52](#_Toc58848074)

[PowerShell Workflows: Вопросы структуры 53](#_Toc58848075)

[PowerShell Workflows: Пример практического использования 60](#_Toc58848076)

[PowerShell Workflows: идентификация результатов запроса удаленных компьютеров 70](#_Toc58848077)

# PowerShell Workflows: Основы

Если вы читали что-нибудь из того, что я написал за последние 6 лет, то вы знаете, что я считаю, что Windows PowerShell это лучшее из всего, что когда-либо появлялось в экосистеме Windows. С появлением 3 версии все стало еще лучше. Одной из основных сфер моих интересов в последние несколько лет был WMI, но одна вещь в Windows PowerShell 3.0 меня действительно восхищает – это рабочие процессы.

Когда я начинаю разбираться с новой технологией, я хочу знать три вещи:

1. Что это и как оно работает?

2. Почему это важно?

3. Что мне это даст?

Эта серия статей расскажет об основах рабочих процессов, в чем их отличие от того, как вы обычно использовали Windows PowerShell, и как использовать их максимально эффективно. После завершения этой серии вы сможете ответить на приведенные мной вопросы и поймете, подходят ли рабочие процессы для использования в вашей стратегии автоматизации.

Одно из определений рабочих процессов – это последовательность автоматизированных шагов или активностей, которые могут выполнять задачи или получать данных с локальной или удаленных машин.

Это определение вполне может подойти к функции или скрипту Windows PowerShell, и нельзя сказать, что вы будете полностью неправы, если так решите, но – и это весьма значительное «но» — рабочие процессы Windows PowerShell созданы для использования в сценариях, где требуются следующие атрибуты:

* Долгосрочные активности
* Повторяющиеся активности
* Часто выполняющиеся активности
* Параллельный запуск активностей на одной или нескольких машинах
* Прерываемые активности, которые могут быть остановлены и перезапущены, включая продолжение выполнения после перезагрузки машины.

Хорошо, я уже вроде объяснил вам, что рабочие процессы – это хорошая вещь – но как же они выглядят? Существует традиция, что знакомство с новым языком программирования начинается с программы «hello world». Итак, самый простой рабочий процесс будет выглядеть так.

workflow helloworld {

«Hello World»

}

Запуск этого кода создает рабочий процесс, но не выполняет его – в точности как и функция Windows PowerShell. Для эго выполнения просто наберите его имя в командной строке и нажмите «Enter». При первом запуске рабочего процесса будьте готовы к небольшой задержке перед тем как он выполнится.

Наш рабочий процесс может выглядеть как функция, начинающаяся с другого слова – но не дайте себя одурачить – самая важная вещь, которую нужно понимать это то, что рабочие процессы выглядят как Windows PowerShell, но на самом деле им не являются. Я буду повторять это утверждение по мере нашего знакомства с рабочими процессами, и после завершения этой серии статей вы будете понимать эту разницу.

Вы заметили, что Windows PowerShell 3.0 привнес новое ключевое слово – workflow. Существует еще несколько ключевых слов, имеющих отношение к рабочим процессам, о которых вам следует знать:

* Workflow
* Parallel
* Foreach –parallel
* Sequence
* InlineScript
* Checkpoint-workflow
* Suspend-workflow

Одна из ключевых функций workflow – это то, что они могут выполнять команды параллельно. Давайте рассмотрим следующий набор команд:

Get-CimInstance –ClassName Win32\_OperatingSystem

Get-Process –Name PowerShell\*

Get-CimInstance –ClassName Win32\_ComputerSystem

Get-Service –Name s\*

В каком порядке будут возвращены результаты, если вы запустите эти команды в скрипте или функции Windows PowerShell?

Если не возникнет никаких проблем, я полагаю что я увижу:

1. Данные, возвращенные классом Win32\_OperatingSystem

2. Информацию о процессе Windows PowerShell

3. Данные, возвращенные классом Win32\_ComputerSystem

4. Информацию о службах, начинающихся с буквы «s»

Теперь давайте превратим это в рабочий процесс с параллельным выполнением.

workflow paralleltest {

   parallel {

      Get-CimInstance –ClassName Win32\_OperatingSystem

      Get-Process –Name PowerShell\*

      Get-CimInstance –ClassName Win32\_ComputerSystem

      Get-Service –Name s\*

   }

}

Если вы хотите, чтобы команды выполнялись параллельно, все что вам нужно сделать – это добавить ключевое слово «parallel», и весь код, заключенный в фигурные скобки будет выполняться параллельно.

Как вы думаете, в каком порядке будут возвращаться данные?

Это вы мне скажите! Вы можете запустить этот рабочий процесс несколько раз и каждый раз данные могут быть выведены в разной последовательности!

Если вы запускаете параллельное выполнение активностей в рабочем процессе – нет никаких гарантий относительно того, в каком порядке будут возвращаться данные. Вы не можете ожидать, что одна часть данных будет возвращена перед другой.

Я несколько раз использовал словосочетание активности рабочего процесса – и я сделал это умышленно. Вспомните как я говорил, что рабочий процесс выглядит как Windows PowerShell, но на самом деле им не является. Вы можете подумать, что код, составляющий рабочий процесс – это PowerShell, но это не так.

Функциональность рабочих процессов Windows PowerShell строится на .NET Framework Windows Workflow Foundation (WWF). Большинство (однако, не все) командлетов сопоставлено активностям рабочих процессов (я коснусь несопоставленных командлетов позже). И это именно то, что вы запускали – активности рабочих процессов. Для запуска активностей, код Windows PowerShell транслируется в XAML. Если потребуется, вы даже можете импортировать XAML в Visual Studio.

Для просмотра XAML-кода наберите следующую команду:

Get-command paralletest | format-list \*

Синтаксис активностей рабочих процессов большей частью похож на тот, что мы используем при работе с командлетами, но есть и отличия — наиболее заметное из них – это параметр **‑ComputerName**. В активностях рабочих процессов он заменен на **‑PSComputerName**. Я проиллюстрирую это в следующем рабочем процессе.

workflow foreachptest {

   param([string[]]$computers)

   foreach –parallel ($computer in $computers){

      Get-WmiObject –Class Win32\_OperatingSystem –PSComputerName $computer

   }

}

Вы можете запустить этот рабочий процесс следующим образом:

foreachptest -Computers «server01», «server02», «server03»

И опять, нет никаких гарантий относительно того, в каком порядке будут возвращены данные.

А что же насчет ситуаций, когда команды должны быть выполнены в определенном порядке? Этот функционал предоставляется посредством ключевого слова «sequence». Все, что входит в блок sequence{} выполняется в том порядке, в каком вы это указали. Вы можете видоизменить последний рабочий процесс следующим образом:

workflow foreachpstest {

   param([string[]]$computers)

   foreach –parallel ($computer in $computers){

      sequence {

         Get-WmiObject -Class Win32\_ComputerSystem -PSComputerName $computer

         Get-WmiObject –Class Win32\_OperatingSystem –PSComputerName $computer

      }

   }

}

foreachpstest -Computers «server01», «server02», «server03»

В этом рабочем процессе вы параллельно обращаетесь к трем компьютерам, но команды

Get-WmiObject -Class Win32\_ComputerSystem -PSComputerName $computer

Get-WmiObject –Class Win32\_OperatingSystem –PSComputerName $computer

выполняются на каждом компьютере именно в этом порядке.

Еще не запутались? Отлично! Теперь давайте попробуем вот так.

workflow foreachpsptest {

   param([string[]]$computers)

   foreach –parallel ($computer in $computers){

      sequence {

         Get-WmiObject -Class Win32\_ComputerSystem -PSComputerName $computer

         Get-WmiObject –Class Win32\_OperatingSystem –PSComputerName $computer

         $disks = Get-WmiObject -Class Win32\_LogicalDisk `

         -Filter «DriveType = 3» –PSComputerName $computer

         foreach -parallel ($disk in $disks){

            sequence {

               $vol = Get-WmiObject -Class Win32\_Volume `

              -Filter «DriveLetter = ‘$($disk.DeviceID)'» `

              –PSComputerName $computer

              Invoke-WmiMethod -Path $($vol.\_\_PATH) -Name DefragAnalysis

           }

         }

      }

   }

}

foreachpstest -Computers «server01», «server02», «server03»

Рабочий процесс принимает список имен компьютеров и параллельно выполняет набор действий. Эти действия выполняются в указанном порядке, поскольку мы использовали ключевое слово sequence. С получением экземпляров **Win32\_ComputerSystem** и **Win32\_OperatingSystem** все понятно, поскольку вы получаете по одному экземпляру каждого класса. Затем рабочий процесс получает список логических дисков, но поскольку их несколько, мы хотим, чтобы они обрабатывались параллельно. Поэтому мы используем:

foreach -parallel ($disk in $disks)

Для каждого диска существует экземпляр класса **Win32\_Volume**. Мы используем свойство **\_\_PATH** этих экземпляров для вызова метода **DefragAnalysis**.

Обычно, вы могли бы сделать следующее:

Get-WmiObject -Class Win32\_Volume -Filter «DriveLetter = ‘C:'» |

Invoke-WmiMethod -Name DefragAnalysis

Однако, рабочие процессы работают через WSMAN (так же как и удаленное выполнение команд), и на выходе вы получаете десериализованные объекты, поэтому вы не можете передать их по конвейеру. В данном случае, чтобы обойти это ограничение, вы можете воспользоваться свойством **\_\_PATH**.

Еще одна вещь, которую, я надеюсь, вы заметили – это то, что я всегда использую имена параметров в активностях (командлетах). Это потому, что рабочие процессы не поддерживают использование позиционных параметров. Так что используйте имена параметров.

Одна из проблем, с которой сталкиваются люди при использовании WMI, это то, что форматирование по умолчанию выводит не все данные. Один из подходов к этой проблеме – это использование командлета Format-List \*. Так что вы, возможно, захотите сделать следующее:

workflow foreachptest {

   param([string[]]$computers)

   foreach –parallel ($computer in $computers){

      Get-WmiObject –Class Win32\_OperatingSystem –PSComputerName $computer | Format-List

   }

}

К сожалению, это не сработает и вы увидите сообщение об ошибке, подобное следующему:

At line:5 char:5

+ Format-List

Cannot call the ‘Format-List’ command. Other commands from this module have been packaged as workflow activities, but this command was specifically excluded. This is likely because the command requires an interactive Windows PowerShell session, or has behavior not suited for workflows. To run this command anyway, place it within an inline-script (InlineScript {Format-List }) where it will be invoked in isolation.

   + CategoryInfo : ParserError: (:) [], ParseException

   + FullyQualifiedErrorId : CommandActivityExcluded

Как нам объясняет сообщение об ошибке — **Format-List** – это один из командлетов, который не был реализован в качестве активности рабочих процессов. Также это сообщение рекомендует нам использовать **InlineScript**, что по сути является скриптблоком Windows PowerShell внутри рабочего процесса. Вы можете использовать его следующим образом:

workflow foreachpitest {

   param([string[]]$computers)

   foreach –parallel ($computer in $computers){

      InlineScript {

         Get-WmiObject –Class Win32\_OperatingSystem –ComputerName $using:computer | Format-List

      }

   }

}

В этом рабочем процессе следует отметить две вещи. Первое, он снова использует параметр **–ComputerName**. Это потому, что внутри **InlineScript** находится командлет, а не активность рабочего процесса. Второе – это то, каким образом передается переменная **$computer** — **$using:computer**. Это вследствие того, как организованы области (scopes) рабочих процессов. Более полно я объясню это позже, но сейчас запомните – если вы собираетесь использовать в блоке **InlineScript** переменную, определенную выше в структуре рабочего процесса, вам нужно использовать модификатор **$using**.

Рабочие процессы, любите вы их или ненавидите, они уже пришли. В этой вводной статье я показал вам, что, хотя рабочие процессы выглядят как Windows PowerShell, на самом деле все же от него отличаются. Также вы познакомились со следующими ключевыми словами:

* Parallel
* Foreach –parallel
* Sequence
* InlineScript

В следующий раз мы рассмотрим некоторые ограничения рабочих процессов, а также то, как их преодолеть.

~Richard.

# PowerShell Workflows: Ограничения

В прошлом посте я рассказал об основах рабочих процессов Windows PowerShell. Я также коснулся нескольких ограничений рабочих процессов. Думаю, пришла пора рассмотреть их поближе и определить обходные пути для их преодоления.

Ограничения рабочих процессов можно разделить на следующие группы:

* Области действия переменных
* Десериализованные объекты
* Командлеты, для которых не было создано активностей
* Неподдерживаемые активности Windows PowerShell

**Неподдерживаемые активности Windows PowerShell**

Некоторые ключевые слова и используемые техники Windows PowerShell не поддерживаются в рабочих процессах. Они приведены в следующей таблице.

|  |  |  |
| --- | --- | --- |
| **Неподдерживаемые техники Windows PowerShell** | | |
| Begin,Process,End | Break,Continue | Подвыражения |
| Множественное назначение переменных | Изменение переменных цикла | Динамические параметры |
| Задание свойств | Оператор точка | Расширенная валидация   параметров |
| Позиционные параметры | Выражение Switch | Выражение Trap |
| Встроенная справка | Задание дисковых переменных | Вызов методов из объектов |
| Единственнное #requires |  |  |

Еще одно ограничение которое мне встречалось, это использование пользовательских дисков Windows PowerShell – если они присутствуют, рабочие процессы могут перестать выполняться.

Некоторые их этих ограничений, например использование выражения **trap**, не такая уж большая проблема. Вместо него вы можете использовать **try-catch**. Другие ограничения приносят больше неудобств, например невозможность использовать выражение **switch**. Если вы попробуете использовать **switch** в рабочем процессе:

workflow testswitch {

   param (

   [string]$os

   )

   switch ($os) {

      «XP» {«Time to upgrade»}

      «Windows7» {«OK — but not the lastest»}

      «Windows 8» {«Latest and greatest»}

   }

}

то вы получите море красных сообщений:

At line:5 char:2

+ switch ($os) {

Case-insensitive switch statements are not supported in a Windows PowerShell workflow. Supply the -CaseSensitive flag, and ensure that case clauses are written appropriately. To write a case-insensitive case statement, first convert the input to either uppercase or lowercase, and update the case clauses to match.

   + CategoryInfo : ParserError: (:) [], ParentContainsErrorRecordException

   + FullyQualifiedErrorId : SwitchCaseSensitive

Прочитав сообщение об ошибке вы можете подумать, что можно сделать следующее:

workflow testswitchi {

   param (

   [string]$os

   )

   switch -CaseSensitive ($os.ToUpper()) {

      «XP» {«Time to upgrade»}

      «WINDOWS7» {«OK — but not the lastest»}

      «WINDOWS 8» {«Latest and greatest»}

   }

}

Нет, опять ошибка.

At line:5 char:25

+ switch -CaseSensitive ($os.ToUpper()) {

+ ~~~~~~~~~~~~~

Method invocation is not supported in a Windows PowerShell Workflow. To use .NET Framework scripting, place your commands in an inline script:

InlineScript {

}.

   + CategoryInfo : ParserError: (:) [], ParentContainsErrorRecordException

   + FullyQualifiedErrorId : MethodInvocationNotSupported

Ну что ж, воспользуемся **InlineScript**.

workflow testswitch {

   param (

  [string]$os

  )

  InlineScript {

    switch ($using:os) {

      «XP» {«Time to upgrade»}

      «Windows7» {«OK — but not the lastest»}

      «Windows 8» {«Latest and greatest»}

    }

  }

}

В этом случае все работает так, как от него ожидается.

Еще одно ограничение, невозможность использования блоков **BEGIN**, **PROCESS** и **END**, которые вы можете использовать в функциях, затрагивает вещи, которые просто не укладываются в концепцию рабочих процессов. Другое ограничение, оказывающее значительное влияние – это невозможность использования справки на основе комментариев. Вы должны использовать XML-файлы. Существует несколько способов генерации этих файлов, один из наиболее простых – это [использовать InfoPath, как описывает James O’Neill](http://blogs.technet.com/b/jamesone/archive/2009/07/24/powershell-on-line-help-a-change-you-should-make-for-v2-3-and-how-to-author-maml-help-files-for-powershell.aspx).

Невозможность валидации параметров может стать значительным неудобством, если ваши рабочие процессы используют большое их число. Для этого ограничения не существует обходных путей кроме написания кода валидации в самом рабочем процессе.

Вы уже видели выражение $using:computer в предыдущем посте и $using:os в скрипте, касающемся использования выражения **switch**. Этот синтаксис используется для преодоления ограничений, относящихся ко второй группе – области действия переменных.

**Области действия переменных**

В рабочих процессах существуют следующие ограничения на использование переменных:

* Переменные, определенные в верхней области видны в нижних областях, но не видны в блоках InlineScript
* Невозможно определить переменную в нижней области с именем, совпадающим с переменной в верхней области
* Если определить или переопределить переменную, вы можете вы можете без всяких проблем использовать ее в этой области
* Не существует области $global
* Для использования в блоке InlineScript переменных, определенных в верхней области, нужно указывать «$using»
* Изменение в блоке InlineScript переменной, определенной в верхней области требует использования временной переменной
* Для изменения переменных, определенных в верхней области нужно использовать «$workflow»
* Нельзя использовать подвыражения

Это может показаться несколько запутанным, но демонстрация должна внести некоторую ясность. Я рекомендую вам запустить этот код и рассмотреть полученные результаты.

workflow demo-scope {

   # This is a workflow top-level variable

  $a = 22

  «Initial value of A is: $a»

  # Access $a from Inlinescript (bringing a workflow variable to the Windows PowerShell session) using $using

  inlinescript {«PowerShell variable A is: $a»}

  inlinescript {«Workflow variable A is: $using:a»}

  ## changing a variable value

  $a = InlineScript {$b = $Using:a+5; $b}

  «Workflow variable A after InlineScript change is: $a»

  parallel {

    sequence {

      # Reading a top-level variable (no $workflow: needed)

      «Value of A inside parallel is: $a»

      # Updating a top-level variable with $workflow:

      $workflow:a = 3

    }

  }

  «Updated value of A is: $a»

}

demo-scope

После запуска рабочего процесса вы должны увидеть нечто подобное.

Initial value of A is: 22

PowerShell variable A is:

Workflow variable A is: 22

Workflow variable A after InlineScript change is: 27

Value of A inside parallel is: 27

Updated value of A is: 3

Рабочий процесс начинается с определения переменной, $a=22, затем он выводит ее значение. Если вы попытаетесь получить содержимое переменной, определенной в верхней области, из блока **InlineScript**, ничего не получится, как показано во второй строке вывода. Для доступа к переменной вам нужно использовать **$using**. Если вам нужно изменить значение этой переменной, то для этого потребуется использовать вторую переменную и затем возвратить новое значение.

$a = InlineScript {$b = $Using:a+5; $b}

Вывод показывает, что теперь переменная обладает значением 27.

Доступ к переменной из блока **parallel** происходит без каких-либо сложностей. Если вам нужно изменить значение этой переменной в блоке **parallel**, потребуется указать **$workflow**.

Подводя итоги работы с переменными: не переусложняйте скрипт и будьте осторожны.

**Десериализация объектов**

Третье ограничение заключается в том, что объекты в рабочих процессах десериализованы. Это сродни с удаленным выполнением команд в Windows PowerShell (что неудивительно, так как рабочие процессы используют тот же транспортный механизм). Можно заметить что десериализация объектов встречается все чаще и чаще. Области, где она встречается:

* Удаленная работа в Windows PowerShell
* Удаленный доступ посредством CIM-сессий
* Командлеты WSMAN
* Рабочие процессы

Возможно это становится стандартным подходом к работе в Windows PowerShell.

Десериализованный объект обладает свойствами объекта, но не его методами. Другими словами, он инертный. Огромная часть кода Windows PowerShell делает нечто подобное:

$string = «abcde»

$string.ToUpper()

$os = Get-WmiObject -Class Win32\_OperatingSystem

$os.ConvertToDateTime($os.LastBootUpTime)

Однако в случае с рабочими процессами это не сработает.

workflow test2 {

   $string = «abcde»

  $string.ToUpper()

  $os = Get-WmiObject -Class Win32\_OperatingSystem

  $os.ConvertToDateTime($os.LastBootUpTime)

}

При попытке запуска появится сообщение об ошибке, указывающее, что вызов методов не поддерживается. Однако, вы можете сделать следующее:

workflow test2 {

  inlinescript {

    $string = «abcde»

    $string.ToUpper()

    $os = Get-WmiObject -Class Win32\_OperatingSystem

    $os.ConvertToDateTime($os.LastBootUpTime)

  }

}

Не пытайтесь использовать переменные, содержащие объекты из верхней области действия, для вызова методов – это не сработает.

Подводя итоги, касательно десериализованных объектов – если вам нужно вызвать метод объекта, делайте это из блока **InlineScript**. Помните, что вы можете вкладывать блоки **InlineScript** в блоки **Parallel**.

**Командлеты, для которых не было создано активностей**

Последнее ограничение касается командлетов, которые не для которых не было создано соответствующих активностей. Как вы помните, в предыдущем посте нам встретилась проблема использования командлетов **Format\*** в рабочих процессах. Это можно обойти, используя **InlineScript**.

workflow foreachpitest {

   param([string[]]$computers)

  foreach –parallel ($computer in $computers){

    InlineScript {

      Get-WmiObject –Class Win32\_OperatingSystem –ComputerName $using:computer |

      Format-List

    }

  }

}

Другие командлеты, которые не могут быть использованы в рабочих процессах приведены в таблице. Вы можете использовать их внутри InlineScript.

|  |  |
| --- | --- |
| **Неподдерживаемые командлеты (группы)** | **Причина** |
| \*Alias, \*FormatData, \*History, \*Location, \*PSDrive, \*Transcript, \*TypeDate, \*Variable, Connect/Disconnect-Wsman | Вносят изменения только в текущую сессию, поэтому не требуются в рабочих процессах |
| Show-Command, Show-ControlPanelItem, Get-Credential, Show-EventLog, Out-Gridview, Read-Host, Debug-Process | Рабочие процессы не поддерживают интерактивные командлеты |
| \*BreakPoint, Get-PSCallStack, Set-PSDebug | Рабочие процессы не поддерживают отладку скриптов |
| \*Transaction | Рабочие процессы не поддерживают транзакции |
| Format\* | Отсутствует поддержка форматирования |
| \*PSsession, \*PSsessionoption | Удаленный доступ контролируется рабочим процессом |
| Export-Console,Get-ControlPanelItem, Out-Default, Out-Null, Write-Host, Export-ModuleMember, Add-PSSnapin, Get-PSSnapin, Remove-PSSnapin, Trace-Command |  |

Ниже приведена таблица с командлетами, которые поддерживают только локальное выполнение, при использовании их в рабочем процессе.

|  |  |  |  |
| --- | --- | --- | --- |
| Add-Member | Compare-Object | ConvertFrom-Csv | ConvertFrom-Json |
| ConvertFrom-StringData | Convert-Path | ConvertTo-Csv | ConvertTo-Html |
| ConvertTo-Json | ConvertTo-Xml | ForEach-Object | Get-Host |
| Get-Member | Get-Random | Get-Unique | Group-Object |
| Measure-Command | Measure-Object | New-PSSessionOption | New-PSTransportOption |
| New-TimeSpan | Out-Default | Out-Host | Out-Null |
| Out-String | Select-Object | Sort-Object | Update-List |
| Where-Object | Write-Debug | Write-Error | Write-Host |
| Write-Output | Write-Progress | Write-Verbose |  |

Если вам нужно запустить их удаленно, как вы наверное уже догадались, используйте **InlineScript**.

Узнать больше об ограничениях рабочих процессов вы можете в файлах справки:

about\_ActivityCommonParameters

about\_Checkpoint-Workflow

about\_Foreach-Parallel

about\_InlineScript

about\_Parallel

about\_Sequence

about\_Suspend-Workflow

about\_WorkflowCommonParameters

about\_Workflows

Если вы только запустили PowerShell, вы можете их не увидеть. Импортируйте модуль PSWorkflow и они станут доступными.

В этой статье я попытался рассказать об ограничениях, с которыми вы можете встретиться, используя рабочие процессы. Как я сказал в предыдущей статье, «[PowerShell Workflows: Основы](https://sergeyvasin.wordpress.com/2013/01/03/powershell-workflows-the-basics/)», если вы не знаете об ограничениях, вы можете здорово о них споткнуться.

# PowerShell Workflows: Вложения

Пока что я показывал вам, как использовать рабочие процессы по одному в каждый момент времени. Если же вы задумаетесь на минуту, как вы обычно используете WindowsPowerShell, вы вероятно скажете, что вы создаете несколько функций, которые затем используете повторно, вызывая их из других функций и скриптов. Сама концепция повторного использования кода должна пронизывать все ваши скрипты, так, чтобы повысить отдачу от затраченного времени и усилий на написание кода.

Техники повторного использования кода, которые вы могли бы использовать мы рассмотрим в другой раз. Сейчас же мы сконцентрируемся на том, как функции и рабочие процессы могут быть использованы в других рабочих процессах.

Вопрос о том, как лучше строить рабочие процессы мы рассмотрим в отдельной статье. Перед этим, вам стоит узнать о механизмах, на которые вы можете опираться при повторном использовании кода. Их можно разделить на следующие три группы:

* **Рабочие процессы PowerShell** – (возможна интеграция с рабочими процессами, созданными в Visual Studio, но, поскольку это используется весьма небольшим числом администраторов, мы не будем рассматривать этот вопрос в данной серии статей)
* **Функции PowerShell**– либо в том же самом скрипте, что и рабочий процесс, либо через использование модулей WindowsPowerShell.
* **Скрипты PowerShell**– на локальной или удаленной машине.

Давайте посмотрим, как рабочие процессы могут взаимодействовать с другими рабочим процессами на примере администрирования Active Directory. Рекомендуется очищать неиспользуемые учетные записи в Active Directory. И обычно вы просматриваете Active Directory на предмет отключенных или просроченных аккаунтов, а также учетных записей, со включенным параметром Password Never Expires.

Для определения отключенных учетных записей, запустите следующий скрипт:

Search-ADAccount -AccountDisabled |

Select-Object -Property DistinguishedName |

Export-Csv -Path c:\ADReports\DisabledAccounts.csv –NoTypeInformation

Для просроченных аккаунтов:

Search-ADAccount -AccountExpired |

Select-Object -Property DistinguishedName |

Export-Csv -Path c:\ADReports\ExpiredAccounts.csv –NoTypeInformation

И для учетных записей, пароль которых не устаревает:

Search-ADAccount -PasswordNeverExpires |

Select-Object -Property DistinguishedName |

Export-Csv -Path c:\ADReports\PsswdNeverExpireAccounts.csv –NoTypeInformation

Эти три скрипта должны быть знакомы администраторам Active Directory. Их использовании гораздо более эффективно чем нахождение учетных записей вручную. Но в этом случае, они выполняются последовательно. Могут ли нам помочь рабочие процессы, чтобы выполнить эти три команд параллельно?

При самом простом подходе, эти три скрипта помещаются в один рабочий процесс:

workflow get-ADReport {

 parallel {

 Search-ADAccount -AccountDisabled |

 Select-Object -Property DistinguishedName |

 Export-Csv -Path c:\ADReports\DisabledAccounts.csv -NoTypeInformation

 Search-ADAccount -AccountExpired |

  Select-Object -Property DistinguishedName |

  Export-Csv -Path c:\ADReports\ExpiredAccounts.csv -NoTypeInformation

 Search-ADAccount -PasswordNeverExpires |

  Select-Object -Property DistinguishedName |

  Export-Csv -Path c:\ADReports\PsswdNeverExpireAccounts.csv -NoTypeInformation

  }

 }

Этот рабочий процесс работает вполне неплохо, практически одновременно выдавая три csv-файла. Его недостаток в том, что вы не можете выполнить поиск по какому-то одному атрибуту – вы всегда будете получать информацию о всех трех типах аккаунтов.

Здесь есть пара вариантов. Первый – вы можете создать отдельный рабочий процесс для каждого скрипта и вложить их в один общий рабочий процесс.

workflow get-ADReport {

   workflow get-disabled {

    Search-ADAccount -AccountDisabled |

    Select-Object -Property DistinguishedName |

    Export-Csv -Path c:\ADReports\DisabledAccounts.csv -NoTypeInformation

   }

  workflow get-expired {

    Search-ADAccount -AccountExpired |

    Select-Object -Property DistinguishedName |

    Export-Csv -Path c:\ADReports\ExpiredAccounts.csv -NoTypeInformation

   }

  workflow get-passwordneverexpire {

    Search-ADAccount -PasswordNeverExpires |

    Select-Object -Property DistinguishedName |

    Export-Csv -Path c:\ADReports\PsswdNeverExpireAccounts.csv -NoTypeInformation

   }

  parallel {

    get-disabled

    get-expired

    get-passwordneverexpire

   }

 }

Это загрузит каждый рабочий процесс в память и вы сможете использовать их отдельно.

Более простой способ, который также упрощает обслуживание – это вывести отдельные рабочие процессы из общего рабочего процесса:

workflow get-disabled {

  Search-ADAccount -AccountDisabled |

  Select-Object -Property DistinguishedName |

  Export-Csv -Path c:\ADReports\DisabledAccounts.csv -NoTypeInformation

 }

workflow get-expired {

  Search-ADAccount -AccountExpired |

  Select-Object -Property DistinguishedName |

  Export-Csv -Path c:\ADReports\ExpiredAccounts.csv -NoTypeInformation

 }

workflow get-passwordneverexpire {

  Search-ADAccount -PasswordNeverExpires |

  Select-Object -Property DistinguishedName |

  Export-Csv -Path c:\ADReports\PsswdNeverExpireAccounts.csv -NoTypeInformation

 }

workflow get-ADReport {

  parallel {

    get-disabled

    get-expired

    get-passwordneverexpire

  }

}

Следующим шагом может быть помещение каждого рабочего процесса в отдельный файл и создание файла .psm1 для загрузки их в качестве модуля. Таким образом вы можете расширить функциональность без необходимости изменения кода. Чуть раньше я показывал, как запускать эти рабочие процессы параллельно. Приведенный подход позволит вам при необходимости запустить их последовательно. Это может пригодиться, если вам, например, сначала нужно создать учетные записи, а затем создать для них почтовые ящики.

Функции используются похожим образом.

workflow get-computersystem {

 param([string[]]$computerName)

 function get-fcomputersystem {

 param ([string]$fcomputer)

 Get-WmiObject -Class Win32\_ComputerSystem -ComputerName $fcomputer

}

# The contents of the foreach block will be executed in parallel

  foreach -parallel($computer in $computerName) {

     if (Test-Connection -ComputerName $computer -Quiet -Count 1) {

         get-fcomputersystem -fcomputer $computer

     }

     else {

       «$computer unreachable»

     }

  }

}

get-computersystem -computerName $env:COMPUTERNAME

В рабочем процессе список имен компьютеров передается посредством параметра **ComputerName**. Выражение **foreach –parallel** используется для обработки этого списка. **Test-Connection** используется для проверки доступности компьютера, и если он отвечает, вызывается функция.

В этом случае функция определена внутри рабочего процесса. Вы также можете определить ее за пределами рабочего процесса. Кроме того, вы можете поместить функцию в отдельный скрипт и загрузить ее и рабочий процесс в качестве модуля.

Важный момент состоит в том, что функции и рабочие процессы должны быть загружены или определены до того, как вы их вызываете.

Скрипты – это третий и последний метод повторного использования кода. Поместим скрипты для работы с Active Directory, приведенные в начале статьи в отдельные файлы.

Назовем их:

get-disabledaccount.ps1

get-expiredaccount.ps1

get-passwordNexpire.ps1

Для того, чтобы выполнить их параллельно, можно попробовать запустить следующий скрипт:

workflow get-ADReport {

 parallel {

   c:\adreports\get-disabledaccount.ps1

   c:\adreports\get-expiredaccount.ps1

   c:\adreports\get-passwordNexpire.ps1

 }

}

С сожалению, попытка его выполнения вызовет ошибку:

At line:3 char:4

+    c:\adreports\get-disabledaccount.ps1

+    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Cannot find the ‘c:\adreports\get-disabledaccount.ps1’ command. If this

command is defined as a workflow, ensure it is defined before the workflow

that calls it. If it is a command intended to run directly within Windows

PowerShell (or is not available on this system), place it in an InlineScript:

‘InlineScript { c:\adreports\get-disabledaccount.ps1 }’

    + CategoryInfo          : ParserError: (:) [], ParseException

    + FullyQualifiedErrorId : CommandNotFound

Тогда мы можем попробовать так:

workflow get-ADReport {

 inlinescript {

   c:\adreports\get-disabledaccount.ps1

   c:\adreports\get-expiredaccount.ps1

   c:\adreports\get-passwordNexpire.ps1  }

}

Это сработает, но что насчет параллельного выполнения? Для этого нужно запустить каждый скрипт отдельно.

workflow get-ADReport {

 parallel {

  inlinescript {c:\adreports\get-disabledaccount.ps1}

  inlinescript {c:\adreports\get-expiredaccount.ps1}

  inlinescript {c:\adreports\get-passwordNexpire.ps1  }

}

}

Каждая секция InlineScript будет выполняться параллельно.

Но что, если мы захотим запустить скрипты, находящиеся на удаленной машине?

Положите скрипты в папку C:\ADReports на удаленной машине и запустите рабочий процесс следующим образом:

PS> get-ADReport -PSComputerName dc02

Скрипты будут запущены на удаленной машине и, поскольку мы не изменяли их, выходные файлы будут созданы там же. Все рабочие процессы обладают несколькими параметрами. Подробнее о них прочитать можно в файле справки.

Get-Help about\_WorkflowCommonParameters

**Заключение**

Как вы видели в прошлых статьях, рабочие процессы Windows PowerShell могут содержать несколько элементов:

* Активности рабочих процессов
* Языковые конструкции рабочих процессов, в частности:

                 Parallel

                 Foreach –parallel

* Скрипты PowerShell (InlineScript), включающие командлеты и языковые конструкции, для которых не было создано активностей рабочих процессов.

В этой статье было показано как повторно использовать код с применением:

* Рабочих процессов PowerShell
* Функций PowerShell
* Скриптов PowerShell на локальной и удаленных машинах

Как вы видите, рабочие процессы похожи на WindowsPowerShell, но все же отличаются от него. Они достаточно универсальные, чтобы вы могли использовать их в любой задаче, однако для этого потребуется потратить некоторые усилия.

Одно из серьезных достоинств рабочих процессов PowerShell – это то, что их можно останавливать и перезапускать. Это потому, что они используют механизм задач (Job) PowerShell. Это именно то, о чем мы поговорим в следующий раз, включая сохранение состояния рабочих процессов, а также их остановку и перезапуск.

# PowerShell Workflows: Механизм заданий

**Резюме**: Microsoft PowerShell MVP, Richard Siddaway продолжает свою серию статей рассказом о механизме заданий (job engine).

Microsoft Scripting Guy, Ed Wilson на связи. Сегодня я публикую четвертую часть в серии статей о рабочих процессах Windows PowerShell за авторством Windows PowerShell MVP и [Honorary Scripting Guy](http://blogs.technet.com/b/heyscriptingguy/archive/2012/11/04/announcing-the-2012-honorary-scripting-guys.aspx)Richard Siddaway.

**Заметка**: Первая статья, [PowerShell Workflows: Основы](https://sergeyvasin.wordpress.com/2013/01/03/powershell-workflows-the-basics/)рассказывает об основных концепциях рабочих процессов. Вторая, [PowerShell Workflows: Ограничения](https://sergeyvasin.wordpress.com/2013/01/06/powershell-workflows-restrictions/), рассматривает ограничения, с которыми вы можете столкнуться при написании рабочих процессов. Третья статья, [PowerShell Workflows: Вложения](https://sergeyvasin.wordpress.com/2013/02/05/powershell-workflows-nesting/), рассказывает о вложении рабочих процессов. Перед тем как читать эту статью, вам стоит ознакомиться с предыдущими.

Richard Siddaway написал [несколько гостевых постов](http://blogs.technet.com/b/heyscriptingguy/archive/tags/richard+siddaway/), кроме этого он является автором двух книг по Windows PowerShell. Последняя его книга, [PowerShell in Depth](http://www.manning.com/jones2/)написана в соавторстве двумя другими MVP – Don Jones и Jeffrey Hicks.

Теперь слово тебе, Richard.

Давайте вспомним первый рабочий процесс в этой серии статей – “Hello World”.

workflow hello {

«Hello World»

}

Вы могли запустить его следующим образом:

PS> hello

Hello World

### ****Запускаем рабочий процесс в качестве задания****

Все рабочие процессы обладают способностью запускаться с качестве задания (Job) Windows PowerShell:

PS> hello -AsJob -JobName w1

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

9 w1 PSWorkflowJob NotStarted True localhost

Вы можете указать параметр**–AsJob**, который запускает рабочий процесс в виде задания, а также вы можете использовать параметр **–JobName**, который позволяет вам указать имя для задания. Заметьте, что в других командлетах, которые вы можете запустить в качестве задания, например, командлетах WMI, вы не можете указать имя для задания.

Еще одна важная деталь, это **PSJobTypeName** – он равен **PSWorkflowJob**. Это новая категория заданий, введенная в Windows PowerShell 3.0 специально для рабочих процессов.

После того, как вы запустите рабочий процесс в качестве задания, с ним можно взаимодействовать как с любым другим заданием.

PS> Get-Job

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

9 w1 PSWorkflowJob Completed True localhost

PS> Receive-Job -Name w1

Hello World

По определению, рабочие процессы – это достаточно длительные задания, интерактивное взаимодействие с которыми не требуется. Поэтому они идеально подходят для выполнения в качестве заданий Windows PowerShell.

Естественно это не все, что касается рабочих процессов и их взаимодействия с механизмом заданий Windows PowerShell – об этом я мог бы писать бесконечно.

### ****Останавливаем и запускаем рабочие процессы****

Рабочие процессы построены на основе заданий Windows PowerShell. Благодаря этому вам доступна такая возможность как приостанавливать и возобновлять рабочие процессы. Давайте сымитируем долгосрочный рабочий процесс:

workflow test-wfsuspension {

Start-Sleep -seconds 10

Suspend-Workflow

Get-ChildItem

}

Рабочий процесс подождет 10 секунд, а затем приостановит свое выполнение вызовом **Suspend-Workflow**.

После того как рабочий процесс приостановлен, вы можете увидеть нечто подобное:

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

7 Job7 PSWorkflowJob Suspended True localhost

Заметьте, что **State** установлен в **Suspended**. Данные и состояние рабочего процесса были сохранены на диск.

Вы можете возобновить выполнение рабочего процесса командлетом **Resume-Job**:

PS> Resume-Job -Id 7

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

7 Job7 PSWorkflowJob Suspended True localhost

PS> Get-Job

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

7 Job7 PSWorkflowJob Completed True localhost

Данные, выводимые командлетом **Resume-Job** могут сбить вас с толку. В качестве состояния он показывает Suspended. Однако, командлет сообщает о статусе задания до того, как он попытается возобновить его. После его возобновления задание завершает свою работу и его выходные данные становятся доступными как обычно.

Это полезное свойство, но куда более полезной является возможность приостановить задание рабочего процесса, а затем возобновить его из другой сессии Windows PowerShell! Если вы удалите старые задания и перезапустите **test-wfsuspension**, вы получите нечто подобное:

PS> test-wfsuspension

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

9 Job9 PSWorkflowJob Suspended True localhost

Я запускаю эту демонстрацию в ISE, однако это будет работать и в консоли Windows PowerShell.

А теперь закройте PowerShell ISE и не сохраняйте никаких файлов. Да, верно. Просто нажмите на маленькую кнопку в верхнем правом углу консоли PowerShell или ISE.

Теперь откройте новую сессию (вам нужно будет запустить ее в повышенными правами). Я снова запущу ISE, но вы можете воспользоваться и консолью PowerShell.

Теперь запускаем командлет **Get-Job** и… ничего не происходит.

Так, без паники. Импортируйте модуль рабочих процессов и вы снова увидите ваше задание.

PS> Get-Job

PS> Import-Module PSWorkflow

PS> Get-Job

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

8 Job9 PSWorkflowJob Suspended True localhost

Вы можете возобновить задание и после его завершения получить выдаваемые им данные.

Стоит заметить, что Job ID может измениться, хотя имя задания останется прежним.

### ****Приостанавливаем задание рабочего процесса****

Есть еще один способ приостановить рабочий процесс – используя командлет **Suspend-Job**. Начнем с создания рабочего процесса:

workflow test-wfsr {

Start-Sleep -seconds 30

Checkpoint-Workflow

Get-ChildItem

}

Запустим рабочий процесс как задание:

PS> test-wfsr -AsJob

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

11 Job11 PSWorkflowJob Running True localhost

PS> Suspend-Job -Id 11

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

11 Job11 PSWorkflowJob Suspending True localhost

После этого вы можете использовать командлет **Suspend-Job**, чтобы приостановить задание.

Задание будет находиться в состоянии **Suspending**, пока рабочий процесс не дойдет до инструкции **CheckPoint-Workflow**. После этого, статус задания изменится на **Suspended**.

PS> Get-Job

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

11 Job11 PSWorkflowJob Suspended True localhost

Затем вы можете возобновить задание:

PS> Resume-Job -id 11

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

11 Job11 PSWorkflowJob Suspended True localhost

PS> Get-Job

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

11 Job11 PSWorkflowJob Completed True localhost

В данном случае я возобновил задание в той же сессии.

Важная вещь: командлеты **Suspend-Job** и **Resume-Job** работают только с заданиями рабочих процессов. При попытке приостановить любое другое задание вы получите ошибку.

В последнем примере вам нужно было использовать активность **Chekpoint-Workflow**. Эта активность записывает копию данных и состояния рабочего процесса на диск, чтобы задание могло быть возобновлено. Другими словами, она создает снимок. Если не сделать этого, рабочий процесс не сможет приостановиться и проигнорирует команду приостановления.

Итак, мы рассмотрели две техники приостановления:

* Если вы хотите приостановить выполнение изнутри рабочего процесса – используйте **Suspend-Workflow**.
* Если команда на приостановку должна прийти извне – используйте **Checkpoint-Workflow** и **Suspend-Job**.

### ****Контрольные точки****

Контрольные точки (checkpoints) – это отличный способ сохранения текущего состояния рабочего процесса, чтобы можно было возобновить его в случае, если прервется сессия или что-то случится с компьютером. Вы можете возобновить выполнение рабочего процесса только с последней контрольной точки – вы не можете выбрать контрольную точку для использования.

Данные контрольных точек сохраняются в вашем пользовательском профиле на машине, с которой вы запустили рабочий процесс. Хотя вы можете создавать контрольные точки после каждой активности, рекомендуется соблюдать некий баланс между временем, необходимым для записи данных и состояния рабочего процесса на диск и временем работы самого рабочего процесса.

Вы уже видели, как используется **Checkpoint-Workflow**. Кроме него есть еще несколько способов создания контрольных точек:

* **Checkpoint-Workflow** – может использоваться после каждой активности, но не внутри блока **InlineScript**. Немедленно создает контрольную точку.
* **PSPersist** (параметр рабочего процесса) – создает контрольные точки перед началом выполнения рабочего процесса, после его завершения, а также после каждой активности. Не оказывает влияния на явным образом определенные контрольные точки рабочего процесса.
* **PSPersist** (параметр активности) – создает снимок после завершения активности. Не применяется к выражениям или командам внутри блока **InlineScript**.
* **$PSPersistPreference** (привилегированная переменная) – при установке в true, контрольная точка создается после каждой активности, пока переменная не будет установлена в false. Переменная оказывает влияние только на рабочие процессы.

Если активность находится внутри конвейера, контрольная точка не создается до тех пор, пока не завершится выполнение конвейера. Внутри блока **parallel**, контрольная точка не создается, пока параллельная обработка не завершится для всех данных, в отличие от блока **sequence**, где контрольные точки создаются после каждой активности.

Эти правила лучше проиллюстрировать на примере. Используя уже знакомую вам активность **Checkpoint-Workflow** вы можете создавать контрольные точки после любой активности:

workflow test-wfchkpnt {

Get-WmiObject -Class Win32\_ComputerSystem

Checkpoint-Workflow

Get-WmiObject -Class Win32\_OperatingSystem

Checkpoint-Workflow

Get-WmiObject -Class Win32\_LogicalDisk

Checkpoint-Workflow

}

В следующем примере тот же результат, т.е. создание контрольной точки после каждой активности достигается использованием параметра рабочего процесса **–PSPersist**.

workflow test-wfchkpnt {

Get-WmiObject -Class Win32\_ComputerSystem

Get-WmiObject -Class Win32\_OperatingSystem

Get-WmiObject -Class Win32\_LogicalDisk

}

test-wfchkpnt –PSPersist

В следующем примере параметр активности **–PSPersist** используется для создания контрольной точки после каждой активности.

workflow test-wfchkpnt {

Get-WmiObject -Class Win32\_ComputerSystem -PSPersist

Get-WmiObject -Class Win32\_OperatingSystem -PSPersist

Get-WmiObject -Class Win32\_LogicalDisk -PSPersist

}

Подобного результата можно добиться и использованием привилегированной переменной (preference variable) **$PSPersistPreference**.

workflow test-wfchkpnt {

$pspersistpreference = $true

Get-WmiObject -Class Win32\_ComputerSystem

Get-WmiObject -Class Win32\_OperatingSystem

Get-WmiObject -Class Win32\_LogicalDisk

$pspersistpreference = $false

}

Лично я предпочитаю использовать **Checkpoint-Workflow** для явного указания мест, где мне нужно создать контрольную точку.

Давайте посмотрим как это работает. Этот простой рабочий процесс создает контрольную точку после каждой итерации цикла.

workflow test-wfchkpnt {

$i = 0

while ($true){

$i++

$i

Checkpoint-Workflow

}

}

Вам нужно запустить рабочий процесс в качестве задания.

PS> test-wfchkpnt -AsJob

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

7 Job7 PSWorkflowJob Running True localhost

PS> Get-Job

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

7 Job7 PSWorkflowJob Running True localhost

После того, как задание проработает несколько секунд, закройте Windows PowerShell.

Откройте новую сессию (с повышенными привилегиями), импортируйте модуль **PSWorkflow** и просмотрите существующие задания.

PS> Import-Module PSWorkflow

PS> Get-Job

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

8 Job7 PSWorkflowJob Suspended True localhost

PS> Resume-Job -Id 8

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

8 Job7 PSWorkflowJob Suspended True localhost

PS> Stop-Job -Id 8

PS> Receive-Job -Id 8 -Keep | select -f 3

WARNING: The workflow job «Job7» was stopped. Receive-Job is only displaying par

tial results.

1

2

3

Возобновите задание и дайте ему проработать несколько секунд. Так как наш рабочий процесс – это бесконечный цикл, нам потребуется остановить выполнение задания вручную. Для получения данных введите **Receive-Job**.

### ****Резюме****

Рабочие процессы Windows PowerShell работают с механизмом задач Windows PowerShell. Вы можете запускать рабочие процессы в качестве заданий. Рабочие процессы предоставляют возможность приостанавливать и возобновлять их задания. Создание контрольной точки сохраняет состояние рабочего процесса, что дает возможность возобновить его работу после закрытия сессии Windows PowerShell.

# [PowerShell Workflows: Перезагрузка компьютера](https://sergeyvasin.net/2013/03/14/powershell-workflows-restarting-the-computer/)

Как вы знаете из предыдущей статьи, рабочие процессы Windows PowerShell могут приостанавливаться и возобновляться. В определенных обстоятельствах, рабочие процессы могут безболезненно пережить завершение сессии Windows PowerShell. В этой статье вы узнаете, как рабочие процессы могут пережить перезагрузку компьютера.

Перезагрузка компьютера может потребоваться, например, в следующих случаях:

* Изменение имени компьютера
* Добавление в домен или удаление из него
* Установка программного обеспечения или обновлений
* Запуск Chkdsk на системном диске

Вы можете произвести эти действия в виде нескольких шагов, разделенных перезагрузкой компьютера. При помощи рабочих процессов вы можете запустит процесс и забыть о нем, пока он полностью не завершит свое выполнение.

Давайте рассмотрим следующие сценарии:

1. Перезагрузка компьютера, к которому удаленно подключается рабочий процесс.

2. Перезагрузка компьютера, на котором выполняется рабочий процесс и *ручное* возобновление рабочего процесса.

3. Перезагрузка компьютера, на котором выполняется рабочий процесс и *автоматическое* возобновление рабочего процесса.

Первый случай является наиболее простым, поэтому с него и начнем.

**Перезагрузка удаленного компьютера**

Чтобы вы могли сконцентрироваться на, собственно, перезагрузке компьютера, для примера мы возьмем довольно простой рабочий процесс.

workflow test-restart {

param ([string[]]$computernames)

foreach -parallel ($computer in $computernames) {

Get-WmiObject -Class Win32\_ComputerSystem -PSComputerName $computer

Get-WmiObject -Class Win32\_OperatingSystem -PSComputerName $computer

}

}

Все, что он делает – это получает информацию о компьютере и операционной системе посредством вызовов WMI. Так как мы используем командлеты WMI вместо CIM, нам не важно, работает ли WSMAN на всех машинах. Для тестирования мы используем два удаленных компьютера:

**W12standard** – Windows Server 2012 (Windows PowerShell 3.0)

**WebR201** – Windows Server 2008 R2 (Windows PowerShell 2.0)

Рабочий процесс запускается следующим образом:

PS> test-restart -computernames w12standard, webr201

В данном случае я явно указываю, что активности рабочего процесса должны выполняться на удаленных компьютерах. Нужно сказать, что это не единственный способ. Больше о параметрах рабочих процессов вы узнаете в следующей статье.

Рабочий процесс параллельно подключается к удаленным компьютерам, на каждом выполняя последовательность WMI запросов. Команды блока **foreach** выполняются последовательно, но запуск их на разных компьютерах происходит параллельно.

Теперь добавим в рабочий процесс команду перезагрузки компьютера.

workflow test-restart {

param ([string[]]$computernames)

foreach -parallel ($computer in $computernames) {

Get-WmiObject -Class Win32\_ComputerSystem -PSComputerName $computer

Restart-Computer -Wait -PSComputerName $computer

Get-WmiObject -Class Win32\_OperatingSystem -PSComputerName $computer

}

}

Перезагрузка удаленных компьютеров инициируется в строке:

Restart-Computer -Wait -PSComputerName $computer

Обратите внимание на параметр **–Wait**. Его наличие приводит к тому, что рабочий процесс ожидает пока удаленный компьютер перезагрузится, прежде чем продолжить выполнение команд.

Если вы запустите рабочий процесс из консоли Windows PowerShell или из Windows PowerShell Integrated Scripting Environment (ISE), вы увидите индикатор выполнения, указывающий стадии перезагрузки:

* Waiting for restart to begin
* Verifying computer has restarted
* Waiting for WMI connectivity
* Waiting for Windows PowerShell connectivity
* Waiting for WinRM connectivity

В моем случае Windows Server 2012 перезагружается гораздо быстрее, чем Windows Server 2008 R2.

В статье TechNet [“Restarting the Computer in a Workflow”](http://technet.microsoft.com/en-us/library/jj574130.aspx) можно найти следующий код:

Restart-Computer -Wait -PSConnectionRetryCount 4 -PSConnectionRetryInterval 5

**Заметка**: Параметры **PSConnectionRetryCount** и **PSConnectionRetryInterval** не распознаются как параметры рабочего процесса.

Вы также можете запустить рабочий процесс в качестве задания Windows PowerShell:

PS> test-restart -computernames w12standard, webr201 -AsJob

Id Name PSJobTypeName State HasMoreData Location

— —- ————- —— ———— ———

25 Job25 PSWorkflowJob Running True localhost

После того, как задание завершит свое выполнение, вы можете просмотреть полученные данные. Однако, вы получите только данные WMI, сообщения о состоянии перезагрузки выведены не будут.

**Перезагрузка компьютера, на котором запущен рабочий процесс**

Здесь у нас есть два варианта:

* Перезапуск рабочего процесса вручную
* Перезапуск рабочего процесса автоматически

Давайте начнем с ручного перезапуска.

**Перезапуск рабочего процесса вручную**

Запуск рабочего процесса, перезагружающего компьютер, на котором он выполняется, а затем перезапуск его вручную – это не оптимальное решение. В идеале, нам бы хотелось, чтобы вся работа делалась автоматически. Однако это решение требует меньше усилий по написанию кода, и целом значительно проще. Иногда бывают случаи, когда простое и быстрое решение – это как раз то, что нужно.

Кроме того, используемый этом примере скрипт представляет собой основу для примера с автоматическим перезапуском рабочего процесса.

Здесь мы используем активность **Rename-Computer**. Переименование компьютера – это одно из действий, требующих перезагрузки компьютера.

workflow rename-localsystem {

param (

[string]$newname

)

Rename-Computer -Newname $newname -Force -Passthru

Restart-Computer -Wait

Get-CimInstance -ClassName Win32\_ComputerSystem |

Select-Object -ExpandProperty Name |

Set-Content -Path «C:\Scripts\$newname.txt»

}

Рабочий процесс получает новое имя компьютера в качестве параметра. Он переименовывает компьютер и затем вызывает активность **Restart-Computer** с параметром **–Wait**. После перезагрузки, рабочий процесс получает текущее имя компьютера и сохраняет его в текстовом файле. Имя файла соответствует новому имени компьютера.

Для начала получим текущее имя компьютера. В нашем случае – это “TEST1NOJOBPARAM”.

PS C:\scripts> $env:COMPUTERNAME

TEST1NOJOBPARAM

PS C:\scripts> rename-localsystem -newname W12SUS

Затем мы запустим рабочий процесс, указав в качестве параметра новое имя компьютера. Заметьте, что мы не использовали параметр **–AsJob**. Рабочий процесс изменяет имя и компьютер перезагружается.

После перезагрузки, мы подключимся к компьютеру и проверим, создан ли файл.

PS C:\scripts> ls

Directory: C:\scripts

Mode LastWriteTime Length Name

—- ————- —— —-

d—- 14/12/2012 18:45 Setup

-a— 02/01/2013 11:58 280 rename-localsystem.ps1

Как видите, файла не существует. Также в списке задач присутствует задача в состоянии **Suspended**.

PS C:\scripts> Import-Module PSWorkflow

PS C:\scripts> Get-Job | Format-Table -AutoSize

Id Name PSJobTypeName State HasMoreData Location Command

— —- ————- —— ———— ——— ——-

3 Job2 PSWorkflowJob Suspended True localhost rename-localsystem

Возобновим задачу. Она завершит свое выполнение и создаст текстовый файл. В качестве завершающей проверки, мы получим значение переменной среды – COMPUTERNAME.

PS C:\scripts> ls

Directory: C:\scripts

Mode LastWriteTime Length Name

—- ————- —— —-

d—- 14/12/2012 18:45 Setup

-a— 02/01/2013 11:58 280 rename-localsystem.ps1

-a— 02/01/2013 12:24 8 W12SUS.txt

PS C:\scripts> $env:COMPUTERNAME

W12SUS

Возможны ситуации, когда перезагрузка компьютера будет последней инструкцией в рабочем процессе. Этот сценарий также вполне может быть реализован с помощью рабочих процессов.

Рассмотрим пример:

workflow add-localsystemtodomain{

param (

$domcred,

[string]$ipv4address

)

$index = Get-NetIPInterface -AddressFamily IPv4 -Dhcp Enabled |

Select-Object -ExpandProperty ifIndex

New-NetIPAddress -InterfaceIndex $index -AddressFamily IPv4 `

-IPAddress $ipv4address -PrefixLength 24

Set-DnsClientServerAddress -InterfaceIndex $index -ServerAddresses «10.10.54.201»

Set-DnsClient -InterfaceIndex $index -ConnectionSpecificSuffix «manticore.org»

Add-Computer -Credential $domcred -DomainName Manticore `

-OUPath «OU=Security Servers,OU=Servers,DC=Manticore,DC=org» -Force

Restart-Computer

}

## remove relicit jobs

Get-Job |

where PSJobTypeName -eq «PSWorkflowJob» |

Remove-Job

$cred = Get-Credential

add-localsystemtodomain -domcred $cred -ipv4address «10.10.54.170» -JobName AddToDomain

Для задания статического адреса сетевого адаптера виртуальной машины используются новые командлеты для работы с сетью. Кроме того, указываются адрес DNS-сервера и суффикс подключения. Компьютер добавляется в домен и затем перезагружается.

Скрипт удаляет все существующие задачи, связанные с рабочими процессами, запрашивает учетные данные для добавления компьютера в домен и запускает рабочий процесс. Так как для активности **Restart-Computer** не указывается параметр **–Wait**, задачи не создается и после перезагрузки не выполняется никаких действий.

Теперь давайте рассмотрим ситуацию, когда система сама перезапускает рабочий процесс.

**Автоматический перезапуск рабочего процесса**

В документации по рабочим процессам сказано, что для их автоматического перезапуска могут использоваться задания Windows PowerShell (Scheduled Jobs). Однако мои попытки реализовать это на практике не увенчались успехом. Поэтому я решил использовать назначенные задания (Scheduled Tasks). В Windows PowerShell 3.0 появилось несколько командлетов для работы с назначенными заданиями. Они используют WMI и предназначены для работы только на Windows Server 2012 и Windows 8. В принципе, можно автоматизировать работу с назначенными заданиями и на других операционных системах с использованием COM, но это задача, выходящая за рамки статьи.

Здесь я использую довольно простой подход, однако этого должно быть достаточно, чтобы познакомить вас с основной концепцией. Давайте начнем с рабочего процесса.

workflow test-restart {

Get-WmiObject -Class Win32\_ComputerSystem | Out-File -FilePath C:\Reports\comp.txt

Get-ChildItem -Path C:\Reports | Out-File -FilePath C:\dir.txt

Restart-Computer -Wait

Get-WmiObject -Class Win32\_OperatingSystem | Out-File -FilePath C:\Reports\os.txt

}

Рабочий процесс использует **Get-WmiObject** для получения класса **Win32\_ComputerSystem** и помещает полученные данные в файл. **Get-ChildItem** получает список файлов каталога C:\Report. Таким образом мы сможем удостовериться, что первая часть рабочего процесса выполнена.

Далее используется активность **Restart-Computer** с параметром **–Wait** и компьютер перезагружается.

После перезагрузки, рабочий процесс возобновляется при помощи назначенного задания и создается файл, содержащий данные, полученные при запросе класса **Win32\_OperatingSystem**.

Теперь нам нужно создать назначенное задание.

$actionscript = ‘-NonInteractive -WindowStyle Normal -NoLogo -NoProfile -NoExit -Command «&»c:\reports\test-resume.ps1»»‘

$pstart = «C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe»

Get-ScheduledTask -TaskName Test | Unregister-ScheduledTask -Confirm:$false

$act = New-ScheduledTaskAction -Execute $pstart -Argument $actionscript

$trig = New-ScheduledTaskTrigger -AtLogOn

Register-ScheduledTask -TaskName Test -Action $act -Trigger $trig -RunLevel Highest

Нам нужно, чтобы назначенное задание запускало Windows PowerShell, поэтому я указал параметры запуска PowerShell в переменной **$actionscript**. Переменная**$pstart** содержит команду для запуска Windows PowerShell.

Следующая строка удаляет предыдущий экземпляр задания.

Действие задания задается при помощи командлета **New-ScheduledTaskAction**. Триггер определяет, когда должно запуститься задание. Я установил запуск задания на событие входа в систему, поскольку я хочу видеть его выполнение. Если хотите, вы можете заменить параметр **–AtLogon** на **–AtStartUp**.

Зарегистрируем новое назначенное задание. Убедитесь, что указан **–RunLevel Highest**, чтобы задание запускалось с повышенными привилегиями.

Еще одна недостающая часть мозаики – это скрипт, запускаемый назначенным заданием.

Get-ChildItem -Path C:\Reports | Out-File -FilePath C:\Reports\dir.txt

Import-Module PSWorkflow

Get-Job | Resume-Job

Здесь создается еще один список содержимого папки C:\Reports. Также в скрипте происходит импорт модуля **PSWorkflow** и возобновление приостановленной задачи. Если в вашей системе присутствует несколько приостановленных задач, вы можете выбрать нужную задачу по имени.

После перезагрузки компьютера и входа в систему вы увидите открытие окна Windows PowerShell и возобновление выполнения задачи.

Здесь есть некоторая странность, заключающаяся в том, что время создания файла os.txt соответствует времени создания comp.txt, хотя создание файла os.txt должно происходить уже после перезагрузки компьютера.

После завершения рабочего процесса вы можете удалить задачу обычным способом. Также вам потребуется удалить назначенное задание.

Get-ScheduledTask -TaskName Test | Unregister-ScheduledTask -Confirm:$false

Трудно назвать это элегантным решением, однако оно работает. Я продолжу пробовать использовать для этой цели задания (scheduled jobs) и в случае, если у меня что-нибудь получится, напишу об этом в своем блоге.

**Gotchas**

Одна из вещей, с которыми я столкнулся при тестировании, это то, что не следует запускать рабочие процессы из ISE. В соответствии с документацией, вы можете запустить рабочий процесс, подобный следующему:

workflow test-restart {

Get-WmiObject -Class Win32\_ComputerSystem

Restart-Computer -Wait

Get-WmiObject -Class Win32\_OperatingSystem

}

Однако, если вы запустите его из ISE, вы не обнаружите приостановленного задания – оно будет выполненным. При запуске же его из консоли, все работает как ожидается. Хочу выразить благодарность Steven Murawski за помощь при обнаружении данной особенности.

Еще одна вещь, о которой стоит помнить – сохраняйте информацию на диск, так как при перезагрузке она потеряется.

**Conclusion**

После тестирования всех этих сценариев я пришел к выводу, что указание удаленного компьютера через параметры рабочего процесса – это самый простой способ обработки событий перезагрузки. Перезагрузка локального компьютера из рабочего процесса возможна, но порой требует множества усилий для ее реализации.

~Richard.

**Автор:**

Ed Wilson, Microsoft Scripting Guy

# [PowerShell Workflows: Используем параметры](https://sergeyvasin.net/2013/03/21/powershell-workflows-using-parameters/)

В этой статье мы рассмотрим параметры, доступные для рабочих процессов и активностей, варианты подключения с удаленным компьютерам, а также командлеты для рабочих процессов. Давайте начнем с параметров.

Для рабочих процессов и активностей существует достаточное количество параметров. Они приведены в следующей таблице.

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Workflow** | **Activity** |
| AppendOutput |  | Y |
| AsJob | Y |  |
| JobName | Y |  |
| Debug |  | Y |
| DisplayName |  | Y |
| ErrorAction |  | Y |
| Input |  | Y |
| MergeErrorToOutput |  | Y |
| PSActionRetryCount |  | Y |
| PSActionRetryIntervalSec |  | Y |
| PSActionRunningTimeoutSec |  | Y |
| PSAllowRedirection | Y | Y |
| PSApplicationName | Y | Y |
| PSAuthentication | Y | Y |
| PSAuthenticationLevel | Y |  |
| PSCertificateThumbprint | Y | Y |
| PSComputerName | Y | Y |
| PSConfigurationName | Y | Y |
| PSConnectionRetryCount   \*\* | Y | Y |
| PSConnectionRetryIntervalSec   \*\* | Y | Y |
| PSConnectionURI | Y | Y |
| PSCredential | Y | Y |
| PSDebug |  | Y |
| PSDisableSerialization |  | Y |
| PSElapsedTimeOutSec | Y |  |
| PSError |  | Y |
| PSParameterCollection | Y |  |
| PSPersist | Y | Y |
| PSPort | Y | Y |
| PSPrivateMetaData | Y |  |
| PSProgress |  | Y |
| PSProgressMessage |  | Y |
| PSRemotingBehavior |  | Y |
| PSRequiredModules |  | Y |
| PSRunningTimeOutSec | Y |  |
| PSSessionOption | Y | Y |
| PSUseSSL | Y | Y |
| PSVerbose |  | Y |
| PSWarning |  | Y |
| Result |  | Y |
| UseDefaultInput |  | Y |
| Verbose |  | Y |
| WarningAction |  | Y |

\*\*В документации сказано, что эти параметры могут применяться к активностям, однако тестирование показало, что активности их не поддерживают. Я полагаю, что это параметры рабочих процессов.

Я не буду здесь объяснять каждый из них, так как название большинства из них говорит само за себя. Также вы можете найти более подробную информацию по ним в двух следующих файлах справки:

about\_WorkflowCommonParameters

about\_ActivityCommonParameters

Некоторые параметры вы уже использовали в предыдущих статьях, например параметры для запуска рабочего процесса в качестве задания.

### ****Рабочие процессы и удаленный доступ****

Один из наиболее интересных параметров – это **PSComputerName**, который используется для подключения к удаленному компьютеру. Есть несколько вариантов. Первый – это указание компьютера на уровне рабочего процесса.

workflow test-remoteaccess {

Get-WmiObject -Class Win32\_ComputerSystem

}

Заметьте, что в коде рабочего процесса не указывается имя компьютера. Однако, вы можете запустить этот рабочий процесс следующим образом:

PS> test-remoteaccess -PSComputerName server02, win732 | select Name, Manufacturer, Model

Name Manufacturer Model

—- ———— ——

SERVER02 LENOVO 4318CTO

WIN732 Microsoft Corporation Virtual Machine

Использование параметра рабочего процесса **–PSComputerName** приводит к тому, что имя компьютера передается всем активностям этот рабочего процесса. Этот способ удобен, если вы запускаете достаточно простой рабочий процесс и хотите, чтобы все его активности были выполнены на удаленном компьютере, однако, если вы получаете данные с нескольких компьютеров, вывод может быть довольно запутанным. Давайте рассмотрим следующий пример.

workflow test-remoteaccess {

Get-WmiObject -Class Win32\_ComputerSystem |

Select-Object -Property Name, Manufacturer, Model

Get-Process

Get-Service

}

test-remoteaccess -PSComputerName server02, win732

Я получил данные в следующем порядке:

|  |  |  |
| --- | --- | --- |
| **Data item** | **First** | **Second** |
| WMI | Server02 | Win732 |
| Process | Server02 | Win732 |
| Service | Server02 и Win732 вперемешку |  |

Данные могут возвращаться в любом порядке, так как активности выполняются параллельно на нескольких компьютерах.

Таким образом, использование параметра рабочего процесса **–PSComputerName** отлично подходит для случаев, когда вам не так важен порядок вывода данных, или когда данные не выводятся вообще.

Когда вы используете параметр **–PSComputerName**, он фактически заменяет собой параметр командлета **–ComputerName**. Т.е. использование этого параметра не предоставляет вам еще один способ подключения. Кроме того, подключение происходит посредством механизма, используемого командлетом. Если удаленный компьютер не поддерживает этот механизм, рабочий процесс не сможет выполниться на этом компьютере. Давайте взглянем на пример, где вместо командлета **Get-WmiObject** мы будем использовать **Get-CimInstance**. Также не забудьте изменить имя параметра **–Class** на **–ClassName**.

workflow test-remoteaccess {

Get-CimInstance -ClassName Win32\_ComputerSystem

}

Давайте запустим его на тех же самых компьютерах.

PS> test-remoteaccess -PSComputerName server02, win732

Name PrimaryOwnerName Domain TotalPhysicalMemory Model Manufacture PSComputerName

—- ———— —— ———— —— ———— ————

SERVER02 Windows … Manticor… 17108062208 4318CTO LENOVO server02

The WS-Management service cannot process the request. A DMTF resource URI was used to

access a non-DMTF class. Try again using a non-DMTF resource URI.

+ CategoryInfo : NotSpecified: (root\cimv2:Win32\_ComputerSystem:String) []

, CimException

+ FullyQualifiedErrorId : HRESULT 0x80338139,Microsoft.Management.Infrastructure.CimCmdlets.GetCimInstanceCommand

+ PSComputerName : [localhost]

Рабочий процесс отработал на компьютере Server02, однако попытка его запуска на Win732 привела к появлению ошибки, сообщающей о проблеме с подключением по протоколу WSMan (WinRM). Командлеты WMI для подключения к удаленным компьютерам используют DCOM, однако командлеты CIM используют WSMan, и если быть более точным, WSMan весии 3.0, который входит в состав Windows PowerShell 3.0. Win732 – это компьютер под управлением Windows 7, с установленным Windows PowerShell 2.0. Варианты решения этой проблемы – установить на компьютер Windows PowerShell 3.0, использовать командлеты WMI или создать сессию CIM с использованием DCOM (эту тему мы сейчас рассматривать не будем).

Вернемся к вариантам запуска рабочего процесса на удаленных компьютерах. Вы также можете указать параметр **–PSComputerName** для активности.

workflow test-remoteaccess {

param(

[string[]]$computername

)

foreach -parallel ($computer in $computername) {

Get-WmiObject -Class Win32\_ComputerSystem -PSComputerName $computer

}

}

Этот рабочий процесс содержит параметр, принимающий список имен компьютеров. Конструкция **foreach –parallel** используется для выполнения активностей на каждом из этих компьютеров. Как вы помните из [первой статьи](https://sergeyvasin.wordpress.com/2013/01/03/powershell-workflows-the-basics/), выполнение на компьютерах из списка происходит параллельно, однако команды из блока **foreach –parallel** выполняются последовательно на каждом отдельном компьютере. В этом примере мы снова используем DCOM.

PS> test-remoteaccess -computername server02, win732 | select Name, Manufacturer, Model

Name Manufacturer Model

—- ———— ——

SERVER02 LENOVO 4318CTO

WIN732 Microsoft Corporation Virtual Machine

Теперь давайте рассмотрим сценарий, где командлеты, выполнение которых должно происходить на удаленных компьютерах, расположены в блоке **InlineScript**. Нужно понимать, что в этом случае вы работаете с командлетами, а не активностями, поэтому вы будете использовать присущий командлетам параметр **–ComputerName**.

workflow test-remoteaccess {

param(

[string[]]$computername

)

inlinescript {

foreach ($computer in $using:computername) {

Get-WmiObject -Class Win32\_ComputerSystem -ComputerName $computer

}

}

}

Рабочий процесс обладает параметром **computername**, в качестве значения для которого указывается список имен компьютеров. Внутри блока **InlineScript**, конструкция **foreach** обрабатывает каждый компьютер из списка. Вам потребуется задать **foreach** следующим образом:

foreach ($computer in $using:computername)

Модификатор **$using** позволяет конструкции получить доступ к переменной, которая была определена в области действия (scope) выше по иерархии (информацию по переменным и их областям действия в рабочих процессах вы можете найти во [второй статье](https://sergeyvasin.wordpress.com/2013/01/06/powershell-workflows-restrictions/)). Рабочий процесс вызывается таким же образом, как и в предыдущем примере.

PS> test-remoteaccess -computername server02, win732 | select Name, Manufacturer, Model

Name Manufacturer Model

—- ———— ——

SERVER02 LENOVO 4318CTO

WIN732 Microsoft Corporation Virtual Machine

Как вы видите, существует несколько способов работы с именами компьютерами в рабочих процессах. Есть еще один способ, не перед тем перейти к его рассмотрению, вам нужно познакомиться с командлетами рабочих процессов.

### ****Командлеты рабочих процессов****

Модуль **PSWorkflow** нами уже упоминался. Его импорт, помимо файлов справки и возможности запуска рабочих процессов, также делает доступными два командлета. Существует еще один модуль – **PSWorkflowUtility**, в котором содержится один командлет – **Invoke-AsWorkfow**. Командлеты и модули приведены в следующей таблице.

|  |  |
| --- | --- |
| **Module** | **Cmdlet/Function** |
| PSWorkflow | New-PSWorkflowExecutionOption New-PSWorkflowSession |
| PSWorkflowUtility | Invoke-AsWorkflow |

Командлеты, входящие в модуль **PSWorkflow** предназначены для работы с сессиями рабочих процессов. Вы можете считать сессии рабочих процессов неким аналогом удаленным сессиям, известным вам с Windows PowerShell 2.0. Если вы собираетесь запустить одну или две команды на удаленном компьютере, то, возможно, будет эффективнее создавать подключения по мере необходимости и разрывать их по завершении работы команды. Если же вам нужно запустить множество команд, каждой из которых потребуется создать подключение к удаленному компьютеру, то в этом случае более правильным будет создать удаленную сессию.

В файле справки к командлету **New-PSWorkflowSession** говорится: «Командлет New-PSWorkflowSession создает пользовательскую сессию («PSSession»), которая специальным образом предназначена для запуска рабочих процессов Windows PowerShell. Она использует конфигурацию сессий Microsoft.PowerShell.Workflow, включающую в себя скрипты, файлы типов и форматирования, а также опции, необходимые для рабочих процессов. Если возникнет такая необходимость, командлет **New-PSWorkflowExecutionOption** позволит вам определить дополнительные параметры конфигурации сессии рабочих процессов.

Если вы когда-либо создавали удаленную сессию в Windows PowerShell, вы не найдете ничего необычного в создании сессии рабочих процессов.

$wkfsess1 = New-PSWorkflowSession -ComputerName server02, win732

$wkfsess1

$sb ={

Import-Module PSWorkflow

workflow test-remoteaccess {

Get-WmiObject -Class Win32\_ComputerSystem |

select -Property Name, Manufacturer, Model

}

test-remoteaccess

}

Invoke-Command -Session $wkfsess1 -ScriptBlock $sb

Команды, которые вы собираетесь запустить в рамках сессии помещаются в скриптблок. Для запуска команд в рамках сессии используется командлет **Invoke-Command**.

Invoke-Command -Session $wkfsess1 -ScriptBlock $sb

New-PSSession : [win732] Connecting to remote server win732 failed with the following error

message : The WS-Management service cannot process the request. Cannot find the

Microsoft.PowerShell.Workflow session configuration in the WSMan: drive on the win732 computer.

For more information, see the about\_Remote\_Troubleshooting Help topic.

At C:\Windows\system32\WindowsPowerShell\v1.0\Modules\PSWorkflow\PSWorkflow.psm1:58 char:9

+ New-PSSession -ConfigurationName Microsoft.PowerShell.Workflow @PSBoundP …

+ CategoryInfo : OpenError: (System.Manageme….RemoteRunspace:RemoteRunspace) [New-

PSSession], PSRemotingTransportException

+ FullyQualifiedErrorId : InvalidResourceUri,PSSessionOpenFailed

Id Name ComputerName State ConfigurationName Availability

— —- ———— —— —————— ————

5 Session5 server02 Opened Microsoft.PowerSh… Available

WARNING: [localhost]:This workflow job cannot be suspended because there are no persistence points

in the workflow. To make the workflow job suspendable, add persistence points to the workflow.

For more information, see the Help topics for Windows PowerShell Workflows.

Name : SERVER02

Manufacturer : LENOVO

Model : 4318CTO

PSSourceJobInstanceId : 78ab3c45-a8f6-428d-a4a4-92aff9df257d

PSComputerName : server02

RunspaceId : 6595de20-5cc3-41fa-9cb7-da0f40f1f0e0

Ой. Это один их недостатков сессий рабочих процессов – вы можете их использовать только если на удаленном компьютере установлен Windows PowerShell 3.0. Если мы посмотрим на точки подключения на компьютере с Windows Server 2012, на котором по умолчанию установлен Windows PowerShell 3.0 (WSMan 3.0), то мы увидим:

PS> ls WSMan:\localhost\Plugin | select name

Name

—-

Event Forwarding Plugin

microsoft.powershell

microsoft.powershell.workflow

microsoft.powershell32

microsoft.windows.servermanagerworkflows

SEL Plugin

WMI Provider

На компьютере с Windows 7 и Windows PowerShell 2.0 (WSMan 2.0) мы получим следующее:

PS> ls WSMan:\localhost\Plugin | select name

Name

—-

Event Forwarding Plugin

microsoft.powershell

WMI Provider

К сожалению, обойти это ограничение не получится и вам придется использовать параметры **ComputerName** и **PSComputerName**, рассмотренные в предыдущих примерах.

Теперь давайте рассмотрим командлет **Invoke-AsWorkflow**. Он отлично подходит для тестирования кода посредством рабочих процессов. Возьмем простой скрипт.

$exp = «Get-WmiObject -Class Win32\_ComputerSystem |

select -Property Name, Manufacturer, Model»

Вы можете запустить его в виде рабочего процесса следующим образом:

Invoke-AsWorkflow -Expression $exp -PSComputerName server02, win732

До того, как вы подумали, что командлет **Invoke-AsWorkflow** – это способ превращения любого кода в рабочий процесс, хочу сказать, что этот командлет не предлагает всей функциональности рабочих процессов, потому как запускает переданные команды внутри блока InlineScript. То есть, вы не сможете использовать такие возможности, как, например, параллельное выполнение команд – одно из самых значительных преимуществ рабочих процессов.

### ****Резюме****

В этой статье был приведен ряд параметров рабочих процессов, и то, как они могут быть использованы для запуска рабочих процессов или активностей. Вы увидели, как имена компьютеров могут быть указаны для рабочего процесса, активности или сессии рабочих процессов. Также был рассмотрен командлет **Invoke-AsWorkflow**, позволяющий тестировать код, используя рабочие процессы.

В следующей статье мы рассмотрим некоторые рекомендации по использованию рабочих процессов, а также ряд вопросов, касающихся их построения.

# [PowerShell Workflows: Вопросы структуры](https://sergeyvasin.net/2013/03/26/powershell-workflows-design-considerations/)

Первый вопрос, который вы должны себе задать – «Что я собираюсь написать: рабочий процесс или обычный скрипт Windows PowerShell?»

Мой приятель MVP Don Jones написал отличную статью о том, [когда вам стоит использовать рабочий процесс](http://powershell.org/wp/2012/08/30/powershell-workflow-when-should-you-use-it/). Если же вкратце, то вам стоит использовать рабочие процессы, если:

* Вам потребуется прервать, а затем возобновить выполнение задачи
* Вам потребуется сохранить состояние выполнения задачи
* Если в вашем скрипте присутствуют задачи, требующие как последовательного, так и параллельного выполнения

Также приведу несколько потенциальных причин для использования рабочих процессов:

* Вам нужно выполнить долгосрочное задание, состоящее из множества последовательных шагов
* Вам нужно выполнить задание на нескольких устройствах
* Вам нужно выполнить асинхронное долгосрочное задание
* Вам нужно выполнить долгосрочное задание, включающее в себя шаги, выполнение которых должно происходить параллельно

На самом деле, эти требования могут быть реализованы посредством скриптов или фоновых заданий (Jobs) Windows PowerShell. Одно из качеств, присущих эксперту по определенной технологии, это способность сказать, когда эта технология не должна использоваться. Так что не бойтесь сказать нет использованию рабочих процессов!

Параллельное выполнение заданий включает в себя два аспекта:

1. Одновременное выполнение одного и того же задания на нескольких машинах
2. Одновременное выполнение нескольких заданий на одной машине

В первом случае вы можете использовать как фоновые задания (Jobs), так и рабочие процессы (Workflows). Во втором случае вы определенно находитесь на территории рабочих процессов. Это туманное место.

Представьте, что вам нужно запустить выполнение задачи на множестве машин, или в высокодоступном окружении, что потенциально потребует регулирования количества одновременных подключений. Это вполне может быть выполнено как при помощи фоновых заданий, так и с использованием рабочих процессов. Здесь нет однозначного ответа. Я знаю организации, которые использую фоновые задания, затрагивающие тысячи машин. Рабочие процессы – это все еще довольно новая технология, чтобы мы могли судить о масштабе ее применения с достаточной достоверностью. Однако, если вы использовали рабочие процессы для управления большим количеством машин – будем рады услышать ваши отзывы.

Перед тем как перейти к примерам, хочу сказать еще одно – даже если фоновые задания полностью решают вашу задачу, *нет ничего плохого в том, чтобы пробовать использовать рабочие процессы*. Единственный способ изучения технологии – это ее использование.

Теперь давайте взглянем на несколько примеров. Один из распространенных сценариев – это добавление ключа реестра на несколько удаленных компьютеров. Во многих организациях среда достаточно гетерогенная. Я сымитирую ее использованием следующих машин:

|  |  |  |
| --- | --- | --- |
| **Machine** | **Operating system** | **Windows PowerShell version** |
| W12SUS | Windows Server 2012 | Windows PowerShell 3.0 |
| Server02 | Windows Server 2012 | Windows PowerShell 3.0 |
| WebR201 | Windows Server 2008 R2 | Windows PowerShell 2.0 |
| Win7test | Windows 7 | Windows PowerShell 3.0 |

Создание ключа и записи реестра – это работа для WMI. Вы не думали, что я этого коснусь, ведь правда? На одной из машин установлен Windows PowerShell 2.0, поэтому вместо CIM-командлетов я буду использовать командлеты WMI. Причина этого в том, что командлеты CIM по умолчанию используют WSMan, но это должен быть WSMan v3.

Добавление ключа включает в себя следующее:

$hklm = 2147483650

$key = «SOFTWARE\HSGworkflowDEMO»

Invoke-WmiMethod -Class StdRegProv -Name CreateKey -ArgumentList $hklm, $key

Для создания записи потребуются дополнительные шаги:

$value = «AreYouThere»

$data = «Yes»

Invoke-WmiMethod -Class StdRegProv -Name SetStringValue -ArgumentList $hklm, $key, $data, $value

На случай, если вы удивляетесь порядку параметров, я запрошу командлет **Get-CIMClass**:

$class = Get-CimClass -ClassName StdRegProv

$class.CimClassMethods[«SetStringValue»]

$class.CimClassMethods[«SetStringValue»].Parameters

Вот что мы получим:

Name CimType Qualifiers

—- ——- ———-

hDefKey UInt32 {ID, IN}

sSubKeyName String {ID, IN}

sValue String {ID, in}

sValueName String {ID, in}

Если мы заглянем в документацию, то увидим, что **sValueName** находится перед **sValue**, что логично, но здесь командлет **Invoke-WmiMethod** ведет себя довольно странно и, по-видимому, ожидает, что параметры будут указаны в алфавитном порядке. В подобных случаях, если сомневаетесь, используйте порядок, указанный командлетом **Get-CimClass**.

Следующий пример демонстрирует, возможно, самую важную вещь, касающуюся создания рабочих процессов: *тестируйте код без их использования*. Если вы будете уверены, что код сам по себе не содержит ошибок – это значительно облегчит тестирование рабочих процессов. Запустить код локально – довольно просто. Если вы собираетесь запустить его на удаленной машине – у вас есть два варианта:

1. Указать параметр **–ComputerName** для каждого командлета **Invoke-WmiMethod**.
2. Поместить код в скрипт-блок и запустить на удаленных компьютерах с помощью командлета **Invoke-Command**.

В каждом из случаев вы можете использовать параметр**–AsJob**.

Давайте рассмотрим второй вариант.

$sb = {

$hklm = 2147483650

$key = «SOFTWARE\HSGworkflowDEMO»

Invoke-WmiMethod -Class StdRegProv -Name CreateKey -ArgumentList $hklm, $key

$value = «AreYouThere»

$data = «Yes»

Invoke-WmiMethod -Class StdRegProv -Name SetStringValue -ArgumentList $hklm, $key, $data, $value

}

Invoke-Command -ComputerName server02, w12sus, win7test, webr201 -ScriptBlock $sb

Мы можем запустить все это в виде задания, добавив параметр **–AsJob**. В этом случае последняя строка будет выглядеть так:

Invoke-Command -ComputerName server02, w12sus, win7test, webr201 -ScriptBlock $sb –AsJob

В качестве небольшого наблюдения: я заметил, что команды на машинах с Windows PowerShell 3.0 выполняются быстрее, чем на компьютерах с Windows PowerShell 2.0.

Для того, чтобы превратить это в рабочий процесс, нам нужно изменить код следующим образом:

workflow new-regkey {

$hklm = 2147483650

$key = «SOFTWARE\HSGworkflowDEMO»

Invoke-WmiMethod -Class StdRegProv -Name CreateKey -ArgumentList $hklm, $key

$value = «AreYouThere»

$data = «Yes»

Invoke-WmiMethod -Class StdRegProv -Name SetStringValue -ArgumentList $hklm, $key, $data, $value

}

Далее нам нужно будет запустить наш рабочий процесс, указав параметр **–PSComputerName**.

new-regkey -PSComputerName server02, w12sus, win7test, webr201

Пока что мы имели дело с простой задачей, которая может быт решена несколькими способами. Давайте все чуть-чуть усложним. Нашей задачей будет:

* Создать ключ реестра HKLM:\SOFTWARE\HSGworkflowDEMO
* Создать под этим ключом три записи и присвоить им значения:
  + AreYouThere = Yes
  + AreWeThereYet = No
  + PowerShell = 1

Таким образом, у нас есть набор параллельных и последовательных заданий. Вы должны создать ключ до того, как перейдете к созданию записей, но эти записи вы можете создавать параллельно. Вы можете выполнить эту задачу при помощи скрипта Windows PowerShell, и все сработает, но выполнение инструкций будет происходить последовательно. Использование рабочего процесса позволяет вам выполнять активности параллельно, что может быль более эффективным. Одно из решений этой задачи при помощи рабочих процессов выглядит следующим образом:

workflow new-regkey {

$hklm = 2147483650

$key = «SOFTWARE\HSGworkflowDEMO»

Invoke-WmiMethod -Class StdRegProv -Name CreateKey -ArgumentList $hklm, $key

parallel {

sequence {

$value = «AreYouThere»

$data = «Yes»

Invoke-WmiMethod -Class StdRegProv -Name SetStringValue -ArgumentList $hklm, $key, $data, $value

} # end of sequence

sequence {

$value = «PowerShell»

$data = 1

Invoke-WmiMethod -Class StdRegProv -Name SetDwordValue -ArgumentList $hklm, $key, $value, $data

} # end of sequence

sequence {

$value = «AreWeThereYet»

$data = «No»

Invoke-WmiMethod -Class StdRegProv -Name SetStringValue -ArgumentList $hklm, $key, $data, $value

}# end of sequence

}# end of parallel

}

Запускаем рабочий процесс.

new-regkey -PSComputerName server02, w12sus, win7test, webr201

Заметьте, как изменяется порядок аргументов командлета **Invoke-WmiMethod** во втором блоке sequence. Одна из загадок WMI.

Мне очень нравится возможность указать параметр **–PSComputerName** при вызове рабочего процесса, так как это позволяет автоматически передать имена компьютеров входящим в него активностям.

Этот рабочий процесс можно разделить на следующие части:

* Последовательное выполнение:
  + Создание ключа реестра
  + Параллельное создание трех записей реестра – двух строк и одного числового значения
    - Создание каждого значения происходит в три последовательных шага

Именно в такие моменты вы начинаете чувствовать удовольствие от использования рабочих процессов. А также приобретать опыт в их построении и в решении таких задач, как:

* что должно выполняться параллельно, а что последовательно
* какие потребуются переменные
* взаимодействие между различными областями действия (scopes)

Точно так же, как и при выполнении каких-то больших проектов, я бы рекомендовал вам сначала обдумать структуру вашего будущего рабочего процесса и уже затем перейти к написанию самого кода. Я описал мой подход к написанию скрипта для решения какой-либо проблемы в [комментариях к Scripting Games 2012](http://blogs.technet.com/b/heyscriptingguy/archive/2012/04/20/expert-commentary-2012-scripting-games-advanced-event-5.aspx).

Что касается рабочих процессов, то ваше решение должно основываться на следующих факторах:

* Какие задачи выполняет рабочий процесс?
* В каком порядке эти задачи должны быть выполнены?
* Можно ли что-то выполнить параллельно?
* Нужно ли мне использовать блоки InlineScript?
* Какие переменные мне понадобятся и где они должны быть определены?
  + Как на это повлияют области действия рабочего процесса?
* Потребуется ли выполнение активностей рабочего процесса на удаленных машинах?
  + Будет ли он выполняться на локальной машине и обращаться к удаленным компьютерам или он будет запущен непосредственно на удаленных машинах с использованием возможностей удаленного выполнения команд?
  + Где будут указываться имена компьютеров – при вызове рабочих процессов, активностей или командлетов?
* Что должно возвращаться по завершении рабочего процесса?
* Потребуется ли перезагрузка компьютера в процессе выполнения рабочего процесса?
* Потребуется ли сохранение состояний выполнения рабочего процесса?
  + Если да, то как часто?
* Нужно ли сохранять какие-либо данные за пределами рабочего процесса?
* Будет ли рабочий процесс запускаться в качестве задания?

В предыдущих статьях мы уже рассматривали все эти вопросы. Как именно вы будете строить ваши рабочие процессы – зависит от решаемой вами проблемы.

На случай, если вы задумываетесь, как удалить созданные нами ключи реестра:

$sb = {

$hklm = 2147483650

$key = «SOFTWARE\HSGworkflowDEMO»

Invoke-WmiMethod -Class StdRegProv -Name DeleteKey -ArgumentList $hklm, $key

}

Invoke-Command -ComputerName server02, w12sus, win7test, webr201 -ScriptBlock $sb

Удалив ключ реестра, мы также удаляем все входящие в него ключи и записи.

В этой статье, возможно, было слишком много текста, однако я обещаю вам исправить положение в следующей статье, где мы разберем довольно большой рабочий процесс, затрагивающий несколько уже рассмотренных нами тем.

# [PowerShell Workflows: Пример практического использования](https://sergeyvasin.net/2013/04/01/powershell-workflows-a-practical-example/)

Это последняя статья в серии, касающейся рабочих процессов Windows PowerShell, однако это вовсе не значит, что писать о них я больше не собираюсь. Рабочие процессы – это новая концепция в Windows PowerShell, и, как и все нововведения, она требует глубокого тестирования для определения тех областей, где ее применение более всего обосновано.

В этой статье я собираюсь объединить все рассмотренные нами темы и попробовать создать рабочий процесс. В сценарии рассматривается расширяющаяся организация, регулярно открывающая новые офисы и хотелось бы автоматизировать этот процесс настолько, насколько это возможно.

Требуется выполнить следующие действия:

1. Создать родительское OU (организационное подразделение) для нового офиса
   1. Создать дочернее OU пользователей
   2. Создать дочернее OU компьютеров
2. Создать UPN
3. Создать учетные записи
   1. Создать 35 учетных записей компьютеров в компьютерном OU
   2. Создать 35 учетных записей пользователей в пользовательском OU
   3. Создать домашние каталоги
4. Создать сайт Active Directory
   1. Создать Subnet (подсеть) и присоединить с сайту
   2. Создать Site Link (связь сайтов)
5. Связать GPO
   1. Связать GPO (объект групповой политики) с OU пользователей
   2. Связать GPO (объект групповой политики) с OU компьютеров

Первое, и самое важное, что нужно сказать, что все это вовсе не обязательно должно быть выполнено с использованием рабочих процессов. И вообще, я бы удивился, узнав, что существует какая-либо задача, которую можно выполнить только с их помощью. И хотя реализация нашей задачи при помощи скриптов была бы чуть сложнее – однако это вполне возможный вариант. Теперь, когда я это сказал, мы можем переходить к созданию рабочего процесса.

Одна из основных вещей, которую привносят рабочие процессы – это возможность параллельного выполнения команд и поэтому вам потребуется решить, что должно быть выполнено параллельно, а что последовательно. Еще один важный фактор, относительно дизайна рабочего процесса – в каком порядке должны происходить действия. Например, вам нужно будет создать OU перед тем как создавать в нем учетные записи.

В программировании существует концепция *псевдокода*. Она используется для задания логики части кода в процессе разработки. Я не думаю, что она активно используется скриптерами, однако в сложных ситуациях, навроде этой, она вполне может помочь.

Если вы представите список задач в виде псевдокода, подучится следующее:

workflow pseudocode {

parallel {

создать родительское OU

создать UPN

создать сайт AD

}

parallel {

sequence {

создать пользовательское OU

получить данные пользователей

parallel {

создать учетные записи пользователей

создать домашние каталоги

}

связать GPO с пользовательским OU

}

sequence {

создать компьютерное OU

получить данные компьютеров

parallel {

создать учетные записи компьютеров

}

связать GPO с компьютерным OU

}

sequence {

создать подсеть AD

создать связь сайтов AD

}

}

}

Начнем с блока parallel, создающего родительское OU, сайт AD и UPN. Все это можно выполнить параллельно, поскольку эти задачи никак не связаны друг с другом. Основная часть рабочего процесса состоит из блока parallel, содержащего три блока sequence – один для учетных записей пользователей, один для учетных записей компьютеров и один для задач топологии Active Directory. Внутри блоков sequence некоторые задачи выполняются параллельно, например, создание пользовательских учетных записей.

Хочу сказать, что структура, предложенная мной — это не единственный вариант построения рабочего процесса.

После разработки и тестирования наш рабочий процесс примет следующий вид.

workflow new-location {

param (

[string]$locationname,

[string]$subnet

)

parallel {

New-ADOrganizationalUnit -Path «DC=manticore,DC=org» -Name $locationname

Set-ADObject -Identity «CN=Partitions,CN=Configuration,DC=manticore,DC=org» -Add @{upnsuffixes = «$locationname-manticore.org» }

New-ADReplicationSite -Name $locationname

}

parallel {

sequence {

New-ADOrganizationalUnit -Path «OU=$locationname,DC=manticore,DC=org» -Name «Users»

##

## получаем данные о пользователях

$password = ConvertTo-SecureString -AsPlainText -String «Pa55W0rd1!» -Force

$users = Import-Csv -Path ./userdata.csv

##

## создаем пользователей и домашние каталоги

foreach -parallel ($user in $users){

New-ADUser -AccountPassword $password -Name ($user.Lname + » » + $user.Fname) -Path «OU=Users,OU=$locationname,DC=manticore,DC=org» -SamAccountName $user.id -UserPrincipalName «$user.id@$locationname-manticore.org»

New-Item -Path «\\w12standard\home» -Name $user.id -ItemType Directory

}

##

## связываем GPO с пользовательским OU

New-GPLink -Name «Management Configuration» -Target «OU=Users,OU=$locationname,DC=manticore,DC=org»

}

##

## создаем компьютерное OU, учетные записи компьютеров и связываем GPO с OU

sequence {

New-ADOrganizationalUnit -Path «OU=$locationname,DC=manticore,DC=org» -Name «Computers»

$computers = 1..35

foreach -parallel ($computer in $computers) {

$cname = «$locationname-{0:000}» -f $computer

New-ADComputer -Path «OU=Computers,OU=$locationname,DC=manticore,DC=org» -Name $cname

}

New-GPLink -Name «W12Updates» -Target «OU=Computers,OU=$locationname,DC=manticore,DC=org»

}

sequence {

New-ADReplicationSubnet -Name $subnet -Site $locationname

New-ADReplicationSiteLink -Name «Site1-$locationname» -SitesIncluded «Site1», $locationname -InterSiteTransportProtocol IP -ReplicationFrequencyInMinutes 15

}

}

}

Потребуется указать два параметра – местонахождение нового офиса и адрес подсети. Запуск рабочего процесса может выглядеть следующим образом:

new-location -locationname Seattle -subnet «10.10.19.0/24»

Первое что делает рабочий процесс – создает родительское OU, сайт AD и UPN.

parallel {

New-ADOrganizationalUnit -Path «DC=manticore,DC=org» -Name $locationname

Set-ADObject -Identity «CN=Partitions,CN=Configuration,DC=manticore,DC=org» -Add @{upnsuffixes = «$locationname-manticore.org» }

New-ADReplicationSite -Name $locationname

}

Эти задачи могут быть выполнены параллельно. Они никак не зависят друг о друга, поэтому вы можете не беспокоиться о порядке их выполнения. Командлет **New-ADReplicationSite** появился в Windows Server 2012. Если в вашей среде он недоступен, вы можете видоизменить скрипт следующим образом:

$for = [System.DirectoryServices.ActiveDirectory.Forest]::GetCurrentForest()

$fortyp = [System.DirectoryServices.ActiveDirectory.DirectoryContexttype]»forest»

$forcntxt = new-object System.DirectoryServices.ActiveDirectory.DirectoryContext($fortyp, $for)

## создание нового сайта

$site = new-object System.DirectoryServices.ActiveDirectory.ActiveDirectorySite($forcntxt, «MyNewSite2»)

$site.Save()

Вам потребуется запустить это код в блоке InlineScript.

Переходим к основной части скрипта. У нас есть три блока sequence, которые выполняются параллельно. Вот где мы видим настоящую мощь рабочих процессов – вы можете создавать учетные записи пользователей и компьютеров и изменять топологию Active Directory одновременно. Первый блок sequence создает учетные записи пользователей.

sequence {

New-ADOrganizationalUnit -Path «OU=$locationname,DC=manticore,DC=org» -Name «Users»

##

## получаем данные о пользователях

$password = ConvertTo-SecureString -AsPlainText -String «Pa55W0rd1!» -Force

$users = Import-Csv -Path ./userdata.csv

##

## создаем пользователей и домашние каталоги

foreach -parallel ($user in $users){

New-ADUser -AccountPassword $password -Name ($user.Lname + » » + $user.Fname) -Path «OU=Users,OU=$locationname,DC=manticore,DC=org» -SamAccountName $user.id -UserPrincipalName «$user.id@$locationname-manticore.org»

New-Item -Path «\\w12standard\home» -Name $user.id -ItemType Directory

}

##

## связываем GPO с пользовательским OU

New-GPLink -Name «Management Configuration» -Target «OU=Users,OU=$locationname,DC=manticore,DC=org»

}

Создается OU для пользовательских учетных записей. В нескольких местах кода мы ссылаемся на переменную **locationname**. Она определена в более высокой области действия (scope), но так как мы используем ее в блоках parallel и sequence, то нам нужно просто указать ее имя.

Также мы задаем пароль для учетной записи пользователя. И нет, я не стал бы это использовать в промышленной среде. Я бы указал пароль в качестве еще одного параметра. Никогда не указывайте пароли в тексте скрипта!

Информация о пользователе получается из файла CSV. В нашем случае файл содержит имя, фамилию и идентификатор пользователя. Вы можете добавить необходимые данные и задействовать еще несколько параметров командлета **New-ADUser**. Если вы не обладаете опытом использования данного командлета, я бы посоветовал вам обратиться к файлу справки, чтобы узнать, что с его помощью можно сделать. Странность заключается в способе задания имени пользовательской учетной записи – подстановка строк не работает, кроме того в рабочих процессах вы не можете использовать подвыражения.

Я указал имя файла явным образом, но вы можете это изменить. Как вам идея использовать механизм событий Windows PowerShell для наблюдения за определенным каталогом? Как только в него попадает новый файл, срабатывает событие и запускается рабочий процесс! Это привнесет еще большую степень автоматизации.

Создание учетных записей пользователей происходит с использованием конструкции foreach –parallel. Она ужасающе мощная. В нашем случае создание 35 учетных записей происходит параллельно. Создание учетной записи и домашнего каталога происходит последовательно, но эти действия происходят параллельно для 35 разных пользователей. Попробовать предположить, чем именно занимается рабочий процесс в определенный момент времени – интересная задача. Мне кажется, это возможно только в том случае, если вы добавите с рабочий процесс механизм журналирования.

Последнее, что выполняется в этом блоке sequence – это связывание GPO с пользовательским OU.

Следующий блок sequence создает учетные записи компьютеров.

sequence {

New-ADOrganizationalUnit -Path «OU=$locationname,DC=manticore,DC=org» -Name «Computers»

$computers = 1..35

foreach -parallel ($computer in $computers) {

$cname = «$locationname-{0:000}» -f $computer

New-ADComputer -Path «OU=Computers,OU=$locationname,DC=manticore,DC=org» -Name $cname

}

New-GPLink -Name «W12Updates» -Target «OU=Computers,OU=$locationname,DC=manticore,DC=org»

}

Сначала создается OU, затем в нем создаются 35 учетных записей компьютеров. Для простоты я явным образом указал их имена, но вы можете использовать для этого отдельный параметр. Кроме того, вы можете подсчитать число создаваемых учетных записей пользователей и создать по одной учетной записи компьютера на каждого пользователя.

Я использую форматирование строк, чтобы создать имя компьютера, отвечающее правилам именования компьютеров в организации. Затем оно передается командлету **New-ADComputer**. И снова, чтобы воспользоваться преимуществами параллельного выполнения, я использую конструкцию foreach –parallel.

Далее мы используем командлет **New-GPLink** для связи GPO с компьютерным OU.

Последний блок sequence вносит изменения в топологию Active Directory.

sequence {

New-ADReplicationSubnet -Name $subnet -Site $locationname

New-ADReplicationSiteLink -Name «Site1-$locationname» -SitesIncluded «Site1», $locationname -InterSiteTransportProtocol IP -ReplicationFrequencyInMinutes 15

}

Эти два командлета доступны только в Windows Server 2012. Если в вашей среде используются серверы более ранних версий, вы можете заменить их следующими командами.

Создание подсети:

## получаем объект леса и задаем контекст именования

$for = [System.DirectoryServices.ActiveDirectory.Forest]::GetCurrentForest()

$fortyp = [System.DirectoryServices.ActiveDirectory.DirectoryContexttype]»forest»

$forcntxt = new-object System.DirectoryServices.ActiveDirectory.DirectoryContext($fortyp, $for)

$site = «MyNewSite2»

$subnetlocation = «Building X»

$subnetname = «10.55.0.0/24»

## создаем подсеть и присоединяем к сайту

$subnet = New-Object System.DirectoryServices.ActiveDirectory.ActiveDirectorySubnet($forcntxt, $subnetname, $site)

$Subnet.Location = $subnetlocation

$subnet.Save()

Создание связи сайта:

## получаем объект леса и задаем контекст именования

$for = [System.DirectoryServices.ActiveDirectory.Forest]::GetCurrentForest()

$fortyp = [System.DirectoryServices.ActiveDirectory.DirectoryContexttype]»forest»

$forcntxt = new-object System.DirectoryServices.ActiveDirectory.DirectoryContext($fortyp, $for)

$link = New-Object -TypeName System.DirectoryServices.ActiveDirectory.ActiveDirectorySiteLink -ArgumentList $forcntxt, «MyNewSite3-MyNewSite4»

$site1 = [System.DirectoryServices.ActiveDirectory.ActiveDirectorySite]::FindByName($forcntxt, «MyNewSite3»)

$site2 = [System.DirectoryServices.ActiveDirectory.ActiveDirectorySite]::FindByName($forcntxt, «MyNewSite4»)

$link.Sites.Add($site1)

$link.Sites.Add($site2)

$link.Cost = 150

$link.ReplicationInterval = «01:00:00» ## 1 час 24×7

$link.Save()

$linkde = $link.GetDirectoryEntry()

$linkde.Description = «Links sites MyNewSite3 and MyNewSite4»

$linkde.SetInfo()

Этот код нужно будет поместить в блок InlineScript. И помните, вам для доступа к переменным, определенным в верхних областях действия (scope), вам нужно использовать модификатор Using.

Я надеюсь, что эта серия статей показала вам мощь рабочих процессов и то, как их можно использовать. Рабочие процессы – это очень полезный ресурс, с которым сообщество Windows PowerShell еще только начало разбираться. Эта серия – всего лишь шаг к пониманию принципов их работы, поэтому я советую вам не останавливаться и продолжать экспериментировать с ними. Также не забывайте делиться результатами с остальной частью сообщества. Со мной вы можете связаться через [мой блог](http://msmvps.com/blogs/richardsiddaway/default.aspx).

Я не думаю, что это последнее мое слово о рабочих процессах, так что – до новых встреч.

# PowerShell Workflows: идентификация результатов запроса удаленных компьютеров

Рабочие процессы (workflows) позволяют работать с множеством компьютеров параллельно, однако при определенных запросах бывает сложно идентифицировать с какого компьютера какая информация получена. В этом посте я приведу несколько вариантов решения этой задачи.

Для примера возьмем следующий скрипт.

workflow Get-WFInfo  
{  
parallel  
{  
Get-CimInstance -ClassName win32\_BaseBoard  
Get-CimInstance -ClassName win32\_bios  
Get-NetAdapter  
Get-Disk  
Get-Partition  
}  
}

Если мы запустим его с параметром **-PSComputerName**, то том самым мы проинформируем рабочий процесс о том, что мы хотим получить результаты указанных в тексте запросов с определенных компьютеров. Однако, если мы его запустим, например, следующим образом:

Get-WFInfo -PSComputerName comp1,comp2

То получим информацию об экземплярах классов: **win32\_BaseBoard**, **win32\_bios**, **MSFT\_NetAdapter**, **MSFT\_Disk** и **MSFT\_Partition** с компьютеров comp1 и comp2, как нам и требовалось, однако не все из этих объектов будут содержать информацию о том, с какого компьютера они получены.

Для примера, давайте зададим следующую команду.

Get-WFInfo -PSComputerName comp1,comp2 | select CimClass,SystemName,PSComputerName

Результаты этой команды будут содержать только название класса — **CimClass**, и два значения двух свойств — **SystemName** и **PSComputerName**, которые в определенных случаю будут содержать имя компьютера, что поможет нам уменьшить количество выводимой информации и сконтентрировать внимание только на тех значениях, которые нам нужны.

Как видно из рисунка, свойство **PSComputerName** содержит имя компьютера только в случае использования командлета (хотя, в данном случае — активности) **Get-CimInstance**.  
Кроме того, класс **MSFT\_NetAdapter** обладает свойством **SystemName**, что в данном случае также помогает идентифицировать источник информации.  
Однако, что касается запросов **Get-Disk** и **Get-Partition**, то здесь соотнести полученные результаты и запрашиваемые компьютеры представляется затруднительным.

Мы можем модифицировать рабочий процесс следущим образом.

workflow Get-WFInfo  
{  
foreach -parallel ($comp in $PSComputername)  
{  
parallel  
{  
Get-CimInstance -ClassName win32\_BaseBoard  
Get-CimInstance -ClassName win32\_bios  
Get-NetAdapter  
Get-Disk -CimSession $comp  
Get-Partition -CimSession $comp  
}  
}  
}

Тогда свойство **PSComputerName** будет содержать имена компьютеров для всех запросов — кроме **Get-NetAdapter**, поскольку в этом случае для идентификации мы можем использовать свойство **SystemName**. Однако, в целях единообразия нам ничто не мешает добавить параметр **-CimSession $comp** и к команде **Get-NetAdapter**.

Принимая во внимание, что при работе с несколькими компьютерами, обращения к ним происходят параллельно без каких-либо дополнительных указаний в тексте рабочего процесса, использование **foreach -parallel** для всех запросов мне кажется излишним. Поэтому мы можем вынести первые три запроса в отдельный блок **parallel**.

workflow Get-WFInfo  
{  
parallel  
{  
Get-CimInstance -ClassName win32\_BaseBoard  
Get-CimInstance -ClassName win32\_bios  
Get-NetAdapter  
}  
  
foreach -parallel ($comp in $PSComputername)  
{  
parallel  
{  
Get-Disk -CimSession $comp  
Get-Partition -CimSession $comp  
}  
}  
}

И все таки, мы можем еще больше упростить наш рабочий процесс.

workflow Get-WFInfo  
{  
parallel  
{  
Get-CimInstance -ClassName win32\_BaseBoard  
Get-CimInstance -ClassName win32\_bios  
Get-NetAdapter  
  
Get-Disk -CimSession $PSComputerName  
Get-Partition -CimSession $PSComputerName  
}  
}

Как уже было сказано, при запросе нескольких компьютеров, рабочий процесс обращается к каждому из них параллельно, и также параллельно выполняет активности указанные в блоке **parallel**.  
Тот факт, что мы указали переменную **$PSComputerName** в качестве значения параметра **-CimSession** позволяет нам получить имя компьютера в качестве значения свойства **PSComputerName** возвращаемого объекта.

Cтоит отметить, что в данном случае при обращении к каждому конкретному компьютеру переменная **$PSComputerName** в командах **Get-Disk** и **Get-Partition** будет содержать имя только этого компьютера.