Оглавление

ОБЗОР ТЕХНОЛОГИИ MICROSOFT WINDOWS POWERSHELLMICROSOFT ® 3

НАЗНАЧЕНИЕ POWERSHELL 4
УСТАНОВКА WINDOWS POWESSHELL 8
НАЗНАЧЕНИЕ POWERSHELL 8

ИСПОЛЬЗОВАНИЕ POWERSHELL В КАЧЕСТВЕ ИНТЕРАКТИВНОЙ ОБОЛОЧКИ 9

ИЕРАРХИЧЕСКИЕ ХРАНИЛИЩА 12
ПСЕВДОНИМЫ 15
ИСПОЛЬЗОВАНИЕ СПРАВКИ 16
РАСШИРЕНИЕ ОБОЛОЧКИ 20

ИСПОЛЬЗОВАНИЕ КОНВЕЙЕРОВ 21

ТЕРМИНОЛОГИЯ 25
СВОЙСТВА ОБЪЕКТОВ 27

ОСНОВЫ СИСТЕМЫ ФОРМАТИРОВАНИЯ 31

ФОРМАТИРОВАНИЕ ПО УМОЛЧАНИЮ. 31
OUT-DEFAULT 34
OUT- И FORMAT- 36
ВАРИАНТЫ ОТОБРАЖЕНИЯ 37
ДОПОЛНИТЕЛЬНЫЕ ДАННЫЕ В ВЫВОДЕ 39
СОЗДАНИЕ HTML 41

ОСНОВНЫЕ КОМАНДЛЕТЫ WINDOWS POWERSHELL 43

СОРТИРОВКА ОБЪЕКТОВ 45
ГРУППИРОВКА ОБЪЕКТОВ 47
ИЗМЕРЕНИЕ ОБЪЕКТОВ 48
ВЫБОР ОБЪЕКТОВ И СВОЙСТВ 49
РАБОТА С CSV И XML ДАННЫМИ 51
ЭКСПОРТ ОБЪЕКТОВ В CSV-ФАЙЛ 52
ИМПОРТ ОБЪЕКТОВ ИЗ CSV-ФАЙЛА 52
ИМПОРТ И ЭКСПОРТ XML-ФАЙЛОВ 53
СРАВНЕНИЕ ОБЪЕКТОВ 53
СРАВНЕНИЕ, ФИЛЬТРАЦИЯ И ПЕРЕЧИСЛЕНИЕ 54
СРАВНЕНИЕ 54
ФИЛЬТРАЦИЯ КОНВЕЙЕРОВ 57
ПЕРЕЧИСЛЕНИЕ 58
РАСШИРЕННЫЕ ВОЗМОЖНОСТИ КОНВЕЙЕРОВ 60
ПОЗИЦИОННЫЕ ПАРАМЕТРЫ 61
ПРИВЯЗКА ДАННЫХ КОНВЕЙЕРА ПО ЗНАЧЕНИЮ 64
ПРИВЯЗКА ДАННЫХ КОНВЕЙЕРА ПО ИМЕНИ СВОЙСТВА 66
ПЕРЕИМЕНОВАНИЕ СВОЙСТВ 68
PASSTHRU 70

WINDOWS® MANAGEMENT INSTRUMENTATION 71

ОБЗОР WMI 72
ВЗАИМОДЕЙСТВИЕ WMI 72
СТРУКТУРА WMI 73
КЛАССЫ 75
ЭКЗЕМПЛЯРЫ 76
КАК НАЙТИ НУЖНЫЙ КЛАСС. 76
ДОКУМЕНТАЦИЯ 79
БЕЗОПАСНОСТЬ WMI 79
ПРОСМОТР WMI ИЗ POWERSHELL 80

Использование WMI 81
Опрос WMI 81
Отбор данных 86
WQL-синтаксис 87
Системные свойства 87
WMI и ForEach-Object 88

АДМИНИСТРИРОВАНИЕ ACTIVE DIRECTORY 88

Добавление модуля 89
AD PSDrive 89
Ранние версии Windows 90
Управление пользователями и группами 91
Фильтрация 91
Управление компьютерами и другими объектами AD 91

СКРИПТЫ POWERSHELL 92

Безопасность скриптов 92
Основы безопасности 93
Политики выполнения 94
Доверенные скрипты 96
Прописывание скриптов 97
Простейшие скрипты 98
Читаемость скрипта 99
Объявление параметров 100
Использование параметров 101
Запрос параметра у пользователя 102

ФОНОВЫЕ ЗАДАНИЯ И ЗАДАНИЯ УДАЛЕННОГО ТИПА 102

Фоновые задания 103
Фоновый WMI 104
Управление задачами 104
Вывод задания 105
Другие команды как фоновые задания 107
Удаленное исполнение 108
Типы удаленного администрирования 109
Windows PowerShell Remoting 111
WinRM 112
1-to-1 shell 114
1-to-Many Remoting 115
Привязка конвейера 117
Работа с результатами 117
Многокомпьютерные задания 120
Распределение нагрузки 122
Restricted Runspaces 123
Постоянное соединение 123
Ключевые моменты 124
Управляйте сессиями! 124
implicit remoting 124

АДМИНИСТРИРОВАНИЕ WINDOWS SERVER® 2008 R2 125

Обзор модулей Windows Server® 2008 R2 125
Использование модулей 127
Командлеты Server Manager 128
GPO командлеты 128
Командлеты TroubleshootingPack 129
BPA командлеты 130
IIS командлеты 131

НАПИСАНИЕ СКРИПТОВ (ОБЗОР ЯЗЫКА) 132

ХРАНЕНИЕ ДАННЫХ 132
ПЕРЕМЕННЫЕ 132
СТРОКИ 135
МАССИВЫ 136
ХЕШ-ТАБЛИЦА 137
СПЛАТТИНГ 137
АРИФМЕТИЧЕСКИЕ ОПЕРАТОРЫ 138
ТИПЫ ПЕРЕМЕННЫХ 138
СЛОЖНЫЕ ОПЕРАТОРЫ СРАВНЕНИЯ 140
ОБЛАСТИ ДЕЙСТВИЯ 140
ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ И STRICT MODE 144
УПРАВЛЯЮЩИЕ ОПЕРАТОРЫ ЯЗЫКА 146
IF….ELSEIF…..ELSE 146
НАБОР ОПЕРАТОРОВ 148
SWITCH 150
BREAK 150
ОПЦИИ SWITCH 151
FOR 151
WHILE-DO-UNTIL 152
FOREACH 153
СТИЛЬ РАБОТЫ 154
ОБРАБОТКА ОШИБОЧНЫХ СИТУАЦИЙ 155
$ERRORACTIONPREFERENCE 156
ПОРОЧНАЯ ПРАКТИКА 157
–ERRORACTION 157
ПЕРЕХВАТ ОШИБОК 157
TRAP 159
TRY…. CATCH 161
ИЗВЛЕЧЕНИЕ ОШИБОК 162
ОТЛАДКА 162
ОТЛАДОЧНЫЙ ВЫВОД 163
ПОШАГОВЫЙ ОТЛАДЧИК 164
КОНТРОЛЬНЫЕ ТОЧКИ 164
ОТЛАДКА В ISE 165
МОДУЛЯРИЗАЦИЯ 165
БАЗОВЫЕ ФУНКЦИИ 166
ПАРАМЕТРИЗОВАННЫЕ ФУНКЦИИ 167
ПЕРЕДАЧА ДАННЫХ КОНВЕЙЕРОМ 169
ФУНКЦИИ-ФИЛЬТРЫ 169
САМОСТОЯТЕЛЬНОЕ КОНСТРУИРОВАНИЕ ВЫХОДНЫХ ДАННЫХ 170
РАЗНЫЕ СПОСОБЫ РЕШЕНИЯ 171
РАСШИРЕННЫЕ ФУНКЦИИ 172

СПИСОК ЛИТЕРАТУРЫ 173

Обзор технологии Microsoft Windows PowerShellMicrosoft ®

Windows PowerShell ® является современной, стандартизированной оболочкой командной строки, которая предлагает администраторам большую гибкость и выбор того, как они управляют своими компьютерами и продуктами на базе Windows. Windows PowerShell не является прямой заменой старых технологий, таких как Microsoft Visual Basic ® Script Edition (VBScript), хотя оболочка имеет

возможности сценариев, и обеспечивает такой же функциональный набор, как VBScript и много чего еще. Windows PowerShell была создана в результате вдохновения лучшими административными инструментами и процессами из операционной системы Windows и других операционных систем. Она предлагает уникальную, ориентированную на Windows модель администрирования.

Возможно впервые, Windows PowerShell предлагает действительно ориентированные на администратора средства автоматизации для повторяющихся или трудоемких административных задач, без необходимости более сложных программ или сценариев системы, которые требовались старыми технологиями.

Windows PowerShell - это не просто новый скриптовый язык. На самом деле это совершенно новый способ управления Windows и продуктами на базе Windows, и новый способ для Microsoft подумать об администраторах и том, как они работают. Многие из вещей, которые вы можете сделать в Windows PowerShell, сразу же покажутся знакомыми, например, возможность просматривать содержимое каталогов на диске. Тем не менее, Windows PowerShell, по сути, является чем-то совершенно другим. Для того, чтобы успешно работать с Windows PowerShell, необходимо иметь базовое понимание оперативной стратегии оболочки, ее технологической базы, и ее дизайна.

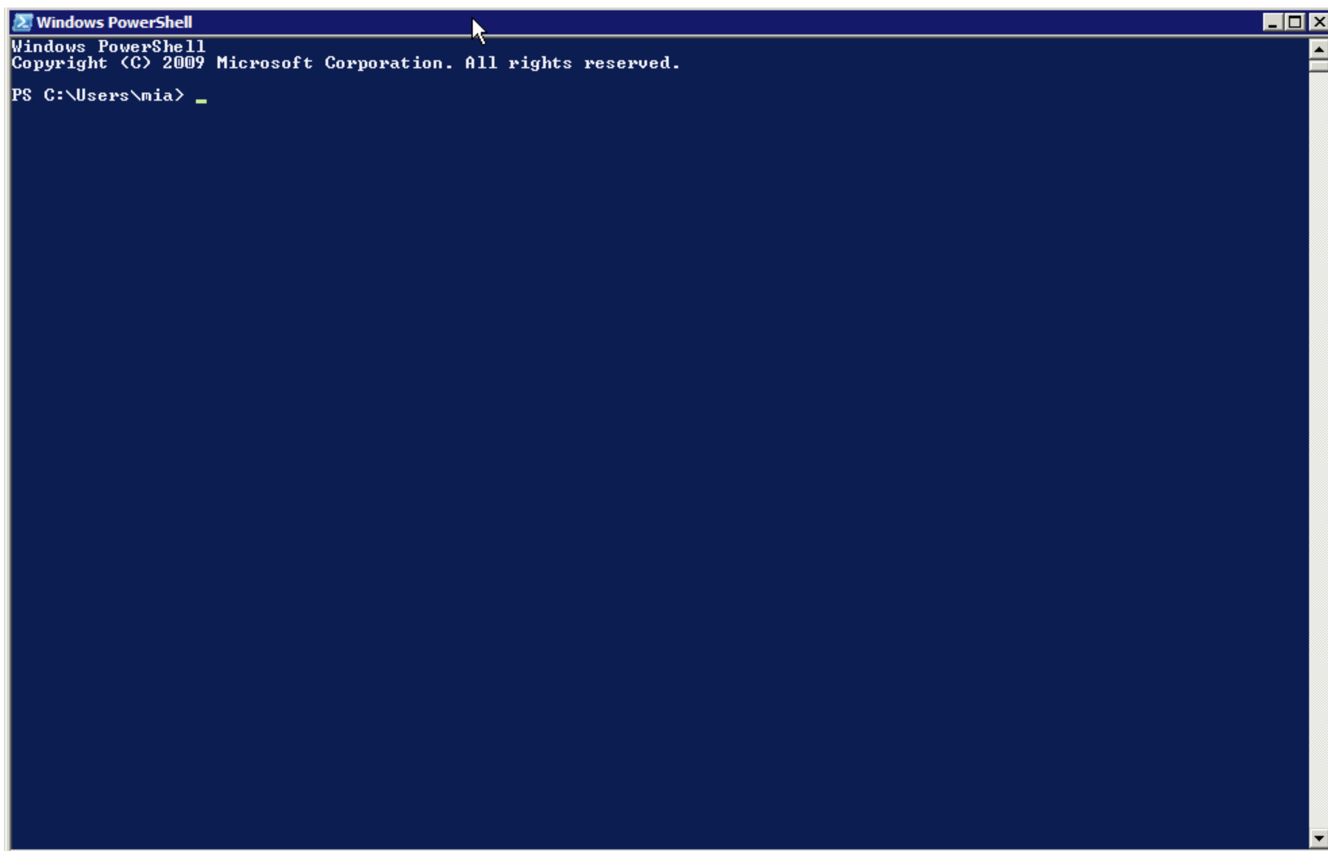
Назначение PowerShell

Назначение PowerShell

- Windows PowerShell – это не язык скриптов или, по крайней мере, это не только язык скриптов.
- Windows PowerShell – это целый механизм, созданный для того, чтобы выполнять команды, которые решают административные задачи, такие как
  - создание нового пользовательского аккаунта
  - конфигурация сервисов
  - удаление почтовых ящиков
- PowerShell предоставляет возможность проектировать GUI интерфейсы на своей базе. Вы можете выполнять одни команды двумя путями
  - Набирать команды в консоли
  - Выбирать графические элементы, которые выполнят те же команды.

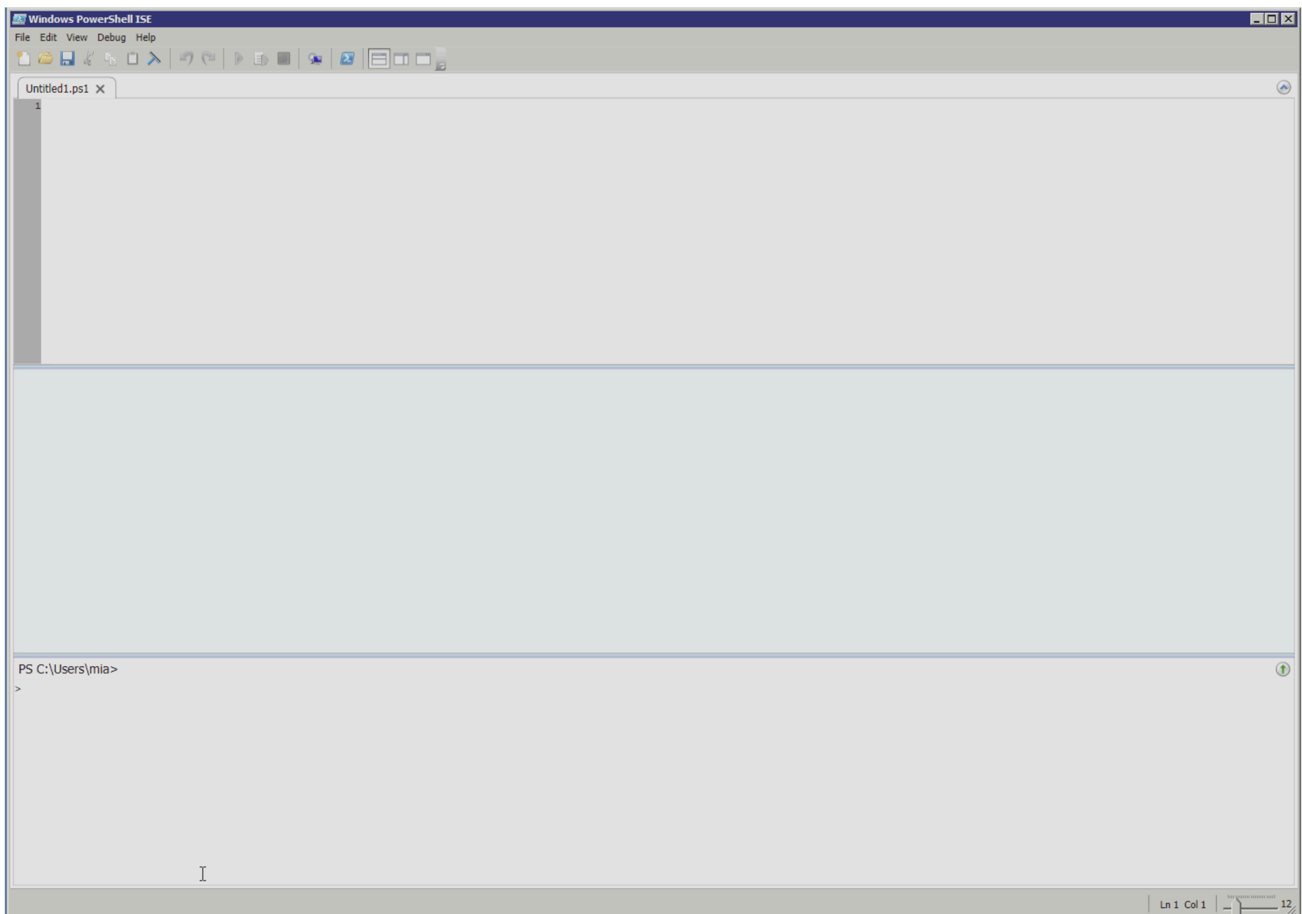
Windows PowerShell – это не язык скриптов или, по крайней мере, это не только язык скриптов. Windows PowerShell – это целый механизм, созданный для того, чтобы выполнять команды, которые решают административные задачи, такие как

создание нового пользовательского аккаунта, конфигурация сервисов, удаление почтовых ящиков, и так далее. В действительности оболочка Windows PowerShell дает массу способов сообщить системе о том, какие команды необходимо выполнить. Например, можно вручную напечатать команду в командной строке.



Командная строка PowerShell

Также вы можете печатать команды в графической консоли ISE (integrated scripting environment), которая обладает более содержательным графическим интерфейсом командной строки.



Windows PowerShell ISE

Если вы являетесь разработчиком программного обеспечения, вы можете встроить Windows PowerShell в свою собственную программу и запрограммировать ее на выполнение определенных команд в ответ на действия пользователя, например, кликание по кнопке или иконке. Можно набрать серию команд в текстовом редакторе и дать оболочке инструкцию выполнять команды из этого файла. В идеале Windows PowerShell должен являться единым, централизованным источником всей административной деятельности. Также в идеале вы должны использовать графический пользовательский интерфейс (GUI) с кнопками, иконками, диалоговыми окнами и прочими элементами, которые выполняют команды Windows PowerShell в фоновом режиме. В данном случае, если бы GUI не позволил вам выполнить ту или иную задачу именно тем способом, которым вы хотите, вы могли бы дать команду на выполнение той же самой задачи нужным вам способом непосредственно в командной строке, минуя GUI. Многие продукты Microsoft, в том числе Microsoft Exchange Server 2007 и Microsoft Exchange Server 2010, построены именно по этой схеме. Другие продукты, например, серверная ОС Windows Server, пока не являются построенными полностью на базе данной системы, хотя их отдельные компоненты – являются. Например, Active Directory Administrative Center в Windows Server 2008 R2 построен именно по данной идеальной схеме – это означает, что вы можете по своему усмотрению пользоваться графическим интерфейсом, который выполняет команды Windows PowerShell в фоновом режиме, или вводить команды самостоятельно непосредственно в консоли Windows PowerShell или ISE.

Где используется Windows PowerShell?

- Windows PowerShell используется с большим количеством продуктов Microsoft, например:
  - Microsoft Exchange Server 2007 and later
  - Microsoft System Center Data Protection Manager
  - Microsoft System Center Operations Manager
  - Microsoft System Center Virtual Machine Manager
  - Microsoft SQL Server 2008 and later

Get-Mailbox | Sort Size | Select –first 100 | Move-Mailbox Server2

Именно возможность выбора между вводом команд напрямую и выполнением этих команд посредством GUI является одной из главных причин привлекательности Windows PowerShell. С помощью данной оболочки Microsoft получает информацию о том, что некоторые задачи, особенно те, которые выполняются не слишком часто, проще выполнять через GUI. GUI может провести вас сквозь сложные операции и помочь быстрее разобраться в ваших возможностях, а также определиться с выбором. Однако Microsoft также получает информацию и о том, что GUI может оказаться неэффективным для выполнения задач, с которыми вы сталкиваетесь регулярно, например, для создания аккаунтов новых пользователей. Посредством создания максимальной административной свободы в форме команд Windows PowerShell Microsoft позволяет выбирать то, что больше подходит для выполнения тех или иных задач: простота использования GUI или мощь и широкие возможности персонализации оболочки с интерфейсом командной строки.

Со временем Windows PowerShell сможет заменить прочие низкоуровневые административные инструменты, которыми вы пользовались ранее. К примеру, Windows PowerShell уже сейчас может вытеснить Visual Basic Script Edition (VBScript), так как оболочка обеспечивает доступ к тем же самым функциям, что и VBScript, во многих случаях предлагая более простые способы выполнения тех же самых задач. Также Windows PowerShell может стать заменой Windows Management Instrumentation (WMI). Несмотря на то, что WMI остается очень полезным, зачастую он оказывается слишком сложным в использовании. Windows PowerShell может строить выполнение специфических задач на базе

существующего инструментария WMI. Технически вы по-прежнему будете использовать WMI, но само использование станет проще. Все это происходит сейчас и будет происходить и видоизменяться в будущем, по мере того, как Windows PowerShell будет развиваться.

Установка Windows PowesShell

Windows PowerShell v2 установлена по умолчанию в операционных системах Windows Server 2008 R2 и Windows 7. В Windows Server 2008 R2 можно установить опционную оболочку Windows PowerShell ISE, которая является более графически ориентированной. Windows PowerShell v2 также доступна для загрузки и установки в Windows XP, Windows Server® 2003, Windows Vista®, и Windows Server 2008. Windows PowerShell v2 включена в пакет Windows Management Framework Core, который помимо этого включает другие сходные технологии управления. Загрузить пакет можно с

http://go.microsoft.com/fwlink/?LinkId=193574

Отдельные версии доступны для разных операционных систем и архитектур (32-битной и 64-битной). Пакет для загрузки включает в себя Windows PowerShell ISE, а также более традиционную консоль с командной строкой.

Windows PowerShell v2 может быть установлена на следующих операционных системах.

- Windows Server 2008 with Service Pack 1
- Windows Server 2008 with Service Pack 2
- Windows Server 2003 with Service Pack 2
- Windows Vista with Service Pack 2
- Windows Vista with Service Pack 1
- Windows XP with Service Pack 3
- Windows Embedded POSReady 2009
- Windows Embedded for Point of Service 1.1

Windows PowerShell v2 требует Microsoft .NET Framework 2.0 с Service Pack 1; Windows PowerShell ISE требует Microsoft .NET Framework 3.5 с Service Pack 1. Обратите внимание, что Windows PowerShell может приобретать новые черты и получать новые функции из новых версий Microsoft .NET Framework. В идеале следует устанавливать последнюю доступную версию Framework, чтобы добиться максимальной функциональности. Также Windows PowerShell приобретает новые возможности в новых версиях Windows. Это означает, что на более старых версиях Windows набор функций будет чуть менее полным просто потому, что данные версии не содержат необходимых компонентов.

Назначение PowerShell

Windows PowerShell является ключевым инструментом управления, в отличие от Microsoft Management Console (MMC). Возможно, вы знаете, что консоль MMC сама по себе во многих случаях оказывается бесполезной. Для того, чтобы сделать ее полезной, необходимо добавлять оснастки. Оснастки дают специфические

возможности управления продуктами и технологиями, например, возможность администрирования активной директории или Exchange Server.

Windows PowerShell действует по такому же принципу. Несмотря на то, что оболочка выполняет множество полезных функций, ее возможности предполагается расширять с помощью всевозможных оснасток и модулей. Windows PowerShell сама по себе не содержит эти оснастки и модули; они устанавливаются параллельно с тем продуктом, функцией или технологией, к которой они относятся. Другими словами, если вы хотите с помощью Windows PowerShell управлять службами Active Directory Domain Services, вам необходимо установить соответствующий модуль. Этот модуль устанавливается вместе с ролью Active Directory® Domain Services. Некоторые модули также могут быть установлены параллельно с удаленным сервером Windows Remote Server Administration Toolkit (RSAT), что делает возможным установить модули, имеющие отношение к серверу, на клиентской операционной системе, например, Windows 7.

Некоторые модули можно загрузить отдельно. Например, на сайте CodePlex Web (**www.codeplex.com**) можно найти ряд проектов, имеющих отношение к Windows PowerShell со сторонних открытых источников. Многие из этих проектов имют вид оснастки или модуля. К ним относится, например, популярный проект Windows PowerShell Community Extensions, который можно скачать с **www.codeplex.com/powershellcx**. Отдельно стоит упомянуть, что некоторые оснастки и модули помимо требований Windows PowerShell могут иметь свои собственные системные требования. Например, модулям может потребоваться конкретная версия Windows или Microsoft .NET Framework. В документации к каждому модулю или оснастке должны быть указаны специфические системные требования, помимо базовых требований оболочки.

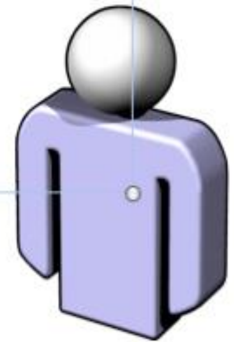
Использование PowerShell в качестве интерактивной оболочки

Хотя Windows PowerShell включает простой и мощный язык для написания скриптов, одним из преимуществ этой технологии является то, что его можно использовать в качестве удобной командной оболочки без написания сложных скриптов. Большое количество административных задач можно выполнять используя PowerShell для ввода интерактивных команд.

Windows PowerShell был разработан для того, чтобы его немедленно могли использовать администраторы, которые уже знакомы со старой оболочкой, в том числе Cmd.exe оболочкой, входящей в Windows, а также обычными оболочками Unix. В то время как некоторые из команд, работают немного по новому принципу, также вы найдете много знакомых команд, доступных для вас.

Какие команды Вы уже знаете?

- Например команды, которые вы бы использовали для выполнения каждой из следующих задач в оболочке cmd или Unix shell:
  - Изменить каталоги.
  - Список файлов и подкаталогов в каталоге.
  - Скопировать файл.
  - Отображение содержимого текстового файла.
  - Удалить файл.
  - Перемещение файлов.
  - Переименовать файл.
  - Создать новый каталог.



Например команды, которые вы бы использовали для выполнения каждой из следующих задач в оболочке cmd или Unix shell:

- Изменить каталоги.
- Список файлов и подкаталогов в каталоге.
- Скопировать файл.
- Отображение содержимого текстового файла.
- Удалить файл.
- Перемещение файлов.
- Переименовать файл.
- Создать новый каталог.

Команды

- С Windows PowerShell можно также запускать большинство внешних команд, которые могут быть знакомы вам:
  - Ipconfig.exe
  - Ping.exe
  - Tracert.exe
  - Nslookup.exe
  - Pathping.exe
  - Net.exe (например, Net Use)
- Windows PowerShell распознает многие из имен команд, с которыми вы, вероятно, уже знакомы, в том числе Cd, Dir, Ls, Cat, Type, MkDir, RmDir, Rm, Del, Cp, Copy, Move и так далее. Одновременно доступны наиболее общие file-and-folder команды управления Cmd.exe (которая использует MS-DOS Синтаксис команды) и оболочки Unix.

С Windows PowerShell можно также запускать большинство внешних команд, которые могут быть знакомы вам:

• Ipconfig.exe

• Ping.exe

• Tracert.exe

• Nslookup.exe

• Pathping.exe

• Net.exe (например, Net Use)

Windows PowerShell распознает многие из имен команд, с которыми вы, вероятно, уже знакомы, в том числе Cd, Dir, Ls, Cat, Type, MkDir, RmDir, Rm, Del, Cp, Copy, Move и так далее. Одновременно доступны наиболее общие file-and-folder команды управления Cmd.exe (которая использует MS-DOS Синтаксис команды) и оболочки Unix.

В большинстве случаев, однако, параметры этих команд разные. Например, в Cmd.exe вы можете выполнить следующую команду:

Dir /s

В Windows PowerShell, что та же команда выдаст ошибку, потому что / с, не признается действительным параметром.

Тем не менее, наличие этих знакомых имен команд поможет вам начать

использовать оболочку немедленно.

Внешние команды, такие как Ipconfig.exe, Pathping.exe, и так далее, продолжают работать как всегда. Вы можете, например, запустить эту общую команду из Windows PowerShell:

Ipconfig / all

Разница в том, что Ipconfig.exe является внешним исполняемым, в то время как команды, такие как Dir являются внутренними командами. Это верно также и для Cmd.exe. Внутренние команды, как правило, имеют синтаксис параметров, который отличается от того, к чему вы привыкли, и внешние исполняемые файлы будут продолжать функционировать как всегда.

Иерархические хранилища

Файловая система Windows и, если уж на то пошло, файловая система большинства компьютеров является иерархической. Это означает, что она состоит из отдельных ячеек, которые называются папками или директориями, которые, в свою очередь, содержат либо файлы, либо другие папки. Папки содержат подпапки, которые, опять же, содержат свои подпапки, и так далее.

Иерархические хранилища

- Файловая система Windows и, если уж на то пошло, файловая система большинства компьютеров является иерархической
  - Это означает, что она состоит из отдельных ячеек, которые называются папками или директориями, которые, в свою очередь, содержат либо файлы, либо другие папки. Папки содержат подпапки, которые, опять же, содержат свои подпапки, и так далее
- Однако файловая система – это не единственная иерархическая система в Windows.
  - The registry
  - The certificate store
  - Active Directory
- Одна из фич PowerShell – единая система работы со всеми иерархическими хранилищами.

Большинство администраторов Windows помнят основные команды, необходимые для управления иерархической файловой системой: Cd, Dir, Copy, Move и другие. Однако файловая система – это не единственная иерархическая система в Windows. Реестр, хранилище сертификатов, Active Directory и многие другие системы хранения также имеют иерархическую структуру. Помимо этого Windows

включает в себя ряд линейных (не-иерархических) хранилищ, таких как переменная среды операционной системы.

Одной из ключевых конструктивных задач Windows PowerShell является адаптация комплекса техник или процессов с целью использования данного комплекса в дальнейшем для решения сходных задач. Например, если реестр и файловая система имеют одну и ту же иерархическую структуру, то почему бы не использовать одни и те же команды для навигации и управления тем и другим? Другими словами, если администраторы уже знают стандартный набор команд, которые применяются в работе с иерархическим хранилищем одной формы, почему бы не адаптировать эти команды для работы с другими формами хранилищ, в том числе линейной?

Возможность использования единого набора команд для навигации по хранилищам, имеющим различную форму, становится возможным благодаря функции Windows PowerShell, которая называется провайдер PSDrive или просто провайдер. Провайдер представляет собой разновидность адаптера, который устанавливает соединение с системой хранения и представляет ее Windows PowerShell в форме дискового накопителя. Когда вы подаете команду, такую как Cd или Dir, она передается провайдеру данного диска, и провайдер делает все необходимое для того, чтобы выполнить эту команду.

Один набор команд для различных хранилищ

- Возможность использования единого набора команд для навигации по хранилищам, имеющим различную форму, становится возможным благодаря функции Windows PowerShell, которая называется провайдер PSDrive или просто провайдер.
- Провайдер представляет собой разновидность адаптера, который устанавливает соединение с системой хранения и представляет ее Windows PowerShell в форме дискового накопителя.
- PowerShell поставляется с набором провайдеров:

The file system	IIS
The registry	SQL Server
The environment variables	Active Directory
The certificate store	

Windows PowerShell оснащена несколькими провайдерами, в том числе для:

• Файловой системы,

• Реестра,

• Переменной среды,

• Хранилища сертификатов

Также она включает провайдеров для собственных хранилищ оболочки, включая хранилище функций, псевдонимов и переменных, о которых вы узнаете позже.

Оболочка может быть расширена и модернизирована для использования других провайдеров. Уже существуют провайдеры для IIS, SQL Server, Active Directory и многих других форм хранилищ. Все провайдеры придают соответствующим хранилищам форму дискового накопителя. Эти дисковые накопители внутри оболочки официально называются PSDrives или Windows PowerShell Drives. Они не отображаются за пределами оболочки в Windows Explorer и могут иметь названия, состоящие более, чем из одной буквы. Например, диски реестра называются HKCU: и HKLM:. Вы можете создавать новые диски, уточняя каждый раз имя нового диска, провайдера, который будет использоваться, например, провайдер файловой системы и начальную ячейку, с которой провайдер будет устанавливать соединение, например, папку или сетевое имя. Обратите внимание, что оболочка всегда начинает работу с одного и того же диска, когда вы начинаете новую сессию; любые новые созданные вами диски сбрасываются при завершении работы с оболочкой.

Управление PSDrives

- Get-PSDrive - выводит все доступные хранилища. По умолчанию, этот список включает все доступные накопители и логические диски
- New-PSDrive - создает новое хранилище. Вы должны указать имя хранилища (без двоеточия на конце), имя провайдера и стартовую точку или путь. Тип такой стартовой точки или пути зависит от типа используемого провайдера
- Remove-PSDrive - уничтожает хранилище. Вы также можете удалить хранилища, присутствующие по умолчанию, например HKCU: или ENV:, однако они будут воссозданы при запуске нового экземпляра оболочки.

Командная оболочка включает полный набор команд для управления хранилищами PSDrives:

Get-PSDrive: выводит все доступные хранилища. По умолчанию, этот список включает все доступные накопители и логические диски.

New-PSDrive: создает новое хранилище. Вы должны указать имя хранилища (без двоеточия на конце), имя провайдера и стартовую точку или путь. Тип такой стартовой точки или пути зависит от типа используемого провайдера.

Remove-PSDrive: уничтожает хранилище. Вы также можете удалить хранилища, присутствующие по умолчанию, например HKCU: или ENV:, однако они будут воссозданы при запуске нового экземпляра оболочки.

Псевдонимы

Большинство обычных команд, с которыми вы встречались в оболочке, такие как Dir, Cd, Ls, Copy, Rm и другие на самом деле являются псевдонимами Windows PowerShell. Псевдоним – это укороченное название (никнейм) команды. Они создаются для того, чтобы упростить и ускорить набор команд, а также, чтобы имена команд, которыми вы пользовались в других оболочках, были легко узнаваемыми. Например, Dir – это псевдоним команды Get-ChildItem. Ls – это псевдоним той же самой команды, так же как и Gci. Псевдонимы не меняют принципа действия команд или их параметров. Псевдоним – это всего лишь никнейм.

В Windows PowerShell встроенные команды называются командлеты. Они отличаются от внешних команд, таких как Ipconfig.exe по нескольким параметрам:

• Командлеты имеют четко определенную структуру названия: глагол-существительное в единственном числе (англ). Глаголы берутся из конкретного строго оговоренного списка. Такая структура позволяет сделать названия более упорядоченными и легко запоминающимися. Существительные всегда идут в единственном числе: ChildItem” вместо “ChildItems,” или “PSDrive” вместо “PSDrives.” Единственное число используется даже тогда, когда командлет относится к нескольким компонентам.

• Командлеты пишутся на одном из языков .NET Framework, например, Visual Basic или C# и бывают упакованы в пакет .NET Framework с расширением .dll. Эти пакеты называются оснастки или PSSnapin. В действительности оболочка не содержит встроенных командлетов; она автоматически загружает несколько оснасток, когда вы открываете новое окно в оболочке.

Тот факт, что Dir – это псевдоним, объяснят причину, по которой Dir /s не работает: вы используете не привычную команду Dir, а командлет Get-ChildItem, для которого Dir – это псевдоним.

Вы не ограничены использованием псевдонимов, предустановленных в Windows PowerShell. Вы можете создавать свои собственные псевдонимы, импортировать и экспортировать их, делиться ими с коллегами и сотрудниками. Вы можете даже удалять псевдонимы – в Windows PowerShell они хранятся в ALIAS:, поэтому удаление той или иной единицы с данного диска удаляет псевдоним. Однако не стоит беспокоиться о том, что вы случайно можете удалить один из базовых псевдонимов: все изменения действуют лишь в пределах одной сессии, каждый раз при открытии нового окна восстанавливаются исходные параметры. Точно так же

созданные вами новые псевдонимы будут работать только в рамках текущей сессии. Экспорт созданных псевдонимов в отдельный файл упростит их импортирование обратно в будущем.

Вы можете также создать автоматизированный профиль, который будет определять псевдонимы и выполнять прочие стартовые задачи каждый раз при открытии нового окна.

Псевдонимы

- **Get-Alias** – отображение список всех псевдонимов. Также можно использовать Dir Alias: чтобы увидеть все содержимое папки ALIAS: drive.
- **New-Alias** – создание нового псевдонима. Здесь обязательно указать имя нового псевдонима, а также название команды, для которой он предназначен.
- **Del или Rm** – удаление псевдонима из ALIAS: drive.
- **Import-Alias и Export-Alias** – импорт и экспорт псевдонимов в файл и из файла.

Вот несколько команд, которые вы можете использовать для управления псевдонимами:

- **Get-Alias** – отображение список всех псевдонимов. Также можно использовать Dir Alias: чтобы увидеть все содержимое папки ALIAS: drive.
- **New-Alias** – создание нового псевдонима. Здесь обязательно указать имя нового псевдонима, а также название команды, для которой он предназначен.
- **Del или Rm** – удаление псевдонима из ALIAS: drive.
- **Import-Alias и Export-Alias** – импорт и экспорт псевдонимов в файл и из файла.

Использование справки

Одно из главных достоинств графического пользовательского интерфейса (GUI) – это его простота. Возможно, вы уже имели возможность убедиться в этом на собственном опыте, пусть даже и не осознавая это: блуждая по меню, чтобы посмотреть доступные функции, кликая правой кнопкой мыши, чтобы увидеть,

какие действия предлагаются в контекстном меню или наводя курсор на иконки, чтобы увидеть всплывающие подсказки. Все это можно сделать с помощью графического интерфейса, не прилагая особых усилий.

В интерфейсе командной строки этого удобства явно недостает. Традиционно это делает такие оболочки более сложными в изучении, поскольку стартовая точка отсутствует. В Windows PowerShell, однако, предусмотрены некоторые подсказки, которые помогут вам изучить оболочку и выяснить, какие функции вам доступны.

Одной из таких подсказок является встроенная справочная система. Большая часть командлетов Microsoft включает подробное описание. Просто запустите команду **Help**, дополненную названием командлета или псевдонимом, и вы увидите подробное описание данного командлета. В случае, если вы указываете псевдоним, вам будет показано описание командлета, благодаря чему вы сможете узнать его полное название (если все, что вы знали до этого, был псевдоним).

Попробуйте выяснить название командлета для таких псевдонимов, как **Type**, **Cat**, и **Del**.

Если вы знакомы с названием команд в Unix, вы можете использовать **Man** вместо **Help** – результат будет тот же.

Help

Help commandName

- С помощью некоторых параметров команды Help можно получить еще более подробную информацию:
- **-detailed** – показ более детального описания
- **-examples** – показ примеров использования
- **-full** – показ полной информации, включая детальное описание, описание каждого параметра и примеры использования
- **-online** – открытие браузера и показ описания командлетов на сайте Microsoft. На сайте может содержаться обновленная или расширенная информация, которая еще не была выпущена в служебном пакете.

С помощью некоторых параметров команды Help можно получить еще более подробную информацию:

- -**detailed** – показ более детального описания
- -**examples** – показ примеров использования

- **-full** – показ полной информации, включая детальное описание, описание каждого параметра и примеры использования
- **-online** – открытие браузера и показ описания командлетов на сайте Microsoft. На сайте может содержаться обновленная или расширенная информация, которая еще не была выпущена в служебном пакете.

Например, чтобы увидеть полное описание командлета Get-ChildItem, вам необходимо запустить команду

Help gci –full.

Если той информации, что вы почерпнули из данного курса, вам кажется недостаточно, учитесь использовать справочную систему оболочки. Сроднитесь с ней. Пользуйтесь ей как можно чаще, чтобы не только запоминать синтаксис командлетов, но и осваивать их возможности.

Прочитайте описание каждого командлета, который встретится вам в процессе изучения курса. Большинство командлетов имеют дополнительные функции и возможности, знание которых может рано или поздно пригодиться. Также возьмите в привычку использовать параметр – example в совокупности с командой Help. Примеры использования – это лучший способ узнать о возможностях командлетов, запомнить написание и.т.д. Нет нужды искать примеры в Интернете – все необходимые примеры встроены в справочную систему оболочки; ими можно воспользоваться в любое удобное время.

Иногда очень неудобно заглядывать в справку, особенно, если вы уже почти закончили написание длинной команды. Помните, что в таком случае вы можете открыть новое окно и заглянуть в справочник там, не прерывая работы в первом окне.

В процессе изучения справочной информации вы заметите, что командлеты Windows PowerShell обычно поддерживают один или несколько параметров. Краткий справочник по синтаксису в файле справки дает ряд полезных советов о том, как эти параметры работают.

Некоторые параметры не требуют значения. Они называются переключателями и используются, чтобы указать, должна текущая команда отправить дополнительные запросы или активировать другой параметр для запуска команды или нет. В справочнике эти параметры можно узнать по отсутствию значения после названия параметра:

-Recurse

Другие параметры требуют ввода значения – в справочнике указано, какое значение (например, число или строка) необходимо тому или иному параметру:

-Exclude <string[]>

Некоторые параметры являются обязательными – без них командлет не будет функционировать. Если вы забыли указать обязательный параметр, оболочка, скорее всего, потребует указать недостающую информацию:

PS C:\> new-aduser
cmdlet New-ADUser at command pipeline position 1

Supply values for the following parameters:
Name:

Другие параметры являются опционными – их легко отличить по тому, что и сам параметр и его значение заключены в квадратные скобки:

[-Include <string[]>]

Такая форма значения как string[], с пустыми квадратными скобками сразу после значения говорит о том, что параметр может иметь больше одного значения. Один из способов придать параметру несколько значений – напечатать эти значения в ряд, разделяя запятой.

Многие параметры имеют имя, что означает, что вы обязательно должны указывать и имя самого параметра, и его значение. Их можно узнать по отсутствию квадратных скобок вокруг имени параметра, даже если он является опционным:

[-Exclude <string[]>]

Другими словами, если вы захотите использовать этот параметр, вы обязательно должны указать его имя. Перед именем параметра всегда ставится дефис, а значение всегда отделяется от имени пробелом:

-Exclude "*.dll"

Помимо этого, есть так называемые позиционные параметры. Их отличительной особенностью является наличие квадратных скобок только вокруг имени параметра:

[-Path] <string[]>

Позиционные параметры позволяют печатать только значение, при условии, что печатаете вы в соответствующем месте (позиции). То есть, если ранее был напечатан параметр –LiteralPath, вы можете напечатать просто его значение на первой позиции:

Get-ChildItem *.*

Вы можете определить верную позицию, вызвав полную справку с помощью команды –full help:

-Path <string[]>
Specifies a path to one or more locati
Required? false
Position? 1
Default value
Accept pipeline input? true (ByV
Accept wildcard characters? false

Когда вы набираете имя параметра, вам необязательно печатать его полностью. Достаточно набрать лишь первые буквы, чтобы Windows PowerShell определила, какой именно параметр вы имеете в виду. Например, вместо того, чтобы набирать –computerName, вы можете напечатать просто –comp, или даже –c, если командлет не имеет других параметров, имя которых начинается с буквы с.

Расширение оболочки

Командлеты, присутствующие по умолчанию в оболочке, не являются единственными доступными для вас командлетами. Microsoft, так же, как и сторонние разработчики программного обеспечения, могут создавать дополнительные командлеты и провайдеры PSDrive, и предоставлять их вам в виде оснасток или модулей. Управление оснастками осуществляется с помощью набора командлетов, название которых включает существительное PSSnapin:

- Get-PSSnapin
- Add-PSSnapin
- Remove-PSSnapin

Расширение оболочки

- Командлеты, присутствующие по умолчанию в оболочке, не являются единственными доступными для вас командлетами. Microsoft, так же, как и сторонние разработчики программного обеспечения, могут создавать дополнительные командлеты и провайдеры PSDrive, и предоставлять их вам в виде оснасток или модулей. Управление оснастками осуществляется с помощью набора командлетов, название которых включает существительное PSSnapin:
  - Get-PSSnapin
  - Add-PSSnapin
  - Remove-PSSnapin
  - Import-Module
  - Remove-Module
  - Get-Module

Get-PSSnapin –registered
Get-Module –list

Оснастки, как правило, следует устанавливать с использованием установочного пакета, так как они должны быть зарегистрированы в оболочке. Чтобы увидеть список зарегистрированных оснасток (кроме тех, что установлены по умолчанию), запустите команду:

Get-PSSnapin –registered

О прочих возможностях командлета Get-PSSnapin можно узнать в справочнике.

Модули, в свою очередь, могут не требовать установки, хотя это зависит от того, какие именно функции выполняет тот или иной модуль. Windows PowerShell располагает тремя основными командлетами для управления модулями:

• Import-Module

• Remove-Module

• Get-Module

Чтобы увидеть список установленных модулей, запустите команду:

Get-Module –list

Windows PowerShell ищет модули в конкретных папках. Чтобы увидеть, в каких папках осуществляется поиск, запустите команду:

Type Env:\PSModulePath

Вы можете модифицировать эту переменную среды, изменяя или добавляя папки, в которых хранятся модули. Обратите внимание, что расширения иногда могут изменять командлеты, присутствующие в оболочке по умолчанию, тем самым меняя их поведение.

Использование конвейеров

Акт передачи выходных данных одного командлета во входные данные другого командлета называется конвейеризацией. Возможно, в других оболочках вам уже приходилось использовать конвейер. Например, это стандартная команда в Cmd.exe:

Dir | More

Здесь выходные данные команды Dir перенаправляются во входные данные команды More, которая создает постраничное отображение выходных данных.

Конвейер нашел широкое применение в Windows PowerShell. Весьма распространенным явлением здесь является строка из полдюжины командлетов, связанных между собой конвейером. Данные переходят из одного командлета в другой, при этом они постепенно уточняются, детализируются и превращаются именно в ту информацию, которая вам требуется.

Конвейеризация (Piping)

- Акт передачи выходных данных одного командлета во входные данные другого командлета называется конвейеризацией. Возможно, в других оболочках вам уже приходилось использовать конвейер. Например, это стандартная команда в Cmd.exe:

```
dir | more
```

- Здесь выходные данные команды Dir перенаправляются во входные данные команды More, которая создает постраничное отображение выходных данных.
- Конвейер нашел широкое применение в Windows PowerShell. Весьма распространенным явлением здесь является строка из полдюжины командлетов, связанных между собой конвейером. Данные переходят из одного командлета в другой, при этом они постепенно уточняются, детализируются и превращаются именно в ту информацию, которая вам требуется.

Командлеты в Windows PowerShell ориентированы на такой способ работы в большей степени, чем команды в любой традиционной оболочке Windows или Unix. Конвейер позволяет выполнять очень сложные и мощные задачи без необходимости написания длиннейших скриптов и программ. Конвейеризация является настолько мощной и важной концепцией, что следующий раздел курса будет полностью посвящен ей.

Все команды в рамках Shell выполняются в виде конвейера. Чтобы лучше понять принцип действия такого конвейера, представьте трубу, по которой течет поток информации. Каждый командлет – это насосная станция, а труба соединяет каждую станцию с последующей. Каждая станция забирает информацию из трубы и добавляет туда новые данные, после чего видоизмененный поток движется к следующей станции.

Вывод командлетов

- Основные командлеты для модификации вывода
  - Format-Table, имеет alias Ft
  - Format-List, имеет alias Fl
  - Format-Wide, имеет alias Fw

```
Get-Service | Format-List
Get-Process | Fw
```

Команды выполняются в одностороннем порядке – слева направо, а знак (“|”) является разграничителем между командлетами. Вы уже видели такие примеры, скажем, этот:

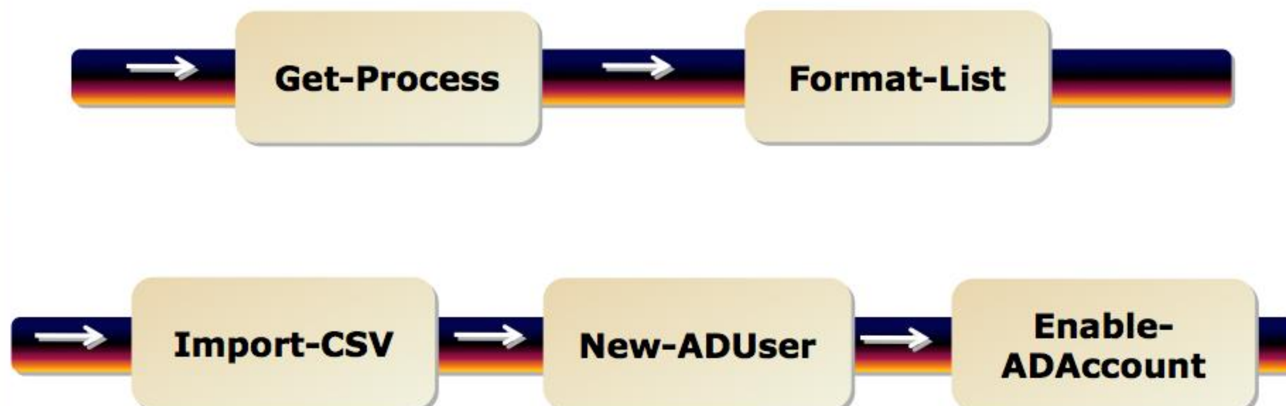
Get-Process | Format-List

Это простейший пример использования конвейера. В действительности с его помощью можно создавать значительно более сложные и мощные команды. Например, чуть позже вы узнаете, как написать командную строку вроде этой:

Import-CSV c:\new_users.csv | New-ADUser –passThru | Enable-ADAccount.

Эта простая трехкомпонентная командная строка позволяет создать сотни новых пользователей Active Directory и активировать их аккаунты. При старых административных технологиях выполнение этой задачи могло потребовать десятки строк на скриптовом языке программирования; с командлетами и конвейером Windows PowerShell данная задача выполняется исключительно с помощью синтаксиса, который вы только что видели (конечно, если были выполнены некоторые предварительные условия, о которых вы узнаете чуть позже).

Конвейер



Windows PowerShell – это, конечно, не первая оболочка, в которой используется технология конвейера. В оболочках Unix уже давным-давно информация от команды к команде передается по конвейеру, да и в Windows (в частности, в Cmd.exe) используется нечто подобное. Проблема же с этими старыми оболочками заключалась в том, что они передавали от одной команды к другой не что иное, как текст. Например, представьте себе, что команда под названием Plist передает текстовое описание текущего процесса, которое выглядит приблизительно так:

Pid Name Image
--- ------------------------ -------------
092 Notepad notepad.exe
098 Windows Paint mspaint.exe
112 Calculator calc.exe
164 Windows PowerShell powersehll.exe

Предположим, вашей целью является завершить работу всех копий Windows Notepad. У вас есть команда под названием Kill, которая принимает ID (PID) процесса и завершает этот процесс. Но вы не можете просто запустить команду:

Plist | Kill

Для начала вы должны определить, какая строка выходных данных Plist относится к Notepad; затем вам нужно сократить эту строку до PID, который требуется команде Kill. В Unix все это осуществляется с использованием утилит, анализирующих текст, таких как Sed, Grep, и Awk. Например, обычной для Unix является такая команда:

```
ps auxwww | grep httpd
```

Проблема при таком подходе состоит в том, что вам приходится тратить больше времени на анализ и модификацию текста, нежели на выполнение самой задачи. Скриптовые языки вроде Perl приобрели популярность исключительно из-за их мощных возможностей анализа текста. Администраторы, использующие такие оболочки, должны быть экспертами в области анализа текста, хотя технически эти манипуляции не имеют ничего общего с административными задачами.

Windows PowerShell позволяет отказаться от текстовых выходных данных в пользу другой их формы. Командлет Windows PowerShell всегда производит элементы, которые представляют собой компоненты операционной системы вместо текста.

Что же это за элементы? По сути элемент – это самостоятельная структура данных, которая описывает часть функционирования операционной системы или серверной технологии. Процесс, например, является частью функционирования операционной системы. Сервис – это часть операционной системы. Файл или папка – часть операционной системы. Пользовательский аккаунт – это элемент операционной системы, точно так же, как и группа, организационная единица или сам домен. Вы можете воспринимать элемент как разновидность структурной единицы, с которой Windows PowerShell работает очень легко и быстро. Вместо того, чтобы давать оболочке команду «пропустить 10 символов в каждом ряду, чтобы найти колонку, которая содержит название процесса», вы можете просто дать ей задание обратиться к атрибуту Name, поскольку структура, используемая в конвейере, создана таким образом, что атрибуты становятся легкодоступными.

Терминология

Windows PowerShell использует терминологию, которая иногда кажется тесно связанной с разработкой программного обеспечения. Помните, что это всего лишь слова, и их употребление не означает, что вам придется стать программистом, чтобы начать пользоваться Windows PowerShell. Например, в предыдущем разделе вы видели, что слово «элемент» используется для обозначения того, что командлет отправляет по конвейеру. Например, командлет Get-Process помещает элементы процесса в конвейер. Более официальным термином здесь будет слово «объект», т.е. вы можете сказать, что командлет Get-Process помещает объекты процесса в конвейер.

Терминология

- командлет Get-Process помещает элементы процесса в конвейер. Более официальным термином здесь будет слово «объект», т.е. вы можете сказать, что командлет Get-Process помещает объекты процесса в конвейер.
- Эти объекты имеют атрибуты. Для процесса атрибутами могут быть имя, ID, объем занимаемой памяти и.т.д. Формальным названием атрибута будет термин «свойство». Другими словами, вы можете сказать, что объект процесса обладает свойством имени, свойством ID, и.т.д.

Pid	Name	Image
092	Notepad	notepad.exe
098	Windows Paint	mspaint.exe
112	Calculator	calc.exe
164	Windows PowerShell	powershell.exe

Эти объекты имеют атрибуты. Для процесса атрибутами могут быть имя, ID, объем занимаемой памяти и.т.д. Формальным названием атрибута будет термин «свойство». Другими словами, вы можете сказать, что объект процесса обладает свойством имени, свойством ID, и.т.д.

Объекты обычно имеют более одного свойства, каждое из которых описывает тот или иной атрибут объекта. Другие оболочки могут отображать эти атрибуты в форме списка или таблицы. Например, объект процесса можно отобразить в виде такого текста:

Pid	***Name***	***Image***
092	***Notepad***	***notepad.exe***
098	***Windows Paint***	***mspaint.exe***
112	***Calculator***	***calc.exe***
164	***Windows PowerShell***	***powershell.exe***

Однако, как вы уже знаете, совершение различных манипуляций с текстом занимает слишком много времени. Структура данных объекта, в свою очередь. Во многом упрощает доступ как к одному свойству объекта, так и ко всему набору его свойств. В рамках данного курса мы будем употреблять термин «свойство» наряду с термином «атрибут», чтобы время от времени напоминать читателям, что название – это не главное. Намного важнее – понять, что Windows PowerShell обеспечивает простой и быстрый доступ к данной информации через конвейер.

Свойства объектов

Поскольку свойства стали доступными без необходимости текстово-синтаксического анализа, они стали обеспечивать более быстрый и простой доступ к определенным частям информации. Командлеты Windows PowerShell производят объекты, а также могут принимать объекты как входящие данные. Конкретные параметры командлетов могут создаваться для поиска конкретных свойств входящих объектов, а если эти свойства найдены, командлет может использовать данную информацию в качестве входящих данных параметра.

Например, представьте, что командлет Get-Service имеет параметр, который называется –computerName. Данный параметр будет уточнять имя удаленного компьютера, с которого вы хотите получить службы:

Get-Service –computerName SEA-SRV2

Данный параметр также может быть создан разработчиком командлета для поиска свойства «Имя компьютера» у входящего объекта. Предположим, у вас есть командлет под названием Get-ComputerInventory, который извлекает список компьютеров из конфигурационной базы данных. Каждый компьютер в базе данных будет выступать в качестве объекта, а каждый объект может иметь несколько свойств, включая свойство «Имя компьютера» (ComputerName). В данном случае для извлечения служб из всех обслуживаемых компьютеров можно запустить следующую команду:

Get-ComputerInventory | Get-Service

Нет нужды отображать инвентаризационную информацию в виде текстовой таблицы, подвергать анализу колонку «Computer Name», и.т.д., так как вы работаете не с текстом. Windows PowerShell передает объекты по конвейеру от одного командлета к другому и устанавливает связь между свойствами объекта и параметрами второго командлета.

В действительности командлета Get-ComputerInventory, встроенного в Windows PowerShell нет, но командлет Get-Service функционирует именно так, как было описано. Вы увидите примеры его использования чуть позже.

Другим исследовательским командлетом PowerShell является командлет Get-Member. Если вы поместите объект в конвейер к нему, то получите список членов класса объекта, который будет включать все свойства объекта. Например

Get-EventLog Security –newest 10 | Get-Member

Вы также можете поместить объект в конвейер к командлету Format-List * для просмотра списка объектов, который будет включать все свойства объекта.

Get-Process | Format-List *

Другим хорошим путем является использование Get-Member вместе с командлетом Out-GridView

Get-Process | Format-List *

Перенаправления ввода-вывода и форматирование

- Имеется ряд Out- командлетов
  - Out-Gridview
  - Out-Printer
  - Out-File
  - Out-Host
  - Out-Null

Помимо командлета Out-GridView, с которым вы уже встречались, оболочка предлагает еще несколько командлетов Out-. Каждый из них предназначен для перенаправления выходных данных в конкретное место:

- Out-Printer
- Out-File
- Out-Host

Out-Host перенаправляет данные в окно консоли, что является действием по умолчанию, о котором вы уже слышали. Out-File – исключительно полезный командлет; при указании имени файла он перенаправляет данные непосредственно в этот файл. При этом объект не форматируется, а значит, предварительно следует использовать один из форматирующих командлетов:

Get-EventLog Application –newest 100 | Format-List * | Out-File events.txt

Не забывайте обращаться к справочной информации! Out-File, например, может иметь множество параметров, которые позволяют уточнить кодировку символов, желаемое количество знаков в каждой строке выходного текстового файла, и.т.д.

Out-Null – еще одна опция, имеющая специальное предназначение. Иногда командлеты превращают объекты не в то, что вам нужно. Чтобы отменить данные изменения, используется командлет Out-Null:

Get-Service | Out-Null

Вы также можете использовать старый синтаксис > и >> для перенаправления объекта в файл. В данном случае командлет Out- File используется скрыто, что упрощает отправку данных в текстовый файл, хотя при этом вы лишаетесь возможности применить дополнительные параметры Out- File:

```
Get-Process > procs.txt
```

На заметку: Если вы не знакомы с данным синтаксисом, запомните, что > отправляет объект в новый файл, заменяя любой существующий файл с таким же названием (Out-File ведет себя точно так же по умолчанию), а >> добавляет объект в конец существующего объекта (так же, как это делает параметр –append командлета Out-File).

Windows PowerShell передает объекты по конвейеру до его конца. На конце конвейера оставшиеся объекты передаются в специальный командлет, который называется Out-Default. Его не нужно прописывать вручную, поскольку он уже встроен в конвейер. Его работа заключается в том, чтобы принять получившиеся объекты и передать их на консольный командлет Out-Host, который отображает выходные данные на экране в текстовом виде. Поскольку объекты не конвертируются в текст до конца конвейера, вы имеете доступ к их свойствам до самого конца. Это означает возможность создавать сложные команды без необходимости текстового анализа.

Это общее правило имеет два исключения:

- Командлеты Format- производят объекты, но эти объекты являются крайне специфическими – их «понимают» только командлеты Out-. Это означает, что Format- командлет должен быть либо последним в конвейере, либо предпоследним (перед командлетом Out-).
- Большинство командлетов Out- не производят никаких объектов; они лишь отправляют объекты (например, текст) на какое-либо устройство (файл, принтер или окно консоли). Поэтому, командлеты Out- следует располагать в конце конвейера.

Свойства объектов

- PowerShell командлеты возвращают объекты
- PowerShell командлеты используют объекты как входные данные
- Командлеты можно заставить использовать свойства входящих объектов в качестве параметров

```
Get-Service –computerName SEA-SRV2
```

- Но параметр можно получить из свойства, переданного другим командлетом

Во многом мощь и гибкость Windows PowerShell связаны с возможностью передавать объекты от одного командлета к другому по конвейеру. Поскольку эти командные строки – даже очень длинные – представляют собой всего одну логическую цепочку текста внутри оболочки, их часто называют однолинейниками (one-liners). В большинстве случаев один такой однолинейник может выполнить те же самые задачи, для решения которых потребовалось бы множество строк, написанных на скриптовом языке в рамках любой их старых технологий, например, Visual Basic Scripting Edition (VBScript). Например, этот однолинейник с использованием техник, которые мы рассмотрим позже, объединяет все имена компьютеров, перечисленных в c:\computers.txt и даже создает отчет о размере журнала регистрации событий:

```
Get-Content c:\computers.txt | where {($_.Trim()).length -gt 0} | foreach {
Get-WmiObject Win32_NTEventLogFile -computer $_.Trim() `
-filter "NumberOfRecords > 0" | Select-Object `
@{Name="Computername";Expression={$_.CSName}},LogFileName,NumberOfRec
ords,`
@{Name="Size(KB)";Expression={$_.FileSize/1kb}},`
@{Name="MaxSize(KB)";Expression={($_.MaxFileSize/1KB) -as [int]}}, `
@{name="PercentUsed";Expression={"{0:P2}" -f ($_.filesize/$_.maxFileSize)}}
} | Sort Computername | Format-Table -GroupBy Computername `
-property LogFileName,NumberOfRecords,*Size*,PercentUsed
```

Конечно, это сложная команда, но она прекрасно иллюстрирует мощь и гибкость конвейера.

Когда вы начнете работать с Windows PowerShell, основной причиной ошибок и затруднений могут стать ваши предположения об именах свойств или значений, которые они содержат. Например, если вы предположите, что существует свойство с именем DriveLetter, тогда как в действительности его имя – DeviceID, вы можете потерять несколько минут на выяснение этого факта. Создание однолинейника или скрипта, в котором предполагается значение FixedDisk для свойства DriveType, может повлечь проблемы, если на самом деле это свойство имеет значение «3».

Вывод из всего этого заключается в том, что никогда не надо строить предположения. Всегда используйте Get-Member, чтобы уточнить имя свойства, и перенаправляйте объекты в Format-List *, чтобы уточнить содержание этих свойств. Проверка займет совсем не много времени и позволит вам избежать ошибок и сэкономить время.

Основы системы форматирования

Не забывайте, что почти все командлеты Windows PowerShell работают с операционной системой и другими элементами, содержащими огромное количество информации. Поэтому, будет непрактичным пытаться вывести на дисплей всю информацию – на компьютерном мониторе просто не хватит места для этого, а осуществлять горизонтальную прокрутку для поиска нужных данных не очень-то удобно. Поэтому, по умолчанию в оболочке отображается часть информации в форме, наиболее удобной для восприятия. Например, процесс может иметь несколько свойств, каждое из которых передает потенциально важную информацию.

Однако ни одного по-настоящему эффективного способа отобразить всю эту информацию в удобном формате нет, поэтому, командлет Get-Process, передавая выходные данные, не пытается этого сделать. Вместо этого оболочка показывает только семь свойств одного процесса, и делает это в довольно простой и удобной форме – в виде таблицы. Тем не менее, эти параметры не являются жестко запрограммированными, и при желании, приложив немного усилий, вы можете отменить их, или даже заменить новыми. Все стандартные настройки вы получаете в файлах XML-формата при установке Windows PowerShell.

Все форматирование по умолчанию основано на имени типа того элемента, который отображается. Имя типа можно увидеть, передав необходимый элемент по конвейеру в командлет Get-Member; после того как конвейер доставляет множество различных типов элементов, оболочка, как правило, выбирает по умолчанию формат выходных данных, основанный на имени типа первого элемента в конвейере.

Когда оболочке требуется использовать форматирование по умолчанию, она следует ряду правил, чтобы достичь этого.

Форматирование по умолчанию.

Правило первое – определен ли формат.

После того, как оболочка определила имя типа того элемента, который требуется отобразить, первым делом она проверяет, определен ли формат просмотра для

данного типа. Форматы просмотра указаны в специальных конфигурационных XML-файлах; некоторые из этих файлов устанавливаются вместе с Windows PowerShell. Попробуйте запустить следующую команду:

Dir $pshome

Rule 1: Определен ли формат?

- После того, как оболочка определила имя типа того элемента, который требуется отобразить, первым делом она проверяет, определен ли формат просмотра для данного типа
  - Форматы просмотра указаны в специальных конфигурационных XML-файлах
  - Файлы, название которых имеет расширение .format.ps1xml, содержат определенный формат просмотра
  - Для нахождения этих файлов используйте команду:

Dir $pshome

Файлы, название которых имеет расширение .format.ps1xml, содержат определенный формат просмотра. Оболочка загружает эти файлы каждый раз при запуске и хранит информацию в памяти в течение одной сессии.

Файлы .format.ps1xml, включенные в установочный пакет оболочки, поставляются компанией Microsoft® и обладают цифровой подписью Microsoft. Эти специфические файлы загружаются даже тогда, когда скрипты отключены, поэтому, оболочка может содержать множество параметров «по умолчанию». Цифровая подпись гарантирует, что файлы не могут быть просто изменены без предупреждения; если же файл каким-либо образом изменить (даже случайно), то подпись прекратит действовать, и оболочка перестанет загружать этот файл автоматически.

Если оболочке необходимо отобразить элемент, для которого уже определен формат просмотра, она отобразит его именно в этом формате. Если предусмотрено несколько вариантов просмотра, то оболочка выберет тот вариант, который первым записан в памяти. Это означает, что порядок, в котором XML-файлы загружены в память, имеет значение, поскольку первый из них всегда будет иметь преимущество над последующими.

Форматы просмотра могут быть очень простыми – список, таблица, широкая или пользовательская форма. Также они могут быть и очень сложными – с ярлыками и расширениями. Они могут даже содержать скриптовые коды для подсчета или получения значений тех или иных колонок таблицы.

Если формат просмотра не определен, в силу вступает **второе правило**.

Rule 2: Какие свойства должны отображаться?

- Ести формат не определен
  - Чтобы принять решение, оболочка проверяет, зарегистрирован ли тип расширения DefaultDisplayPropertySet для того имени, которое необходимо отобразить.
  - Типы расширений хранятся в XML-файлах, так же, как и форматы просмотра
  - Лежат там же:

```
Dir $pshome
```

- Не модифицируйте файлы форматов – они подписаны, создавайте свои

Это правило определяет, какие свойства элемента должны отображаться. Если формат предопределен, то в нем уже указано, какие свойства должны отображаться, следовательно, второе правило не нужно. Если нет – оболочке необходимо решить, какие свойства показать. Чтобы принять решение, оболочка проверяет, зарегистрирован ли тип расширения DefaultDisplayPropertySet для того имени, которое необходимо отобразить. Типы расширений хранятся в XML-файлах, так же, как и форматы просмотра. Если вы запустите команду Dir $pshome, то увидите XML-файлы, определяющие типы расширения. Имя этих файлов имеет расширение .types.ps1xml, вместо .format.ps1xml.

Файлы .types.ps1xml, так же, как и .format.ps1xml, обладают цифровой подписью Microsoft и не подлежат изменению.

В оболочке хранятся типы расширения для многих имен типов элементов. Однако в действительности их намного больше – DefaultDisplayPropertySet является лишь одним из многих доступных расширений. Только то, что имя типа появилось в файле а .ps1xml не обязательно означает, что расширение DefaultDisplayPropertySet

зарегистрировано. Тем не менее, если имя было зарегистрировано, оболочка будет использовать свойства, указанные в расширении, каждый раз. Если нет – будут отображаться все свойства элемента (**правило номер 3**).

После всех перечисленных действий оболочка просто задает себе вопрос: сколько свойств элемента необходимо показать – либо те, что определены в DefaultDisplayPropertySet, либо все.

Rule 3: Table или List?

- сколько свойств элемента необходимо показать – либо те, что определены в DefaultDisplayPropertySet, либо все.
- Если оболочке требуется отобразить не более четырех свойств, используется таблица. Если пять и более – используется список. Это правило гарантирует, что таблица поместится в стандартное окно консоли стандартного размера.
- После того, как оболочка определила, какую форму (таблицу или список) следует использовать, она начинает создавать эту форму. Для этого происходит внутреннее обращение к командлету Format-List или Format-Table, куда и передаются элементы, которые необходимо отформатировать и показать.

Если формат просмотра был определен правилом №1, то правило №3 не учитывается. При заранее определенном формате просмотра всегда уточняется то, в каком виде будут отображаться свойства – список, таблица, и.т.д.

Если оболочке требуется отобразить не более четырех свойств, используется таблица. Если пять и более – используется список. Это правило гарантирует, что таблица поместится в стандартное окно консоли стандартного размера.

После того, как оболочка определила, какую форму (таблицу или список) следует использовать, она начинает создавать эту форму. Для этого происходит внутреннее обращение к командлету Format-List или Format-Table, куда и передаются элементы, которые необходимо отформатировать и показать.

Out-Default

Рассмотрим вкратце, как происходит форматирование в рамках оболочки. В конце каждого командного конвейера находится командлет Out-Default. Он всегда находится там, даже если вы не указали его в командной строке. Его работа заключается в том, чтобы принять окончательные выходные данные из конвейера

и передать их командлету Out-Host, который отвечает за вывод информации на экран. Если вы наберете команду:

Get-Process

То вы запускаете (пусть даже не осознавая этого) команду:

Get-Process | Out-Default

А в обычном окне консоли или Windows PowerShell ISE это означало бы то же самое, что и:

Get-Process | Out-Host

Именно поэтому последние командлеты используются по умолчанию: выходные данные внутренне отправляются на Out-Default, а затем – на Out-Host.

Out-Default

- . В конце каждого командного конвейера находится командлет Out-Default. Он всегда находится там, даже если вы не указали его в командной строке. Его работа заключается в том, чтобы принять окончательные выходные данные из конвейера и передать их командлету Out-Host, который отвечает за вывод информации на экран.
- Если вы наберете команды, то получите одинаковые результаты

Get-Process
Get-Process | Out-Default
Get-Process | Out-Host

- Вы не должны вызывать Out-Default. Однако Вы мажете вызвать другие командлеты для перенаправления вывода.

Но что будет, если вы вручную наберете другое место назначения, скажем, используя командлет Out- File?

Get-Process | Out-File c:\procs.txt

Большинство командлетов Out- не производят никаких выходных данных, а значит, не отправляют ничего в конвейер (Out- String – особенный случай, который является исключением из правила). Поэтому, даже если предыдущая команда была передать данные в файл, в действительности конвейер (даже если вы не осознаете этого) будет выглядеть так:

Get-Process | Out-File c:\procs.txt | Out-Default

А функционально происходить будет следующее:

```
Get-Process | Out-File c:\procs.txt | Out-Host
```

Вы не увидите ничего на экране при запуске команды, поскольку командлет Out-File не производит никаких выходных данных. Он лишь принимает входящие данные и передает текстовую интерпретацию этих данных в файл, но после этого в конвейере ничего не остается. Хотя Out-Default продолжает работать, он не получает никаких данных, поэтому на экран ничего не выводится.

Out- и Format-

Командлеты Out- технически неспособны «понять» такие вещи как процессы или службы. Они работают только со специальными инструкциями по форматированию, которые производятся только командлетами Format-. Как только командлет Out- обнаруживает, что ему приходится иметь дело с чем-то другим, кроме инструкции форматирования, он автоматически запускает форматирующую систему в оболочке, основываясь на трех правилах, которые мы рассмотрели ранее. Данные передаются в форматирующий командлет, после чего поступают обратно в командлет Out-.Поэтому, когда вы запускаете команду:

```
Get-Process
```

В действительности оболочка выполняет намного больше действий, и весь процесс выглядит так:

```
Get-Process | Out-Default | Out-Host | Format-Table | Out-Host
```

Форматирующая система определяет, какой из Format- командлетов следует использовать. В данном случае для выходных командлета Get-Process в оболочке зарегистрирован определенный формат просмотра – таблица, поэтому, используется командлет Format-Table.

Вы можете извлечь пару довольно важных уроков, проанализировав поведение оболочки:

• Как только командлет Format- начинает работу, первоначальные выходные данные теряются, поскольку подвергаются форматированию. Сравните результат двух команд:

```
Get-Process | Get-Member
```

или

```
Get-Process | Format-Table | Get-Member
```

• Командлет Format-Table поглощает элементы процесса и превращает их в инструкции по форматированию, которые понимает только командлет Out-. Поэтому, командлет Format- , как правило, должен быть последним в командной строке.

• Командлеты Out-, как уже упоминалось ранее, не производят никаких выходных данных. Поэтому, если они используются, то располагаются в самом конце командной строки (кроме командлета Out-String, который является исключением). Если в цепочке команд используются и командлет Format, и командлет Out-, то Format- будет предпоследним, а Out- последним.

Все командлеты Out- способны воспринимать одни и те же инструкции по форматированию. Поэтому, обе команды:

```
Get-Process | Fl * | Out-Host
Get-Process | Fl * | Out-File c:\proclist.txt
```

производят идентичные выходные данные; разница заключается лишь в ширине (некоторая информация может быть сокращена или свернута), а также в том, что в пером случае данные выводятся на экран, а во втором – сохраняются в файл.

Варианты отображения

Вы можете создать файл формата просмотра (.format.ps1xml), содержащий более одного варианта просмотра для данного имени типа. Так, файл DotNetTypes.format.ps1xml содержит более одного варианте просмотра для имени System.Diagnostics.Process.

Каждый раз при запуске системы форматирования по умолчанию используется тот формат просмотра, который первым зарегистрирован в памяти. Поэтому, команда Get-Process произведет те же самые выходные данные, что и Get-Process | Format-Table. Данные будут идентичными потому, что в первом случае вы не уточнили формат просмотра. Оболочка следует правилу №1, находит зарегистрированные варианты просмотра в DotNetTypes.format.ps1xml, использует самый первый вариант для имени System.Diagnostics.Process. Этот первый вариант – таблица. Во втором случае вы уточнили, что данные должны быть отображены в таблице, что совпало с форматом просмотра, используемым по умолчанию. Однако вы могли бы вручную указать другой формат просмотра с помощью параметра –view командлета Format-. Здесь существует одно правило. Вариант просмотра, который вы указываете, должен совпадать с параметром командлета Format-. Другими словами, если для имени Diagnostics предусмотрен только один вариант просмотра – таблица, то вы должны использоват командлет Format-Table. Если же вы попытаетесь запустить команду:

```
Get-Process | Format-List –view diagnostics
```

а варианта отображения имени diagnostics в виде списка не существует, вы увидите ошибку, даже если вариант показа в форме таблицы существует.

Например, данная команда будет работать:

```
Get-Process | Ft –view priority
```

так как для имени Priority предусмотрен формат просмотра – таблица. Однако следующая команда работать не будет:

```
Get-Process | Fl –view priority
```

потому что командлет Format-List не может использовать формат таблицы, предусмотренный для имени Priority.

Различные форматы

- ***Get-Process | Format-List –view diagnostics***
  - ***Format-List : View name diagnostics cannot be found.***
- ***Get-Process | Ft –view priority***
  - ***Все ОК***
- ***Get-Process | Fl –view priority***
  - ***Format-List : View name priority cannot be found.***

Вы можете создавать в оболочке свои собственные файлы .format.ps1xml для новых или дополнительных вариантов просмотра имен по умолчанию.

Не пытайтесь изменить файлы .format.ps1xml с цифровой подписью Microsoft, которые поставляются вместе с Windows PowerShell.

Вместо этого вы можете создавать новые необходимые вам файлы в формате XML. Файлы Microsoft можно использовать в качестве образца, но ни в коем случае не нужно пытаться их изменить. После того, как вы создали файл, используйте командлет Update-FormatData, чтобы загрузить его в оболочку и зарегистрировать в качестве варианта просмотра по умолчанию в системе форматирования. При запуске этого командлета вы должны указать путь к вашему XML-файлу. Для этого необходимо следующее:

• Используйте параметр –append, чтобы загрузить ваш файл после существующего в памяти оболочки. Это удобно в тех случаях, когда необходимо создать дополнительный вариант просмотра для того или иного имени. Но если в оболочке уже был предусмотрен другой формат просмотра, то по умолчанию будет использоваться именно этот, существующий вариант.

• Если вы хотите, чтобы созданный вами файл сохранился в памяти перед существующим, используйте параметр –prepend. В данном случае по умолчанию будет использоваться именно ваш файл.

Подробная инструкция создания файлов .format.ps1xml не предусмотрена в рамках данного курса. В некоторых книгах этот вопрос раскрыт более подробно, например, в Windows PowerShell v2.0: TFM под авторством Don Jones и Jeffery Hicks.

Не забывайте о том, что любые изменения, вносимые пользователем в оболочку, действуют только в течение одной сессии; оболочка не загружает созданные вами файлы автоматически каждый раз. Если вы хотите, чтобы созданный вами файл был доступен каждый раз при запуске новой сессии, создайте специальный скрипт для Windows PowerShell, который будет запускать командлет Update- FormatData каждый раз при открытии нового окна.

Дополнительные данные

- Для того чтобы добавить пользовательские свойства к объекту, используется команда Select-Object. Например, чтобы добавить атрибут ComputerName к элементу Computer, у которого уже есть атрибут Name, можно запустить команду:

```
Get-ADComputer –filter * |
 Select
*,@{Label='ComputerName';Expression={$_.Name}}
```

- если вы хотите просто добавить свои собственные колонки в таблицу, вместо того, чтобы придавать новые свойства объекту, вы можете запустить команду:
- ***Get-ADComputer –filter * | Ft DnsHostName,Enabled,@{Label='ComputerName';Expression={$_.Name}}***

Для того чтобы добавить пользовательские свойства к объекту, используется команда Select-Object. Например, чтобы добавить атрибут ComputerName к элементу Computer, у которого уже есть атрибут Name, можно запустить команду:

```
Get-ADComputer –filter * |
Select *,@{Label='ComputerName';Expression={$_.Name}
```

Команда Select-Object обеспечивает доступ ко всем свойствам компьютеров, которые были определены с помощью Get-ADComputer и группового символа *. Помимо этого, Select-Object добавляет новые свойства этим компьютерам. В данном случае новое свойство имеет ярлык ComputerName, что в командной конструкции обозначено с помощью команды Label. Такая часть конструкции как Expression показывает значение, которое содержит данное свойство. Expression сопровождается скриптовым блоком, заключенным в фигурные скобки. Внутри этого скриптового блока структурный нуль $_ указывает, на что направлена команда Select-Object – в данном случае, это компьютеры. Точка после $_ указывает на то, что вы хотите обратиться к существующему свойству, в данном случае, Name.

Командлет Format-Table принимает тот же самый синтаксис для создания новых колонок в таблице. Это означает, что если вы хотите просто добавить свои собственные колонки в таблицу, вместо того, чтобы придавать новые свойства объекту, вы можете запустить команду:

```
Get-ADComputer –filter * |
Ft DnsHostName,Enabled,@{Label='ComputerName';Expression={$_.Name}}
```

Обратите внимание, что для создания новых колонок в таблице используется та же самая конструкция, а такие свойства компьютеров как DnsHostName и Enabled помещаются в собственные новые колонки.

В чем же разница? Если вам необходимо изменить элемент в конвейере, а затем передать его в другой командлет, вы можете использовать опцию Select-Object для добавления новых свойств элементу. Если же вы просто хотите создать новую колонку в таблице, используйте опцию Format- Table. В принципе, не случится ничего страшного, если вы используете опцию Select-Object, а затем передадите результат по конвейеру в командлет Format-Table для вывода данных в виде таблицы. Обе техники дают один и тот же результат. Просто не забывайте о том, что командлет Format-Table обычно идет в самом конце конвейера, и что данные из него можно передавать только командлету Out-. Элемент Expression в конструкции также может содержать математические знаки, такие как +, -, * или /, которые обозначают, соответственно, прибавление, вычитание, умножение или деление. Например, таким образом вы можете добавить значения двух свойств:

```
Get-WmiObject Win32_LogicalDisk |
Ft DeviceID,Size, @{Label='SpaceUsed';Expression={$_.Size - $_.FreeSpace}}
```

Здесь командлету Format-Table дана команда создать колонки для свойств логических дисков DeviceID и Size, которые были извлечены командлетом Get-WmiObject. Вдобавок к этим двум колонкам он создает третью, под названием SpaceUsed. Эта колонка содержит значение, равное разнице между общим размером диска и оставшейся свободной памятью. Как вы видите, здесь используется знак вычитания для осуществления математического вычисления.

Техника использования $_ в качестве структурного нуля для обозначения элемента, на который направлена команда, а также последующая точка, указывающая, к какому свойству следует обратиться, является ключевой техникой в Windows PowerShell. Позже вы еще неоднократно будете встречаться с этим синтаксисом, поэтому, постарайтесь свыкнуться с ним.

Такой элемент командной строки как Expression может содержать практически любой скрипт или команду Windows PowerShell. В первом примере это был просто доступ к атрибуту Name тех элементов, которые поступают в командлет Format-Table, хотя вы можете использовать намного более сложные значения, например:

```
get-adcomputer -filter * |
ft dnshostname,@{
Label="OSVersion";
Expression={
(gwmi win32_operatingsystem -comp $_.Name).caption
}
}
```

Это уже достаточно сложная команда. Вот что здесь происходит:

• Командлет Get-ADComputer извлекает данные обо всех компьютерах в домене (хотя и с осторожностью, так как в большом домене данный процесс может быть ресурсоемким).

• Данные о компьютерах передаются в командлет Format-Table.

• Format-Table показывает атрибут DNSHostName.

• Также Format-Table создает дополнительную колонку под названием OSVersion. Содержимое этой колонки – это данные, которые отображает командлет Get-WmiObject (использует alias gwmi).

• Get-WmiObject соединяется с именем компьютера, указанным в атрибуте Name входящих данных в Format-Table – это то имя, которое мы показываем.

• Командлет Get-WmiObject находится внутри круглых скобок, которые обозначают, каким должен быть результат выполнения команды. В данном случае результатом является информация об операционной системе компьютера.

• Точка после скобок обозначает, что мы хотим получить доступ к свойству информации WMI, полученной в результате выполнения команды. В данном случае мы получаем доступ к атрибуту Caption операционной системы, который представляет собой текстовое описание имени и версии операционной системы.

Выходные данные для отдельно взятого компьютера будут выглядеть так:

```
dnshostname             OSVersion
-----------             ---------
server-r2.company.pri   Microsoft Windows Server 2008 R2 Standard
```

Создание HTML

Иногда может потребоваться отобразить выходные данные в таком виде, чтобы их можно было просматривать через веб-браузер. Командлет ConvertTo-HTML конвертирует элементы конвейера в форму таблицы на базе HTML. Этот командлет не записывает HTML в файл, он просто помещает HTML-текст в конвейер. Однако вы можете сохранить его в файл, используя командлет Out-File. Типичным примером использования данного командлета может быть следующая команда:

```
Get-EventLog Security –newest 20 | ConvertTo-HTML |
Out-File events.htm
```

Не отправляйте выходные данные командлета Format- в ConvertTo-HTML; помните о том, что командлеты Format- производят только инструкции для форматирования, а значит, если вы предпримите такую попытку, в HTML будут конвертированы именно сами инструкции. Если вы хотите посмотреть, каким будет результат, попробуйте запустить команду:

```
Get-Process | Format-Table | ConvertTo-HTML | Out-File confusing.htm
```

HTML, произведенный командлетом ConvertTo-HTML, является хорошо оформленным, «чистым» HTML-текстом, а значит, не содержит никакой информации, касающейся форматирования. В результате мы получаем простой

файл. Однако, используя различные параметры командлета ConvertTo- HTML, можно уточнить некоторую дополнительную информацию, например:

• Текст, который необходимо поместить перед HTML-таблицей.

• Заголовок страницы.

• Текст, который необходимо поместить после HTML-таблицы.

• Ссылка CSS, которая может использоваться для определения дополнительных опций форматирования: шрифт, цвет и.т.д.

Создание HTML

- Иногда хочется смотреть данные в браузере
- Командлет ConvertTo-HTML
  - Преобразует данные в HTML таблицу
  - Не пишет данные в файл, а выкидывает на стандартный вывод
  - Можнг передать конвейером в Out-File для создания файла, который можно будет просмотреть браузером

```
Get-EventLog Security –newest 20 |
ConvertTo-HTML | Out-File events.htm
```

- Параметры ConvertTo-HTML позволяют изменить заголовки и прицепить CSS

Не забывайте, что Windows PowerShell использует официальные термины для описания элементов в конвейере и их свойств. Мы уже упоминали о них ранее. Данные термины являются всего лишь словами и не несут никакого значения сами по себе. И все же, вам следует привыкнуть к регулярному использованию этих терминов – так вам будет проще общаться с другими пользователями оболочки. В данном модуле слово «элемент» использовалось для обозначения данных, которые командлет помещает в конвейер. Например, командлет Get-Process помещает в конвейер элемент «процесс». Более официальный термин, который используется вместо слова «элемент» - это «объект». Поэтому, можно также сказать, что командлет Get-Process помещает в конвейер объекты процесса.

Все эти объекты имеют атрибуты. Для процесса атрибутами могут быть имя, ID, объем занимаемой памяти и.т.д. Официальное название атрибутов – свойства. Поэтому, можно сказать, что процесс имеет свойство имени, свойство ID, и.т.д.

Основные командлеты Windows PowerShell

Тогда как многие встроенные в оболочку командлеты могут считаться ключевыми, другие созданы исключительно для передачи объектов, которые запускаются в конвейер другими командлетами. Поэтому, в названиях таких командлетов чаще всего присутствует слово «объект». В частности, в данном разделе мы рассмотрим следующие из них:

• Sort-Object

• Group-Object

• Measure-Object

• Select-Object

• Compare-Object

Некоторые из ключевых командлетов также предназначены исключительно для экспорта и импорта данных в формат текстовых файлов и из формата текстовых файлов. К ним относятся:

• Import-CSV

• Export-CSV

• Import-CliXML

• Export-CliXML

Основные командлеты

• Преобразование данных
  - Sort-Object
  - Group-Object
  - Measure-Object
  - Select-Object
  - Compare-Object

Эти командлеты преобразут данные, полученные от других командлетов

• Импорт и экспорт данных
  - Import-CSV
  - Export-CSV
  - Import-CliXML
  - Export-CliXML

Эти командлеты экспортируют и импортируют данные из/в другие форматы

Ни один из перечисленных командлетов (кроме командлетов Import-) не генерирует объекты самостоятельно. Вместо этого они принимают объекты, сгенерированные другими командлетами, обрабатывают их (модифицируют или фильтруют), а затем передают их дальше по конвейеру, в следующий командлет.

Самое время вспомнить о том, что командлеты Windows PowerShell не производят простой текст в качестве выходных данных. Вместо этого они производят объекты. Объект – это общий термин в программировании, которым обычно называют самостоятельную сущность, которая предоставляет информацию о себе самой в форме атрибутов или свойств и предлагает методы, побуждающие программное обеспечение выполнить ту или иную задачу или функцию.

Многие командлеты Windows PowerShell, особенно те, в названии которых содержится глагол Get-, производят объекты и помещают их в конвейер:

Get-EventLog Security –newest 20

Передавая эти объекты по конвейеру другому командлету, вы можете манипулировать ими:

Get-EventLog Security –newest 20 | Sort-Object

Некоторые ключевые командлеты, рассмотренные в данном разделе, просто производят определенные действия над входящими объектами и передают их дальше по конвейеру. Другие ключевые командлеты поглощают входящий объект, затем производят новый объект, который передается дальше. Вы поймете разницу между ними, когда испытаете оба вида командлетов в действии.

Сортировка объектов

Сортировка объектов

- Командлет Sort-Object
  - Командлет Sort-Object позволяет изменить порядок, в котором перечисляются объекты.
  - Он может принимать входящие данные любого типа .
  - Необходимо уточнить одно или несколько свойств, в соответствии с которыми будет сформирован список объектов .
  - по умолчанию объекты могут быть отсортированы в восходящем порядке .
  - указав параметр –descending, можете изменить порядок на нисходящий. .

```
Get-Service | Sort-Object status
Get-Service | Sort-Object status –descending

Get-Service | Sort-Object status, name
```

Когда вы запускаете командлет, он самостоятельно определяет порядок, в котором будут перечислены исходящие объекты. Список в журнале регистрации входящих событий, в свою очередь. Обычно формируется в хронологическом порядке.

Порядок, предлагаемый командлетами по умолчанию, иногда может вас не устраивать. Например, работа со списком всех остановленных сервисов может быть неудобной, если список сформирован в алфавитном порядке по названию сервиса. Командлет Sort-Object позволяет изменить порядок, в котором перечисляются объекты. Он может принимать входящие данные любого типа; вам же остается уточнить одно или несколько свойств, в соответствии с которыми будет сформирован список объектов. Например, по умолчанию объекты могут быть отсортированы в восходящем порядке, а вы, указав параметр – descending, можете изменить порядок на нисходящий. Операционная система вашего компьютера распознает указанный порядок сортировки, поэтому результаты могут немного отличаться на разных компьютерах из-за особенностей операционной системы каждого из них. Например:

```
Get-Service | Sort-Object status
```

Или, чтобы отсортировать объекты в нисходящем порядке:

```
Get-Service | Sort-Object status -descending
```

Sort – это встроенный псевдоним командлета Sort-Object, который может использоваться вместо полного имени командлета:

```
Get-Service | Sort status
```

Обратите внимание, что при написании имен свойств регистр клавиатуры не имеет значения: status, Status, и STATUS будет означать одно и то же. Также, когда Sort-Object сортирует текстовые свойства, например, процессы или сервисы, он по умолчанию делает их нечувствительными к регистру. Чтобы более подробно узнать об опциях, предлагаемых данным командлетом, запустите команду Help Sort-Object. Иногда может показаться, что Sort-Object работает слишком медленно, особенно, когда получает по конвейеру большое количество объектов сразу. Это происходит из-за того, что командлет должен дождаться всех объектов до последнего, прежде чем начать сортировку. Как только последний объект получен, командлет моментально производит отсортированные выходные данные. Вы можете указать несколько свойств, разделяя их запятой. Например, если вы хотите получить список всех сервисов, отсортированных по статусу, но чтобы при этом внутри каждого статуса сервисы были отсортированы в алфавитном порядке, запустите команду:

```
Get-Service | Sort status,name
```

Учтите, что Sort-Object оперирует реальными значениями свойств объектов, но Windows PowerShell не всегда отображает эти реальные значения свойств. Примером может быть свойство статуса сервиса. Внутри оболочки статус отображается в виде цифры, однако, гибкая печатная система Windows PowerShell автоматически переводит это число в текстовую строку. Поэтому, вместо статуса 0 или 1, вы видите статус Stopped (остановленные) или Running (запущенные) . Если внимательно присмотреться, то можно заметить, что с помощью следующей команды статус Stopped оказывается перед Running, хотя обычно бывает наоборот:

```
Get-Service | Sort status
```

Это вызвано работой командлета Sort-Object над значением этого свойства – естественно, что при сортировке он ставит сначала 0, а потом 1. Но когда оболочка отображает данные значения, 0 превращается в Stopped, а 1 – в Running.

Группировка объектов

Группировка объектов

- The Group-Object cmdlet…
  - изучает свойства заданных объектов и объединяет объекты в группы по значениям каждого свойства
  - на выходе он показывает, сколько объектов находится в каждой группе Is useful only when object properties have repetitive values

Get-Service | Group-Object status

Иногда вам может потребоваться распределение объектов по группам, чтобы работать с каждой группой отдельно. В этом поможет командлет Group-Object. Он изучает свойства заданных объектов и объединяет объекты в группы по значениям каждого свойства. Например, данная команда:

Get-Service | Group-Object status

обычно позволяет создать две группы объектов: со статусом «остановлен» и со статусом «запущен». Возможно, вы получите и другие группы статусов, в зависимости от того, в каком состоянии находятся сервисы в тот момент, когда вы запускаете программу. Командлет Group-Object обычно бывает полезным, когда свойства объектов имеют повторяющиеся значения. Например, команда Get-Service | Group-Object name будет менее полезной, так как каждый сервис имеет уникальное имя. В данном случае командлет Group-Object будет вынужден создавать отдельную группу для каждого сервиса, и каждая группа будет включать всего один объект. Преимуществом Group-Object является то, что на выходе он показывает, сколько объектов находится в каждой группе, что позволяет вам моментально сориентироваться в том, сколько сервисов остановлено, сколько процессов откликается на запросы Windows и.т.д.

Измерение объектов

Измерения

- Командлет Measure-Object может посчитать количество входящих объектов, а также измерить составные значения числовых свойств объектов.
    - -average
    - -maximum
    - -minimum
    - -sum
- Обратите внимание, что Measure-Object вбирает в себя входящие объекты, что означает, что, поступив в него, они больше не находятся в конвейере.
- Measure-Object обычно является последним командлетом в цепочке

Get-Process | Measure-Object

Get-Process | Measure-Object –property VM –average –sum –minimum -maximum

Командлет Measure-Object может посчитать количество входящих объектов, а также измерить составные значения числовых свойств объектов. В качестве самого простого примера его работы вы можете просто посчитать количество объектов с помощью команды:

Get-Process | Measure-Object

Обратите внимание, что Measure-Object вбирает в себя входящие объекты, что означает, что, поступив в него, они больше не находятся в конвейере. В качестве выходных данных Measure-Object выступают какие-либо числовые значения объектов, а не процессы. Вы можете убедиться в этом, передав выходные данные командлету Get-Member:

Get-Process | Measure-Object | Get-Member

Это означает, что после обработки командлетом Measure-Object все процессы теряются, остаются лишь числовые показатели их значений. На практике Measure-Object обычно является последним командлетом в цепочке, поскольку его выходные данные вы вряд ли захотите передать куда-то еще.

Помимо простого подсчета объектов, Measure-Object может создавать составные значения для числовых свойств объектов, например, таких как виртуальная память (VM). Для этого необходимо указать одно или несколько составных значений и имя свойства, которое вы хотите измерить:

Get-Process | Measure-Object –property VM –average –sum –minimum –maximum

Результатом будет комплексное числовое значение. Другие опции Measure-Object можно найти в справочнике (Help).

Весьма распространенное явление – сокращать имена параметров при печати, например, вводить min вместо minimum. Это обычно срабатывает хорошо, однако, не забывайте, что вы вводите просто урезанное слово, а не его аббревиатуру. Например, -aver можно набрать вместо –average (средний), в то время как –avg не будет работать, хотя это распространенная аббревиатура в английском языке.

Выбор объектов и свойств

Выбор объектов

- Командлет Select-Object
  - Используется для двух целей:
    1. Выбор подмножества объектов: -first, -last, -skip

       Get-Process | Select-Object –first 10

    2. Выбор свойств объектов

       Get-Process | Select-Object name,ID,VM,PM

  - В комбинации

    Get-Process | Select-Object name,ID –first 10

Командлет Select-Object имеет две четких цели, то есть, может использоваться двумя способами. Эти два способа могут использоваться как одновременно, так и по отдельности, в зависимости от ваших потребностей.

Выбор определенных объектов:

Вы можете использовать Select-Object использовать для выбора подмножества объектов в конвейере, применяя такие его параметры как -first, -last и –skip:

• -first – определяет количество объектов, считая с начала, которые нужно выбрать.

• -last - определяет количество объектов, считая с начала, которые нужно выбрать.

• -skip – пропускает указанное количество объектов и отображает оставшиеся.

Например, чтобы выбрать только первые 10 объектов, запустите команду:

Get-Process | Select-Object –first 10

Чтобы выбрать 10 процессов, занимающих больше всего физической памяти:

Get-Process | Sort-Object PM | Select-Object –last 10

При таком использовании командлет Select-Object выпускает те же самые объекты, что были получены им в качестве входящих данных, однако, их количество может быть меньше. Убедиться в этом можно, передав данные из Select-Object в Get-Member:

Get-Process | Select-Object –first 10 | Get-Member

Выбор определенных свойств объектов:

Select-Object может также использоваться для ограничения свойств исходящих объектов. Например, если вы работаете с процессами и хотите вывести только данные о свойствах ID, VM и PM, вы можете использовать Select-Object, чтобы отсеять все остальные свойства:

Get-Process | Select-Object name,ID,VM,PM

Однако при таком использовании вы вынуждаете Select-Object производить новый тип объектов, отличных от входящих данных. Входящие объекты забираются командлетом, но не выпускаются обратно в конвейер. В этом можно убедиться, запустив команду:

Get-Process | Select-Object name,ID,VM,PM | Get-Member

Это происходит каждый раз, когда вы уточняете список свойств с помощью Select-Object. Эта техника может также использоваться с применением параметров –first, -skip и –last, но поскольку это список свойств, а не список объектов, Select-Object будет выдавать в качестве исходящих данных видоизмененные объекты:

Get-Process | Select-Object name,ID,VM,PM –first 10 | Get-Member

Select – это псевдоним, который используется по умолчанию для командлета Select-Object, поэтому, его можно указывать вместо полного названия:

Get-Process | Select name,id –first 10

Создание новых свойств

Select-Object может также использоваться для придания новых, пользовательских свойств объектам. Это может быть просто новое имя для существующего свойства, например, вы можете добавить свойство ComputerName объекту, который уже имеет имя MachineName. С другой стороны, пользовательские свойства могут включать в себя сложные вычисления, например, такое свойство как PercentageFreeSpace (свободное место в процентах) для объекта, который уже обладает свойствами TotalSize (общий размер) и FreeSpace (свободное место).

Для придания объекту новых свойств, вам необходимо предварительно составить хеш-таблицу. Более подробную информацию о хеш-таблицах вы найдете в последующих разделах курса, а пока достаточно просто запомнить корректный синтаксис:

Get-Process | Select name,vm,pm,@{Label="TotalMemory"; Expression={$_.vm + $_.pm}}

Как это работает:

• Командлет Select-Object получает команду выбрать свойства VM и PM, которыми данный объект уже обладает.

• Несмотря на то, что действия Select-Object не зависят от регистра клавиатуры, он запоминает, какой регистр был использован для того или иного имени в последний раз. В частности, в данном случае свойство vm будет выведено на экран строчными буквами, поскольку строчные буквы использовались при написании команды.

• Новое свойство имеет имя (ярлык) TotalMemory (общая память). Значение этого свойства высчитывается исходя из значений существующих свойств VM и PM. Элемент Expression в командной строке определяет значение нового свойства как сумму значений двух существующих свойств - VM и PM. Переменная $_ является структурным нулем, который относится к «текущему объекту».

Вы можете относиться к элементу $_ как к пустой ячейке в таблице, которые вы, вероятно, встречали на бумажных бланках. Оболочка автоматически заполняет бланк, вставляя по умолчанию имя текущего объекта, а значит вручную его вписывать необязательно – вместо этого сюда вставляется символ структурного нуля. $_ может использоваться только в конкретных сценариях, в которых оболочка будет его искать. Блок Expression как раз относится к таким сценариям.

Работа с CSV и XML данными

CSV и XML

- Windows PowerShell обладает возможностью читать и записывать файлы, в которых значения разделены запятой (comma-separated values или CSV), а также простые XML файлы
  - Import-CSV, Export-CSV, Import-CliXML, Export-CliXML

```
Get-EventLog Security –newest 20 | Export-CSV new-events.csv

Get-Process | Sort VM –desc | Select –First 10 | Export-CSV top-vm.csv

Import-CliXML procs.xml | Get-Member
```

Windows PowerShell обладает возможностью читать и записывать файлы, в которых значения разделены запятой (comma-separated values или CSV), а также простые XML файлы. Для XML-файлов используется специальный пользовательский интерфейс под названием Command Line Interface XML или CliXML.

Экспорт объектов в CSV-файл

Когда вы отправляете объекты по конвейеру в командлет Export-CSV и уточняаете имя файла, в который необходимо экспортировать объекты, оболочка изучает все объекты в конвейере. Затем она пишет заголовок в виде комментария в первой строчке файла, где указывается имя типа объекта в конвейере. После этого пишется второй заголовок, где перечисляются все свойства объектов в конвейере в виде списка. Начиная с этого места, каждый объект из конвейера записывается в виде строчки CSV-файла, а его свойства заносятся в соответствующие столбцы таблицы. Готовый файл можно без проблем открыть в Microsoft Office Excel, импортировать в одно из многочисленных приложений базы данных, использовать в качестве фиксированной формы стандартного письма в Microsoft Office Word, и.т.д.Простейший пример использования данного командлета может выглядеть так:

```
Get-EventLog Security –newest 20 | Export-CSV new-events.csv
```

Также вы можете произвести какие-либо манипуляции с объектами в конвейере, прежде чем импортировать их в CSV-файл:

```
Get-Process | Sort VM –desc | Select –First 10 | Export-CSV top-vm.csv
```

Командлет Export-CSV может иметь множество дополнительных параметров, которые применяются для выполнения конкретных задач. Например, вы можете:

• Указать, что первый заголовок, содержащий информацию о типе объектов, следует опустить.

• Указать, что второй заголовок, включающий список объектов, можно опустить.

• Указать другой разделитель, вместо запятой (символ). О прочих возможностях командлета Export-CSV вы можете узнать из справочной системы.

Важно отметить, что CSV-файл является файлом «плоского» формата, т.е. может включать в себя только один уровень данных. Windows PowerShell не может превратить сложную иерархию объектов в CSV-файл. Например, такой объект как папка в файловой системе может содержать другие объекты – файлы или папки; Export-CSV не сможет использовать эту иерархию. Попробуйте изучить файл, созданный с помощью подобной команды, чтобы, что произойдет:

```
Dir C:\ | Export-CSV directories.csv
```

Импорт объектов из CSV-файла

Командлет Import-CSV может читать CSV-файлы (или файлы, где вместо запятых использован другой разделитель), и создавать статичные объекты, отображающие содержимое этих файлов. Каждая строчка CSV файла превращается в отдельный объект, который помещается в конвейер. Свойства объектов конструируются из

столбцов таблицы CSV-файла. Если у CSV-файла есть заголовок (он должен быть по умолчанию), то имена свойств объектов берутся из этого заголовка. В справочнике вы найдете информацию о дополнительных возможностях командлета Import-CSV, среди которых можно отметить возможности:

• Указать, что CSV-файл не содержит заголовка.

• Указать, что вместо запятой используется другой разделитель.

Импорт и экспорт XML-файлов

Командлеты Import-CliXML и Export-CliXML работают во многом аналогично командлетам Import-CSV и Export-CSV. Однако вместо «плоского» CSV-файла могут использовать более сложный формат, который способен отобразить иерархическую структуру. Попробуйте запустить команду:

Dir c:\Windows\System32 -recurse | Export-CliXML directories.xml

Вы можете открыть файл в Windows® Notepad, но намного проще это сделать в Internet Explorer, который умеет форматировать XML-файлы, придавая им более читабельный вид. Перевод объекта в формат XML таким способом называется сериализация; перевод XML-файла обратно в объект оболочки называется десериализация. Десериализованные объекты не являются живыми объектами. Они являются скорее слепками или снимками, сделанными в тот момент, когда они подверглись сериализации. Десериализованные объекты не несут никакой функциональности и методов – они представляют собой просто набор свойств. Чтобы увидеть различие, сначала запустите команду:

Get-Process | Get-Member

Обратите внимание на то, как отображаются объекты – каждый из них обладает множеством методов. Затем запустите следующую команду:

Get-Process | Export-CliXML procs.xml

Откройте Windows Notepad или Windows Calculator и запустите следующую команду:

Import-CliXML procs.xm

На первый взгляд, на дисплее будет отображаться та же картинка, что и после первой команды. Однако обратите внимание, что здесь не будет списков – объекты являются десериализованными из XML, а не импортированными из ОС. Сейчас запустите новую команду:

Import-CliXML procs.xml | Get-Member

Имена типов объектов укажут на о, что эти объекты являются десериализованными и больше не обладают методами – у них остались только свойства, так как свойства – это все, что включает в себя XML-файл.

Сравнение объектов

Windows PowerShell обладает возможностью сравнивать два набора объектов. Сравнение может быть довольно сложным и запутанным, так как объекты сами по себе часто являются сложными. Например, возьмем два идентичных процесса,

запущенных на разных компьютерах. Пусть это будет Windows Notepad. Некоторые аспекты объектов будут идентичными на обоих компьютерах, например, свойство имени. Другие же свойства, такие как ID, VM и PM будут различаться. Являются ли эти объекты идентичными? Это зависит от того, с какой именно целью вы их сравниваете. Одна из целей сравнивания объектов – это организация внесения изменений. Возможно, вы захотите создать базовую линию, которая бы описывала первоначальную конфигурацию сервера. Позже вы захотите сравнить текущую конфигурацию с той, что была изначально, чтобы узнать, какие изменения произошли. . Windows PowerShel предлагает специальный командлет Compare- Object (с псевдонимом Diff), который упрощает процесс сравнения. Начать можно с создания базового файла. Лучше всего для этой цели подходит XML. Для создания файла с описанием текущей конфигурации компьютерных сервисов запустите команду:

Get-Service | Export-CliXML service-baseline.xml

Сейчас попробуйте внести изменения в работу сервисов, например, запустите остановленное приложение или остановите запущенное. Для манипуляций в лабораторных условиях хорошо подходит сервис BITS. Затем сравните новую конфигурацию с базовой:

Compare-Object (Get-Service) (Import-CliXML service-baseline.xml)

В данной команде круглые скобки указывают на то, что командлеты Get-Service и Import-CliXML должны быть запущены в первую очередь. Их выходные данные передаются командлету Compare-Object, который сравнивает два набора объектов. В данном примере Compare-Object будет сравнивать все параметры объектов. Однако при работе с другими типами объектов, например, с процессами, память и значение CPU которых постоянно изменяются, сравнивать все их параметры бессмысленно, так как результаты будут постоянно разными. В таком случае вы можете дать командлету Compare-Object задачу учитывать только определенные параметры при сравнении.

Чтобы узнать, как указать командлету необходимые для сравнения свойства, а также получить информацию о прочих его возможностях, обратитесь к справочнику.

Сравнение, фильтрация и перечисление

Сравнение

Любое сравнение двух или более объектов или свойств предназначено для того, чтобы получить истинное или ложное значение (True or False value). Данный тип сравнения называют Булевым сравнением, поскольку его результатом всегда является одно из Булевых значений – «правда» или «ложь».

Сравнение двух простых значений или объектов – довольно распространенная задача. Например, вы сравниваете два имени компьютера, чтобы проверить, являются ли они одинаковыми или сравниваете значение производительности компьютера с неким пороговым значением, чтобы проверить, какое значение выше.

Знаки сравнения ставятся между двумя единицами, которые вы хотите сравнить. Наверное, вы помните простые знаки сравнения из программы средней школы:

$4 > 10$
$10 = 5$
$15 \le 15$

Windows PowerShell производит сравнение тем же способом, хотя традиционные математические символы здесь не используются. Windows PowerShell выделяет два особых объекта - $True и $False, которые отображают Булевы значения True и False. Например, в процессе сравнения 4 и 10, утверждение о том, что 4 больше 10 будет неверным, значит, результатом будет $False, а утверждение, что 10 равно 10 – верным (результат $True).

Windows PowerShell дает возможность выполнять сравнение непосредственно в командной строке. Для этого просто напечатайте ваше сравнение и нажмите клавишу RETURN, чтобы увидеть результат.

Ввиду того, что оболочка использует символы < и > для своих целей (например, > используется для перенаправления вывода в файл) для операций сравнения используются специальные наборы символов, унаследованные из других интерфейсов командной строки, в частности из UNIX-шелла.

Операторы сравнения

- Основные:
  - -eq : Equal to
  - -ne : Not equal to
  - -le : Less than or equal to
  - -ge : Greater than or equal to
  - -gt : Greater than
  - -lt : Less than
- Чувствительные к регистру:
  - -ceq : Equal to
  - -cne : Not equal to
  - -cle : Less than or equal to
  - -cge : Greater than or equal to
  - -cgt : Greater than
  - -clt : Less than
- Не чевствительные к регистру

  (не используются, вместо них используются основные)
  - -ieq : Equal to
  - -ine : Not equal to
  - -ile : Less than or equal to
  - -ige : Greater than or equal to
  - -igt : Greater than
  - -ilt : Less than

Основные операторы

- -eq – равно
- -ne – не равно
- -le – меньше или равно

- -ge – больше или равно
- -gt больше, чем
- -lt – меньше чем

Если данные операторы используются со строками, то они нечувствительны к регистру, в частности результат выполнения команды

“Hello” –eq “HELLO”

будет истиной. Если Вам нужны операторы, чувствительные к регистру, то необходимо использовать

- -ceq – Равно (чувствительно к регистру)
- -cne – не равно (чувствительно к регистру)
- -cle – меньше или равно (чувствительно к регистру)
- -cge – больше или равно (чувствительно к регистру)
- -cgt больше, чем (чувствительно к регистру)
- -clt – меньше чем (чувствительно к регистру)

Операторы Булевой алгебры (-and, -or, -not)

- В сложных сравнениях вы можете использовать операторы –and и –or
  - ***4 –gt 10 –or 10 –gt 4 # returns True***
  - ***4 –lt 10 –and “Hello” –ne “hello” # returns False***
- Обычно сравнения выполняются слева направо, однако Вы можете группировать выражения.
  - ***(4 –gt 10 –and (10 –lt 4 –or 10 –gt 5)) –and 10 –le 10***

В сложных сравнениях вы можете использовать операторы –and и –or

4 –gt 10 –or 10 –gt 4 # returns True
4 –lt 10 –and “Hello” –ne “hello” # returns False

Обычно сравнения выполняются слева направо, однако Вы можете группировать выражения.

(4 –gt 10 –and (10 –lt 4 –or 10 –gt 5)) –and 10 –le 10

Фильтрация конвейеров

- Командлет Where-Object …
  - Используется для удаления некоторых объектов из конвейера
  - Убирает объекты в соответствии с указанными критероиями
  - Пропускает требуемые объекты далее по конвейеру
- Where-Object использует переменную $_ для указания текущего объекта

```
Get-Service | Where-Object { $_.Status –eq "Running" }
```

Фильтрация – это процесс удаления некоторых объектов из конвейера, обычно по той причине, что они не соответствуют заданным критериям. Как правило, фильтрация оставшихся объектов проводится перед тем, как они передаются другому командлету для дальнейшей обработки.

Фильтрация осуществляется с помощью командлета Where-Object, который имеет псевдоним Where. Главным параметром Where-Object является скриптовый блок, в котором вы уточняете, каким критериям должен отвечать объект для того, чтобы остаться в конвейере. Объекты, которые не удовлетворяют данным критериям, удаляются из конвейера.

Внутри данного блока оболочка ищет специальный элемент, переменную $_, обозначающую структурный нуль. Эта переменная заменяется текущим объектом. По сути скриптовый блок и то сравнение, которое вы поместили в него, будет изучаться отдельно для каждого объекта, который находится в конвейере. Например:

```
Get-Service | Where-Object { $_.Status –eq "Running" }
```

Обратите внимание, что скриптовый блок заключен в круглые скобки. Если бы командлет Get-Service сгенерировал 100 объектов, то сравнение командлетом Where-Object было бы проведено 100 раз. Каждый раз переменная $_ будет заменяться новым объектом. После переменной $_ следует точка, которая

указывает на то, что сравнение следует производить по одному из свойств объекта, в данном случае это статус. Данный пример представляет собой сравнение значений свойства Статус всех объектов в конвейере с заданным значением – Running (запущенный). Если свойство объекта Статус содержит значение RUNNING, Running, running (мы знаем, что параметр –eq нечувствителен к регистру), то данный объект передается по конвейеру в следующий командлет. Если же свойство Статус объекта не соответствует данному критерию, объект просто пропускается.

Иногда можно спутать предназначение командлетов Select-Object и Where-Object. Запомните, что Where- Object используется для фильтрации всех объектов в конвейере; Select-Object – для выбора ряда объектов (с помощью параметров –first или –last) или для уточнения свойств объектов, которые вы хотите видеть. Если вы знакомы с языком Structured Query Language (SQL), который используется в работе с базами данных, то имена командлетов Select-Object и Where-Object должны также быть для вас знакомыми. Так, Select-Object равнозначен утверждение SQL SELECT и используется для выбора необходимых свойств или объектов, а Where-Object равнозначен утверждению SQL WHERE, которое устанавливает критерии фильтрации.

Where-Object – не всегда является лучшим выбором!

Для того чтобы использовать Where-Object, вы обычно начинаете команду с командлета Get-, например, Get-Process или Get-ADUser.Далее вы передаете все эти объекты по конвейеру командлету Where-Object, который удаляет все объекты, не соответствующие вашим критериям. Для большого количества объектов Where-Object может оказаться неэффективным. Во многих случаях командлеты Get-предлагают свою собственную фильтрацию. Если командлет предлагает возможность фильтрации (чаще всего с помощью параметра –filter), его использование всегда предпочтительнее, нежели использование Where-Object. Такая фильтрация обычно происходит на этапе извлечения данных, то есть, командлет Get- изначально извлекает и передает в конвейер только те объекты, которые соответствуют вашим критериям. Это экономит время и производственные мощности. При работе с Active Directory для таких командлетов как Get-ADUser использование параметра –filter является обязательным, поскольку случайное извлечение всех имен пользователей домена может вызвать привести к непосильной нагрузке на контроллер домена.

Перечисление

Многие так называемые командлеты действия (action cmndlets) – те командлеты, которые производят какие-либо действия или выполняют задачу, созданы для работы с полным набором объектов. Например, представьте себе (но не запускайте!) такую команду:

```
Get-Process | Stop-Process
```

Данная команда извлекает все процессы, а затем делает попытку закрыть их все, что может привести к неожиданному выключению компьютера.

Однако может быть такое, что у вас нет возможности или желания работать со всеми объектами сразу. В данном случае вам потребуется возможность работать с каждым объектом индивидуально, извлекая их по одному.

В Windows PowerShell v1 было больше возможностей работать с объектами индивидуально. В v2 Microsoft добавил множество командлетов, предназначенных для работы со всеми объектами сразу, следовательно, количество сценариев, где может потребоваться нумерация объектов или их индивидуальная обработка, сократилось.

Перечисление объектов

- Командлет ForEach-Object …
  - Позволяет выполнять операции над набором объектов
  - Where-Object использует $_ для обозначения текущего объекта
  - Использует блок скрипта {}

```
Get-Service | Where-Object { $_.Status –eq "Stopped" }
    | ForEach-Object { $_.Start() }
```

Перечисление или инумерация объектов производится с помощью командлета ForEach-Object, который имеет два псевдонима - Foreach и % (да, математический символ процента является псевдонимом ForEach-Object). Типичным параметром для ForEach-Object является скриптовый блок, содержащий информацию о том действии, которое вы собираетесь произвести с каждым из входящих объектов. Внутри этого скриптового блока оболочка ищет переменную $_ и подставляет вместо нее все объекты по очереди Например, вот простейший пример команды, где сначала извлекаются все сервисы, затем отсеиваются запущенные, а затем производится попытка применить метод Start() к оставшимся:

```
Get-Service | Where-Object { $_.Status –eq “Stopped” } | ForEach-Object { $_.Start() }
```

В действительности, эту же самую задачу можно выполнить более простым способом:

```
Get-Service | Where-Object { $_.Status –eq “Stopped” } | Start-Service
```

Командлет Start-Service может работать со всеми входящими объектами сразу, поэтому, действительно нет необходимости перечислять их, и ндивидуально применять метод Start() к каждому из них.

В последнее время командлет ForEach-Object становится все менее востребованным, поскольку Microsoft постоянно совершенствует и обновляет Windows PowerShell, выпуская новые командлеты. Если у вас возникла необходимость в использовании ForEach-Object, сначала подумайте – быть может есть какой-то более эффективный способ выполнения той же самой задачи с использованием командлета, который работает со всеми объектами сразу. ForEach-Object остается необходимым лишь в тех ситуациях, где нет возможности использовать другой командлет.

Расширенные возможности конвейеров

Одно из ключевых различий между Windows PowerShell и старыми административными скриптовыми языками заключается в том, что Windows PowerShell – это первая оболочка, призванная ускорить процесс автоматизации административной работы. Хотя оболочка поддерживает мощный скриптовый язык, она в то же время предлагает ряд более простых и эффективных способов решения тех же самых задач. Windows PowerShell дает возможность автоматизировать сложные комплексные задачи без необходимости написания скриптов. Однако для этого вам придется приобрести определенное мастерство в использовании различного вида команд в рамках Windows PowerShell. Наиболее важным здесь является конвейер и процесс передачи объектов от одного командлета к другому. Конструкция командлетов позволяет им производить большие объемы работы при относительно небольшом объеме ввода данных, а это значит, что вам придется намного меньше печатать.

Большинство командлетов Windows PowerShell поддерживают множество параметров; каждый параметр предназначен для приема входящих данных определенного типа. Например, закройте все запущенные копии Windows Notepad, а затем откройте новый документ Windows Notepad. Запустите команду:

```
Get-Process
```

Обратите внимание, что ваш документ Windows Notepad появился в списке. Попробуйте остановить процесс, запустив команду:

```
Stop-Process notepad
```

Почему вы видите ошибку? В сообщении об ошибке указано, что оболочка не смогла привязать параметр ID, потому что она не смогла конвертировать наши входящие данные, notepad, в 32-битное целое число. Сейчас загляните в раздел справочника, где описывается работа Stop-Process:

```
Help Stop-Process
```

Обратите внимание на различные наборы параметров. Набор параметров – это группа параметров, которые могут использоваться вместе. Некоторые параметры могут входить в разные наборы. Например, предположите, что набор параметров A включает параметры –computername и –id. Набор параметров B включает параметры –computername, –name, и –noclobber. Это означает, что вы не можете

использовать одновременно параметр –id и параметр –name, так как они находятся в разных наборах параметров. Однако вы можете использовать –id вместе с –computername, поскольку они оба находятся в одном наборе. В первом наборе параметров командлета Stop-Process параметр ID является не опционным, то есть, вы должны указать значение для него. Однако настоящее имя параметра, -id, является опционным – оно заключено в квадратные скобки. Когда мы запустили Stop-Process без указания параметра, оболочка решила, что мы указали «notepad» в качестве значения параметра –id. Но параметр –id требует Int32 – 32-битного целого числа, а «notepad» не может быть конвертирован в этот тип данных.

Третий набор параметров может принимать имя процесса в виде строки, но если вы решите использовать эту технику, то параметр –name не будет опционным. Попробуйте:

Stop-Process –name notepad

Это сработало, так как оболочка поняла, что «notepad» относится к параметру –name, а не –id. Суть всего описанного заключается в том, что все параметры слегка отличаются друг от друга, и каждый из них предназначен для приема данных определенного типа.

Позиционные параметры

Позиционные параметры

- Позиционные параметры не требуют указания их имен в командной строке
  - Их использование строится на их местоположении (позиции) в командной строке
  - Это упрощает ввод команд, например

```
Stop-Process –id 53  #Executes correctly
Stop-Process 53  #Executes correctly
```

- Имена позиционных параметров в справке указываются в квадратных скобках

```
Stop-Process [-Id] <Int32[]>
```

- Имена параметров, за исключением позиционных могут находиться в любой части командной строки
  - Однако, использование имен параметров делает код чище

Если вы увидите в справочном описании командлета что-то вроде этого:

Stop-Process [-Id] <Int32[]>

это будет означать, что перед вами позиционный параметр. Это означает, что сам параметр необходим, но его имя набирать необязательно. В частности, вы можете напечатать:

```
Stop-Process –id 53
```

Или

```
Stop-Process 53
```

До тех пор, пока соответствующий тип данных расположен в соответствующем месте (в данном случае на первой позиции), оболочка буде знать, что делать. Данную информацию можно найти в разделе с полным описанием командлета в справочнике:

```
Help Stop-Process –full
```

Здесь вы можете увидеть, что параметр является позиционным, что он занимает позицию номер 1 и является обязательным:

```
-Id <Int32[]>
Specifies the process IDs of the processes to be stop
type "get-process". The parameter name ("Id") is optio
(Этот параметр уточняет ID процессов, которые необходимо остановить.
Наберите "get-process". Параметр имени («Id») является опционным:)
Required? true
Position? 1
Default value
Accept pipeline input? true (ByPropertyName)
Accept wildcard characters? False
```

Вот еще один пример из раздела описания командлета Get-ChildItem в справочнике:

```
Get-ChildItem [[-Path] <string[]>] [[-Filter] <string>] [-Exclude <string[]>]
```

В данном примере весь параметр –path является опционным. Как вы видите, весь параметр, включая тип данных string[], заключен в квадратные скобки. Однако само имя параметра –path находится в отдельных квадратных скобках. Это означает, что параметр является опционным, и если вы захотите использовать его, вы можете опустить его имя, если его значение указано на первой позиции. Параметр –filter также является опционным и позиционным, параметр –exclude – опционным, но не позиционным. Если вы захотите использовать –exclude, вы будете должны указать его имя.

Позиционные параметры призваны упростить процесс печати. Представьте, насколько утомительным было бы указывать параметр –path каждый раз, когда вам потребуется обратиться к директории:

```
Dir –path c:\
```

Но поскольку –path является позиционным параметром, вы можете набрать более простую и привычную команду:

```
Dir c:\
```

Однако использование имен параметров имеет два преимущества:

• Это делает команду намного более понятной для кого бы то ни было, поскольку имя параметра позволяет лучше разобраться в том. Что происходит.

• Когда вы используете имя параметра, вы можете расположить параметры в любом порядке, и вам не придется каждый раз запоминать нужный порядок.

Например, обратите внимание на команду:

Add-Member NoteProperty ComputerName LON-DC1

Данный командлет еще не был рассмотрен в данном курсе, и вы, возможно, не поймете его назначения. Но если бы тот же самый командлет был написан с использованием имени параметра, вы бы скорее догадались, для чего он служит:

Add-Member –memberType NoteProperty –Name ComputerName –Value LON-DC1

Возможно, вы все равно предпочтете обратиться в справочник за помощью, но в любом случае имя параметра позволяет быстрее определить, какие данные и для чего здесь используются.

Вы использовали позиционные параметры в течение всего курса. Dir, который, конечно, является псевдонимом для командлета Get-ChildItem может стать ярким примером. Where-Object – еще одним примером. Рассмотрим команду:

Get-Service | Where-Object { $_.Status –eq “Running” }

Вот та же самая команда, но все имена параметров в ней напечатаны полностью:

Get-Service | Where-Object –FilterScript { $_.Status –eq “Running”}

Привязка данных конвейера по значению

Привязка данных конвейера по значению

- Многие параметры предназначены для того, чтобы принимать данные из конвейера. Этот процесс называется *binding*.
  - Вы этот метод уже использовали в команде

```
Get-Service | Where-Object { $_.Status –eq "Running" }
```

```
-InputObject <psobject>
   Specifies the objects to be filtered. You can...

   Required?                       false
   Position?                       named
   Default value
   Accept pipeline input?          true (ByValue)
   Accept wildcard characters?     False
```

До настоящего времени мы фокусировали ваше внимание на командлетах, для которых вы указывали параметры, либо по имени, либо по позиции. Однако многие параметры предназначены для того, чтобы принимать данные из конвейера. В действительности, когда вы передаете данные от одного командлета к другому, командлет, который принимает данные, должен присоединить или привязать входящие объекты к одному из своих параметров. Рассмотрим команду:

Get-Service | Where-Object { $_.Status –eq “Running” }

Здесь вы передаете объекты командлету Where-Object. Но что именно делает Where-Object с этими объектами? Заглянем в справочник.

-InputObject <psobject>
Specifies the objects to be filtered. You can a
(Указывает, какие объекты следует отфильтровать.)
Required? false
Position? named
Default value
Accept pipeline input? true (ByValue)
Accept wildcard characters? False

Командлет поддерживает только два параметра: -FilterScript и –InputObject. Критерий фильтрации передается параметру –FilterScript по позиции. Что же происходит с –InputObject? –InputObject привязывается к входящим из конвейера

данным. Из справочника вы можете узнать, что этот параметр принимает входящие данные по значению. Тип входящих данных, который он требует, указан прямо рядом с именем параметра - <psobject>.. Это означает, что принимаются объекты любого типа. В справочнике говорится: «Указывает, какие объекты следует отфильтровать». Это означает, что этот параметр содержит объекты, подлежащие фильтрации.

Как это работает:

1. Get-Service обрабатывает и производят ряд служебных объектов.

2. Служебные объекты передаются командлету Where-Object.

3. Where-Object «видит», что есть данные для параметра –FilterScript.

4. Where-Object также «видит», что по каналу входящих данных поступает множество объектов. Where-Object проверяет, может ли какой-либо из его параметров принять эти объекты по значению.

5. Where-Object определяет, что его собственный параметр может принять этот тип данных по значению.

Многие командлеты могут привязывать входящие данные по значению. Найдите в справочнике полную информацию по таким командлетам как Stop- Process, Stop-Service, Start-Service, Remove-ChildItem и другим.

Обратите внимание, что привязка по значению всегда происходит в первую очередь. Вскоре вы узнаете о привязке по имени свойства. Эта привязка встречается только в том случае, если ни один параметр не смог принять входящие данные по значению.

Привязка данных конвейера по имени свойства

Привязка данных объекта по имени свойства

- В рамках данной техники оболочка ищет имя параметра, после чего проверяют, обладают ли входящие объекты соответствующими свойствами. Если да – соответствующее свойство привязывается к параметру. Обратите внимание, что данный тип привязки встречается только тогда, когда оболочка не смогла привязать входящие данные по значению. Например, изучите справочную информацию по командлету Get-Service:

```
-ComputerName <string[]>
  Gets the services running on the specified computers...

  Required?                       false
  Position?                       named
  Default value                   Localhost
  Accept pipeline input?          true (ByPropertyName)
  Accept wildcard characters?     False
```

Этот тип привязки входящих данных является более сложным. В рамках данной техники оболочка ищет имя параметра, после чего проверяют, обладают ли входящие объекты соответствующими свойствами. Если да – соответствующее свойство привязывается к параметру. Обратите внимание, что данный тип привязки встречается только тогда, когда оболочка не смогла привязать входящие данные по значению. Например, изучите справочную информацию по командлету Invoke-Command. Обратите внимание, что параметр –inputObject привязывает входящие данные по значению и принимает такой тип входящих данных, как объекты [object]. Этот тип объектов [object] является общим типов. По сути все что угодно можно назвать объектом. Это означает, что –inputObject всегда будет привязывать входящие данные по значению. То есть, для параметра не будет возможности привязать входящие данные по имени свойства.

Изучите полную информацию по командлету Get-Service, и вы обнаружите следующий параметр:

```
-ComputerName <string[]>
Gets the services running on the specified computers
Type the NetBIOS name, an IP address, or a fully qua
, a dot (.), or "localhost".
This parameter does not rely on Windows PowerShell or
configured to run remote commands.
Required? false
```

Position? named
Default value Localhost
Accept pipeline input? true (ByPropertyName)
Accept wildcard characters? False

(Здесь текст был урезан, чтобы поместиться в формат книги; чтобы прочитать его полностью, изучите раздел справочника по командлету Get-Service).

В справочнике указано, что параметр –computername принимает входящие данные из конвейера по имени свойства. Одним из способов запустить данную команду будет:

Get-Service –computername LON-DC1

Можно даже перечислить имена компьютеров через запятую:

Get-Service –computername LON-DC1,SEA-DC1

Однако можно использовать и другую технику – принять входящие данные, используя свойство ComputerName. Например, вы можете создать с помощью Windows Notepad такой файл:

ComputerName
LON-DC1
SEA-DC1

Его следует сохранить как CSV-файл. Несмотря на то, что здесь нет запятых, это полноценный CSV-файл: здесь есть заголовок с именем столбца и две строчки данных. Здесь не нужны запятые, поскольку столбец всего один. Файл можно импортировать в оболочку с помощью команды:

Import-CSV computernames.csv

Командлет Import-CSV произведет в качестве выходных данных два объекта, каждый из которых будет обладать свойством ComputerName. Эти объекты можно передать командлету Get-Service:

Import-CSV computernames.csv | Get-Service

Поскольку свойство объектов ComputerName совпадает с параметром –computername, ваши два имени будут отправлены в качестве входящих данных параметру –computername, и командлет Get-Service попытается извлечь данные из обоих компьютеров.

Параметр, осуществляющий привязку по имени свойства, работатет только тогда, когда входящие объекты имеют имя свойства, которое полностью и в точности совпадает с именем параметра (хотя регистр здесь не учитывается). Так, свойство под названием comp не совпадет с параметром –computername.

Переименование свойств

Предположим, вы используете командлет, который генерирует имена компьютеров, и хотите передать эти имена компьютеров другому командлету, который обладает параметром –computername. Проблема в том, что первый командлет может не суметь превратить имена компьютеров в свойство computername. Примером может быть командлет Get-ADComputer.

Мы еще не рассматривали командлеты Active Directory; мы приступим к этому лишь в следующей главе. Пока просто используйте Get-ADComputer в качестве примера. Можете попробовать запустить следующие команды, если вы используете контроллер домена классной комнаты и пользуетесь при этом либо консолью Windows PowerShell, либо Windows PowerShell ISE.

Запустите команды:

```
Import-Module ActiveDirectory
Get-ADComputer –filter *
```

Результат будет примерно следующим:

```
DistinguishedName : CN=SERVER-R2,OU=Domain
Controllers,DC=company,DC=pri
DNSHostName : server-r2.company.pri
Enabled : True
Name : SERVER-R2
ObjectClass : computer
```

```
ObjectGUID : 0f04c9d6-44af-4e4f-bb7e-4ebc52ab343f
SamAccountName : SERVER-R2$
SID : S-1-5-21-1842503001-3345498097-2301419571-1000
UserPrincipalName
```

Данные объекты имеют свойство Name. Это не соответствует параметру – computerName, который привязывает входящие данные по имени свойства. Поэтому, например, данная команда не увенчается успехом:

```
Get-ADComputer –filter * | Get-Service
```

Тем не менее, вы можете сделать нечто подобное. Например, использовать Select-Object, чтобы добавить новые свойства объектам. Свойство получит имя computername и будет содержать то же значение, что и существующее свойство name. Вот команда:

```
Get-ADComputer –filter * | Select
@{Label=”ComputerName”;Expression={$_.Name}}
```

В результате вы получите объекты, обладающие только одним свойством – ComputerName. Если вы хотите сохранить существующие свойства, вы должны дать команду Select-Object включить эти свойства в выходные данные:

```
Get-ADComputer –filter * | Select *,
@{Label=”ComputerName”;Expression={$_.Name}}
```

Добавление символа * к списку свойств означает команду сохранить все свойства из входящих данных. С новым свойством computername данная команда сработает (конечно, учитывая, что у вас есть разрешение на соединение с удаленными компьютерами):

```
Get-ADComputer –filter * | Select *,
@{Label=”ComputerName”;Expression={$_.Name}} | Get-Service
```

Это довольно мощная техника, которая избавит вас от необходимости писать длинные сложные скрипты.

passThru

Understanding Passthrough

- Some “action” cmdlets accept pipeline input, but do not provide output.

```
New-ADUser –name JohnD –samaccountname JohnD

# This command has no output. Results can't be piped to other commands as is.
```

- Cmdlets that provide no output by default require an extra parameter (-passThru) to pass on its objects.

```
New-ADUser –name JohnD –samaccountname JohnD –passThru | Enable-ADAccount

#New-ADUser result is now piped to Enable-ADAccount
```

- -passThru is used by many cmdlets. See help for details.

Большинство так называемых командлетов действия могут принимать входящие данные, но не могут производить что-либо. Хорошим примером является командлет New-ADUser. Например, вы можете создать простой пользовательский аккаунт с помощью команды:

New-ADUser –name JohnD –samaccountname JohnD

Однако никаких выходных данных произведено не будет, и имя пользователя будет находиться в отключенном состоянии. Если вы хотите «оживить» пользователя, то после предыдущей команды запустите такую:

Get-ADUser –filter “name –eq JohnD” | Enable-ADAccount

Однако командлет New-ADUser, так же как и многие другие командлеты действия, поддерживает параметр –passThru. Этот параметр дает командлету команду отправлять в конвейер в качестве выходных данных тот объект, с которым он работал или который он создал. Поэтому, можете попробовать что-то в этом роде:

New-ADUser –name JohnD –samaccountname JohnD –passThru | Enable-ADAccount

Stop-Service – еще один командлет, который поддерживает –passThru. Обычно Stop-Service не производит никаких выходных данных. Однако с помощью –passThru он меняет поведение. Попробуйте:

Get-Service –name BITS | Stop-Service –passthru

Данная команда сгенерирует выходные данные, и вы увидите сервисы, которые командлет пытается закрыть.

Windows® Management Instrumentation

WMI

- Windows® Management Instrumentation или WMI – это технология управления, являющаяся частью операционной системы Microsoft® Windows. Впервые она появилась еще в Windows NT® 4.0, и обеспечивала более стабильный и постоянный доступ к настройкам конфигурации как локального, так и удаленных компьютеров.
- Однако практическое использование WMI не всегда было простым – более ранние технологии, такие как Microsoft Visual Basic® Scripting Edition (VBScript), требовали программного подхода к использованию WMI, и далеко не все администраторы могли справиться с этой задачей.
- Windows PowerShell предлагает администраторам самый легкий и доступный во многих отношениях способ работы с WMI.

Windows® Management Instrumentation или WMI – это технология управления, являющаяся частью операционной системы Microsoft® Windows. Впервые она появилась еще в Windows NT® 4.0, и обеспечивала более стабильный и постоянный доступ к настройкам конфигурации как локального, так и удаленных компьютеров.

Долгие годы WMI была одним из самых мощных и доступных инструментов администратора для получения управленческой информации и осуществления тех или иных изменений в конфигурации, особенно на удаленных компьютерах. Однако практическое использование WMI не всегда было простым – более ранние технологии, такие как Microsoft Visual Basic® Scripting Edition (VBScript), требовали программного подхода к использованию WMI, и далеко не все администраторы могли справиться с этой задачей. Windows PowerShell предлагает администраторам самый легкий и доступный во многих отношениях способ работы с WMI.

Важно понимать, что WMI может служить совершенно разным целям для разных групп пользователей. Например, разработчики программного обеспечения взаимодействуют с WMI при создании приложений, таких как Microsoft System

Center Configuration Manager. В этом модуле мы будем рассматривать WMI как административный инструмент, и будем говорить только о тех элементах WMI, которые имеют отношение к повседневной работе администратора.

Обзор WMI

WMI – это технология управления, которая внедряется в операционную систему Windows со времен Windows NT 4.0. Каждая новая версия Windows, так же, как и каждая новая версия многих других продуктов Microsoft, обновляет WMI, придавая ей новые черты и наделяя новыми возможностями.

WMI предоставляет информацию в соответствии с моделью Common Information Model или CIM, которая была разработана промышленной группой Distributed Management Task Force или DMTF при участии Microsoft и других поставщиков. Но несмотря на то, что WMI стремится отобразить информацию стандартизированным способом, она почти не устанавливает жестких правил для разработчиков программного обеспечения. В результате WMI зачастую воплощается совершенно по-разному в разных продуктах Microsoft и сторонних продуктах. К сожалению, это делает изучение WMI очень сложным, причем проблема усугубляется тем фактом, что многие варианты реализации WMI нигде не задокументированы. Windows PowerShell может упростить доступ к WMI в случае, если вы будете знать, какой именно элемент WMI вам нужен, но не может полностью решить проблему отсутствия документации или отсутствия жестких стандартов внедрения WMI.

Отчасти сложность WMI объясняется тем фактом, что она отвечает нуждам различных групп пользователей, включая разработчиков ПО, которым часто требуется более детальное и менее качественное отображение управленческой информации. В будущем, вы, возможно, будете замечать, что все больше командлетов Windows PowerShell создается в виде оболочек вокруг технологии WMI. Такие оболочки будут обеспечивать более простые и узкоспециализированные способы работы с WMI, и в то же время скрывать многие из существующих «сложностей». Возможно, что некоторые из командлетов, которыми вы уже пользуетесь, в действительности используют WMI. Но даже полный переход на WMI не представляет собой никакой опасности. Это ценная технология, и, несмотря на некоторые противоречие и прочие недостатки, она дает довольно мощные возможности.

В качестве администратора вы, вероятно, будете сталкиваться с WMI в основном для извлечения информации о Windows, аппаратном обеспечении компьютера и продуктах, запущенных в Windows. В отдельных случаях вам также может потребоваться WMI для произведения изменений в конфигурации.

Взаимодействие WMI

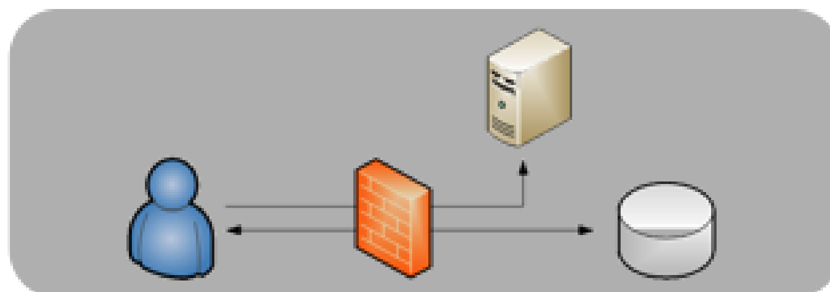
Когда вы используете WMI для установления соединения с удаленным компьютером, вы, по сути, используете подсистему WMI на своем локальном компьютере для поключения к WMI сервису на удаленном компьютере. Взаимодействие двух компьютеров происходит с использованием протокола Remote Procedure Call или RPC. RPC – это старый, проверенный временем протокол, который используется в Windows более десяти лет. Однако по мере того,

как локальные брандмауэры получали большее распространение, RPC становился все сложнее в использовании и управлении.

Как правило, первое RPC соединение компьютера происходит с распределителем конечной точки удаленного компьютера. Этот распределитель конечной точки является частью сервиса RPC на удаленном компьютере и занимает собой хорошо известный TCP порт. Вы можете представить, что вы просто открываете порт в локальном брандмауэре, но в действительности на порт не ложится вся нагрузка по осуществлению RPC взаимодействия. Вместо этого распределитель конечной точки выбирает новый произвольный TCP порт для всего оставшегося взаимодействия. Так как порт выбирается произвольно, сложно создать статичные правила брандмауэра, разрешающие RPC трафик.

Комммуникации WMI

- Коммуникации WMI используют протокол Remote Procedure Call (RPC)
  - Использует распределитель конечной точки
  - распределитель конечной точки выбирает новый произвольный TCP порт для всего оставшегося взаимодействия.
  - сложно создать статичные правила брандмауэра, разрешающие RPC трафик



- Windows Firewall поддерживает исключение *Remote Management*
  - Это исключение позволяет динамически открывать порты для WMI RPC траффика

Windows Firewall поддерживает в качестве исключения сервис Remote Management, который может разрешить RPC, необходимый для WMI. Прочие брандмауэры могут иметь аналогичные возможности – уточнить это можно у их поставщиков. Другие настройки конфигурации, включая User Account Control (UAC) и Distributed Component Object Model (DCOM) могут также задействовать WMI-коммуникации; веб-страницы, расположенные по посещенным ранее URL адресам, выдают информацию при конфигурировании этих настроек.

Структура WMI

Между тем, WMI – это не просто цельная программа. Она включает в себя несколько провайдеров, каждый из которых соединяет WMI-сервис с конкретным продуктом, технологией, функцией и.т.д. WMI провайдер действует почти как

драйвер устройства, обеспечивая WMI сервису доступ к различным продуктам Windows. Например, на компьютерах с операционной системой Windows Server, на которых установлен сервис Windows DNS, WMI провайдер позволяет WMI сервису запрашивать и создавать ресурсные записи DNS, а также настройки конфигурации DNS.

Вся информация, которую извлекает WMI-провайдер, регистрируется в WMI-хранилище, централизованной конфигурационной базе данных, которая указывает WMI, что информация доступна на том или ином конкретном компьютере. Важно понимать, что информация, которую вы получаете с помощью WMI, в действительности не находится в хранилище: хранилище содержит лишь перечень доступной информации, и WMI извлекает ее динамически, по запросу. Хранилище состоит из пространств имен, которые приблизительно соответствуют отдельным продуктам и технологиям. Пространства имен расположены иерархически, а значит, они могут включать в себя подпространства.

Структура WMI

Root\Cimv2
Root\MicrosoftDNS
Root\MicrosoftActiveDirectory
Root\SecurityCenter

Win32_Account
Win32_BIOS
Win32_Desktop
Win32_Fan
Win32_Group
Win32_Keyboard
Win32_LogicalDisk
Win32_NetworkAdapterConfiguration
Win32_NTDomain
Win32_Product
Win32_Service

Пространство имени верхнего уровня является корневым. Другие пространства могут содержать:

- Root\Cimv2
- Root\MicrosoftDNS
- Root\MicrosoftActiveDirectory
- Root\SecurityCenter

Не все компьютеры Windows содержат одни и те же пространства имен. Например, клиентский компьютер не будет содержать Root\MicrosoftActiveDirectory, а серверный компьютер не будет содержать Root\SecurityCenter.

Для того, чтобы работать с пространством имен на удаленном компьютере, это пространство не обязательно нужно устанавливать на свой компьютер. Например, если вы используете компьютер под управлением Windows 7 и хотите работать с Windows DNS на удаленном сервере, вы можете сделать это через пространство имен Root\MicrosoftDNS, установленное на этом удаленном компьютере, несмотря на то, что на вашем компьютере его нет.

Классы

Внутри каждого пространства имен WMI выделяет один или несколько классов. Класс – это абстрактное значение компонента управления. Например, пространство имен Root\Cimv2 содержит класс, который называется Win32_TapeDrive. Этот класс определяет свойства ленточного накопителя и существует в пространстве имени независимо от того, подключен данный носитель к компьютеру или нет.

Классы пространства имен The Root\Cimv2 почти всегда имеют префикс Win32_, даже на 64-битных компьютерах. Этот префикс не используется в названиях классов других пространств имен. В других пространствах имен названия классов вообще редко имеют какой-либо префикс.

К другим классам пространства имен Root\Cimv2 относятся:

- Win32_Account
- Win32_BIOS
- Win32_Desktop
- Win32_Fan
- Win32_Group
- Win32_Keyboard
- Win32_LogicalDisk
- Win32_NetworkAdapterConfiguration
- Win32_NTDomain
- Win32_Product
- Win32_Service

Как вы видите, пространство имен Root\Cimv2 содержит классы, имеющие отношение к операционной системе Windows и аппаратному обеспечению компьютера. Это одно из немногих пространств имен, содержащихся на каждом компьютере с Windows, хотя классы внутри этого пространства имен различаются на серверных и клиентских компьютерах, а также на различных версиях Windows ОС. Классы определяют два важных элемента: свойства и методы. Каждый класс имеет как минимум одно свойство, и подобно другим объектам в Windows PowerShell, свойства класса обеспечивают доступ к управленческой и

конфигурационной информации. Некоторые классы обладают методами, которые требуют определенных действий, например, перезагрузки компьютера или изменения конфигурации.

Экземпляры

Реальное существование класса называют экземпляром. Например, если на вашем компьютере есть четыре логических диска, то вы имеете четыре экземпляра класса Win32_LogicalDisk. Если на вашем компьютере нет подключенного ленточного накопителя, у него будет нулевой экземпляр класса Win32_TapeDrive, хотя сам класс будет существовать в виде абстрактного значения.

Экземпляры являются такими же объектами, как и любые другие объекты, которые используются в Windows PowerShell. Экземпляры обладают свойствами и методами, которые определены их классом, и вы можете работать с этими свойствами и методами внутри конвейера оболочки.

Во многих случаях экземпляры предназначены только для чтения, что означает, что вы можете извлечь значение свойства, но не можете изменить его. Особенно это касается классов в пространстве имен Root\Cimv2. Для того, чтобы вы могли изменить настройки конфигурации, класс должен обладать методом, который вы сможете использовать для осуществления изменений. За пределами пространства имен Root\Cimv2 принцип работы может быть другим. В некоторых случаях вы можете изменить значение свойства путем прикрепления к нему нового свойства. Примером этого может служить пространство имен для Internet Information Services (IIS) 6 внутри WMI. В этом заключается одно из главных противоречий WMI: в разные времена разные группы специалистов, занимающихся разработкой новых продуктов Microsoft, высказывали разные идеи по поводу того, как WMI может использоваться для управления продуктами, технологиями и функциями. Иногда идеи одной группы находили поддержку у специалистов другой группы и были адаптированы ими, что приводило к некоторой стабильности и схожести различных пространств имен; в других случаях эти идеи никогда более не были использованы, и пространства имен, использованные той или иной группой специалистов, оставались уникальными внутри WMI.

Как найти нужный класс.

При существовании десятков тысяч WMI классов найти тот единственный, который помог бы выполнить конкретную задачу, очень сложно. Центральной директории WMI классов не существует, к тому же, существует очень мало способов (конечно, кроме использования поисковых машин) поиска классов по ключевым словам. Для поиска нужного класса следует основываться на трех базовых принципах:

• Опыт. По мере того, как вы будете приобретать опыт в использовании WMI, вы научитесь угадывать имена классов, после чего с помощью специальных инструментов или поисковых машин вы сможете подтвердить свои догадки. Также может помочь обращение к более опытным администраторам за помощью. Такую возможность предоставляют некоторые онлайн-сообщества, например:

• The “Windows PowerShell” Internet newsgroup (доступна по адресу: http://go.microsoft.com/fwlink/?LinkId=193516)

• www.PoshComm.org

• www.PowerShell.com

• www.ScriptingAnswers.com

• www.ConcentratedTech.com

• Исследования. Различные «исследовательские» и «поисковые» инструменты WMI позволяют просматривать WMI хранилища как на локальном, так и на удаленных компьютерах. В хранилищах классы обычно располагаются в алфавитном порядке, что упрощает поиск класса по названию и просмотр доступных классов.

• Примеры. Используя поисковые интернет-машины для поиска по ключевым словам (например, “WMI BIOS” или “WMI disk space), зачастую можно обнаружить примеры, написанные другими людьми. Эти примеры, даже если они и не являются в точности тем, что вам нужно, могут помочь вам найти классы, имеющие отношение к выполняемой вами задаче. Хорошей стартовой точкой для поиска примеров, имеющих отношение ко многим видам административной работы, является Microsoft TechNet Script Center Repository (http://go.microsoft.com/fwlink/?LinkId=193517).

Обратите внимание, что даже примеры, имеющие отношение к другим технологиям, таким как VBScript, JScript или Perl, могут быть полезными для поиска названий WMI классов. Не исключайте такие варианты примеров лишь потому, что они не относятся исключительно к Windows PowerShell.

Поиск классов внутри шелла

- Командлет Get-WmiObject

```
Get-WmiObject –namespace root\cimv2 –list

Get-WmiObject –namespace root\cimv2 –list
–computername SEA-DC1

Get-WmiObject –namespace root\cimv2 –list
–computername SEA-DC1 –credential CONTOSO
\Administrator

Get-WmiObject –namespace root –class
“__namespace” | ft name
```

Помимо этого, вы можете дать Windows PowerShell команду отобразить список классов:

Get-WmiObject -list

Эта команда отображает список классов пространства имен, используемого по умолчанию. Чтобы отобразить классы из другого пространства имен, следует добавить параметр – namespace и уточнить имя пространства имен, например, Root\SecurityCenter. Также оболочка может помочь найти классы, название которых содержит определенную строку символов:

Get-WmiObject -list -class *user*

Поскольку параметр –class является позиционным, вы можете даже опустить имя параметра –class:

Get-WmiObject -list *user*

WMI развивалась годами, и продолжает развиваться. Это означает, что конкретный WMI класс может вести себя слегка по-разному в разных версиях Windows, могут иметь меньше свойств или полностью отсутствовать в старых версиях Windows. Обязательно учитывайте это, особенно при работе с удаленными компьютерами. Пространства имен на разных компьютерах могут очень сильно различаться, не считая пространства имен Root\Cimv2. Клиентский и серверный компьютер обычно имеют разные наборы пространств имен; пространства имен часто бывают доступными только когда соответствующий продукт или технология установлены на компьютер.

Документация

Было бы замечательно, если бы для каждого пространства имен и WMI класса существовало бы централизованное хранилище. К сожалению, это не так.

Главные классы операционной системы, такие как классы в пространстве имен Root\Cimv2, последовательно задокументированы; эта документация хранится в Интернете как часть библиотеки Microsoft Developer Network (MSDN®). Поскольку библиотека периодически подвергается реорганизации, меняя URL адреса, самый простой способ найти документацию для класса – воспользоваться поисковыми системами. Например, если вы введете в строку поиска Win32_Service, вы обнаружите, что первые результаты выдачи будут вести именно в Библиотеку, а уже оттуда вы можете начать поиск других классов в пространстве имен, используя таблицу дерева контента.

Вне пространства имен Root\Cimv2 документация намного менее стабильна и согласованна, как по форме, так и в плане доступности. У многих групп разработчиков просто не было времени, чтобы подготовить необходимую документацию. В других случаях группы разработчиков могли создать WMI классы для собственного использования, но не думали о том, что эти классы станет использовать кто-то еще, поэтому, документация так и не была опубликована для широкого доступа.

Если документация доступна – а это чаще всего бывает с классами пространства имен Root\Cimv2 – обращайте пристальное внимание на детали, указывающие, с какими продуктами ли с какой версией Windows данный класс работает. Однако не забывайте, что классы могут различаться в зависимости от версии Windows. Несмотря на то, что класс Win32_Processor присутствует, например, на Windows NT 4.0, он ведет себя совсем не так, как на Windows Server 2008 R2 (хотя в данном случае в документации объясняется разница).

Безопасность WMI

WMI содержит мощную систему безопасности. Защита может устанавливаться как для целого пространства имен, так и для отдельного атрибута класса. По умолчанию большая часть защитной информации конфигурируется в пространстве имен Root, тогда как прочие пространства имен и их классы перенимают это корневое разрешение.

Также по умолчанию защита WMI конфигурируется таким образом, чтобы локальные пользователи могли запрашивать почти любую информацию из WMI. Члены локальной административной группы (которая в доменной среде обычно включает доменные админские группы доменного уровня), могут запрашивать информацию удаленно. Не стоит менять настройки конфигурации системы безопасности WMI до тех пор, пока вы не будете осознавать на 100%, что вы делаете, и каковы возможные последствия. Модификация системы безопасности может привести к полной остановке WMI или негативно повлиять на продукты, основанные на WMI, такие как System Center Configuration Manager. Настройки безопасности, заданные по умолчанию, подходят для большинства ситуаций, и чаще всего самым мудрым решением будет не трогать их совсем.

Настройки безопасности WMI можно просмотреть и модифицировать с помощью оснастки WMI Control в консоли Microsoft Management Console (MMC). По умолчанию консоль не содержит эту оснастку; вам необходимо открыть новое MMC-окно или существующую консоль и добавить оснастку вручную.

Обратите внимание, что WMI чувствительна к User Account Control (UAC) на тех версиях Windows, которые поддерживают эту технологию защиты. Если вы планируете запросить WMI информацию с удаленного компьютера, вы должны обладать действующим администраторским токеном. Обычно это означает, что вы должны либо работать в Windows PowerShell как «Администратор» (чтобы иметь необходимую привилегию), либо обеспечить альтернативный мандат после установления WMI соединения.

Просмотр WMI из PowerShell

Windows PowerShell также может помочь вам изучить все классы в заданном пространстве имен, так как она может выдать список всех классов, присутствующих в пространстве имен. Чтобы получить список всех классов, содержащихся в пространстве имен, используемом по умолчанию, запустите команду:

```
Get-WmiObject –namespace root\cimv2 –list
```

Вы можете заменить название пространства имен Root\Cimv2 другим названием пространства имен для того, чтобы увидеть классы, содержащиеся в нем. А если добавить параметр –computerName, можно увидеть классы пространства имен на удаленном компьютере:

```
Get-WmiObject –namespace root\cimv2 –list –computername LON-DC1
```

Обычно командлет Get-WmiObject использует мандат того пользовательского аккаунта, через который вы открыли окно Windows PowerShell. Это единственный аккаунт, который может запрашивать WMI информацию локально. WMI сама по себе не позволит использование альтернативного мандата для локального соединения. Однако для удаленного соединения вы можете добавить параметр –credential, чтобы указать альтернативный аккаунт. Имя пользователя должно быть в формате DOMAIN\USERNAME:

```
Get-WmiObject –namespace root\cimv2 –list –computername LON-DC1 –credential
CONTOSO\Administrator
```

Наконец, чтобы увидеть список всех пространств имен, запустите в Windows PowerShell следующую команду:

```
get-wmiobject -namespace root -class "__namespace" | ft name
```

Использование WMI

Использование WMI

Get-WmiObject Win32_Service
gwmi Win32_Service

Gwmi Win32_Service | Get-Member

Gwmi Win32_Service
–computerName "LON-DC1", "SEA-DC2"

Gwmi Win32_Service
(Get-Content c:\names.txt)

Windows PowerShell делает доступной большую часть WMI информации с помощью двух простых командлетов, тем самым предоставляя вам легкий способ извлечения WMI информации с локального и удаленного компьютера, а также способ задействовать WMI методы.

Помните, что Windows PowerShell отображает всю WMI информацию в форме объектов, так же, как она поступает с любой другой информацией. Это означает, что командлеты, используемые для сортировки, фильтрации, группировки, экспорта, сравнения, и.т.д. могут использоваться в сочетании с WMI в рамках оболочки.

Опрос WMI

Командлет Get-WmiObject отвечает за извлечение информации либо с локального компьютера, либо с одного или нескольких удаленных компьютеров. Параметр –class уточняет имя класса, которое вы хотите извлечь; по умолчанию извлекаются все экземпляры этого класса. Если класс находится вне пространства имен Root\Cimv2, используйте параметр –namespace для уточнения имени пространства имен, в котором можно найти класс. Также используйте параметр –computername, чтобы уточнить имя удаленного компьютера и параметр –credential, чтобы уточнить альтернативный мандат для удаленного подключения. За один раз можно запросить только один класс.

Командлет извлекает указанные экземпляры и помещает их в конвейер, например:

Get-WmiObject Win32_Service

Обратите внимание, что некоторые классы WMI перехлестываются с другими командлетами Windows PowerShell, например, Get-Service. Однако вы, возможно, уже заметили различия в их работе и способе отображения информации. Например, Get-Service использует Microsoft .NET Framework для извлечения информации о сервисах; эта информация не включает некоторые детали, такие как режим запуска сервиса или его учетная запись, поскольку такая информация не включена в Framework. Класс WMI Win32_Service, в свою очередь, включает информацию о режиме запуска и учетной записи сервиса. Вы можете увидеть эти свойства, передав экземпляр свойства по конвейеру командлету Get-Member:

Get-WmiObject Win32_Service | Get-Member

Также вы можете увидеть все свойства всех экземпляров класса, используя Format-List:

Get-WmiObject Win32_Service | Format-List *

Встроенный псевдоним командлета Get-WmiObject - gwmi.

В разделе справочника, посвященном командлету Get-WmiObject, обратите внимание на параметр –computerName. В справочнике по синтаксису он указывается в следующем виде:

[-ComputerName <string[]>]

Значение <string[]> указывает на то, что параметр может принимать несколько имен компьютеров. Одним из способов указать эти имена является их перечисление через запятую, поскольку Windows PowerShell автоматически интерпретирует такой список как набор объектов:

-computerName "LON-DC1","SEA-DC2"

Однако к параметру –computerName можно прикрепить все, что обозначает набор объектов. Например, вы можете создать текстовый файл, в котором в каждой строчке указано одно имя компьютера:

LON-DC1
SEA-DC2
NYC-SRV7

Для чтения этого файла вы можете использовать командлет Get-Content. Командлет обрабатывает каждую строчку текстового файла как объект строки – именно это необходимо параметру –computerName:

-computerName (Get-Content c:\names.txt)

Скобки указывают оболочке на то, что команду Get-Content необходимо выполнить в первую очередь. Результаты этого командлета передаются параметру –computerName.

При соединении с несколькими удаленными компьютерами командлет Get-WmiObject все же может работать с альтернативным мандатом, на который указывает параметр –credential. Однако для всех компьютеров должен использоваться один и тот же мандат. Множественные соединения производятся

последовательно, а не параллельно, поэтому, может потребоваться некоторое время для извлечения списка имен. Если командлен по каким-то причинам не может получить доступ к компьютеру (компьютер находится в режиме оффлайн, доступ заблокирован или не получено разрешение), то ситуация воспринимается как бесконечная ошибка, в результате чего командлет выдает сообщение об ошибке и продолжает работу, пытаясь подключиться к следующему в списке компьютеру.

Объекты, возвращаемые командлетом Get-WmiObject, всегда обладают свойством __SERVER, которое содержит имя компьютера, из которого поступил данный объект. Это свойство может быть полезным, когда вы отправляете запрос нескольким компьютерам, так как позволяет отсортировать и сгруппировать объекты по имени компьютера:

```
Get-WmiObject Win32_Service –computer (Get-Content c:\names.txt) | Sort
__SERVER | Format-Table –groupBy __SERVER
```

Командлет Get-WmiObject всегда использует Remote Procedure Calls (RPCs) для подключения к удаленным компьютерам; «родные» способности удаленной работы Windows PowerShell не задействуются. По этой причине Get-WmiObject может подключиться к любому удаленному компьютеру, на котором запущен сервис WMI, даже если на нем не установлена Windows PowerShell.

Еще одним способом запросить данные с нескольких компьютеров является использование командлета ForEach-Object. Помните, что командлет Get- Content читает текстовый файл и возвращает каждую строчку в виде объекта. ForEach-Object дает возможность перечислить все эти объекты. Если бы каждая строчка текста содержала имена компьютеров, вы бы перечисляли имена компьютеров. Преимущество этой техники состоит в том, что командлет ForEach-Object может выполнять множественные команды по отношению к каждому компьютеру:

```
Gc c:\names.txt | ForEach-Object {
gwmi win32_computersystem;
gwmi win32_operatingsystem } |
select buildnumber,servicepackmajorversion,totalphysicalmemory
```

Скриптовый блок ForEach-Object содержит два WMI запроса, каждый из которых возвращает разные объекты. Объекты обоих типов отправляются по конвейеру командлету Select-Object. Select-Object затем выбирает три указанные свойства. Выходные данные будут выглядеть примерно так:

```
buildnumber      servicepackmajorversion         totalphysicalmemory
-----------      -----------------------         -------------------
                                                 3220758528
7600             0
```

Если вы посмотрите внимательно, то заметите, что выходные данные расположены не в одну строчку. Первый объект в конвейере – это Win32_ComputerSystem, который не имеет свойства BuildNumber или ServicePackMajorversion – поэтому, данные колонки в первой строчке будут пустыми. Второй объект в конвейере не имеет свойства TotalPhysicalMemory, поэтому, эта колонка во второй строчке

является пустой. Другими словами, вы увидите пример, в котором эти отдельные куски информации объединяются в один поток выходных данных.

Tips and Tricks

- Вычисления

```
gwmi win32_logicaldisk |
 select deviceid,drivetype,
 @{Label='freespace(gb)';Expression={$_.freespace/1GB}}
```

```
Gwmi win32_operatingsystem |
 select caption,
 @{Label='PhysMemory';
 Expression={(gwmi
win32_computersystem).totalphysicalmemory}}
```

- Фильтрация

```
Gwmi Win32_Service | Where { $_.Name –eq 'BITS' }
Gwmi Win32_Service –filter "Name = 'BITS'"
```

Вы уже знаете, как использовать командлет Select-Object и добавлять новые свойства объекту. Часть значения свойства под названием Expression может содержать почти любой код Windows PowerShell, в том числе и абсолютно новый командлет. Например, следующая команда позволит извлечь сведения о логических дисках компьютера и отобразить информацию об имеющемся на них свободном месте в гигабайтах:

```
gwmi win32_logicaldisk |
select deviceid,drivetype,
@{Label='freespace(gb)';Expression={$_.freespace/1GB}}
```

Чуть более усовершенствованный вариант этой техники может задействовать параметр –as, чтобы привести полученные данные об имеющемся свободном месте к целому числу. Поскольку целое число не имеет дробей, результат округляется в сторону ближайшего целого числа, а десятичные значения игнорируются:

```
gwmi win32_logicaldisk |
select deviceid,drivetype,
@{Label='freespace(gb)';Expression={$_.freespace/1GB –as [int]}}
```

Другой вариант этой техники позволяет объединять информацию из двух или более WMI классов в единую строку выходных данных. Предположим, вы хотите

извлечь информацию о версии Windows и объеме физической памяти компьютера. Вы можете начать с запроса информации о классе Win32_OperatingSystem:

```
Gwmi win32_operatingsystem | select caption
```

Затем вы можете добавить новое пользовательское свойство в выходные данные, и значением этого свойства будет абсолютно новый WMI запрос к классу

```
 Win32_ComputerSystem:
Gwmi win32_operatingsystem |
select caption,
@{Label='PhysMemory';
Expression={(gwmi win32_computersystem).totalphysicalmemory}}
```

Что мы здесь видим? Часть команды Expression выполняет новый WMI запрос. Обратите внимание, что запрос помещен в круглые скобки. Благодаря этому, результаты запроса воспринимаются как объект, что особенно важно по той причине, что этот запрос возвращает только один WMI объект (в конце концов, на любом компьютере установлена только одна операционная система). Точка после закрывающей скобки говорит о том, что вы хотите получить доступ к составной части WMI объекта, возвращенного в результате выполнения запроса, в частности, к свойству TotalPhysicalMemory. Значение этого свойства становится значением нашего пользовательского свойства PhysMemory. Попробуйте запустить вышеуказанную команду и посмотрите результат.

Что если вы пытались бы связаться с удаленным компьютером? Один из способов сделать это – просто уточнить имя компьютера для обоих WMI запросов.

```
Gwmi win32_operatingsystem –comp Server1 |
select caption,
@{Label='PhysMemory';
Expression={
(gwmi win32_computersystem –comp Server1).totalphysicalmemory}
}
```

Но есть способ лучше – уточнить имя компьютера только для начального WMI соединения. После того как WMI объект станет доступным, вы можете использовать специальное свойство __SERVER, чтобы получить доступ к имени компьютера. Это означает, что вам придется изменить имя компьютера только в одном месте, для того чтобы отправить запрос другому компьютеру:

```
Gwmi win32_operatingsystem –comp Server1 |
select caption,
@{Label='PhysMemory';
Expression={
(gwmi win32_computersystem –comp $_.__SERVER).totalphysicalmemory}
}
```

В данном примере заполнитель $_ , расположенный в скриптовом блоке Expression, содержит WMI_объект, который был передан по конвейеру командлету Select-Object. Вслед за ним идет точка, которая указывает, что мы хотим получить доступ к части объекта, содержащегося в данном заполнителе. Мы получаем

доступ к параметру __SERVER, который содержит информацию об имени компьютера, из которого пришел WMI-объект. Это имя компьютера станет значением параметра –computerName в нашем следующем WMI запросе.

Отбор данных

В некоторых случаях вам, возможно, не понадобятся все экземпляры классов. Например, вы хотите запросить информацию только об одном сервисе, BITS. Одним из способов добиться этого станет использование командлета Where-Object:

```
Gwmi Win32_Service | Where { $_.Name –eq 'BITS' }
```

Однако эта техника имеет один недостаток – в данном случае оболочка отправляет запрос в каждый экземпляр класса, возвращает результаты и проверяет каждый из них, чтобы определить, какой из них соответствует вашим критериям. Когда вы извлекаете данные о большом количестве объектов, особенно с удаленных компьютеров, этот процесс может занять очень много времени и привести к большой нагрузке на процессор.

Командлет Get-WmiObject предлагает параметр –filter, который отправляет ваш критерий WMI-сервису. WMI-сервис может отфильтровать информацию намного быстрее и вернуть только те результаты, которые соответствуют вашим критериям. К сожалению, WMI-сервис не принимает символы сравнения Windows PowerShell, поэтому, вам придется вспомнить несколько другой синтаксис для указания критериев:

```
Gwmi Win32_Service –filter "Name = 'BITS'"
```

Пару слов об этом синтаксисе:

• Строки могут быть заключены в одинарные кавычки. Поэтому, весь критерий обычно заключен в двойные кавычки, как показано выше.

• К знакам сравнения относятся:

= (equality)

<> (inequality)

>= (greater than or equal to)

<= (less than or equal to)

> (greater than)

< (less than)

• LIKE (разрешает групповые символы)

• Не используйте заполнитель $_, поскольку он работает только в Windows PowerShell, но не в WMI. Для указания критерия WMI указывайте имя свойства.

• Можно использовать ключевые слова AND и OR, чтобы указать несколько критериев.

Символ LIKE является особенно гибким. Применяя его, вы можете:

• Использовать % в качестве группового символа. “Name LIKE ‘%windows%’”

• Использовать _ в качестве 1-символьного группового знака.

• Использовать [и], чтобы обозначить ряд или набор символов: “DriveType LIKE ‘[abcdef]’”

WMI, как и Windows PowerShell, обычно нечувствительна к регистру. Ключевые слова, такие как LIKE могут быть напечатаны и строчными буквами – like; символы в командной строке, например, названия сервисов, также нечувствительны к регистру. Некоторые WMI-провайдеры чувствительны к регистру, но это, скорее, исключения.

И, наконец, лучший способ произвести фильтрацию – это расположить критерии фильтрации как можно ближе к левому краю командной строки Windows PowerShell. Когда вы можете произвести фильтрацию путем использования параметра –filter командлета, предпочтительнее использовать Where-Object для выполнения этой задачи.

WQL-синтаксис

Иногда вы можете столкнуться с языком написания запросов WMI Query Language (WQL), например, в рамках более старых технологий, таких как VBScript, где WQL был приоритетным способом составления запросов по извлечению данных. Windows PowerShell позволяет использовать такие запросы. Для этого нужно просто прикрепить строку запроса к параметру –query командлета Get-WmiObject:

Gwmi –query "SELECT * FROM Win32_Process"

Параметр –query может быть использован в сочетании с –namespace, –credential, –computername и многими другими параметрами, но не с – class, так как строка запроса должна уточнять, к какому классу обращен запрос.

В разделе справочника, посвященному Get-WmiObject, можно найти более подробную информацию об этих параметрах.

Использование параметра –query не имеет конкретных достоинств или недостатков по сравнению с другими формами командлета Get-WmiObject.

Системные свойства

Если вы попробуете передать WMI-объект по конвейеру командлету Format-List *, вы заметите несколько свойств, имена которых начинаются с двойного нижнего подчеркивания, например, __SERVER и __PATH:

Get-WmiObject Win32_Process | Format-List *

Эти системные свойства относятся непосредственно к WMI и могут содержать весьма ценную информацию. Например, вы уже видели, что свойство __SERVER содержит информацию об имени компьютера, с которого «пришел» WMI-объект. Это работает даже в том случае, если вы указали локальный хост или IP адрес для WMI-соединения:

Gwmi Win32_BIOS –computer localhost | Format-List *

__SERVER всегда содержит настоящее имя компьютера. Свойство __PATH также может быть полезным: оно содержит указатель, который может использоваться для уникальной ссылки на WMI-объект. Вы даже можете использовать его для повторного извлечения этого WMI_объекта.

ForEach-Object

- Так как WMI не является частью Windows PowerShell, оболочка содержит ограниченное количество командлетов для работы с WMI-объектами. В большинстве случаев основные командлеты оболочки могут манипулировать WMI-объектами так, как это требуется для выполнения ваших задач. Но иногда, однако, вы будете вынуждены отправлять WMI_объекты командлету ForEach-Object, чтобы выполнить определенные действия над каждым из них по очереди:

```
Gwmi Win32_Process | ForEach-Object { Something }
```

Зачастую приходится извлекать несколько WMI-объектов, после чего работать с каждым из них индивидуально. Так как WMI не является частью Windows PowerShell, оболочка содержит ограниченное количество командлетов для работы с WMI-объектами. В большинстве случаев основные командлеты оболочки могут манипулировать WMI-объектами так, как это требуется для выполнения ваших задач. Но иногда, однако, вы будете вынуждены отправлять WMI_объекты командлету ForEach-Object, чтобы выполнить определенные действия над каждым из них по очереди:

```
Gwmi Win32_Process | ForEach-Object { $_ }
```

В данном примере каждый WMI-объект отправляется в конвейер, что означает, что ForEach-Object в действительности не выполняет никакой важной функции. Однако в этот скриптовый блок вы можете поместить практически любую команду Windows PowerShell или даже несколько задач.

Администрирование Active Directory

Active Directory® - это технология, «внутри» которой многие администраторы проводят большую часть своего времени, выполняя повседневную административную работу – добавление новых пользователей, обновление объектов директории и.т.д. С внедрением в Windows Server® 2008 R2 командлетов, ориентированных на Active Directory, у администраторов появилась

возможность экономить массу времени и сил, используя Windows PowerShell® для автоматизации выполнения многих повторяющихся задач, которые раньше отнимали львиную долю времени. Автоматизация также позволяет повысить безопасность и стабильность работы, так как влияние человеческого фактора, а значит, и количество ошибок сводится к минимуму.

В Windows Server 2008 R2 впервые появился модуль Active Directory для Windows PowerShell. Этот модуль установлен на всех контроллерах домена и является частью Remote Server Administration Toolkit (RSAT) для Windows® 7. Модуль содержит командлеты, которые позволяют осуществлять управление Active Directory из командной строки. Также эти командлеты обеспечивают поддержку технологии, которая отвечает за работу новой графической консоли Active Directory Administrative Center. Эти новые командлеты взаимодействуют с веб-сервисом, который является частью Active Directory в Windows Server 2008 R2. Этот же веб-сервис можно добавить в контроллеры доменов Windows Server 2008 и Windows Server 2003, загрузив его с официального сайта Microsoft. Найти его можно здесь:

http://go.microsoft.com/fwlink/?LinkId=193581

Нет необходимости устанавливать этот веб-сервис на все контроллеры домена. Вполне достаточно того, чтобы он был на контроллере домена того сайта, на котором находится административный центр. Точно так же, нет нужды устанавливать сами командлеты на ваши контроллеры домена. Командлеты нужны только на том клиентском компьютере, на котором вы хотите их использовать.

Добавление модуля

Import-Module используется для загрузки модуля Active Directory в оболочку:

```
Import-Module ActiveDirectory
```

Remove-Module используется для удаления модуля, после завершения работы с ним. Обратите внимание, что модуль остается установленным до тех пор, пока вы либо не удалите его, либо не завершите текущую сессию работы в оболочке. Вы можете извлечь список командлетов, включенных в модуль, запустив команду:

```
Get-Command –module activedirectory
```

AD PSDrive

Модуль Active Directory включает провайдера PSDrive. По умолчанию при импортировании модуля появляется виртуальный диск под названием AD:, через который осуществляется вход в домен. Главная цель этого диска – обеспечение защиты выполняемых командлетов.

Каждый из командлетов Active Directory позволяет вам соединиться с удаленным доменом и получить разрешение на это. В справочнике по командлетам Active Directory перечислены соответствующие параметры. Например:

```
[-AuthType {Negotiate | Basic}] [-Credential <PSCredential>]
```

Указание полномочий для каждого командлета может оказаться утомительным. Здесь на помощь приходит провайдер PSDrive. Попробуйте запустить команду help

New-PSDrive, когда вы находитесь на диске C:, а затем переключитесь на диск AD: и запустите ее снова:

```
Cd c:
Help New-PSDrive
Cd AD:
Help New-PSDrive
```

Эти командлеты меняют параметры, когда вы переключаетесь на диск AD:. Эта альтернативная версия New-PSDrive разработана специально для обеспечения доступа к другим доменам, позволяя вам использовать альтернативный мандат. Этот мандат «запоминается» на тот период времени, пока PSDrive отображается.

Когда вы запускаете командлет Active Directory, PSDrive автоматически использует мандат и домен текущего диска. Таким образом, вместо того, чтобы указывать мандат каждый раз, когда вы запускаете командлет, вы можете просто привязать диск к определенному домену, переключиться на этот диск и запустить командлет оттуда. Командлет будет автоматически использовать тот мандат, который ассоциируется с этим диском. Чтобы использовать другой набор полномочий, подключите другой диск к требуемому домену, переключитесь на этот диск и запускайте командлет. Только командлеты Active Directory могут работать таким способом. Другие командлеты, поддерживающие параметр –credential, не перенимают полномочия PSDrive, который подключен к домену.

Если вы часто работаете с одним и тем же набором доменов, вы можете импортировать модуль Active Directory и привязывать диски к этим доменам каждый раз при запуске оболочки. Это можно сделать в профильном скрипте.

Ранние версии Windows

Не забывайте, что командлеты Active Directory предназначены для установки только на Windows Server 2008 R2 и Windows 7. Тем не менее, вы можете пользоваться этими командлетами и на более ранних версиях Windows, при условии, что они установлены хотя бы на одном компьютере из сетевого окружения.

Предположим, вы пользуетесь компьютером с running Windows XP, на котором установлена Windows PowerShell v2. Вы можете установить на домен контроллер домена Windows Server 2008 R2 и задействовать удаленную работу Windows PowerShell. Таким образом вы сможете установить командлеты Active Directory, а также контроллер домена, с которым взаимодействуют эти командлеты.

Далее, вы можете использовать неявный удаленный доступ с клиентского компьютера с Windows XP к командлетам, установленным на компьютере с Windows Server 2008 R2. Командлеты будут вести себя так, как будто они установлены на локальном компьютере с Windows XP, но выполняться в действительности они будут на удаленном компьютере. Эта техника будет рассмотрена позже, в модуле, посвященном удаленной работе с помощью Windows PowerShell.

Управление пользователями и группами

Управление пользователями и группами, несомненно, является одной из главных задач администратора в Active Directory. В рамках данного занятия вы узнаете, как осуществлять это управление с помощью командлетов. Опять же, техники, о которых вы узнаете здесь, могут применяться к другим командлетам, которые поставляются как Microsoft, так и сторонними поставщиками.

Работа с командлетами Active Directory, по сути, сводится к тому, чтобы вычислить, какие командлеты существуют и как пользоваться каждым из них. Для этого вы можете использовать команды Get-Command и Help соответственно.

Мы специально уделяем минимум времени изучению синтаксиса каждого из командлетов Active Directory. Ключевым навыком при работе с Windows PowerShell должно стать умение самостоятельно определять имена командлетов и искать их описание в справочнике. В этом и последующих модулях вы будете развивать этот навык.

Возьмите в привычку читать полное описание командлета в справочнике. Обращайте внимание на то, к каким параметрам происходит привязка конвейера по значению, а к каким – по имени свойства. Обладание этой информацией позволит вам быстрее научиться использовать короткие команды вместо длинных и сложных скриптов.

Фильтрация

По правде говоря, запрашивание всех объектов директории одновременно – неверный подход. Поступая таким образом, вы устанавливаете непомерную нагрузку на клиентский компьютер, на сеть, и особенно, на контроллеры домена. По этой причине большинство командлетов Active Directory, которые отвечают за извлечение объектов (например, командлеты с глагольной частью Get в названии), имеют обязательный параметр –filter. Вы можете установить значение * для данного параметра, но в большинстве случаев вам необходимо задать более точные критерии фильтрации, чтобы извлечь лишь те объекты, которые вам действительно нужны.

Параметр –filter командлетов Active Directory принимает критерии, оформленные в стиле Windows PowerShell:

```
Get-ADUser -Filter 'Name -like "*SvcAccount"'
Get-ADUser -Filter {Name -eq "GlenJohn"}
```

Вы можете найти и другие примеры в соответствующем разделе справочника. Большинство командлетов также могут ограничить свой радиус действия определенным подкаталогом директории, который будет называться поисковой базой. Найдите в справочнике информацию о том, как задать организационную единицу или другой контейнер в качестве поисковой базы.

Управление компьютерами и другими объектами AD

Active Directory хранит намного больше информации о доменных компьютерах, чем вы можете представить. Не забывайте отправлять объекты в командлеты Get-Member или Format-List *, чтобы просмотреть, что именно здесь содержится.

Также помните и том, что во избежание чрезмерных нагрузок командлеты Active Directory обычно по умолчанию извлекают не все свойства, содержащиеся в директории. Поэтому, вы всегда можете уточнить, какие именно атрибуты необходимо извлечь. О том, как сделать это – читайте в разделе справочника, посвященном определенному командлету.

Командлеты, рассмотренные в данном разделе, действуют во многом аналогично командлетам, рассмотренным в предыдущих разделах.

Скрипты PowerShell

Несмотря на то, что Windows PowerShell® - это отличная оболочка с интерфейсом командной строки, постоянный набор длинных команд вручную, во-первых, отнимает много времени, а во-вторых, не гарантирует отсутствие ошибок. Поэтому скрипты Windows PowerShell дают возможность запускать последовательную цепочку команд много раз, что избавляет от необходимости набирать эти команды вручную каждый раз. Также скрипты обеспечивают простой способ автоматизации длинных цепочек команд или задач, которые требуют логических решений, повторяющихся действий и.т.д.

Выполнение скрипта Windows PowerShell контролируется набором объектов, которые по умолчанию конфигурируются в безопасном состоянии. Когда вы начнете работать со скриптами, вам, возможно, придется изменить конфигурацию некоторых из этих объектов, исходя из особенностей выполнения ваших скриптов, а также политики безопасности вашего окружения.

Безопасность скриптов

Объекты безопасности скриптов для Windows PowerShell созданы специально для предотвращения возникновения проблем с безопасностью скриптов, которые могут возникнуть при использовании более старых технологий, таких как Microsoft® Visual Basic® Scripting Edition (VBScript). Windows PowerShell по умолчанию хранит их в безопасном состоянии, позволяя при необходимости менять это состояние в соответствии с вашими скриптовыми потребностями или для обеспечения выполнения различных задач, связанных с безопасностью.

IT-индустрия накопила обширный опыт, связанный с проблемами безопасности, которые могут создавать скриптовые языки. Главная проблема безопасности состоит в том, что скрипт представляет собой простейший способ внедрения вредоносного ПО или вируса в ваше окружение, в первую очередь, из-за того, что некоторые пользователи используют сценарии, не понимая, к чему именно это может привести или же вовсе не отдавая себе отчет в том, что они используют сценарии.

Объекты безопасности Windows PowerShell предназначены для того, чтобы создать «безопасное по умолчанию» окружение, в котором пользователи не смогут случайно использовать сценарии. Впрочем, нельзя сказать, что оболочка делает абсолютно невозможным использование сценариев пользователями. Просто по умолчанию это будет сделать сложно, по крайней мере, если пользователь не осознает, что именно он делает. Однако конфигурация настроек по умолчанию

часто оказывается неудобной для тех, кто часто пользуется сценариями, поэтому, оболочка предусматривает возможность изменения конфигурации. В результате такой реконфигурации вредоносным скриптам становится проще проникнуть в ваше окружение, поэтому, оболочка предлагает ряд специальных настроек, которые позволяют поддерживать баланс между удобством и безопасностью.

Основы безопасности

Оболочка предлагает три ключевых объекта безопасности, имеющих отношение к скриптам.

Основы безопасности

- Расширение файла .ps1, используемое для идентификации сценариев Windows PowerShell регистрируется в Microsoft Windows® как невыполнимый тип файла .
- Оболочка не ищет скриптовые файлы в текущем каталоге.
- Внутри оболочки есть скрипт под названием Execution Policy, который определяет перечень разрешенных сценариев. По умолчанию разрешенные сценарии отсутствуют.

Не стоит менять конфигурацию настроек безопасности оболочки до тех пор, пока вы не будете отдавать себе полный отчет о возможных последствиях.

• Расширение файла .ps1, используемое для идентификации сценариев Windows PowerShell регистрируется в Microsoft Windows® как невыполнимый тип файла. По умолчанию, двойной клик по файлу с расширением .ps1 не запускает скрипт (хотя он может открыться редакторе, например, Windows Notepad или Windows PowerShell ISE).

• Оболочка не ищет скриптовые файлы в текущем каталоге. Так, если вы наберете myscript в оболочке, она не откроет файл myscript.ps1, который может храниться в текущей директории. Вместо этого вам придется указать точный путь к файлу, например, ./myscript. Это позволяет предотвратить атаки, получившие название «команда хакеров», когда вместо внутренней команды выполняется сценарий, имеющий аналогичное название.

• Внутри оболочки есть скрипт под названием Execution Policy, который определяет перечень разрешенных сценариев. По умолчанию разрешенные сценарии отсутствуют.

Очень важно понимать, что не стоит менять конфигурацию настроек безопасности оболочки до тех пор, пока вы не будете отдавать себе полный отчет о возможных последствиях. Ваша организация должна адаптировать настройки безопасности Windows PowerShell, не нарушая оптимальный баланс между безопасностью и удобством.

Политики выполнения

Политики исполнения

- Всего существует пять настроек политики выполнения:
  - Restricted: сценарии не выполняются, за исключением некоторых скриптов с цифровой подписью от Microsoft
  - RemoteSigned: Разрешено выполнение, но удаленные скрипты должны быть подписаны
  - AllSigned: Разрешено выполнение, все скрипты должны быть подписаны
  - Unrestricted: Разрешено выполнение любых скриптов
  - Bypass: Полностью обходят политику безопасности
- Три метода изменить политику:
  - Group Policy
  - Administrators: командлет Set-ExecutionPolicy
  - Users: используя параметр powershell.exe

Политику выполнения внутри оболочки можно изменить тремя способами:

• Групповой политикой. Загружаемый административный шаблон групповой политики доступен на http://www.microsoft.com/downloads /details.aspx?displaylang=en&FamilyID=2917a564-dbbc-4da7-82c8-fe08b3ef4e6d; этот шаблон встроен в Windows Server 2008 R2. Любые настройки, заданные групповой политикой, доминируют над любыми локальными настройками.

• Администратором. Запуск командлета Set-ExecutionPolicy позволяет администратору изменить политику выполнения внутри оболочки на данном компьютере, при условии, что компьютер не находится под управлением настроек групповой политики, как было описано выше. Политика выполнения системы хранится в реестре HKEY_LOCAL_MACHINE, который обычно пишется только администратором локального компьютера.

• Пользователем. Пользователи могут работать в Windows PowerShell, используя powershell.exe и параметр командной строки, который меняет политику выполнения на время текущей сессии и доминирует над любыми локальными настройками.

Сам факт, что пользователям разрешается менять установленные администратором настройки политики выполнения, может показаться странным, но не забывайте, что политика выполнения создана с целью предотвращения случайного использования сценариев. Она не ставит своей целью запретить опытным пользователям работать со сценариями, но защищает от случайного запуска скриптов, который может произойти без ведома пользователя. Запуск Powershell.exe и использование параметра командной строки явно не относится к тем действиям, которые можно с легкостью осуществить по незнанию.

Всего существует пять настроек политики выполнения:

• Ограниченные (Restricted). Такие настройки используются в Windows PowerShell по умолчанию. При этом сценарии не выполняются, за исключением некоторых скриптов с цифровой подписью от Microsoft, которые содержат информацию о конфигурации оболочки по умолчанию.

• С подписью для удаленных файлов (RemoteSigned). Эти настройки разрешают выполнение всех сценариев. Однако удаленные файлы – загруженные из Интернета через Internet Explorer, переданные по электронной почте через Microsoft Office Outlook или по сети должны иметь цифровую подпись надежного поставщика.

• С подписью для всех файлов (AllSigned). Такие настройки позволяют выполнение любых сценариев при условии, что они имеют цифровую подпись надежного поставщика.

• Неограниченные (Unrestricted). Эти настройки позволяют выполнение любых сценариев.

• Обходные (Bypass). Такие настройки позволяют полностью обойти политику выполнения и разрешают выполнение всех сценариев. Предназначены они в первую очередь для разработчиков программного обеспечения, которые встраивают оболочку в другие приложения и планируют создание собственной модели безопасности, вместо используемой в оболочке по умолчанию.

Продвинутого пользователя установленная политика выполнения не остановит от использования сценариев в Windows PowerShell. Но это и не является целью политики выполнения. Точно так же политика выполнения не является формой защиты от вредоносных скриптов. Политика выполнения лишь позволяет предотвратить случайный запуск сценариев пользователями по незнанию, а также помогает определить, какие сценарии являются «надежными» и более безопасными.

Помните, что ни один пользователь не может использовать скрипты оболочки для выполнения задач, на которые он не имеет полномочий. Другими словами, пользователь, не являющийся администратором, не сможет запустить сценарий удаления всех пользователей из Active Directory, поскольку у него нет прав на это.

Возможность запустить сценарий не дает пользователю больше полномочий, чем он имеет.

Доверенные скрипты

Два вида настроек политики выполнения - RemoteSigned и AllSigned используют цифровые сертификаты и обладают уровнем доверия, позволяющим идентифицировать «надежные» сценарии.

Итак, что такое доверие?

Доверие начинается с корневого центра сертификации или root CA. Существуют публичные CA и частные, которые оказывают услуги за определенную плату. У многих компаний есть свои собственные CA. Когда вы говорите, что доверяете данному CA, это означает, что вы ознакомились с его политикой проверки подлинности людей или компаний, которым CA выдает цифровые сертификаты.

Доверенные скрипты

- Доверенный скрипт – подписанный цифровым сертификатом, которому доверяет Ваш компьютер
  - Требует Certification Authority
  - Требует digital certificate
  - Требует Цепочки доверия к доверенному CA
  - Политики RemoteSigned или AllSigned


```
<!-- SIG # Begin signature block -->
<!-- MIIXXAYJKoZIhvcNAQcCoIIXTTCCF0kCAQExCzAJBgUrDgMCGgUAMGkGCisGAQQB -->
<!-- gjcCAQSgWzBZMDQGCisGAQQBgjcCAR4wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR -->
<!-- AgEAAgEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQUVjLbCjAz9Hb6bPt9NeoVABfG -->
<!-- fvKgghIxMIIEYDCCA0ygAwIBAgIKLqsR3FD/XJ3LwDAJBgUrDgMCHQUAMHAxKzAp -->
<!-- BgNVBAsTIkNvcHlyaWdodCAoYykgMTk5NyBNaWNyb3NvZnQgQ29ycC4xHjAcBgNV -->
<!-- BAsTFU1pY3Jvc29mdCBDb3Jwb3JhdGlvbjEhMB8GA1UEAxMYTWljcm9zb2Z0IFJv -->
<!-- b3QgQXV0aG9yaXR5MB4XDTA3MDgyMjIyMzEwMloXDTEyMDgyNTA3MDAwMFoweTEL -->
<!-- MAkGA1UEBhMCVVMxEzARBgNVBAgTCldhc2hpbmd0b24xEDAOBgNVBAcTB1JlZG1v -->
<!-- bmQxHjAcBgNVBAoTFU1pY3Jvc29mdCBDb3Jwb3JhdGlvbjEjMCEGA1UEAxMaTWlj -->
<!-- cm9zb2Z0IENvZGUgU2lnbmluZyBQQ0EwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAw -->
<!-- ggEKAoIBAQC3eX3WXbNFOag0rDHa+SU1SXfA+x+ex0Vx79FG6NSMw2tMUmL0mQLD -->
<!-- TdhJbC8kPmW/ziO3C0i3f3XdRb2qjw5QxSUr8qDnDSMf0UEk+mKZzxlFpZNKH5nN -->
<!-- sy8iw0otfG/ZFR47jDkQOd29KfRmOy0BMv/+J0imtWwBh5z7urJjf4L5XKCBhIWO -->
<!-- sPK4lKPPOKZQhRcnh07dMPYAPfTG+T2BvobtbDmnLjT2tC6vCn1ikXhmnJhzDYav -->
<!-- 8sTzILlPEo1jyyzZMkUZ7rtKljtQUxjOZlF5qq2HyFY+n4JQiG4FsTXBeyS9UmY9 -->
```


Цифровой сертификат – это форма цифрового удостоверения личности. Цифровой сертификат удостоверяет подлинность или реальное существование человека или компании, но степень доверия этому удостоверению зависит от того, насколько вы доверяете данному центру сертификации. Другими словами, если компания связывается с вами в режиме онлайн и сообщает, что она является корпорацией Microsoft, вы должны проверить ее цифровой сертификат. Если сертификат подтверждает, что компания действительно является корпорацией Microsoft, вы должны проверить, кем был выдан этот сертификат, и доверяете ли вы этому центру сертификации.

Если вы уверены, что центр сертификации провел достаточно тщательное изучение для того, чтобы удостовериться в подлинности заявлений данной компании, значит, вы доверяете этому центру сертификации. Если этот сертификат использован для цифровой подписи сценария, вы также доверяете этому сценарию. Это не означает, что данный сценарий имеет какие-либо преимущества или не содержит вредоносных программ. Это означает, что вы просто можете идентифицировать того, кто подписал данный сценарий, а «надежная» подпись гарантирует, что скрипт не был изменен, поскольку имеет цифровую подпись.

Цифровая подпись производится с помощью ключа шифрования, который является частью цифрового сертификата. Подпись включает информацию о сертификате, в том числе, о подлинности владельца сертификата. Также подпись содержит зашифрованную информацию, которая может быть использована для выяснения содержимого скрипта. Если подпись и содержимое скрипта совпадают, вы знаете, кто подписал скрипт, и знаете, что скрипт не изменился с того времени, как был подписан. Опять же, это не означает, что скрипт является безопасным – просто в случае, если он окажется вредоносным, то вы сможете, используя информацию сертификата, выследить того, кто его подписал.

Если вы хотите посмотреть, как выглядит цифровая подпись, запустите в Windows PowerShell команду: type $pshome/types.ps1xml. Информация в самом конце файла и будет являться цифровой подписью. Проверить статус цифровой подписи файла можно с помощью команды get-authenticodesignature $pshome/types.ps1xml.

Прописывание скриптов

Чтобы подписать скрипт, вам нужно в первую очередь получить цифровой сертификат доверия. Получить его можно в публичных или частных центрах сертификации. Публичные центры сертификации обычно взимают ежегодную плату за сертификат. К частным центрам сертификации относятся центры, принадлежащие определенной компании – они обычно основываются на Windows Certificate Services или другом продукте управления сертификатами.

Вам понадобится сертификат класса 3, также известный как сертификат шифрования подписи. В публичных центрах сертификации они обычно стоят дороже, чем сертификаты класса 1, необходимые для кодирования или подписи электронной почты, а также требуют более тщательной проверки для идентификации личности. Многие центры сертификации предлагают другие варианты сертификатов для шифрования подписи, но вам нужен сертификат Microsoft AuthentiCode.

Также вы можете сгенерировать сертификат локально с помощью инструмента MakeCert.exe, который является частью Windows Platform SDK. Чтобы узнать о MakeCert.exe и о том, как им пользоваться, запустите в Windows PowerShell команду:

```
Help about_signing
```

Локальный сертификат может использоваться только на данном локальном компьютере. Скрипты, подписанные с помощью этого сертификата, могут выполняться только на этом компьютере.

После того, как вы установили сертификат, вам необходимо запустить командлет Set-AuthenticodeSignature, чтобы прикрепить подпись к сценарию. Раздел справочника, посвященный этому командлету, содержит описание деталей его использования и примеры.

Прочитайте справочные материалы по Set-AuthenticodeSignature и изучите некоторые его опции. Придумайте несколько примеров того, как можно использовать его для создания подписи к скрипту.

Простейшие скрипты

Скрипт – это не что иное, как текстовый файл, содержащий одну или несколько команд, которые выполняются последовательно. Проще всего создать базовый скрипт посредством копирования и вставки команд из командной строки в текстовый файл.

Скрипты – это текстовые файлы с расширением .ps1. Эти файлы содержат одну или несколько команд, которые оболочка должна выполнять в определенном порядке. Вы можете редактировать скрипты в Windows Notepad, но Windows PowerShell ISE предлагает более удобный способ редактирования. В нем вы можете печатать команды в интерактивном режиме, немедленно просматривать результаты и вставлять эти результаты в скрипт для долгосрочного использования. Или вы можете набирать команды непосредственно в скрипт, выделять необходимые и затем с помощью клавиши F8 требовать у оболочки выполнения выделенных задач. Если результат вас удовлетворил – просто сохраняете скрипт.

Сторонние поставщики программного обеспечения часто предлагают различные коммерческие и бесплатные редакторы, которые, по их словам, обеспечивают большее удобство при написании сценариев. Честно говоря, существует очень немного различий между тем, что вы можете сделать в сценарии и тем, что вы можете сделать в командной строке. Команды точно так же работают в сценарии, что означает, что вы можете просто вставлять в текстовый файл команды, скопированные из командной строки.

Написание сценария во многом упрощается, когда вы собираете его из уже написанных и проверенных команд. Для этого существует три техники:

• Запускать команды в окне консоли, после чего копировать и вставлять проверенные команды в сценарий.

• Запускать команды в Windows PowerShell ISE, после чего копировать и вставлять проверенные команды в сценарий.

• Писать команды непосредственно в сценарии, в Windows PowerShell ISE и проверять их путем выделения и нажатия F8. если команда работает, значит сценарий уже готов и его можно сохранять.

Сторонние скриптовые редакторы обычно также предлагают различные способы быстрого переноса команд из командной строки в файл сценария. Обратите внимание, что возможен и обратный процесс – вы можете скопировать команду из сценария и вставить ее в командную строку для запуска. Конечно, в Windows PowerShell ISE можно сделать проще – выделить необходимую команду и нажать F8 для ее быстрого запуска.

Однако скрипты позволяют также добавлять к командам описание. По мере добавления команд в файл сценария, вы можете также добавлять комментарии, касаемые того, для чего та или иная команда предназначена. Таким образом, если кто-то другой (или даже вы сами через несколько месяцев) будет читать этот документ, намного меньше времени уйдет на выяснение того, какую роль играет данная команда. Чтобы создать комментарий, просто наберите знак решетки, а затем ваш комментарий:

```
# Retrieve WMI object for the operating system
Get-WmiObject Win32_OperatingSystem
```

Читаемость скрипта

Команды в командной строке не всегда легко прочитать. Например, данная команда может показаться достаточно сложной:

```
Gwmi win32_operatingsystem –comp (gc names.txt) | % { $_.Reboot() }
```

Но при наборе команды в командной строке, вас не должна волновать их читаемость. Ваша цель – максимально сократить длину команды. Именно для этого используются псевдонимы и сокращенные имена параметров. Сценарии же, в свою очередь, предназначены для долгосрочного использования, и простота их использования в дальнейшем будет зависеть от простоты прочтения. Поэтому, при их написании следуйте некоторым рекомендациям:

- Всегда используйте полные имена командлетов, а не псевдонимы;
- Всегда используйте полные, а не сокращенные имена параметров;
- Старайтесь разбивать длинные строчки на несколько коротких посредством нажатия Return после символа конвейера.

Последняя рекомендация может оказаться особенно полезной для улучшения читаемости сценария. Например, вместо команды:

```
Ps | sort vm –desc | select –fir 10 | export-csv procs.csv
```

Можно написать скрипт:

```
Get-Process |
Sort-Object VM –descending |
Select-Object –first 10 |
Export-CSV procs.csv
```

Такое структурное разделение первой, второй, третьей и четвертой строки позволяет визуально связать каждую из них с командлетом, расположенным в начале. Некоторые пользователи оболочки используют знак перехода, который используется в качестве знака продолжения строки:

```
Get-Process | `
Sort-Object VM –descending | `
Select-Object –first 10 | `
Export-CSV procs.csv `
```

Эта техника работает, но осложняет чтение сценария, так как знак перехода – обратный апостроф – читается не во всех шрифтах.

Если вы работаете, в основном, в Windows PowerShell ISE или аналогичном стороннем редакторе, вы можете перемещаться из командной строки в файл сценария намного эффективнее. Поскольку вы уже пишете скрипт, нет необходимости копировать и вставлять команды – просто сохраняйте файл.

Также ISE поддерживает разнообразные наборы символов, включая двухбайтовый (DBCS), который необходим для некоторых языков. Поэтому, для пользователей, работающих на этих языках, ISE намного предпочтительнее окна консоли. ISE выполняет команды так же, как и консоль, но командлеты, такие как Read- Host, генерируют диалоговое окно вместо командной строки. Если вы планируете написать скрипт для выполнения определенной задачи, то после написания в ISE, следует проверить его в окне консоли, чтобы убедиться, что все работает так, как нужно.

Если вы уже научились использовать окно консоли, переходите на использование Windows PowerShell ISE. Он обладает очень простым графическим пользовательским интерфейсом, и на его изучение понадобится минимум времени.

Параметризованные скрипты

Иногда вам может понадобиться сценарий, содержащий переменную информацию, например, возможно, вам потребуется менять имя компьютера или имя файла. Конечно, вы можете редактировать скрипт каждый раз при запуске. Но параметризация скрипта позволить вводить переменную информацию без необходимости редактирования в дальнейшем.

Когда скрипты содержат жестко запрограммированные значения, такие как имена компьютеров, IP адреса, имена файлов, их становится сложнее использовать повторно, к тому же, они сложнее в использовании для начинающих пользователей. Необходимость редактирования сценария каждый раз при запуске команды нередко приводит к ошибкам. Если редактирование производит неопытный пользователь, он может внести поправки не туда, куда нужно, что вызовет дополнительную путаницу. Использование параметров позволяет встраивать переменные в сценарий, не нарушая его структуру. При этом отпадает необходимость в его редактировании при повторном использовании.

Объявление параметров

Параметры переменной указываются в блоке Param() в начале сценария. Этот блок должен предшествовать любой команде в сценарии, несмотря на то, что перед ним могут идти комментарии, объясняющие, для чего он предназначен.

Иногда бывает очень полезно прописывать комментарии, в которых объясняется, что представляет собой каждый параметр, и какое значение от него ожидается.

После того, как параметры указаны, они могут использоваться в качестве переменной вместо жестко запрограммированного значения. Имя параметра обычно начинается со знака $ и может включать в себя буквы, цифры и символ нижнего подчеркивания. Также имя параметра может содержать пробелы, если вы заключаете его в фигурные скобки:

${My Parameter}

Однако данная техника затрудняет чтение сценария в дальнейшем, поэтому, Windows PowerShell не рекомендует использовать пробелы. Например:

```
Param (
$computerName,
$userName
)
```

Обратите внимание, что имена параметров в списке разделены запятыми, а читаемость сценария можно улучшить, поместив каждое имя в отдельную строчку. Но при этом не забывайте ставить запятую после каждого имени параметра, кроме первого. Также обратите внимание, что в блоке Param() используются круглые скобки, а не фигурные. Для каждого параметра можно задать значение по умолчанию:

```
Param (
$computerName = 'localhost',
$userName
)
```

Таким образом, если сценарий запускается без указанного значения для параметра $computerName, будет использоваться значение по умолчанию, а именно ‘localhost’. Если значения по умолчанию не предусмотрено, сценарий запускается с нулевым значением параметра.

Использование параметров

Когда вы запускаете параметризированный сценарий, вы можете разместить параметры либо позиционно, либо используя имена. Если скрипт под названием MyScript.ps1 содержит блок параметров

```
Param (
$computerName,
$userName
)
```

То вы можете запустить команду:

```
./myscript 'localhost' 'Administrator'
```

или такую (более легкую для чтения):

```
./myscript –computerName 'localhost' –username 'Administrator'
```

Если вы используете технику позиционирования, разделяйте значения параметров пробелами, а не запятыми. Если вы используете имена, то параметры могут быть расположены в любом порядке, а их имена – как полными, так и урезанными.

Запрос параметра у пользователя

Запрос параметра у пользователя

- FИногда вам может потребоваться запрос данных у пользователя, в случае, если вам не хватает значений некоторых параметров. Для решения этой задачи используется командлет Read-Host. Например, чтобы запросить у пользователя параметр, значение которого у вас отсутствует, используйте Read-Host как часть значения по умолчанию:

```
Param (
        $computerName = $(Read-Host 'Enter computer name'),
        $userName = $(Read-Host 'Enter user name')
        )
```

Иногда вам может потребоваться запрос данных у пользователя, в случае, если вам не хватает значений некоторых параметров. Для решения этой задачи используется командлет Read-Host. Например, чтобы запросить у пользователя параметр, значение которого у вас отсутствует, используйте Read-Host как часть значения по умолчанию:

```
Param (
$computerName = $(Read-Host 'Enter computer name'),
$userName = $(Read-Host 'Enter user name')
)
```

Больше информации о командлете Read-Host вы можете узнать из соответствующего раздела справочника.

Фоновые задания и задания удаленного типа

Фоновые задания и задания удаленного типа являются двумя ключевыми чертами, представленными Windows PowerShell ® v2.0. Каждая из этих черт придает более высокий уровень зрелости Windows PowerShell, помогая вам выполнять более сложные задачи и позволяя расширять административные достижения.

Interactive vs. Background

- При выполнении некоторых длительных команд в оболочке, вы можете предпочесть, чтобы они происходили в фоновом режиме, так что вы могли бы продолжать использовать оболочку для других задач.
- Одним из способов достижения этой цели могло бы быть простое открытие второго окна оболочки, но это связано с использованием немного большего объема памяти и процессорного времени на вашем компьютере, и дает вам еще одно окно в управлении. Другим вариантом является выполнение команд в качестве фонового задания.

Фоновые задания

При выполнении некоторых длительных команд в оболочке, вы можете предпочесть, чтобы они происходили в фоновом режиме, так что вы могли бы продолжать использовать оболочку для других задач.

Одним из способов достижения этой цели могло бы быть простое открытие второго окна оболочки, но это связано с использованием немного большего объема памяти и процессорного времени на вашем компьютере, и дает вам еще одно окно в управлении. Другим вариантом является выполнение команд в качестве фонового задания.

Как правило, команды Windows PowerShell завершаются в интерактивном режиме, или синхронно. Если вы выполняете длительный команду, вы должны ждать его завершения перед запуском других команд в оболочке.

Фоновые задания просто перемещает обработку команд в фоновом режиме. Команда продолжает выполнить, но вы можете начать использовать оболочку для других задач. Windows PowerShell предоставляет командлеты для проверки состояния фонового задания.

Когда команда работает в фоновом режиме, ее результаты сохраняются в памяти как часть работы объекта. Вы можете получить результаты выполненных работ и работать с ними в канале.

Фоновый WMI

Фоновый WMI

- Командлет Get-WmiObject…
  - часто занимает много времени для запуска, особенно, когда вы извлекаете информацию из нескольких удаленных компьютеров.
  - поддерживает -AsJob параметр, который приводит к выполнению его в фоновом режиме.

Get-WmiObject –computer (gc names.txt) –class CIM_DataFile –AsJob

Get-WmiObject является командой, которая часто занимает много времени для запуска, особенно, когда вы извлекаете информацию из нескольких удаленных компьютеров. Командлет поддерживает -AsJob параметр, который проходит в качестве команды фонового задания Вы можете использовать Get-WmiObject в точности так, как вы это обычно делаете, просто указывать -AsJob в качестве дополнительного параметра для создания фоновых заданий:

Get-WmiObject –computer (gc names.txt) –class CIM_DataFile –AsJob.

Вместо того, чтобы ждать завершения WMI, команда будет немедленно размещать объект работы в конвейер и начинать выполнение работы в фоновом режиме.

Управление задачами

После того, как задание запущено, вы можете контролировать ее ход, удалять полную работу, или получать результаты завершенной работы. Оболочка предоставляет несколько командлетов:

Управление заданиями

- Четыре важных командлета для управления задагиями
  - Get-Job
  - Remove-Job
  - Wait-Job
  - Stop-Job

• Get-Job будет получать все доступные задания. Укажите идентификатор или имя-параметра для получения конкретного задания.

Передайте эти командные данные Get-Member или Format-List *, чтобы увидеть свойства задания.

• Remove-Job удаляет работу. Укажите id или укажите параметр для удаления задания в этом командлете.

• Wait-Job сделает паузу и дождется работы, чтобы закончить. Укажите id или назвате параметра, чтобы указать, какое задание в этот командлете стоит ждать для выполнения.

Этот командлет может быть полезным в сценариях, когда необходимо выполнение скрипта, чтобы приостановить, пока данное фоновое задание будет завершено.

• Stop-Job будет останавливать выполнение задания, в том числе задания, которые зависли и не будут завершены самостоятельно.

Укажите либо id или назовите параметр, чтобы указать, какое задание должно быть остановлено в этом командлете.

Вывод задания

Когда работа завершена, выводы ее команды будут сохраняться в памяти как часть рабочего объекта. Вы можете использовать Receive-Job для получения этих результатов с кэш-памяти.

Receive-Job помещает объекты в конвейер. Предположим, вы начинаете работу с помощью следующей команды:

```
Get-WmiObject Win32_Service -comp Server2 -AsJob
```

И в результате работы выглядит следующим образом:

```
Id    Name    State        HasMoreData.
2     Job2    Completed    True
```

Тогда вы сможете получать результаты в конвейер, фильтровать их и сортировать их, а именно:

```
Receive-Job  –id 2 | Sort State | Where { $_.State –ne "Stopped" }
```

Вы также можете присвоить результаты Receive-Job переменной:

$results = Receive-Job –id 2

Обратите внимание, что в предыдущем примере на самом деле довольно плохая практика. Он запрашивает от WMI больше информации, чем это действительно необходимо, а затем фильтрует эту информацию локально. Лучшей техникой является:

Gwmi Win2_Service –comp Server2 –filter "State <> 'Stopped'" –asjob

Этот пример выполняет фильтрацию на источнике, следуя лучшей практике левого фильтра, имея в виду размещение критериев фильтрации так далеко влево от командной строки, как это возможно.

Job Output

- Когда работа завершена, выводы ее команды будут сохраняться в памяти как часть рабочего объекта. Вы можете использовать Receive-Job для получения этих результатов с кэш-памяти.
- Receive-Job помещает объекты в конвейер. Предположим, вы начинаете работу с помощью следующей команды:

Receive-Job –id 2 | Sort State | Where { $_.State –ne "Stopped" }

- Вы также можете присвоить результаты Receive-Job переменной .

$results = Receive-Job –id 2

- Когда вы получаете итог работы, оболочка удаляет итог из кэш-памяти задания.
- Указывая параметр сохранения, однако, вы можете поручить оболочке оттавить результаты в кэше задания используя параметр -keep

Когда вы получаете итог работы, оболочка удаляет итог из кэш-памяти задания. Это означает, что вы можете использовать Receive-Job один раз для любой работы, а после этого нельзя будет получить никакие другие результаты.

Указывая параметр сохранения, однако, вы можете поручить оболочке оттавить результаты в кэше задания используя параметр -keep.

Receive-Job –id 4 –keep

Еще один способ сохранить результаты выполнения задания, это экспортировать их в XML-файл:

Receive-Job –id 7 | Export-CliXML jobresults.xml

Вы можете затем использовать Import-CliXML, чтобы повторно импортировать результаты работы в оболочке.

Оболочка сохраняет информацию о in-prosess и завершенных работах, даже после того, как вы получили результаты этой работы.

Вы можете очистить список заданий, удалив задания, которые были завершены. Отметим, что таким образом устраняют любые кэшированные выводы, вы не сможете получить результаты выполнения задания после удаления работы.

Укажите задание либо по ID:

```
Remove-Job -id 1
```

Либо по названию:

```
Remove-Job –name Job1
```

Или в конвейере камендлета:

```
Get-Job | Where { $_.Name –like "AD*" } | Remove-Job
```

Другие команды как фоновые задания

Фоновые задания не ограничиваются WMI. Другие командлеты поддерживают параметр AsJob, и вы можете использовать любые из этих команд, чтобы начать новую фоновую работу.

Commands as Jobs

- Фоновые задания не ограничиваются WMI. Другие командлеты поддерживают параметр AsJob, и вы можете использовать любые из этих команд, чтобы начать новую фоновую работу
- любая команда может стать фоновой работой с помощью Start-Job командлета. Is

Start-Job -scriptblock {Get-Process}

Кроме того, любая команда может стать фоновой работой с помощью Start-Job командлета. Просто предоставьте блок сценария (то есть, одну или несколько команд). Можно дополнительно предоставить альтернативные учетные данные для работы. Start-Job всегда начинает работу, поэтому нет AsJob параметров. Однако есть параметр Name, который позволяет указать имя задания. Это имя будет использоваться вместо автоматически генерируемого имени задания, такого как job1 или Job2. Указывая собственное название задания легче сослаться на это задание с помощью параметра -Name командлетов, таких как Get-Job, Wait-Job, Remove-Job, Receive-Job, и Stop-Job.

Вы также можете заметить, что Invoke-Command имеет -AsJob параметр. Тем не менее, не используйте это, чтобы начать местную работу, так как он предназначен для того, чтобы начать фоновые задачи, общаться с удаленными компьютерами.

Этот командлет будет рассмотрен далее.

Удаленное исполнение

В то время как Windows PowerShell командлеты, такие как Get-WmiObject или Get-Service уже реализуют свои возможности удаленного подключения, Windows PowerShell также предоставляет свои особенности удаленного взаимодействия, которые позволяют практически любую команду или сценарий запускать на одном или нескольких удаленных компьютерах. Эти функции помогают расширить административный доступ, охватив любой компьютер сети под управлением Windows PowerShell v2, и они позволяют вам управлять несколькими компьютерами с помощью одной команды.

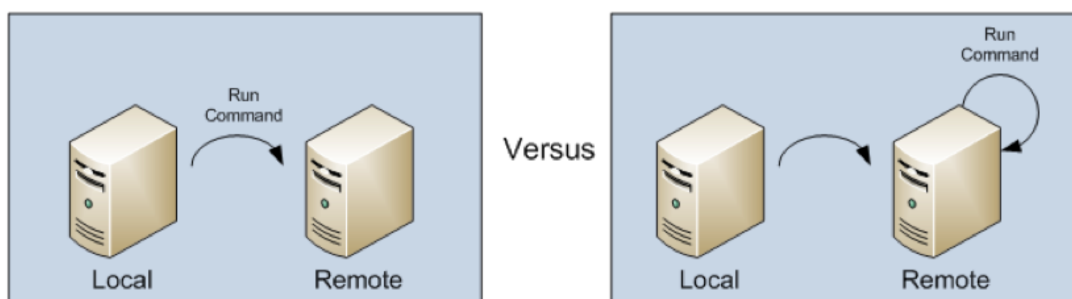
Вход в консоль сервера либо физически, либо с помощью Remote Desktop connection не является хорошей практикой управления. Иногда это неизбежно, но так как сервер генерирует и поддерживает графическую среды пользователя, например, рабочий стол и приложения, которые работают на нем, то использование серверного диска, процессора и памяти является неэффективным.

Microsoft постепенно добавлены возможности удаленного управления для многих административных инструментов и консолей, которые администраторы используют наиболее, в том числе Active Directory Users and Computers, Server Manager и так далее. Тем не менее, это, как правило, графические инструменты, то есть просты в использовании, но они не могут быть легко автоматизированы.

С дистанционныме возможностями Windows PowerShell в управлении, вы можете запустить почти любую ОС Windows PowerShell команду и получить ее выполнение на одном или нескольких удаленных компьютерах. Эта возможность позволяет управлять удаленными компьютерами более легко и с меньшими затратами, чем консольной сессией или соединением Remote Desktop.

Удаленное исполнение

- Существует несколько форм удаленного исполнения:
  - командлеты, которые имеют computerName parameter и не используют Windows PowerShell Remoting .
  - Некоторые командлеты предназначены созданы для связи с удаленным компьютером. Active Directory командлеты, например .
  - Windows PowerShell Remoting способен делать удаленное подключение к одному или нескольким компьютерам и запускать команды, которые находятся на этих компьютерах .



Термин удаленное администрирование означает много разных вещей в Windows PowerShell.

• За исключением некоторых специфических команд вы узнаете на этом уроке, командлеты, которые имеют computerName parameter и не используют Windows PowerShell Remoting. Такие командлеты, как Get-WmiObject или Get-Service, реализуют собственное удаленное подключение. Get-WmiObject, например, использует Remote Procedure Call (RPC) протокол, присущий технологии WMI.

• Некоторые командлеты предназначены созданы для связи с удаленным компьютером. Active Directory командлеты, например, знают, как подключиться к контроллеру домена, чтобы выполнить их работу. Microsoft Exchange Server командлеты знают, как подключиться к компьютеру Exchange Server, даже если командлеты будут работать на Windows ® 7 клиентском компьютере. Эти команды используют коммуникационные протоколы, связанные с их конкретными технологиями и не используют Windows PowerShell Remoting.

• Windows PowerShell Remoting, о котором вы узнаете в этом уроке, способен делать удаленное подключение к одному или нескольким компьютерам и запускать команды, которые находятся на этих компьютерах. Windows PowerShell Remoting может быть использован с любым командлетом, который установлен на удаленных компьютерах и не полагается на командлеты, имеющие computerName parameter или любые другие возможности удаленного подключения.

Вы часто можете комбинировать эти три способа. Например, предположим, что вы хотите получить WMI информацию с удаленного компьютера, вы можете использовать любой из этих методов:

• Используйте Get-WmiObject и его -computerName параметр. Командлет запускается на вашем локальном компьютере, и использует RPC, для подключения к службе WMI на удаленном компьютере. Удаленный сервис обрабатывает запрос WMI и возвращает результаты в компьютер.

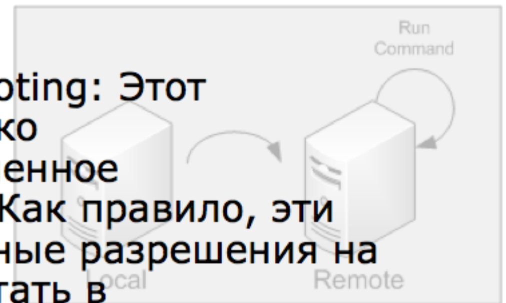
• Использование Windows PowerShell Remoting чтобы установить HTTP соединение с удаленным компьютером. Используя эту связь, укажите на удаленном компьютере запустить Get-WmiObject. Командлет запускается на удаленном компьютере и сообщается с WMI для выполнения запроса. WMI результаты передаются обратно на компьютер по HTTP-соединению.

Вы можете видеть, что есть некоторые тонкие различия между методами. Другой протокол используется для сообщений, которые могут повлиять на вашу способность работать через брандмауэры. Обработка происходит в различных местах для каждой ситуации. Во втором примере, большая часть обработки происходит на удаленном компьютере, в то время как в первом, нагрузка делится между двумя компьютерами.

Идея состоит в том, что различные формы удаленного администрирования имеют применение в различных сценариях. Ни одна из форм не лучше по своей сути, чем другая, хотя в определенных ситуациях, например, когда вам нужно распространять обработку или когда вам нужно общаться через брандмауэр, одна из форм удаленного администрирования может быть более подходящей.

Windows PowerShell Remoting

- 1-to-1 Remoting: В этом случае, вы подключаетесь к одному удаленному компьютеру и запускаете команды оболочки на нем, точно так, как если бы вы зашли в консоль и открыли Windows PowerShell окно.
- 1-to-Many Remoting, или Fan-Out Remoting: В этом случае, вы даете команду, которая будет выполнена на одном или нескольких удаленных компьютерах одновременно. Вы не работаете с каждым удаленным компьютером интерактивно, а, скорее, ваши команды выдаются и выполняются в пакетном режиме, и результаты возвращаются на компьютер для вашего пользования.
- Many-to-1 Remoting или Fan-In Remoting: Этот сценарий предполагает, что несколько администраторов осуществляют удаленное подключение к одному компьютеру. Как правило, эти администраторы будут иметь различные разрешения на удаленном компьютере и могут работать в ограниченном пространстве внутри оболочки.



Целью Windows PowerShell Remoting является подключение к одному или нескольким удаленным компьютерам, для запуска команд на этих компьютерах, и доведение результатов обратно на компьютер. Это позволяет одному администратору управлять компьютерами в сети с клиентского компьютера без физической необходимости посещать каждый компьютер. Основной целью Remoting Windows PowerShell является обеспечение пакета управления, который позволяет выполнять команды на весь набор удаленных компьютеров одновременно, в то время как старые технологии, такие как удаленный рабочий стол не позволяют это сделать.

Windows PowerShell Remoting (или просто Remoting для краткости) не использует те же технологии соединения и протоколы как более старые дистанционных методы введения, такие как Remote Desktop, и Remoting, как правило, меньше весит, что означает, что способ требует меньше накладных расходов, чем некоторые из тех технологий, которые старше. Это не означает, что эти старые технологии не полезны и не имеют места быть. Это означает лишь то, что Remoting может быть более эффективным и действенным в некоторых сценариях способом.

Существуют три основных способа использования Remoting:

• 1-to-1 Remoting: В этом случае, вы подключаетесь к одному удаленному компьютеру и запускаете команды оболочки на нем, точно так, как если бы вы зашли в консоль и открыли Windows PowerShell окно.

• 1-to-Many Remoting, или Fan-Out Remoting: В этом случае, вы даете команду, которая будет выполнена на одном или нескольких удаленных компьютерах одновременно. Вы не работаете с каждым удаленным компьютером интерактивно, а, скорее, ваши команды выдаются и выполняются в пакетном режиме, и результаты возвращаются на компьютер для вашего пользования.

• Many-to-1 Remoting или Fan-In Remoting: Этот сценарий предполагает, что несколько администраторов осуществляют удаленное подключение к одному компьютеру. Как правило, эти администраторы будут иметь различные разрешения на удаленном компьютере и могут работать в ограниченном пространстве внутри оболочки. Этот сценарий обычно требует разработки пользовательских ограниченной пространства и далее в этом курсе рассматриваться не будет.

Remoting нуждается в том, чтобы Windows PowerShell v2, наряду с поддержкой таких технологий, как Windows, Remote Management (WinRM) и. NET Framework v2.0 или выше, могла быть установлена как на вашем компьютере и на любом удаленном компьютере, к которому требуется подключиться. В целях безопасности Remoting должны быть включены, прежде чем вы начнете пользование. Вы узнаете, как включить Remoting позже в этом уроке.

WinRM

Вместо того, чтобы использовать старые протоколы, такие как RPC для общения , Remoting использует Windows Remote Management или WinRM. WinRM представляет собой реализацию Microsoft веб-служб для управления, или WS-MAN, набор протоколов, которые получили широкое распространение в различных операционных системах. Как следует из названия, WS-MAN—and WinRM использует протоколы на базе Web. Преимущество этих протоколов в том, что они используют один определенный порт, что дает им возможность легче проходить через брандмауэры, чем старые протоколы, которые редко выбирали порт.

WinRM сообщается через Hypertext Transport Protocol, или HTTP. Обратите внимание, что версия WinRM включена и используется Windows PowerShell v2 не по стандартными портами HTTP 80 или 443. По умолчанию, текущая версия (v2.0) из WinRM использует TCP порт 5985 для входящих незашифрованных соединений. Этот порт назначения настраивается, хотя и рекомендуется оставить его значения по умолчанию, так как Windows PowerShell Remoting в связанных командлетах учитывает тот же порт по умолчанию.

Примечание: Старые версии WinRM использовали TCP порт 80 (или 443 для зашифрованного соединения) по умолчанию. Это не верно с WinRM v2. Тем не менее, WinRM v2 может быть поручено работать в режиме совместимости, где он также прислушивается к портам 80 и / или 443 для поддержки старых приложений, которые знаю только как отправить WinRM запросы к этим портам. Мы не будем рассматривать эту конфигурацию в этом курсе.

WinRM может использовать Transport Layer Security (TLS, иногда неточно называется SSL) для шифрования данных, передаваемых по сети. Отдельные приложения, которые используют WinRM, такие как Windows, PowerShell, могут

также применять свои собственные методы шифрования данных, которые передаются в службу WinRM.

WinRM поддерживает аутентификацию и, по умолчанию, использует собственный протокол Kerberos Active Directory в среде домена. Kerberos не передает учетные данные по сети и поддерживает взаимную аутентификацию чтобы убедиться, что вы действительно подключаетесь к удаленному компьютеру, который вы указали.

В целях безопасности, WinRM по умолчанию отключен и должен быть включен. Он может быть включен локально на одном компьютере или он может быть включен и настроен с помощью объекта групповой политики (GPO).

WinRM запускается как сервис. Когда эта функция включена, и необходимые порты брандмауэра открыты, WinRM служба прислушивается к входящим запросам WinRM на назначенный порт. Когда запрос получен, WinRM рассматривает запрос, чтобы определить приложения, для которого и делался запрос.

Заявки должны регистрироваться в конечном итоге с WinRM. Регистрация сообщает WinRM, что приложение доступно и способно обрабатывать входящие запросы WinRM. Эта регистрация является тем, что позволяет WinRM найти применение при входящем получении запроса для этого приложения.

Когда входящий запрос получен, WinRM передает запрос назначенному приложению.

Приложение делает то, что оно должен делать, а затем передает любые результаты обратно в WinRM. WinRM передает эти результаты обратно на компьютер и приложение, которое послало первоначальный запрос WinRM.

Windows PowerShell , по умолчанию, не регистрирует себя в качестве конечной точки, опять же, для безопасности. Вы должны разрешить оболочке зарегистрировать себя в качестве конечной точки с WinRM. Это может быть выполнено локально на одном компьютере или с помощью в Group Policy в области домена. Вы должны также убедиться, что все необходимые конфигурации брандмауэра были изменены, чтобы разрешить WinRM трафик.

Вам нужно всего лишь включить WinRM, зарегистрироваться в конечной точки и изменить настройки брандмауэра на компьютерах, которые получат WinRM запросы. Любой компьютер с установленным Windows PowerShell v2 может отправить WinRM запросы.

Есть два способа настройки Windows PowerShell и WinRM. Первый и, возможно, самый простой, заключается в запуске Enable-PSRemoting командлета.

Примечание: Избегайте запуска Set-WSManQuickConfig. Этот командлет настраивает часть Remoting, но далеко не все. На самом деле, Enable-PSRemoting запускает Set-WSManQuickConfig внутренне, а затем выполняет другие действия, необходимые для регистрации Windows PowerShell, как WinRM конечной точки.

Вы должны войти в систему как администратор (и, если контроль учетных записей включен, необходимо запустить оболочку от имени администратора) для успешного запуска Enable-PSRemoting командлета:

• Включить WinRM.

• Начать WinRM обслуживание.

• Установить WinRM автоматический запуск службы.

• Изменить конфигурацию брандмауэра Windows, чтобы разрешить входящие соединения WinRM.

• Зарегистрировать обе 64 - и 32-разрядные версии Windows PowerShell, как WinRM конечные точки.

• Так как оказываемый уровень влияния высок, то вам будет предложено подтвердить свой выбор, чтобы командлет мог быть запущен.

1-to-1 shell

1-to-1 Shell

- 1-to-1 shell позволяет быстро получить удаленную командную строку
  - Аналогично Telnet, SSH, или PSExec \\computerName cmd

```
Enter-PSSession –comp server-r2
```

- Для выхода используйте

```
Exit-PSSession
```

1-to-1 shell позволяет быстро получить удаленную командную строку, похожую на SSH или Telnet в операционной системе Unix. 1-to-1 оболочка также похожа по функциям на то, как работает PSExec.

После того как вы открыли 1-to-1 удаленное соединение оболочки, вы можете эффективно использовать удаленно компьютерные Windows PowerShell сессии, как если бы вы физически, сидя перед компьютером, открыли Windows PowerShell окно.

Используйте Enter-PSSession для установления удаленного соединения оболочки. Эта команда принимает-параметр computerName, который принимает имя или IP-адрес компьютера, к которому вы хотите подключиться. Другие параметры

позволяют указать альтернативные порты TCP, указать альтернативные учетные данные пользователя, и так далее. Прочитайте помощь для командлета, чтобы узнать об этих и других возможностях и особенностях.

После того, как удаленная оболочка стала активной, ваши строки Windows PowerShell будут изменяться, отражая имя компьютера, то, что вы подключены к, а также текущий путь:

```
PS C:\> enter-pssession -comp server-r2
[server-r2]: PS C:\Users\Administrator\Documents>
```

Когда вы закончите использование удаленного подключения, запустите Exit-PSSession, чтобы вернуться к местной командной строке:

```
[server-r2]: PS C:\Users\Administrator\Documents> exit-pssession
PS C:\>
```

При создании удаленного соединения оболочки, помните, что профиль скриптов не выполняется, даже если сценарий профиля существует.

При использовании удаленного соединения оболочки, у вас есть только доступ к командам, которые установлены на удаленном компьютере и тем, которые вы загрузили в оболочку с помощью Add-PSSnapin или Import-Module. Доступные команды на удаленном компьютере могут отличаться от имеющихся на локальном компьютере.

Будьте осторожны: когда вы находитесь в работе с удаленной оболочкой, каждая команда запуска будет выполняться удаленным компьютером. Примите во внимание эту последовательность команд:

```
Enter-PSSession –computerName Server1
Enter-PSSession –computerName Server2
Enter-PSSession –computerName Server3
```

Первая команда устанавливает соединение с компьютера на Server1. Вторая происходит в течение этой сессии, то есть вы подключаетесь с Server1 к Server2. Третья команда, осуществляется внутри сессии, то есть вы подключаетесь с Server2 на Server3. Старайтесь не делать этого, так как каждая сессия включает в себя определенный объем памяти, обработки, сетевые накладные расходы. Вместо этого, выйдите из сессии до создания новой:

```
Enter-PSSession –computerName Server1
Exit-PSSession
Enter-PSSession –computerName Server2
Exit-PSSession
Enter-PSSession –computerName Server3
Exit-PSSession
```

Эта последовательность гарантирует, что каждое подключение к Server1, Server2, и Server3 будет с вашего компьютера.

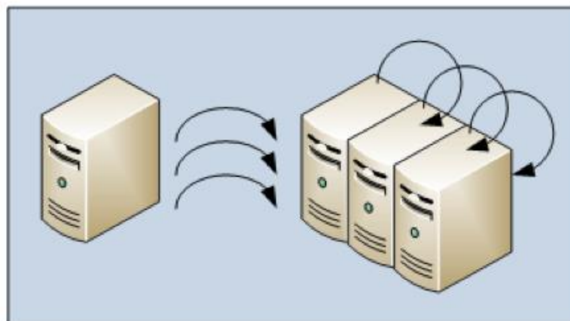
1-to-Many Remoting

1-to-1 Remoting полезен, когда вам нужно всего лишь выполнить команды на удаленном компьютере. Тем не менее, если вам нужно выполнить те же команды

на нескольких удаленных компьютерах с помощью 1-to-1 Remoting это будет неэффективно. Необходимо ввести сессию на первый компьютер, запустить команду, а затем выйти. Тогда вы введите второй сессии, выполните команду, и выйдете. Вам придется повторить этот процесс для каждого удаленного компьютера, и он займет слишком много времени.

1-to-Many Remoting

- 1-to-1 Remoting полезен, когда вам нужно всего лишь выполнить команды на удаленном компьютере. Тем не менее, если вам нужно выполнить те же команды на нескольких удаленных компьютерах с помощью 1-to-1
- Invoke-Command исполняются синхронно



Invoke-Command –scriptblock { Dir c:\demo }
–computerName Server1,Server2,Server3

1-to-Many Remoting позволяет представить команду, несколько команд, или даже целый сценарий с удаленных компьютеров. Команды передаются на удаленный компьютер с помощью WinRM. Каждый компьютер выполняет команды самостоятельно и возвращает результат, опять же с помощью WinRM, на ваш компьютер.

По умолчанию, оболочка не будет пытаться связаться с более чем 32 компьютерами одновременно. Если вы укажете более 32 компьютеров, то она будет работать с первыми 32, и потом, когда эти команды будут завершены, оболочка свяжется с дополнительными компьютерами. Эта функция называется дросселированием, и вы можете изменить дроссельный уровень с помощью дроссельных параметров.

Использование 1-to-Many Remoting осуществляется с помощью Invoke-Command командлета. Командлет принимает либо блок сценария или путь к файлу сценария и может принимать одно или несколько имен компьютеров:

```
Invoke-Command –scriptblock { Dir c:\demo }
–computerName 'Server1','Server2'
```

Дополнительные параметры Invoke-Command дают возможность указать альтернативный порт TCP, указать, что шифрование будет используется, указать альтернативные учетные данные, и так далее. Просмотрите помощь для команды, чтобы узнать больше.

Обратите внимание, что Invoke-Command обычно работает синхронно, то есть вы должны ждать, пока оболочка закончит работу со всеми удаленными компьютерами перед запуском дополнительных команд. Оболочка отображает результаты, как они приходят с удаленных компьютеров. Указывая-AsJob параметр, Invoke-Command может работать асинхронно, как фоновое задание..

Привязка конвейера

Параметр –computerName Invoke-Command связывает ByPropertyName канальные входные данные, то есть любой ввод объектов, которые имеют свойство ComputerName, будет иметь значения этого свойства, соотносящиеся с параметром computerName.

Если вы запросите компьютеры из Active Directory, вы заметите, что они имеют свойство Name:

```
Import-Module ActiveDirectory
Get-ADComputer –filter *
```

Вы можете использовать Select-Object для создания пользовательского объекта, который имеет ComputerName, а затем отправить эти объекты в Invoke-Command:

```
Get-ADComputer –filter * |
Select @{Label='ComputerName';Expression={$_.Name}} |
Get-Service –name *
```

Эта команда вернет список всех сервисов со всех компьютеров в AD

Работа с результатами

Имейте в виду, что Windows PowerShell командлеты возвращают объекты, как их выводы. К сожалению, объекты программного обеспечения не могут быть переданы по сети. Вместо этого, WinRM сериализирует объекты в XML-формате, так как XML это просто текст, и текст может быть передан по сети довольно легко. Объекты принимаются на компьютере, а затем десериализируются обратно в объекты. Такое преобразование в и из XML отражается на том, как вы можете использовать объекты.

Работа с результатами

- Имейте в виду, что Windows PowerShell командлеты возвращают объекты, как их выводы. К сожалению, объекты программного обеспечения не могут быть переданы по сети. Вместо этого, WinRM сериализирует объекты в XML-формате, так как XML это просто текст, и текст может быть передан по сети довольно легко. Объекты принимаются на компьютере, а затем десериализируются обратно в объекты. Такое преобразование в и из XML отражается на том, как вы можете использовать объекты

```
Invoke-Command –script { Get-Service } –computer
Server1,Server2 | Sort PSComputerName | Format-
Table –groupby PSComputerName
```

Рассмотрим, например, вывод этой команды:

```
PS C:\> get-service | get-member
TypeName: System.ServiceProcess.ServiceController
Name                      MemberType      Definition
----                      ----------      ----------
Name                      AliasProperty   Name = ServiceN
RequiredServices          AliasProperty   RequiredService
Disposed                  Event           System.EventHan
Close                     Method          System.Void Clo
Continue                  Method          System.Void Con
CreateObjRef              Method          System.Runtime.
Dispose                   Method          System.Void Dis
Equals                    Method          bool Equals(Sys
ExecuteCommand            Method          System.Void Exe
GetHashCode               Method          int GetHashCode
GetLifetimeService        Method          System.Object G
GetType                   Method          type GetType()
InitializeLifetimeService Method          System.Object I
Pause                     Method          System.Void Pau
Refresh                   Method          System.Void Ref
Start                     Method          System.Void Sta
Stop                      Method          ystem.Void Sto
WaitForStatus             Method          System.Void Wai
```

```
CanPauseAndContinue     Property        System.Boolean
CanShutdown             Property        System.Boolean
CanStop                 Property        System.Boolean
Container               Property        System.Componen
DependentServices       Property        System.ServiceP
DisplayName             Property        System.String D
MachineName             Property        System.String M
ServiceHandle           Property        System.Runtime.
ServiceName             Property        System.String S
ServicesDependedOn      Property        System.ServiceP
ServiceType             Property        System.ServiceP
Site                    Property        System.Componen
Status                  Property        System.ServiceP
```

Это объект нормального обслуживания. Сравните его с объектом службы, который вернулся из Invoke-Command:

```
PS C:\> invoke-command -script { get-service } -computer server-r2 | get-member
TypeName: Deserialized.System.ServiceProcess.ServiceController
Name      MemberType     Definition
----    ----------     ----------
ToString      Method      string ToString(), st
Name      NotePropertу      System.String Name=AD
PSComputerName      NoteProperty      System.String PSCompu
PSShowComputerName NoteProperty      System.Boolean PSShow
RequiredServices  NoteProperty      Deserialized.System.S
RunspaceId      NoteProperty      System.Guid RunspaceI
CanPauseAndContinue  Property      System.Boolean {get;s
CanShutdown      Property      System.Boolean {get;s
CanStop      Property      System.Boolean {get;s
Container    Property      {get;set;}
DependentServices      Property      Deserialized.System.S
DisplayName      Property      System.String {get;se
MachineName      Property      System.String {get;se
ServiceHandle      Property      System.String {get;se
ServiceName      Property      System.String {get;se
ServicesDependedOn      Property      Deserialized.System.S
ServiceType  Property      System.String {get;se
Site   Property      {get;set;}
```

Вы можете видеть, что типовое имя теперь указывает, что это объекты десериализации. Кроме того, объективные методы больше не доступны (за исключением встроенных ToString методов, которые прилагается ко всем объектам), и только свойства объектов выжили в преобразовании в и из XML. Эти объекты больше не являются прикрепленными к реальным, функционирующим частям программного обеспечения. Вместо этого, они по сути являются текстовыми снимками, или этих объектов.

Отметим также, что несколько дополнительных свойств были присоединены к объектам PsComputerName, например, которые содержат имя компьютера, с которого пришел каждый объект. Когда вы получаете результаты с нескольких компьютеров, это дополнительное свойство может позволить вам сортировать и группировать объекты в зависимости от компьютера, с которого они пришли. Например:

Invoke-Command –script { Get-Service } –computer Server1,Server2 | Sort PSComputerName | Format-Table –groupby PSComputerName

Если вы используете the –AsJob параметр Invoke-Command, то каждый компьютер получит свою группу работ:

Invoke-Command –script { Get-Process } –computer Svr1,Svr2 –asjob

Создаются три задания: верхнего уровня, работа для Invoke-Command , а потом дочерняя работа каждая для SVR1 и SVR2.

Результаты каждого компьютера, хранятся в соответствующем дочернем задании. Вы можете получить результаты от каждого по отдельности или вы можете получить все результаты, от верхнего уровня заданий.

Многокомпьютерные задания

Многокомпьютерные задания

- Задания, выполняемые на нескольких удаленных компьютерах сожержат дочерние задания
  - Задания верхнего уровня представляют задание в целом
  - Дочерние задания представляют задачи на каждом конкретном компьютере
- Чтобы посмотреть имена дочерних заданий…

```
Get-Job -id 1 | Select childjobs
```

- Получить их…

```
Get-Job -name job2
```

- Получить их результаты…

```
Receive-Job –name Job2 –keep
```

При использовании Invoke-Command и его-AsJob параметрjd, вы создаете верхний уровень работы:

PS C:\> invoke-command -script { get-service }

```
-computer server-r2,localhost -asjob
Id      Name        State
1       Job1        Running
```

Вы можете проверить статус этого высшего уровня работы, запустив Get-Job, и после того, как работа завершена, вы можете получить я результаты всех компьютеров с помощью Receive-Job:

```
Receive-Job –id 1 –keep
```

Однако, это также можно увидеть в имена дочерних заданий, которые были созданы:

```
get-job -id 1 | select childjobs
ChildJobs
---------
{Job2, Job3}
```

После того, как вы узнали имена дочерних названий, вы можете взять их по одному:

```
PS C:\> get-job -name job2
Id      Name            State
2       Job2            Completed
```

Вы также можете получить результаты каждого задания индивидуально по желанию:

```
Receive-Job –name Job2 –keep
```

Invoke-Command командлет также имеет-JobName параметр, который позволяет указать пользовательское имя для работы, а не с использованием автоматически сгенерированных имен, как Job2.

Распределение нагрузки

Распределение нагрузки

- Избегайте…

```
Invoke-Command –script { Get-Service } –computer Server1 |
 Where { $_.Status –eq "Running"} |
 Sort Name |
 Export-CSV c:\services.csv
```

- Делайте так…

```
Invoke-Command –script {
  Get-Service |
  Where { $_.Status –eq "Running" } |
  Sort Name
} | Export-CSV c:\services.csv
```

В качестве лучшей практики, попытайтесь сделать все ваши обработки на удаленном компьютере. Результаты, которые вы получите от Invoke-Command должно быть в окончательном виде, который вам нужен, также вы можете отформатировать их, экспортировать их в файл, или направить их на различные устройства вывода на локальный компьютер.

Например, следует избегать ситуаций, когда извлекаете все службы с удаленного компьютера, а затем выполняете сортировку и фильтрацию на месте:

```
Invoke-Command –script { Get-Service } –computer Server1 |
Where { $_.Status –eq "Running"} |
Sort Name |
Export-CSV c:\services.csv
```

Вместо этого, сделайте так много работы на удаленном компьютере, как это возможно:

```
Invoke-Command –script {
Get-Service |
Where { $_.Status –eq "Running" } |
Sort Name
} | Export-CSV c:\services.csv
```

При подаче команды на несколько компьютеров, каждый компьютер должен выполнять поставленные перед ним задачи на высшем уровне.

Restricted Runspaces

Ранее вы узнали о концепции ограниченных пространств выполнения, которая распространяется на сценарии Many-to-1 Remoting.

Ограниченное пространство выполнения, как правило, создано разработчиком программного обеспечения и развернуто на сервере, который запускает Windows PowerShell. Цель ограниченного пространство выполнения заключается в том, чтобы дать удаленным администраторам ограниченные административные возможности на этом сервере.

Например, в среде Exchange Server, вы можете поделить место на Exchange Server Computer с другими клиентами. Возможность использовать Windows PowerShell Remoting для управления частями Exchange configuration , такие как почтовые ящики пользователей, полезна, но хостинговая компания может ограничить вас в изменениях конфигурации частей других клиентов на сервере.

Ограниченное пространство выполнения предоставляет такую возможность.

Ограниченное пространство выполнения может добавить новые командлеты, удалить командлеты, добавлять, удалять или изменять параметры и так далее. Тут самое главное понимать, что некоторые командлеты могут вести себя не так, как вы привыкли.

Постоянное соединение

Итак, вы указали информацию о соединении как для 1-to-1, так и для 1-to-Many Remoting. То есть, когда работает или Invoke-Command или Enter-PSSession, вы должны предоставить имена компьютеров и, возможно, номера портов, альтернативные учетные данные, параметры проверки подлинности, и так далее. Если делать это повторяемо, то это может привести к ошибкам, поэтому Windows PowerShell предлагает способ для вас, чтобы создавать и повторно использовать соединение. Оболочка предоставляет эту возможность через PSSession.

Вы создаете новое, постоянное соединение с помощью New-PSSession командлета. Его параметры аналогичны для Enter-PSSession и Invoke-Command: Вы указываете имена компьютеров, учетные данные, порты и так далее. Результатом является постоянный объектом сессии, который может быть сохранен в переменной для дальнейшего использования.

Сессионные объекты могут быть переданы и Invoke-Command и Enter-PSSession, хотя Enter-PSSession принимает только один объект сессии, например:

```
$sessions = New-PSSession –computer Svr1,Svr2,Svr3
Invoke-Command –script { Get-Service } –session $sessions
```

Эта команда выполнит Get-Service на всех трех компьютерах. Или:

```
$sessions = New-PSSession –computer Svr1,Svr2,Svr3
Enter-PSSession –session $sessions[1]
```

Это позволит ввести 1-to-1 Remoting session на Svr2.

Ключевые моменты

Всегда используйте Start-Job, чтобы начать работу местных фоновых работ-никогда не используйте Invoke-Command и его-AsJob

параметр. Start-Job гарантирует, что работа создана локально, и что WinRM не используется.

Если вы используете Invoke-Command, чтобы начать локальное задание, знайте, что вы используете локальную петлю. То есть, WinRM выходит в сеть, а затем общается сам с собой по сети. Это может создать сложные ситуации для безопасности. Например, эта команда:

Invoke-Command –script { Get-EventLog Security } –comp localhost

на самом деле включает в себя коммуникационную сеть, и может поставить некоторые ограничения на то, что в состоянии сделать представленные команды.

Управляйте сессиями!

Сессии требуют некоторой памяти, процессора и сетевых накладных расходов, поэтому вы не должны оставлять их открытыми без надобности. Используйте Remove-PSSession для закрытия сессий и используйте Get-PSSession для получения сессий, которые в настоящее время открыты.

Закрытие всех сессий, которые у вас хранятся в переменной, легко осуществимо при использовании этой команды:

$sessions | Remove-PSSession

Вы можете легко создавать и поддерживать несколько наборов сессий в различных переменных. Чтобы закрыть все открытые сессии, используйте следующую команду:

Get-PSSession | Remove-PSSession

implicit remoting

Последнее приложение Remoting называется неявной implicit remoting. Идея состоит в том, чтобы признать, что различные компьютеры в среде содержат разные команды. Например, контроллер домена может содержат модуль ActiveDirectory. Этот модуль не будет работать на компьютере-клиенте Windows XP, хотя Windows XP может иметь Windows PowerShell v2. Предположим, у вас Windows XP на клиентском компьютере и вы хотите использовать командлеты в модуле ActiveDirectory.

Основные этапы implicit remoting таковы:

1. Используйте New-PSSession для установления сеанса с компьютера Windows XP на контроллер домена, который содержит модуль ActiveDirectory.

2. Вызовите команду на контроллере домена для загрузки модуля ActiveDirectory.

3. Используйте Import-PSSession для импорта удаленных команд на компьютере Windows XP. Укажите, что только команды, чьи имена начинаются с AD должны быть экспортированы, убедившись, что только Active Directory командлеты включены в экспорте.

Этот процесс не устанавливает модуль ActiveDirectory на Windows XP компьютер, этот модуль не может работать на Windows XP. Для использования Active Directory командлетов, вам нужно запускать их удаленно. Вы можете выполнять удаленные команд, указав нормальное имя командлетов.

Конечно, implicit remoting работает в любом сценарии, в котором вы хотите использовать командлеты, установленные на другом Компьютере - вы не ограничены в модуле ActiveDirectory или в Windows XP, хотя это хороший

Пример того, как implicit remoting может быть полезен.

Обратите внимание, что implicit remoting доступны только в то время как оригинальная сессия активна. Если закрыть оболочку или эту сессию, вы потеряете удаленные командлеты.

Администрирование Windows Server® 2008 R2

Windows Server® 2008 R2 поставляется с множеством оснасток и модулей Windows PowerShell®, разработанных для того, чтобы облегчить административную работу по управлению различными функциями Windows Server 2008 R2. Изучение этих оснасток и модулей даст вам возможность автоматизировать выполнение многих задач, связанных с Windows Server 2008 R2, а также позволит получить дополнительный опыт в использовании самой оболочки Windows PowerShell, ее ключевых командлетов и прочих техник, рассмотренных ранее в этом курсе.

Обзор модулей Windows Server® 2008 R2

Иногда одного знания о существовании того или иного инструмента бывает достаточно для того, чтобы сэкономить массу времени. Особенно это справедливо для Windows PowerShell: если вы знаете, что инструмент или возможность реально существует, такие функции как Help или командлет Get-Member помогут быстро получить информацию об особенностях использования данного инструмента или возможности. Цель данного урока – познакомить вас с многообразием модулей и оснасток, доступных в Windows Server 2008 R2. Обратите внимание, что некоторые из этих модулей и оснасток также доступны в Windows® 7, в частности, после установки Remote Server Administration Toolkit (RSAT). Модули и оснастки в Windows 7 и Windows Server 2008 R2 обычно работают идентично – функциональных различий между модулем, установленным на клиентском компьютере и тем же модулем, установленным на серверном компьютере практически нет.

Модули Windows Server 2008 R2 устанавливаются в папку модулей Windows PowerShell в масштабе всей системы. Чтобы посмотреть содержимое папки модулей внутри Windows PowerShell, наберите следующую команду:

```
Cd $pshome\modules
```

Также Windows PowerShell определяет переменную среды, которая содержит пути, по которым оболочка будет находить расположение модулей. Содержимое этой переменной среды можно посмотреть, набрав команду:

Get-Content env:psmodulepath

Также вы можете сгенерировать список установленных модулей, набрав команду:

Get-Module -list

Большинство расширений Windows PowerShell, включенных в Windows Server 2008 R2, представлено в виде модулей, однако, некоторые из них представлены в виде оснасток. Чтобы просмотреть список установленных оснасток, запустите команду:

Get-PSSnapin –registered

Обратите внимание, что не все модули и оснастки установлены в Windows Server 2008 R2 по умолчанию. Как правило, модуль или оснастка устанавливается одновременно с установкой соответствующей функции или роли. Например, модуль ActiveDirectory обычно устанавливается одновременно с установкой роли the Active Directory® Domain Services. Поэтому, в типичной производственной среде могут наблюдаться некоторые расхождения между модулями, установленными на разных серверах, так как на каждом сервере установлен индивидуальный набор ролей и функций.

Также не забывайте, что вам не нужно копировать или устанавливать одни и те же модули на каждый компьютер в вашем окружении. Некоторые модули, например, работают только на Windows Server 2008 R2, и при копировании на Windows 7 или на любой клиентский компьютер они просто не будут функционировать. Одна из задач обучения удаленному управлению, рассмотренная ранее, заключается в том, чтобы научиться использовать модули и оснастки, расположенные на удаленных компьютерах.

Модули и оснастки, включенные в Windows Server 2008 R2, могут относиться к следующим функциям и ролям:

- Active Directory Domain Services (AD DS)
- Active Directory Rights Management Services (AD RMS)
- AppLocker (Application ID Policy Management)
- Best Practices Analyzer
- Background Intelligent Transfer Service (BITS)
- Group Policy and Group Policy objects (GPOs)
- Network Load Balancing
- Windows PowerShell Diagnostics (PSDiagnostics)
- Remote Desktop Services
- Server Manager
- Server Migration

• Troubleshooting Packs

• Web Administration (Internet Information Services)

• Web Services for Management (WS-MAN)

• Windows Cluster Service (Failover Clusters)

• Windows Server Backup

Помимо этого, управление некоторыми функциями Windows Server 2008 R2 может осуществляться с использованием сторонних модулей и оснасток. Например, http://pshyperv.codeplex.com/ является открытым источником командлетов, разработанных, чтобы облегчить управление установками Microsoft Hyper-V™.

Использование модулей

Чтобы добавить модуль в оболочку, используйте командлет Import-Module и имя модуля. Обратите внимание, что модули остаются загруженными в продолжение всей текущей сессии работы оболочки, если только вы не используете командлет Remove-Module для их удаления. Просмотреть список загруженных на данный момент модулей можно с помощью командлета running Get-Module. Не забывайте, что все загруженные модули исчезают при завершении сеанса работы оболочки и не загружаются автоматически при начале следующей сессии. С помощью командлета Import-Module в профильном сценарии (мы рассмотрим его позже) можно автоматически загружать те модули, которые нужны вам для работы постоянно. Например:

Import-Module ActiveDirectory

Когда модуль загружен, можно просмотреть список командлетов, которые были добавлены, используя Get-Command:

Get-Command –module ActiveDirectory

Некоторые модули также добавляют новых провайдеров PSDrive. Просмотреть список всех установленных провайдеров можно с помощью командлета Get-PSProvider.

Некоторые расширения Windows Server 2008 R2, например, Windows Server Backup представлены не в виде модулей, а в виде оснасток. Загрузить их можно, используя командлет Add- PSSnapin и полное имя оснастки. Так же, как и модули, они остаются загруженными в продолжение текущей сессии работы оболочки. Чтобы увидеть, какие командлеты были добавлены в результате загрузки оснасток, запустите команду:

Get-Command –pssnapin snap-in-name

Большинство расширений, поставляемых Microsoft, имеют стандартный для Windows PowerShell файл «help». Следовательно, после загрузки модуля или оснастки вы можете с помощью Get-Command уточнить, какие командлеты были добавлены, а затем найти в справочном файле информацию об их предназначении и особенностях использования.

Командлеты Server Manager

Модуль Server Manager является базовым. Он включает всего три командлета, один из которых генерирует список всех доступных функций, указывая, какие из них установлены. Два других предназначены для добавления и удаления функций.

Список всех доступных ролей и функций, создаваемый командлетом Get-WindowsFeature, содержит официальные имена всех ролей и функций. Эти имена следует использовать и с командлетами Add-WindowsFeature и Remove-WindowsFeature для добавления или удаления функций.

Несмотря на то, что выходные данные командлета Get-WindowsFeature выглядят как хорошо отформатированный текстовый список, в действительности это набор объектов, так же, как выходные данные любого командлета в оболочке. Предустановленный элемент View, включенный в модуль ServerManager, форматирует эти объекты и выводит на экран в виде иерархичного списка. Однако это форматирование происходит только в самом конце конвейера. Это означает, что вы можете передавать эти объекты по конвейеру другим командлетам, чтобы манипулировать ими или выводить в виде отдельного файла.

Если вас интересует конструкция модуля, запустите сначала cd $pshome/modules/servermanager, затем gc servermanager.psd1, а после ls. Вы увидите содержимое модуля ServerManager, а именно его бинарный файл, файл справки в формате XML и значение вида .format.ps1xml. Содержимое будет представлять собой просто список компонентов, так что оболочка «знает», как загрузить их все.

9-20

Модуль GroupPolicy включает командлеты для создания, резервного копирования, хранения объектов групповой политики (Group Policy) и манипулирования ими и ссылками GPO (Group Policy Objects) внутри домена.

GPO командлеты

Group Policy состоит из двух различных компонентов:

- Файл GPO, который постоянно хранится на диске, содержащий настройки GPO.
- Канал связи между GPO и контейнером, сайтом, организационной единицей или доменом. Этот канал связи указывает, какие пользователи и компьютеры испытывают воздействие GPO.

Всестороннее изучение особенностей работы GPO лежит за пределами данного курса, хотя, если вы являетесь опытным администратором Windows Server, вы должны уже знать, как они работают. Для работы с GPO существует три основных командлета:

- Get-GPO: Извлекает существующие GPO, обычно по имени.
- New-GPO: Создает новый GPO.
- Remove-GPO: Удаляет GPO.

С помощью других командлетов можно осуществлять такие функции как переименование, резервное копирование и хранение GPO. К командлетам, предназначенным для работы с линками GPO, относятся:

• New-GPLink

• Remove-GPLink

• Set-GPLink

Помимо этого, существуют командлеты для работы с настройками наследования политик, с результирующей групповой политикой (RSoP) и с индивидуальными настройками внутри GPO. Например, Set-GPPrefRegistryValue и Set-GPRegistryValue позволяют изменять индивидуальные настройки внутри GPO. Принцип работы с GPO с помощью оболочки во многом схож с принципом работы с помощью Group Policy Management Console.

1. Начните с выбора существующего или создания нового GPO.

2. Измените одну или несколько настроек или настроек реестра внутри GPO.

3. Соедините GPO с одним или несколькими местами в домене.

Вспомнив процедуру работы в графической консоли, вы быстрее запомните последовательность команд, необходимых для выполнения тех же самых задач с помощью командной строки. Также рекомендуется потратить некоторое время на изучение командлетов, добавляемых модулем GroupPolicy. Некоторые командлеты умеют производить отчеты в HTML-формате, управлять разрешениями GPO и многое другое. Для того, чтобы узнать больше о возможностях всех предлагаемых командлетов и ознакомиться с примерами их использования, прочитайте соответствующий рздел справочника.

Командлеты TroubleshootingPack

Модуль TroubleshootingPack содержит командлеты, которые управляют Windows Troubleshooting Packs из командной строки. Эти командлеты не только обеспечивают удобный способ автоматизации расширения Troubleshooting Pack, но и позволяют через Windows PowerShell Remoting связаться с удаленным сервером или клиентскими компьютерами и запустить Troubleshooting Packs на них. Пакеты Troubleshooting представляют собой новую функцию Windows 7 и Windows Server 2008 R2, которая обеспечивает интерактивную диагностику и последующее решение проблем.

Модуль TroubleshootingPack содержит всего два командлета:

• Get-TroubleshootingPack загружает пакет troubleshooting pack с диска.

• Invoke-TroubleshootingPack выполняет загруженный пакет troubleshooting.

Чтобы загрузить пакет, необходимо знать путь к нему. Большинство пакетов устанавливаются в \Windows\Diagnostics\System, поэтому, можно запустить такую команду:

Get-TroubleshootingPack c:\windows\diagnostics\system\aero

Также вы можете просмотреть директорию c:\windows\diagnostics для поиска установленных на компьютер пакетов. Обратите внимание, что набор траблшутеров на разных компьютерах будет различаться – он зависит от версии операционной системы, установленных ролей и функций и.т.д. После того, как вы загрузили пакет Troubleshooting, вы можете передавать его по конвейеру

командлету Invoke-TroubleshootingPack для начала процесса решения проблем. Многие пакеты являются интерактивными и предлагают меню в виде командной строки, которую можно использовать для уточнения проблемы. Пакеты можно автоматизировать с помощью создания файла ответов в XTML. Информацию о том, как создать такой файл, вы найдете в разделе справочника, посвященном Get-TroubleshootingPack. После настройки файла ответов в соответствии с вашими потребностями (используя Windows Notepad или Windows PowerShell ISE), вы можете применять параметры –unattend и –answer командлета Invoke-TroubleshootingPack, чтобы диагностика проблем осуществлялась в автоматическом режиме. Хорошим сценарием будет использование Windows PowerShell 1-к-1 для обеспечения совместимости на удаленном компьютере. Это позволяет выполнять поиск проблем на удаленном компьютере, даже в то время, когда пользователь входит в систему на этом компьютере. Данная техника позволяет осуществлять поиск и решение проблем удаленных пользователей.

BPA командлеты

Модуль BestPractices включает командлеты, предназначенные для создания и изучения отчетов Best Practices Analyzer (BPA).

BPA – это функция Windows Server 2008 R2, предназначенная для анализа конфигураций ролей и функций, установленных на сервере и их сравнения с рекомендованными Microsoft конфигурациями. Используя командлеты BPA, вы сможете автоматизировать создание и анализ BPA отчетов, таким образом, поддерживая оптимальную конфигурацию сервера для большей надежности и работоспособности.

Командлеты BPA работают с моделями BPA. Каждая модель представляет собой набор рекомендаций, имеющих отношение к определенной функции или роли сервера. Запуск командлета Get-BpaModel без указания каких-либо параметров позволяет получить список доступных моделей. Этот список может варьироваться в зависимости от того,какие функции и роли установлены на сервер. В списке отображается официальный ID каждой модели. ID может понадобиться для извлечения конкретной модели с целью ее выполнения.

После того, как вы извлекли одну или несколько моделей, вы можете передать их командлету Invoke-BpaModel для их выполнения и создания отчета BPA. Отчеты BPA отображаются не моментально. Они передаются на хранение в операционную систему, откуда их необходимо извлекать. Для этого используйте командлет Get-BpaResult, уточнив ID той BPA модели, результаты для которой вы хотите просмотреть. Конечно, результатов не будет до тех пор, пока данная модель не будет извлечена.

Результаты возвращаются не в виде текстового списка. Как и почти все остальное в оболочке, результаты BPA представляют собой набор объектов, которые вы можете сортировать, фильтровать, и.т.д., а также экспортировать их в файл или конвертировать в другие форматы, например, HTML. Детальные инструкции по использованию командлетов BPA содержатся в соответствующем разделе справочника.

IIS командлеты

Модуль WebAdministration включает 74 командлета, большая часть которых используется для выполнения сложных IIS задач высокого уровня, таких как создание веб-сайтов, FTP-сайтов, приложений и пр. Также они позволяют запускать, останавливать или перезапускать сайт или пул. Более детальная информация об особенностях использования этих командлетов содержится в соответствующих разделах справочника.

Однако большая часть настроек конфигурации IIS в рамках Windows PowerShell не выполняется с помощью этих командлетов. Вместо этого модель, созданная разработчиками IIS, основывается на использовании IIS: диска.

Один из ключевых фактов об IIS 7.5 (версия, вошедшая в состав Windows Server 2008 R2) заключается в том, что она обладает модульной структурой. Это означает, что в IIS можно добавить новые функции посредством установки дополнительных программных модулей и других дополнений через консоль Server Manager или из интернета (например, с сайта http://iis.net).

Так как версии установленных IIS могут слегка отличаться друг от друга, за счет разных дополнений и разных возможностей, было бы довольно сложно создать единый набор командлетов для управления всеми возможными вариантами конфигурации. Поэтому, разработчики IIS создали командлеты для управления ключевыми, неизменными элементами, которые поддерживает любой сервер IIS: пулы приложений, веб-сайты, FTP сайты и.т.д.

Детальная настройка конфигурации сервера и его сайтов, пулов и других элементов осуществляется с помощью иерархического IIS диска, который выглядит и работает во многом как обычный жесткий диск или диск реестра, который отображается в Windows PowerShell. Например, верхний уровень IIS диска содержит три папки:

```
AppPools
Sites
SslBindings
```

Папка AppPools содержит сконфигурированные пулы приложений, такие как:

```
PS IIS:\> cd apppools
PS IIS:\apppools> ls
Name                    State          Applications
----                    -----          ------------
Classic .NET AppPool    Started
DefaultAppPool          Started        Default Web Site
```

Далее вы можете изменить директорию на один из этих пулов с помощью команды:

```
PS IIS:\apppools> cd defaultapppool
PS IIS:\AppPools\defaultapppool> ls
Name
----
WorkerProcesses
```

Дальнейшая навигация по иерархичной системе осуществляется таким же образом. Обратите внимание, что в действительности вы не используете диск – вместо него вы имеете дело с хранилищем параметров конфигурации, которое выполнено в виде диска. PSDrive провайдер, входящий в состав модуля WebAdministration, отвечает за придание этому хранилищу формы обычного диска. Вы используете командлеты –Item и –ChildItem для навигации по хранилищу, извлекаете существующие значения конфигурации и вносите изменения в эти значения.

Написание скриптов (обзор языка)

Вы уже знаете, что в простейшем смысле сценарий Windows PowerShell® представляет собой цепочку последовательных команд, которые выполняются в порядке строгой очередности. Сценарий такого типа во многом схож с командными файлами или пакетными скриптами, которые используются в большинстве других оболочек, включая Cmd.exe в Windows. На более сложном уровне сценарии Windows PowerShell могут напоминать программы, написанные на таких языках как Microsoft® Visual Basic® Scripting Edition (VBScript), JavaScript, JScript®, Perl и так далее. При необходимости можно создавать такие сценарии, которые будут предугадывать и исправлять конкретные ошибки – в этом случае вам потребуется потратить немало времени на настройку ваших скриптов, чтобы добиться их безупречной работы. Также сценарии можно моделировать, составляя их из отдельных блоков, чтобы использовать тот или иной блок в конкретной ситуации. В данном модуле вы узнаете, как использовать эти более сложные скриптовые элементы.

Поскольку большая часть функций в оболочке выполняется с помощью проблемно-ориентированных командлетов, ее скриптовый язык в действительности является очень простым и состоит из относительно небольшого количества (менее двух десятков) ключевых слов и конструкций.

Хранение данных

Вы уже сталкивались с переменными, которые используются в Windows PowerShell, в предыдущих модулях. В рамках данного занятия вы узнаете больше о том, что они собой представляют, как работают и для чего используются. Также вы ознакомитесь с родственными тематиками, такими как «массивы» и «дополнительные операции».

Переменные

Задумайтесь на минуту о ярлыках в Windows Explorer. Вам, несомненно, приходилось создавать ярлыки для файлов и папок, а меню «Пуск» представляет собой нечто большее, нежели просто набор папок и ярлыков. Ярлыки обладают несколькими важными признаками:

• У них есть свойства, такие как текущий каталог, который устанавливается, когда ярлык открыт и путь файла, на который ярлык указывает.

• Ими можно манипулировать: перемещать, удалять, копировать, и.т.д.

• Сами по себе ярлыки не представляют интереса – они лишь указывают на что-либо, представляющее интерес (приложение или документ).

• Они указывают на то, что физически хранится в определенном месте.

В Windows PowerShell переменная выполняет функцию, схожую с функцией ярлыков. Несмотря на то, что переменная сама по себе является объектом, наибольший интерес представляет то, на что она указывает. Переменные выступают в роли ярлыка чего-либо, что хранится в памяти оболочки – строки, числа, объекта или даже группы объектов.

Как и большинство динамически создаваемых элементов в оболочке, переменная существует только до тех пор, пока текущая сессия оболочки не завершена. Windows PowerShell создает диск variable:, на котором хранятся все переменные, которые были определены в оболочке. Этот диск содержит не только переменные, которые вы создаете, но и огромное количество переменных, которые определяются оболочкой и используются для хранения параметров конфигурации, влияющих на поведение оболочки. Оболочка предоставляет множество командлетов для управления переменными: увидеть их все можно, запустив команду:

```
Get-Command –noun variable
```

Однако во многих случаях вам не придется использовать эти командлеты. Помимо них, оболочка предлагает специальный синтаксис, который позволяет определять и модифицировать переменные без использования командлетов. Также для управления переменными можно использовать диск variable:, например, вы можете удалить элемент с диска, чтобы удалить соответствующую переменную из памяти.

Имена переменных обычно состоят из букв, цифр и символа нижнего подчеркивания. Имя переменной не включает символ $. Этот символ выполняет функцию ключа, который указывает оболочке, что то, что идет вслед за ним, является именем переменной. Когда вы передаете имя переменной параметру командлета, который ожидает получить имя переменной, символ $ указывать не нужно:

```
Remove-Variable –name var
```

Если вы поставите символ $ перед именем переменной, оболочка передаст параметру содержимое этой переменной:

```
$var = 'computername'
Remove-Variable –name $var
```

Данная команда удалит переменную $computername, а не переменную $var.

Имена переменных могут содержать пробелы и другие символы, но для этого имя следует заключить в фигурные скобки:

```
${My Variable} = 5
```

Использование пробелов считается не самым лучшим решением, так как использование фигурных скобок увеличивает объем работы при наборе команды и затрудняет чтение скрипта в дальнейшем.

Windows PowerShell использует знак равенства (=) в качестве оператора присваивания. Это означает, что все, что находится справа от этого знака, является значением того, что находится справа:

```
$var = 5
```

Оболочка позволяет создавать новые переменные путем придания того или иного значения переменной, как в примере выше. При этом нет необходимости объявлять переменную заранее, до присвоения ей значения.

На заметку: Если вы знакомы с VBScrip, у вас может возникнуть вопрос, есть ли здесь эквивалент Option Explicit. Об этом мы поговорим чуть позже.

К переменной можно привязать любой объект или группу объектов:

```
$services = Get-Service
```

Переменная будет хранить свое содержимое до тех пор, пока вы не замените его, не закроете оболочку, либо не удалите переменную.

Вы можете создавать и использовать переменные в оболочке, ссылаясь на переменную:

```
$var = 100
$var
$var = "Hello"
```

Обратите внимание, что очистка переменной – это не то же самое, что ее удаление. В следующем примере переменная $var существует, но имеет нулевое значение:

```
$var = 100
$var = ''
$var = $null
```

Также вы можете создавать, устанавливать и удалять переменные с помощью командлетов:

```
New-Variable –name var –value (Get-Process)
Set-Variable –name var –value $null
Clear-Variable –name var
Remove-Variable –name var
```

Последняя команда в данном примере в действительности удаляет переменную. Вы можете убедиться в этом, запустив команду:

```
Get-Variable
```

Или изучив содержимое диска variable:

```
Dir variable
```

Переменные могут передаваться по конвейеру другим командлетам; при этом в действительности передаются не сами переменные, а их содержимое:

```
$services | Sort Status | Select –first 10
```

И, наконец, переменная обеспечивает доступ к элементам объекта – свойствам и методам. Для этого необходимо обращаться с переменной, как с объектом: после

ее имени ставить точку, а далее указывать имя элемента, к которому вы хотите получить доступ:

```
$wmi = Get-WmiObject Win32_OperatingSystem
$wmi.Caption
$wmi.Reboot()
```

Строки

Оболочка позволяет использовать как одинарные, так и двойные кавычки для определения границ строки. В большинстве ситуаций можно использовать любой тип кавычек. В следующем примере обе команды будут идентичными с функциональной точки зрения:

```
$var = "Hello"
$var = 'World'
```

Ключевым различием здесь будет то, что при использовании двойных кавычек оболочка начинает искать символ $. Когда она находит этот символ, она решает, что все последующие символы, вплоть до пробела, являются именем переменной и заменяет всю переменную ее содержимым. Следующий пример иллюстрирует это различие:

```
PS C:\> $var = 'Hello'
PS C:\> $var2 = "$var World"
PS C:\> $var3 = '$var World'
PS C:\> $var2
Hello World
PS C:\> $var3
$var World
```

Как правило, двойные кавычки используются только в тех случаях, когда вам действительно необходима функция замещения переменной. Во всех остальных случаях принято использовать одинарные кавычки.

11-13

Как и во многих других оболочках, в Windows PowerShell есть символ перехода (экранирующий символ). Под символом перехода подразумевают обратный апостроф или `. На английской клавиатуре этот символ обычно расположен вверху слева под или рядом с клавишей Esc или Escape, как правило, на одной клавише со знаком тильды (~), но на других клавиатурах его расположение может варьироваться. По ссылке, указанной ниже, можно посмотреть раскладку клавиатуры в разных странах.

На заметку: при использовании некоторых шрифтов бывает сложно отличить обратный апостроф ` от обычного ‘. Старайтесь не путать их. Обратный апостроф может отменять особенное значение символов, которые идут вслед за ним. Например, в Windows PowerShell символ пробела имеет особое значение и используется в качестве разделителя. Однако при использовании обратного апострофа пробел теряет это значение и превращается в обычный буквенный символ:

```
Cd c:\program` files
```

Вы уже знаете, что символ $ имеет особое значение, и что оболочка ищет его внутри двойных кавычек для выполнения функции замещения переменной. Попробуйте отменить это особое значение – и вы получите просто знак доллара:

```
PS C:\> $var = 'Hello'
PS C:\> $var2 = "$var World"
PS C:\> $var3 = "`$var contains $var"
PS C:\> $var2
Hello World
PS C:\> $var3
$var contains Hello
```

Символ перехода может даже отменить особенное значение перехода каретки в исходное положение, которое обычно используется для определения логических границ строки:

```
$var = "This `
is a `
singe line"
```

Однако, как правило, этот метод не нужен, так как оболочка автоматически определяет, как правильно проанализировать такую многострочную команду.

И, наконец, символ перехода может придавать особенные значения отдельным символам, которые обычно не имеют никакого значения. Например, `t может обозначать табуляцию, `n – новую строку, `r – возврат каретки в исходное положение, и.т.д. Более подробную информацию обо всех возможных значениях можно найти в соответствующем разделе справочника.

Ссылка: • Windows International Keyboard Layouts, http://go.microsoft.com/fwlink/?LinkId=200517.

Массивы

Массив – это переменная, которая содержит более одного объекта. Например:

$var = 5

В данном примере $var не является массивом, так как она содержит всего один объект – число 5. Однако данная команда помещает массив объектов в переменную.

$var = Get-Service

В области разработки программного обеспечения существует отчетливое различие между массивом значений и набором объектов. Windows PowerShell абстрагируется от этих отличий, поэтому, во многих случаях термины «массив» и «набор» будут означать одно и то же. Оболочка автоматически воспринимает любой список, элементы которого разделены запятой, как массив:

```
$myarray = 1,2,3,4,5,6,7,8,9,0
```

Существует и более формальный способ обозначения новой переменной: использование символа @ и заключение списка значений в круглые скобки:

```
$myarray = @(1,2,3,4,5,6,7,8,9,0)
```

Функциональной разницы между этими двумя способами создания массива нет.

Работая с массивом, можно использовать порядковые номера для того, чтобы ссылаться на отдельные элементы. Первый объект в массиве имеет порядковый номер (индекс) 0. Использование индексов с отрицательным значением обеспечивает доступ к объектам с конца: -1 будет обозначать последний объект, -2 – предпоследний, и.т.д:

```
$services = Get-Service
$services[0]
$services[1]
$services[-1]
```

Массивы также обладают свойством Count, которое показывает количество объектов в массиве:

```
$services.count
```

Хеш-таблица

Вы уже знаете, как использовать хеш-таблицы с такими командлетами, как Select-Object и Format-Table. Хеш-таблица представляет собой разновидность массива. Каждый элемент в этом массиве состоит из ключа и значения, формируя пару ключ-значение. Например, когда вы использовали хеш-таблицу с командлетом Select-Object, вы имели два элемента: Lable (ключ) и Expression (значение):

```
Get-Process | Select @{Label='TotalMem';Expression={$_.VM + $_.PM}}
```

Символ @ в сочетании с фигурными скобками используется для создания хеш-таблицы. Когда вы создаете свою хеш-таблицу, вы можете использовать любые ключи и значения – ваш выбор не ограничивается только Label и Expression. После того, как вы создали хеш-таблицу, вы можете использовать ключи для извлечения значений:

```
PS C:\> $hash = @{"Server1"="192.168.15.4";"Server2"="192.168.15.11";
"Server3"="192.168.15.26"}
PS C:\> $hash.Server1
192.168.15.4
```

Символ @ используется в качестве оператора для создания как массивов, так и хеш-таблиц (которые, также, называются ассоциативными массивами); в простых массивах перечень значений указывается в скобках, тогда как в хеш-таблицах используются пары ключ-значение в фигурных скобках. Обращайте внимание на это различие. Хеш-таблица, как и любой массив, имеет свойство Count:

```
$hash.count
```

Вы можете передать объекты хеш-таблицы по конвейеру командлету Get-Member, чтобы увидеть другие свойства и методы:

Сплаттинг

Сплаттинг (Splatting) – это способ перемещения параметров и значений в хеш-таблицу и дальнейшей передачи всей хеш-таблицы командлету. Данная техника

может быть полезной для передачи динамически создаваемых параметров командлетам. Для начала создаем обычную хеш-таблицу, где в качестве ключей выступают имена параметров, а в качестве значений – значения этих параметров:

```
$parms = @{"ComputerName"="Server-R2";
"Class"="Win32_BIOS";
"Namespace"="root\cimv2"
}
```

Затем используем символ @ в качестве сплат-оператора (оператора подстановки) и имя переменной без символа $ для передачи этой хеш-таблицы командлету:

```
Get-WmiObject @parms
```

Командлет работает как обычно, принимая имена параметров и значения из хеш-таблицы.

На заметку: Различные примеры использования символа @ могут сбить с толку. Его конкретное предназначение зависит от контекста, в котором он используется, но в любом случае, его использование так или иначе связано с массивами или хеш-таблицами (ассоциативными массивами).

Арифметические операторы

Арифметические символы используются для осуществления подсчетов в оболочке. Они оцениваются оболочкой в соответствии со стандартными правилами в порядке старшинства:

- Умножение (*) и деление (/) слева направо
- Сложение (+) и вычитание (-) слева направо
- Выражения в скобках рассчитываются слева направо и изнутри наружу.

Оболочка распознает такие величины как Кб, Мб, Гб и Тб. Кб равен 1,024, Мб - 1,048,576 и.т.д. Обратите внимание, что эти значения являются двоичными, а не десятичными (это означает, что 1 Кб не равняется ровно 1000). Символ + также является оператором для сцепления строк:

```
$var1 = "One "
$var2 = "Two"
$var3 = $var1 + $var2
$var3
One Two
```

Типы переменных

Обычно оболочка позволяет помещать в переменную объекты любого типа. Если вы выполняете операцию, которая требует особого типа объектов, оболочка пытается временно конвертировать или принудительно преобразовать объекты в необходимый вид. Например:

```
PS C:\> $a = 5
PS C:\> $b = "5"
PS C:\> $a + $b
10
```

```
PS C:\> $b + $a
55
```

Переменная $ - это целое число, однако, переменная $b – это строка, так как ее значение было ограничено кавычками. В данном примере конструкция $a + $b дает в результате 10. Объект в $b был временно конвертирован в целое число, чтобы арифметическая операция была успешно завершена. Оболочка самостоятельно решила сделать это, так как первая переменная содержала целое число, а вторая содержала объект, который не мог быть воспринят как целое число.

В примере $b + $a можно наблюдать обратную ситуацию. Оболочка сначала «увидела» строку, после чего скорректировала вторую переменную, превратив ее в подходящую для сцепления строк форму.

Такое поведение оболочки может иногда сбить с толку и привести к неожиданным результатам. Решением проблемы может стать детальное информирование оболочки о том, какой тип объекта вы планируете использовать в переменной. Это делается путем уточнения имени типа .NET Framework в квадратных скобках:

```
PS C:\> [int]$a = "5"
PS C:\> $a | gm
TypeName: System.Int32
```

В данном примере, несмотря на то, что "5" – это строка, оболочка в принудительном порядке конвертировала ее в целое число, чтобы поместить в переменную $a. Если бы конвертация была невозможна, вы увидели бы ошибку:

```
PS C:\> $a = "Hello"
Cannot convert value "Hello" to type "System.Int32". Error: "Input string was not in
a correct format."
```

После того, как вы указали тип объектов для переменной, оболочка будет учитывать это до тех пор, пока вы не измените указания:

```
PS C:\> [string]$a = "Hello"
```

Наиболее распространенными видами объектов являются:

• String (строка)

• Int (integer) (целое число)

• Single и Double (плавающие числа)

• Boolean (Булев или логический тип данных)

• Array (Массив)

• Hashtable (хеш-таблица)

• XML

• Char (одинарный символ).

При написании сценариев считается оптимальным указывать тип объектов вручную. Это позволяет избежать ошибок и неожиданных результатов и упрощает настройку скрипта.

Сложные операторы сравнения

Вы уже знаете о базовых сравнительных операторах, таких как –eq и –gt. Но помимо них, оболочка предлагает ряд дополнительных операторов:

Оператор –contains позволяет узнать, содержится ли тот или иной объект в массиве. При этом он не просто сравнивает строки, например, он не скажет, содержит ли строка “Server1” подстроку “ver”. Вместо этого, происходит проверка всего объекта целиком:

```
$collection = "One","Two","Three"
$collection –contains "One"
```

Оператор –contains сравнивает более сложные объекты, изучая все свойства этих объектов. Чтобы использовать обратную логику и проверить, действительно ли массив не содержит тот или иной объект, используется оператор –notcontains.

Оператор –like производит сравнение с использованием подстановочного знака и обычно нечувствителен к регистру.

```
$var = "Server1"
$var –like "server*"
```

Чувствительной к регистру является версия –clike. Опять же, для использования обратной логики применяются операторы –notlike и –cnotlike. Все четыре оператора в качестве подстановочного знака используют *.

Оператор –is определяет, относится ли объект к определенному типу:

```
$var = "CONTOSO"
$var –is [string]
```

Тесно связан с ним оператор –as, который пытается конвертировать данный объект в объект другого типа. Если конвертация невозможна, он обычно выдает пустое значение, такое как пустая строка или ноль:

```
PS C:\> $var = "CONTOSO"
PS C:\> $var2 = $var -as [int]
PS C:\> $var2
PS C:\>
```

Области действия

Как и многие другие языки программирования и скриптовые языки, Windows PowerShell поддерживает иерархию областей действия. Эта иерархия контролирует то, когда и где те или иные элементы будут видимы и пригодны, а также как к этим элементам можно обратиться или изменить их. Поскольку правила области действия функционируют всегда, даже когда вы не догадываетесь об этом, использование оболочки без понимания, что такое область действия может привести к путанице и неожиданным ошибкам.

Область действия разработана для того, чтобы обозначать границы вокруг выполняемых элементов в оболочке. При правильном использовании она предотвращает нежелательное взаимодействие элементов и обеспечивает большую независимость элементов.

Область действия верхнего уровня в оболочке называется глобальной. Когда вы работаете непосредственно в командной строке, а не в сценарии, вы работаете в глобальной области действий. Эта область действий содержит определенные псевдонимы, отображения PSDrive, переменные и даже определенные функции. Все остальные области являются «детьми» или «внуками» (или даже «правнуками») глобальной области действий. Новая область действий появляется каждый раз, когда вы запускаете сценарий или выполняете функцию. Это может привести к образованию огромного количества вложенных областей и разветвленной иерархии. Например, представьте, что вы:

• Запустили сценарий;

• Сценарий содержит функцию;

• Функция выполняет другой сценарий

Этот последний сценарий будет уже областью действий четвертого поколения: глобальная область действия – область действия для первого сценария, область действия для функции – область действия для второго сценария.

Области действия «живут» столько, сколько необходимо – когда выполнение сценария заканчивается, его область действия разрушается или удаляется из памяти. Области действия создаются для следующих элементов оболочки:

• Переменные

• Функции

• Псевдонимы

• PSDrives

Оболочка устанавливает по умолчанию ряд правил, касающихся областей действия.

• Как только доступ к элементу получен, оболочка проверяет, существует ли этот элемент в рамках текущей области действия. Если да – оболочка использует этот элемент.

• Если элемент не существует в текущей области действия, оболочка обращается к «родительской» области действия и ищет элемент уже там.

• Таким образом, оболочка проверяет все области действия по восходящей до тех пор, пока элемент не будет найден. Если элемент не обнаружен в глобальной области действий, значит он не существует.

Например, представьте, что вы запустили сценарий, содержащий всего одну команду:

```
Gwmi win32_service
```

Каждая область действия начинает работу в виде пустой, «свежей» сессии. Вы не указали псевдоним Gwmi в этой области действия, поэтому, когда вы используете его, оболочка обращается к «родительской», глобальной области действия, чтобы проверить, существует ли этот псевдоним там. По умолчанию он существует, и оболочка использует псевдоним Gwmi, который указывает на Get-WmiObject. Такое поведение области действия является причиной того, что все псевдонимы,

PSDrive диски и переменные, присутствующие в оболочке по умолчанию, работают в вашем сценарии: они обнаруживаются в глобальной области действия.

Поведение меняется, когда вы изменяете элемент. Обычно оболочка не позволяет изменять элементы в «родительской» области действия из «дочерней» области действия. Вместо этого элементы создаются в текущей области действия. Например, вы запускаете следующую команду из командной строки, то есть, из глобальной области действия:

```
$var = 100
```

Вы получаете переменную $var внутри глобальной области действия, и эта переменная содержит целое число 100. Иерархия области действия будет выглядеть так:

```
• Global: $var = 100
```

Далее вы запускаете сценарий, который содержит следующее:

```
Write $var
```

Эта область действия не содержит переменную $var, поэтому, оболочка обращается к «родительской» области действия – глобальной. Переменная $var обнаруживается там, поэтому, целое число 100 возвращается в сценарий и сценарий выдает 100 в качестве выходных данных. Предположим, далее вы запустили сценарий, который содержит следующее:

```
$var = 200
write $var
```

Переменная $var сейчас создается в области действия сценария, и целое число 200 помещается в нее. Иерархия области действия сейчас выглядит так:

- Global: $var = 100
- Script: $var = 200

Новая переменная с именем $var сейчас была создана в области действия сценария, но переменная в глобальной области действия осталась неизменной. Сценарий выдает в качестве выходных данных 200, так как $var содержит именно это число. Это не влияет на глобальную область действия. Если, после того как сценарий будет выполнен, вы запустите из командной строки команду:

```
Write $var
```

Выходными данными будет являться число 100, так как переменная $var в глобальной области действия содержит именно его. Такая модель поведения относится ко всему, что создается в «дочерней» области действия. Здесь необходимо запомнить два базовых правила:

- Если вы читаете что-либо, не существующее в текущей области действия, оболочка пытается извлечь это из «родительской» области действия.
- Если вы пишете что-либо, то это всегда создается в текущей области действия, даже если в «родительской» области действия есть элемент с аналогичным именем.

Запомните следующие общие правила:

• «Дочерняя» область действия может ЧИТАТЬ элементы, находящиеся в «родительской» области действия.

• «Родительская» область действия никогда не сможет прочитать или написать что-либо в «дочерней».

• Вы можете ПИСАТЬ только в текущей области действия.

• Если вы пытаетесь поменять что-либо в родительской области действия, результатом будет появления нового элемента с таким же именем в текущей области действия.

Windows PowerShell не запрещает изменять элементы в «родительской» области действия. Однако обычно это считается не очень удачным методом работы. Если сценарий изменяет что-либо в глобальной области действия, бывает сложно предугадать, как эти изменения скажутся на работе других элементов оболочки. Все командлеты, имеющие отношение к элементам областей действия, имеют параметр –scope. К таким командлетам относятся:

• New-Alias

• New-PSDrive

• New-Variable

• Set-Variable

• Remove-Variable

• Clear-Variable

• Get-Variable

Параметр –scope может принимать несколько значений:

• Global – означает, что командлет относится к глобальной области действия.

• Script – означает, что командлет относится к области действия сценария. Если командлет запущен внутри скрипта, он влияет на область действия этого скрипта. Если он запущен в дочерней по отношению к скрипту области действия, он будет влиять на этот скрипт.

• Local – означает, что командлет относится к текущей области действия, которая запущена в данный момент.

• Также значением может быть целое число, где 0 обозначает текущую область действия, 1 – область действия, являющуюся родительской по отношению к текущей, и.т.д.

Если бы следующая команда выполнялась в сценарии, запущенном из командной строки, переменная была бы создана в глобальной области действия (на один уровень выше текущей области действия):

New-Variable –name var –value 100 –scope 1

Также вы можете использовать специальный синтаксис:

• $global:var относится к переменной $var в глобальной области действия.

• $script:var относится к переменной $var в текущей области действия (если текущая область действия - сценарий), или к ближайшей родительской области действия, являющейся сценарием.

• $local:var относится к переменной $var в текущей области действия.

Здесь, опять же, не рекомендуется модифицировать элементы за пределами текущей области действия, поскольку сложно сказать, как эти изменения могут отразиться на других задачах и процессах.

Чтобы предотвратить возникновение сложных, запутанных ситуаций, требующих тщательной настройки сценария, старайтесь следовать нескольким основным рекомендациям:

• Никогда не ссылайтесь на переменную до тех пор, пока вы не прикрепили к ней объект в текущей области действия. Это позволит ограничить доступ к переменным «родительской» области действия, которые могут содержать объекты, о которых вы не знали.

• Никогда не пытайтесь модифицировать псевдонимы, PSDrive элементы, функции или переменные в «родительской» области действия. Изменения можно производить только в текущей области действия.

Эти правила, в частности, относятся к областям действия, которые находятся за пределами вашей видимости, например, глобальной области действия. Старайтесь никогда не менять то, что может различаться в зависимости от ситуации, времени, используемого компьютера и.т.д.

Объявление переменных и Strict Mode

Нет необходимости объявлять переменные заранее. Этот вопрос часто интересует администраторов, имеющих опыт работы с VBScript, так как в этом языке предусмотрена команда Option Explicit, которая требует заблаговременного объявления переменных. В Windows PowerShell нет точного аналога Option Explicit, однако, здесь есть командлет Set-StrictMode. Set-StrictMode позволяет задавать строгий режим, а версия 2.0 выполняет функции, сходные с Option Explicit. Сама команда выглядит так:

```
Set-StrictMode –version 2
```

В этом режиме оболочка запрещает использование неинициализированных переменных. Например, при выключенном строгом режиме или при использовании версии 1.0 командлета, следующая команда будет разрешена:

```
Function MyFunction {
Write $var
$var = 1
Write $var
}
MyFunction
Write $var
```

Однако при выполнении следующей команды оболочка выдаст ошибку:

```
Set-StrictMode -version 2.0
```

```
Function MyFunction {
Write $var
$var = 1
Write $var
}
MyFunction
Write $var
```

Ошибка возникает из-за того, что переменная $var упоминается дважды до того, как ей было присвоено значение в данной области действий (вы узнаете больше об областях действия на следующем занятии). Целью отображения ошибки является предупреждение простых синтаксических ошибок. Например, рассмотрим команду:

```
Function MyFunction {
$computer = 'Server2'
gwi Win32_ComputerSystem -comp $compter
}
MyFunction
```

В этом примере переменная $computer во второй раз была напечатана с ошибкой. Данный скрипт будет выполняться, но скорее всего появится сообщение об ошибке, указывающее на то, что WMI не мог установить соединение, так как пытался связаться с пустым значением имени компьютера. Это сообщение об ошибке может ввести в заблуждение, так как корректное имя сервера присутствует в скрипте, а команда Get-WmiObject была напечатана правильно, отдельно от переменной. Решить проблему поможет следующее изменение:

```
Set-StrictMode -version 2.0
Function MyFunction {
$computer = 'Server2'
gwi Win32_ComputerSystem -comp $compter
}
MyFunction
```

Сейчас оболочка выдает ошибку об использовании неинициализированной переменной $compter. Эта ошибка заставит вас обратить более пристальное внимание на настоящую проблему – опечатку в имени переменной.

Установка строгого режима версии 2.0 крайне рекомендуется. Это позволяет быстрее и с большей точностью выявить существующие ошибки и устранить их. Обратите внимание, что строгий режим версии 2.0 не требует объявления переменной заранее; он запрещает ссылаться не переменную до тех пор, пока ей не присвоено значение. Например, следующий пример работает при выключенном строгом режиме:

```
Set-StrictMode -off
Function MyFunction {
Write $var
$var = 1
Write $var
```

```
}
$var = 3
MyFunction
Write $var
```

Он работает потому, что в первой выполняемой строчке кода переменной $var присваивается значение 3. Когда функция запущена, она может извлечь это значение из родительской области действия. Другими словами, переменной $var было присвоено значение в области действия скрипта и всех дочерних областях, поэтому, ошибки не возникает.

Управляющие операторы языка

Windows PowerShell предлагает языковые конструкции, которые являются идентичными многим другим скриптовым языкам. В основе ее синтаксиса лежит язык C#, хотя имеются сходства и со многими другими языками на основе C, такими как Java, JavaScript или PHP. У многих администраторов получается использовать Windows PowerShell для выполнения сложных задач даже без использования этих скриптовых конструкций. Данные конструкции необходимы, в первую очередь, для написания скриптов, предназначенных для выполнения сложного, многоэтапного процесса, где требуется принятие решений или выполнение повторяющихся мелких подзадач. Также они могут быть полезны для переходящих скриптов, основанных на более старых технологиях, таких как VBScript.

Около полдюжины конструкций, которые будут рассмотрены на этом уроке, составляют формальный скриптовый язык Windows PowerShell. Они дают возможность сценариям предпринимать различные действия, основываясь на динамических критериях и выполнять повторяющиеся задачи. Большинство этих конструкций основано на сравнительных критериях, а сравнение обычно производится с помощью одной или нескольких сравнительных операций оболочки.

Следует помнить, что Windows PowerShell предназначена для широкой аудитории. Например, не все администраторы свободно владеют языками программирования, поэтому, использование конструкций при знакомстве с Windows PowerShell может вызвать у них определенные затруднения. Но оболочка позволяет выполнять достаточно сложные задачи и без использования скриптовых элементов. В то же время, другие администраторы имеют некоторый опыт программирования – у них использование этих конструкций не вызовет сложностей. Но главное здесь то, что оболочка позволяет выполнять множество сложных задач без знания того, что мы привыкли относить к программированию. Конструкции доступны для всех, кто хочет и может пользоваться ими, но они не являются обязательными для работы с оболочкой. Тем не менее, они позволяют создавать скрипты для выполнения еще более сложных задач, а изучив их, вы сможете автоматизировать многие административные действия.

If….ElseIf…..Else

Рассмотрим конструкцию, используемую для принятия решений. Она предназначена для того, чтобы производить одно или несколько логических

сравнений и выполнять одну или несколько команд в отношении первого сравнения, которое окажется верным:

```
$octet = Read-Host 'Enter the third octet of an IP address'
if ($octet –lt 15 –and $octet –gt 1) {
$office = 'New York'
} elseif ($octet –ge 15 –and $octet –lt 20) {
$office = 'London'
} elseif ($octet –ge 21 –and $octet –lt 34) {
$office = 'Sydney'
} else {
throw 'Unknown octet specified'
}
```

Несколько замечаний, касающихся этой конструкции:

• Конструкция должна начинаться с блока if.

• Она может не включать блока elseif, а может включать несколько.

• Блоком else можно заканчивать конструкцию.

• Выполняется только первый блок со значением True, получившимся в результате сравнения.

• Блок else выполняется только в том случае, если этого не сделал ни один из предыдущих блоков.

Условие или сравнительная часть скриптовой конструкции помещается в скобки. Условие может быть любой сложности и может содержать подвыражения в скобках. Однако целая конструкция должна в обязательном порядке иметь одно из двух решений: True (верное) или False (ложное). Обратите внимание, что Windows PowerShell отображает значение False в виде нуля, а значение True любым другим числом. Поэтому, если результатом сравнения будет ноль, утверждение будет признано ложным (False). Например, следующая конструкция будет иметь результат True:

```
if (5+5) {
write 'True'
} elseif (5-5) {
write 'False'
}
```

Код условия, он же код команды, которая выполняется, когда результатом сравнения будет значение True, заключен в фигурные скобки. При этом оболочка не устанавливает жестких правил форматирования – например, следующее оформление конструкции будет вполне приемлемым:

```
if (5+5) { write 'True' } elseif (5-5) { write 'False' }
```

Однако такую конструкцию не очень удобно читать. Поэтому, чаще всего рекомендуется структурировать код и располагать скобки в соответствии с общепринятыми стандартами. Например, распространенным стандартом форматирования является расположение открывающей фигурной скобки в новой

строке после условия, а закрывающей – в новой строке после последнего условного утверждения или команды. Именно такой формат мы чаще всего использовали в данном курсе ранее и будем использовать в дальнейшем. Еще один часто используемый формат выглядит так:

```
if (5+5)
{
write 'True'
}
elseif (5-5)
{
write 'False'
}
```

Не имеет значения, какой формат вы выберете, если вы правильно структурируете условный код и уверены в корректном размещении фигурных скобок.

Вложенные конструкции

Все конструкции могут быть вложенными в другие. Единственное правило здесь заключается в том, что они должны быть полностью вложенными. Это означает, что внутренняя конструкция должна быть закрыта раньше внешней. Написание каждого условного уровня с новой строки поможет вам убедиться, что вы все сделали правильно; многие сторонние редакторы предлагают дополнительные визуальные ключи, которые помогают контролировать правильность вложения конструкций. Вот пример вложенной конструкции If:

```
If ($ping.pingstatus –eq 0) {
	$wmi = gwmi win32_operatingsystem –computer $computer
	if ($wmi.buildnumber –ne 7600) {
		write 'Incorrect build – ignoring computer'
	}
}
```

Обратите внимание, как структурирование текста помогает выделить вложенную конструкцию. Windows PowerShell не принуждает производить именно такое форматирование, тогда как в Windows PowerShell ISE оно является обязательным. Например, следующая конструкция разрешена, но является неудобной для чтения и для работы:

```
If ($ping.pingstatus –eq 0) {
$wmi = gwmi win32_operatingsystem –computer $computer
if ($wmi.buildnumber –ne 7600) {write 'Incorrect build – ignoring computer'}}
```

Набор операторов

Когда Windows PowerShell запускает сценарий, это происходит точь-в-точь так же, как если бы вы вручную набирали каждую строчку сценария в диалоговом окне. Таким образом, нет никакой разницы между запуском сценария и запуском отдельных команд вручную, за исключением того, что сценарий запускается только в том случае, если политика выполнения оболочки позволяет это. Но даже если включен режим Restricted политики выполнения, вы можете открыть

сценарий, скопировать его содержимое в буфер обмена и вставить в интерактивную оболочку.

На заметку: Возможность копирования и вставки скрипта в оболочке не считается проблемой, связанной с безопасностью, так как для того, чтобы сделать это, вы должны предпринять ряд целенаправленных действий. Вероятность того, что вы случайно скопируете и вставите нужные команды, практически исключена.

Вы можете использовать любую скриптовую конструкцию напрямую из командной строки, даже из окна консоли вместо ISE. Когда вы открываете конструкцию путем набора { и нажатия клавиши Return, подсказка в оболочке меняется и указывает на то, что вы находитесь внутри конструкции, и что оболочка ждет от вас ее завершения:

```
PS C:\> if ($var -eq 5) {
>>
```

Когда вы закончили ввод конструкции и закрыли ее, снова нажмите Return, чтобы запустить выполнение кода:

```
PS C:\> if ($var -eq 5) {
>> write 'is 5'
>> } else {
>> write 'is not 5'
>> }
>>
is not 5
PS C:\>
```

Включая конструкцию такого типа в скриптовый блок командлета, вы можете отказаться от рекомендуемого форматирования и расположить всю команду в одну строку, например:

```
gwmi win32_logicaldisk | select deviceid, @{Label='DriveType';Expression={ if
($_.drivetype -eq 3) { write 'LocalFixed' } else { write 'NonFixed' }}}, Size
```

Эта однострочная команда включает командлет Select-Object, который использует пользовательские колонки. Выражение для этих колонок включает конструкцию If, но она не отформатирована в соответствии с рекомендациями. Выходные данные этой команды будут выглядеть примерно так:

```
deviceid    DriveType    Size
--------    ---------    ----
A:          NonFixed
C:          LocalFixed   64317550592
D:          NonFixed
Z:          NonFixed     413256384512
```

Как вы видите, скриптовая конструкция может использоваться напрямую из командной строки, а не только в сценарии.

Switch

Эта конструкция также предназначена для принятия логических решений. Она сравнивает один входящий объект с рядом возможных объектов или сравнений. По умолчанию она выполняет все удовлетворяющие критерию предложения, а не только первое. Примером такой конструкции может быть:

```
$computer = 'LON-DC1'
switch ($computer) {
	'LON-DC1' {
		write 'London Domain Controller 1'
	}
	'SEA-DC1' {
		write 'Seattle Domain Controller 1'
	}
	'LON-DC1' {
		write 'London Domain Controller 1'
	}
	default {
		write 'Unknown'
	}
}
```

Несколько замечаний по поводу этой конструкции:

• Обратите внимание, что в выходных данных London Domain Controller 1 присутствует дважды, так как здесь есть два удовлетворяющих критерию предложения.

• Блок default является опционным и выполняется только в том случае, если в предыдущих блоках не было найдено удовлетворяющих критерию предложения.

• Каждое возможное совпадение представляет собой отдельную подконструкцию, код которой должен заключаться в фигурные скобки.

Break

Ключевое слово Break прерывает выполнение любой конструкции кроме If…ElseIf…Else. Break можно использовать в сочетании с Switch для того, чтобы убедиться, что выполняется только первое удовлетворяющее критерию предложение.

```
$computer = 'LON-DC1'
switch ($computer) {
'LON-DC1' {
write 'London Domain Controller 1'
break
}
'SEA-DC1' {
write 'Seattle Domain Controller 1'
break
}
```

```
'LON-DC1' {
write 'London Domain Controller 1'
break
}
default {
write 'Unknown'
}
}
```

Выходными данными в этом примере является один London Domain Controller 1, так как ключевое слово Break немедленно прервало конструкцию после первого удовлетворяющего критерию предложения.

Опции Switch

Конструкция Switch включает несколько опций, влияющих на ее поведение. Одной из них является использование параметра –wildcard switch, который позволяет удовлетворяющему критерию предложению, являющемуся строкой, обрабатываться как строке с подстановочным символом. В следующем примере выходными данными будут “London” и “Domain Controller.” Обратите внимание, что ключевое слово Break не используется, а значит возможно несколько вариантов предложений, удовлетворяющих критериям:

```
$computer = 'LON-DC1'
switch -wildcard ($computer) {
'LON*' {
write 'London'
}
'SEA*' {
write 'Seattle Domain Controller 1'
}
'*DC*' {
write 'Domain Controller'
}
'*SRV*' {
write 'Member Server'
}
default {
write 'Unknown'
}
}
```

О других возможностях конструкции Switch можно узнать в соответствующем разделе справочника.

For

Эта конструкция обычно используется для повторения одной и той же команды указанное количество раз:

```
For ($i=0; $i –lt 10; $i++) {
```

```
Write $i
}
```

Несколько замечаний по поводу конструкции:

• Условия или критерии сравнения конструкции в действительности состоят из трех частей, разделенных точкой с запятой.

• В первой части задается стартовое условие, в данном случае $i=0.

• Вторая часть представляет собой условие, при котором цикл должен повторяться, в данном случае он будет повторяться до тех пор, пока $i меньше 10.

• Третья часть – это операция, которая будет выполняться каждый раз при завершении цикла, в данном случае это увеличение $i на один.

Разрешается менять переменную counter внутри самого цикла:

```
For ($i=0; $i –lt 10; $i++) {
Write $i
$i = 10
}
```

В данном примере цикл будет выполняться только один раз, так как в процессе выполнения $i выйдет за пределы обозначенного диапазона (меньше 10).

Не забывайте о правилах, касающихся областей действия. Например, следующее делать не рекомендуется:

```
For ($i; $i –lt 10; $i++) {
Write $i
}
```

В данном примере $i идентифицирована как переменная цикла, но ей не присвоено стартовое значение. Оболочка вынуждена передвигаться по областям действия вверх и проверять, определена ли переменная $i в какой-либо из родительских областей. Если нет – переменной $i присваивается значение по умолчанию, ноль. Однако если $i была создана в одной из родительских областей действия, она может получить ненулевое значение или даже значение, не являющееся числом. Это может привести к неожиданным результатам и к длительным поискам ошибки. Поэтому, всегда рекомендуется задавать стартовое значение числовой переменной.

While-Do-Until

Эти ключевые слова могут использоваться в разных комбинациях для создания немного разных эффектов. Вот три варианта:

```
$var = 1
while ($var -lt 10) {
write $var
$var++
}
```

```
$var = 1
do {
```

```
write $var
$var++
} while ($var -lt 10)
```

```
$var = 20
do {
write $var
$var--
} until ($var -lt 10)
```

Несколько заметок об этих конструкциях:

• Когда критерии сравнения или условие находятся в конце конструкции, условный код внутри конструкции всегда выполняется как минимум один раз.

• Когда критерии сравнения или условие находятся в начале конструкции, Условный код внутри конструкции выполнятся только в том случае, если сравнение начинается с True. Это означает, что в некоторых ситуациях код не выполняется вообще.

• While может использоваться как в начале, так и в конце конструкции, а Until – только в конце.

• While и Until в конце конструкции выполняют одну и ту же функцию, но имеют противоположную логику.

ForEach

Данная конструкция выполняет ту же самую базовую задачу, что и командлет ForEach-Object: перечисляет ряд объектов. Условный код конструкции выполняется один раз для каждого входящего объекта. Во время каждого выполнения цикла следующий объект извлекается из набора и помещается в обозначенную вами переменную. Например6

```
$services = Get-Service
ForEach ($service in $services) {
Write $service.name
}
```

Обратите внимание, что синтаксис здесь слегка отличается от командлета ForEach-Object. Эта конструкция не содержит переменной $_, а ее условие содержит ключевое слово in. Это различие иногда вызывает путаницу, так как оболочка предусматривает псевдоним ForEach для командлета ForEach- Object. Это означает, что “foreach” является одновременно скриптовой конструкцией и псевдонимом для командлета. Оболочка может различить их, исходя из того, где они используются. Псевдоним используется только в цепочке команд, и обычно обладает объектами, которые передаются ему по конвейеру.

```
Get-Service | ForEach { Write $_.name }
```

Важно не путать эти две формы синтаксиса. Но действительно ли вам так нужен ForEach? Во многих случаях администраторы, имеющие некоторый опыт в программировании, в частности, в VBScript, используют ForEach- Object или

конструкцию ForEach даже тогда, когда в этом нет острой необходимости. Конечно, в этом нет ничего страшного, кроме того, что они делают лишнюю работу. Например, рассмотрим отрывок кода, который удаляет все объекты в папке, если их размер превышает указанный:

```
$folder = 'c:\demo'
$files = dir $folder
foreach ($file in $files) {
if ($file.length –gt 100MB) {
del $file
}
}
```

Это очень «скриптовый» подход к созданию кода, который часто можно встретить в таких языках как VBScript. Однако Windows PowerShell позволяет создать куда более простую модель, так как большинство командлетов могут обрабатывать наборы объектов, не требуя их перечисления:

```
Dir c:\demo | where { $_.Length –gt 100MB } | Del
```

Каждый раз, когда вы собираетесь использовать ForEach или ForEach-Object, не забывайте подумать – а действительно ли это так необходимо? В некоторых случаях, например в тех, которые были рассмотрены в модуле, посвященном Windows Management Instrumentation, это необходимо. Однако во многих других ситуациях можно обойтись и другими средствами.

Стиль работы

При наборе конструкции в командной строке, например, для создания части скриптового блока, который будет передаваться командлету, вы можете отказаться от рекомендуемого форматирования в пользу краткости. В этом случае вам придется печатать множественные команды в одной строке. Windows PowerShell позволяет действовать таким образом, при условии, что команды будут разделены точкой с запятой:

```
gwmi win32_logicaldisk | select deviceid, @{Label='DriveType';Expression={ switch
($_.drivetype) { 2 { Write 'Floppy'; break } 3 { Write 'Fixed'; break } 4 { Write
'Optical'; break } 5 { Write 'Network'; break }}}}, Size, FreeSpace
```

Обратите внимание, что точка с запятой используется для разделения команд Write и Break внутри каждого блока, содержащего условие. Несмотря на то, что это все можно напечатать так, как показано в примере, при использовании форматирования блок будет легче читаться:

```
gwmi win32_logicaldisk |
	select deviceid, @{Label='DriveType';Expression={
		switch ($_.drivetype) {
		2 { Write 'Floppy'; break }
		3 { Write 'Fixed'; break }
		4 { Write 'Optical'; break }
		5 { Write 'Network'; break }
	}
```

```
}
}, Size, FreeSpace
```

Второй пример тоже будет верным, поскольку оболочка знает, как проанализировать такой тип входящих данных. Набор данной команды в оболочке будет выглядеть примерно так:

```
PS C:\> gwmi win32_logicaldisk |
>> select deviceid, @{Label='DriveType';Expression={
>>     switch ($_.drivetype) {
>>             2 { Write 'Floppy'; break }
>>             3 { Write 'Fixed'; break }
>>             4 { Write 'Optical'; break }
>>             5 { Write 'Network'; break }
>>     }
>> }
>> }, Size, FreeSpace
>>
```

Если вы ранее работали со скриптовыми языками, то, возможно, при изучении Windows PowerShell вы будете использовать те же подходы, которые применяются в этих языках. В этом нет ничего страшного – скриптовые конструкции Windows PowerShell созданы специально для того, чтобы вы могли перенести свои старые знания в новые условия. Однако не забывайте, что зачастую Windows PowerShell предлагает более простые способы выполнения тех же самых задач. Изучение этих способов позволить сделать работу с Windows PowerShell более эффективной, а также узнать о новых возможностях, предлагаемых оболочкой. В целом можно порекомендовать стараться чаще использовать цепочки команд вместо формальных скриптов. Для очень сложных задач написание сценария может стать верным решением, но во многих случаях можно обойтись более простыми и короткими командами.

Обработка ошибочных ситуаций

Несмотря на то, что вам придется фиксировать и устранять некоторые ошибки, например, опечатки, во время написания сценария, некоторые ошибки можно предусмотреть заранее. Например, ошибка, возникающая из-за того, что файл или сервер недоступен, относится к тем ошибкам, о возможности возникновения которых можно догадаться заранее, но которые нельзя зафиксировать или предотвратить. Решением в данном случае станет написание сценария, который позволит выявить и исправить такие ошибки.

В сценарии а Windows PowerShell выделяют три большие категории ошибок:

• Синтаксические ошибки, которые, как правило, являются результатом опечаток или неправильного написания.

• Логические ошибки, которые означают, что сценарий выполняется верно, но результаты получаются не те, которые были запланированы.

• Ошибки выполнения, которые можно предусмотреть заранее, но нельзя исправить заранее (например, недоступность удаленного компьютера).

Синтаксические ошибки проще всего выявить и предупредить. Сообщения об ошибках обычно направляют вас в нужное место – вам остается лишь пристально взглянуть на неправильно набранное слово, определить ошибку и устранить ее. Логические ошибки являются более сложными и требуют настройки сценария – об этом вы узнаете чуть позже. Логические ошибки обычно возникают, когда переменная или свойство имеет значение, несущее не ту информацию, которую вы ожидаете. Например, вы предполагаете, что свойство DriveType содержит строку Removable, а в действительности оно содержит числовое значение «5». И, наконец, ошибки выполнения по степени сложности решения находятся примерно посередине между синтаксическими и логическими. Ошибки выполнения не всегда можно предсказать заранее. Например, вы можете написать скрипт для подключения к удаленному компьютеру, но при попытке подключения выяснится, что у вас нет на это разрешения. На этом уроке мы остановимся на ошибках выполнения, которые не всегда можно предупредить, но можно предугадать возможность их возникновения во время написания скрипта.

$ErrorActionPreference

Многие командлеты способны продолжать выполняться даже при возникновении ошибок. Например, рассмотрим такую команду:

```
Get-WmiObject Win32_Service –computer Server1,Server2,Server3
```

Предположим, что во время запуска команды Server2 оказался в режиме оффлайн. Командлет успешно установил соединение с Сервером 1, но попытка подключения к Серверу 2 оказалась неудачной. Это непрерывающая ошибка. Командлет не может выполнить действие для Сервера 2, но он не прекращает работу и переходит к Серверу 3. Когда командлет сталкивается с ошибкой такого типа, он проверяет, что можно предпринять для ее решения. Информация о действиях, предпринимаемых для решения бесконечных ошибок, хранится во встроенной в оболочку переменной $ErrorActionPreference. Эта переменная может принимать одно из четырех значений:

• Continue является значением по умолчанию и дает командлету инструкцию: выдать сообщение об ошибке и продолжить работу.

• Stop принуждает командлет превратить непрерывающую ошибку в прерывающее исключение, в результате чего выполнение командлета полностью прекращается.

• SilentlyContinue дает командлету инструкцию продолжать работу без отображения сообщения об ошибке.

• Inquire – дает командлету инструкцию обратиться к пользователю с запросом, чтобы пользователь мог самостоятельно выбрать, что следует сделать – остановить работу или продолжить.

Не забывайте, что все это относится только к непрерывающим ошибкам, с которыми может столкнуться командлет. Но если командлет столкнулся с прерывающим исключением, его работа незамедлительно прекращается, вне зависимости от того, как настроена переменная $ErrorActionPreference.

Порочная практика

В целом сообщения об ошибках – полезная вещь. Оболочка обычно выдает четкие, информативные сообщения об ошибках, и если вы прочитаете внимательно такое сообщение, вы поймете, где именно произошла ошибка, и что можно предпринять для ее решения. Поэтому, отказываться от сообщений об ошибках нецелесообразно. Однако многие администраторы часто добавляют в начало сценария команду:

```
$ErrorActionPreference = "SilentlyContinue"
```

Обычно они делают это потому, что используют такой командлет, как Get-WmiObject, с помощью которого они могут предусмотреть возникновение ошибок, поэтому, сообщения об ошибках им не нужны. Однако добавление вышеуказанной команды в начало сценария отменяет появление сообщений об ошибках для всего скрипта. Зачастую при выполнении скрипта возникают ошибки совсем другого рода, о которых администратор не узнает, так как сообщения об ошибках отключены. В результате скрипт работает не так, как планировалось, и а его настройку уходит масса времени, так как без сообщения об ошибках весьма сложно определить, в каком месте необходимо внести изменения в сценарий. Существует очень мало ситуаций, в которых сообщения об ошибках не нужны вообще. Поэтому, если вы решили отключить уведомления, лучше сделать это для одного, конкретного командлета.

–ErrorAction

Все командлеты Windows PowerShell поддерживают ряд общих параметров, которые обрабатываются непосредственно самой оболочкой, и которые разработчик командлетов не должен прописывать вручную. Эти параметры не перечисляются в справочнике для каждого командлета, их список можно увидеть в разделе Common Parameters. Прочитать информацию о них можно, набрав команду:

```
Help about_commonparameters
```

Один из общих параметров - это –ErrorAction, имеющий псевдоним EA. Этот параметр может принимать те же четыре значения, что и переменная $ErrorActionPreference. Однако, в отличие от этой переменной, данный параметр осуществляет выявление и отображение ошибок только для одного командлета. Поэтому, если вы используете командлет, например, Get-WmiObject, вы можете отключить уведомления об ошибках именно для этого командлета с использованием параметра –ErrorAction или –EA:

```
Gwmi Win32_Service –computer Server1,Server2,Server3
–EA SilentlyContinue
```

Перехват ошибок

Вы можете дать оболочке указание уведомлять о прерывающих исключениях и запускать команды в ответ на эти ошибки. Это называется «перехват ошибок» и означает, что вы самостоятельно определяете, какое действие предпринять в ответ

на возникшую ошибку, а не доверяете оболочке произвести действие по умолчанию.

Вы можете перехватывать только прерывающие исключения. Непрерывающие ошибки вы перехватывать не можете, даже тогда, когда видите уведомление об ошибке.

Если вы предусмотрели заранее, что командлет может столкнуться с ошибкой, которую вы хотите перехватить, вы должны указать параметр – ErrorAction со значением Stop. Ни одно другое действие не гарантирует возможность перехвата.

Если вы используете –ErrorAction Inquire, перехватываемое исключение генерируется только в том случае, если пользователь выбирает опцию «остановить командлет». После того, как вы дали командлету инструкцию превращать непрерывающие ошибки в прерывающие перехватываемые исключения посредством –EA Stop, у вас есть два варианта перехвата ошибки: с помощью конструкции Trap или конструкции Try…Catch

Прежде, чем мы перейдем к конструкциям Trap и Try…Catch, вспомните, что вы знаете об областях действия в Windows PowerShell.

Сама оболочка – это глобальная область действия. Каждый скрипт, который вы запускаете, создает свою собственную область действия. Функции (которые мы рассмотрим чуть позже) тоже содержатся в своих областях действия. Поэтому, если вы выполняете сценарий, который содержит функцию, вы имеете дело с деревом областей действия, которое выглядит примерно так:

• Глобальная область действия

• Область действия скрипта

• Область действия функции

Область действия функции является дочерней по отношению к скриптовой, а скриптовая, в свою очередь, является дочерней по отношению к глобальной. Если прерывающее исключение происходит внутри функции, оболочка сначала проверяет, собираетесь ли вы перехватить это исключение внутри этой же области, т.е. внутри функции. Если у вас нет способа сделать это, оболочка покидает область действия функции и переносит прерывающее исключение в родительскую область действия, в данном случае в скриптовую. С точки зрения скрипта, функция сама сгенерировала это исключение, поэтому, оболочка проверяет, есть ли внутри скрипта инструменты для того, чтобы перехватить и исправить исключение. Если таких инструментов не обнаружено, оболочка покидает область действия скрипта и обращается к глобальной области действия. С точки зрения глобальной области действия, исключение было сгенерировано всем скриптом, и оболочка начинает искать инструменты для перехвата и исправления ошибки в глобальной области действия. Это может показаться довольно сложным, но очень важно решить, какие действия предпринять в отношении ошибки. Говоря в общем, вы должны стараться перехватить и исправить ошибку в той области действия, в которой она возникла. Например, если функция содержит командлет Get-WmiObject, и вы хотите перехватить и исправить ошибки для этого командлета, конструкция перехвата ошибок должна быть включена в функцию. Таким образом, оболочке не

придется выходить за пределы области действия функции для исправления ошибки.

Trap

Конструкция для перехвата ошибок обычно указывается в скрипте перед исключением, возникновение которого вы предусматриваете. Простая конструкция может выглядеть примерно так:

```
trap {
write-host "Exception trapped!" -fore yellow -back black
continue
}
get-wmiobject win32_process -comp NotOnline -ea stop
```

В конце конструкции вы можете указать одно или два ключевых слова:

• Continue – продолжает выполнение команды, которая указана в скрипте после исключения, не выходя за пределы текущей области действия.

• Break – выходит за пределы текущей области действия и ищет средства для перехвата ошибки и ее исправления в родительской области действия.

Например, рассмотрим короткий скрипт:

```
trap {
write-host "Exception trapped in the script"
-fore green -back black
continue
}
function test {
trap {
write-host "Exception trapped in the function"
-fore yellow -back black
break
}
write-host "I am inside the function"
get-wmiobject win32_process -comp NotOnline -ea stop
write-host "I am still inside the function"
}
write-host "Running the function"
test
write-host "Finished running the function"
```

Ошибка возникает в командлете Get-WmiObject. Так как перехват ошибки указан в области действия функции, этот перехват выполняется. Перехват заканчивается ключевым словом break, поэтому оболочка выходит за пределы области действия функции и передает ошибку в родительскую область. Перехват определен и здесь, поэтому он выполняется. Перехват заканчивается ключевым словом continue, поэтому оболочка переходит к выполнению команды, указанной после перехвата. С точки зрения скрипта, ошибка произошла в функции test, поэтому выполнение

команды продолжается в этой же области действий. Выходные данные скрипта будут выглядеть так:

```
Running the function
I am inside the function
Exception trapped in the function
Exception trapped in the script
Finished running the function
```

Попробуйте запустить скрипт, указанный в примере, и убедитесь, что вы поняли, почему выходные данные будут выглядеть именно так.

Ниже вы видите тот же самый скрипт с одной небольшой разницей: Перехват внутри функции сейчас заканчивается ключевым словом continue, а не break:

```
trap {
write-host "Exception trapped in the script"
-fore green -back black
continue
}
function test {
trap {
write-host "Exception trapped in the function"
-fore yellow -back black
continue
}
write-host "I am inside the function"
get-wmiobject win32_process -comp NotOnline -ea stop
write-host "I am still inside the function"
}
write-host "Running the function"
test
write-host "Finished running the function"
```

Выходные данные сейчас выглядят так:

```
Running the function
I am inside the function
Exception trapped in the function
I am still inside the function
Finished running the function
```

Вы поняли, почему? Когда произошла ошибка, был выполнен перехват внутри функции. Однако эта команда заканчивалась ключевым словом continue, поэтому, оболочка не вышла за пределы области действия функции и продолжила выполнений следующей команды. Поэтому, на этот раз отобразилась строчка I am still inside the function. Функция была выполнена до конца, а ошибка не была передана в скрипт. Поэтому, перехват в скрипте не был выполнен.

При выполнении перехвата ошибок крайне важно помнить о дереве областей действия. Это может показаться сложным, но, потратив некоторое время на

изучение поведения оболочки в вышеописанных ситуациях, вы сможете избежать лишней путаницы и ошибок.

Конструкция перехвата, с которой вы только что познакомились, была общей, то есть, она подходит для любого типа прерывающих исключений. Windows PowerShell также позволяет определять перехват особых видов исключений, но для этого вы должны знать определенные типы классов исключений .NET Framework. Например:

```
trap [System.Management.Automation.CommandNotFoundException]
{"Command error trapped"}
```

Этот перехват выполняется только для исключений типа System.Management.Automation.CommandNotFoundException. Оболочка позволяет указывать несколько конструкций для перехвата, каждая из которых предназначена для конкретного типа исключений. Это один из способов выбирать разные инструменты для решения разных проблем. Однако определить нужное имя класса .NET Framework может оказаться сложно.

Также обратите внимание, что перехват – это отдельная область действия, как и функция. Перехват может получать доступ к переменным за пределами своей области действия, по общим правилам. Однако любые переменные, которые создаются или устанавливаются перехватом, относятся только к нему. Если вы используете перехват, чтобы изменить переменную, расположенную в родительской для него области действия, используйте командлет Set- Variable с параметром –scope.

Try…. Catch

Конструкция Try…Catch является более простой в использовании, чем конструкция Trap. Типичная конструкция такого типа выглядит так:

```
try {
gwmi win32_service -comp notonline -ea stop
} catch {
write-host "Error!"
}
```

Блок Try определяет команду, при выполнении которой, как вы предполагаете, может произойти ошибка, и для которой вы задали ErrorAction команды Stop. Если ошибка произошла, начинает выполняться код, указанный внутри блока Catch.

Так же, как и в случае с конструкцией Trap, вы можете указать несколько блоков Catch, каждый из которых будет выявлять определенный тип ошибок. Чтобы узнать больше об этой технике, запустите Help about_try_catch_finally.

После того, как выполнение кода внутри блока Catch закончилось, начинается выполнение команды, указанной после блока Try…Catch. Также вы можете задать блок Finally:

```
try {
gwmi win32_service -comp notonline -ea stop
} catch {
```

```
write-host "Error!"
} finally {
write-host "This executes either way"
}
```

Код внутри блока Finally выполняется независимо от того, произошла ошибка или нет. Например, вы можете использовать этот блок для завершения соединения с базой данных, вне зависимости от того, произойдет ожидаемая ошибка или нет.

Извлечение ошибок

При использовании конструкций Trap и Try вам, возможно, понадобится доступ к тому исключению, которое стало причиной запуска этой конструкции. Существует два способа добиться этого. Встроенная переменная $error содержит ошибки, встречающиеся в оболочке. $Error[0] – это последняя ошибка, $Error[1] – предпоследняя и.т.д. Также вы можете использовать параметр -ErrorVariable (или –EV) для того чтобы выявить ошибку, сгенерированную этим командлетов в переменную. Например:

```
Get-Content does-not-exist.txt –EA Stop –EV myerr
```

Обратите внимание, что перед именем переменной myerr в данном контексте не используется символ $. Если происходит ошибка, командлет помещает информацию о ней в указанную переменную. Вы можете использовать ее следующим образом:

```
try {
gwmi win32_service -comp notonline -ea stop -ev myerr
} catch {
$myerr | out-file c:\errors.txt -append
}
```

Обратите внимание, что myerr не включает значок доллара, когда указывается в сочетании с параметром –EV, потому что на этом этапе вы лишь указываете имя переменной, а $ технически не является частью ее имени. Позже, когда вы действительно будете использовать саму переменную, знак доллара будет ставиться перед ее названием, чтобы указать оболочке, что это именно переменная, а не что-то другое.

Отладка

Логические ошибки, возможно, являются самыми сложными в исправлении. Обычно их нельзя предусмотреть заранее, и они не всегда влекут за собой уведомление об ошибке. Просто в результате логической ошибки скрипт работает не так, как было запланировано. Цель отладки скрипта – выявить причину возникновения логических ошибок, исправить их и протестировать результат.

Как уже упоминалось на предыдущем занятии, цель отладки – решение логических ошибок. Самая распространенная причина их возникновения – это свойство переменной или объекта, имеющее не то значение, которое вы предполагали. Таким образом, отладка предназначена для того, чтобы помочь вам увидеть, что в действительности содержат переменные и свойства и сравнить ваши ожидания с действительностью. Однако, прежде чем начать отладку, вам необходимо понять,

что должна делать каждая строчка вашего сценария. Вы не сможете определить, были ли ваши ошибочными, пока не определитесь с этими ожиданиями.

Прежде чем приступить к отладке, сядьте с вашим сценарием и листом бумаги. Запишите, какие входящие значения используются в вашем сценарии. Пройдитесь по всему сценарию, строчка за строчкой. Запишите, каких результатов вы ожидаете от каждой строчки, и как вы можете проверить правильность выполнения каждой операции. Запишите значения, которые содержит каждая переменная и каждое свойство.

На заметку: Когда вы приобретете определенный опыт, лист бумаги, возможно, уже не понадобится. Тем не менее, первое время рекомендуется записывать каждое действие, чтобы облегчить себе работу.

Отладочный вывод

Одна из технологий отладки заключается в изучении трассировочных сообщений выходных данных вашего скрипта. Это поможет определить, какие значения в действительности имеют свойства и переменные. При сравнении трассировочных сообщений с вашими ожиданиями, которые вы записали на листе бумаги, несовпадений быть не должно. Если что-то не совпадает, возможно, именно это и послужило причиной ошибки.

Оболочка не создает трассировочные сообщения автоматически. Чтобы получить их, необходимо использовать в сценарии командлет Write-Debug. Например, каждый раз, когда вы меняете значение переменной, вы должны прописывать новое значение таким образом:

```
$var = Read-Host "Enter a computer name"
Write-Debug "`$var contains $var
```

Обратите внимание, что в данном примере использования Write-Debug переменная и ее содержание помещены в двойные кавычки. Однако в первом случае переменная указана без знака $. В результате имя переменной отобразится как есть, вслед за ним идет содержимое переменной и ее значение. Выходные данные такой отладки весьма полезны, так как содержат имя переменной и ее значение. Также следует использовать Write-Debug везде, где скрипт принимает решение. Например, рассмотрим следующий отрывок:

```
If ($var –notlike "*srv*") {
Write-Host "Computer name is not a server"
}
You might modify this as follows:
If ($var –notlike "*srv*") {
Write-Debug "$var does not contain ‘srv’"
Write-Host "Computer name $var is not a server"
} else {
Write-Debug "$var contains ‘srv’"
}
```

Обратите внимание, что был добавлен блок Else, поэтому скрипт будет генерировать трассировочные сообщения независимо от того, какое логическое

решение будет принято. По умолчанию оболочка скрывает выходные данные Write-Debug. Чтобы они отображались, поместите переменную $DebugPreference в начало скрипта:

```
$DebugPreference = 'Continue'
```

Теперь выходные данные Write-Debug будут отображаться. Когда вы закончите отладку скрипта, удалять команды Write-Debug будет не нужно. Вместо этого снова скройте их выходные данные:

```
$DebugPreference = 'SilentlyContinue'
```

Таким образом, команда Write-Debug останется в скрипте на тот случай, если в дальнейшем она снова понадобится для отладки.

На заметку: Выходные данные Write-Debug отличаются от обычных выходных данных в окне консоли Windows PowerShell. Сторонние скриптовые редакторы могут перенаправить эти данные в другую панель, окно или страницу, позволяя таким образом отделить трассировочные сообщения от других данных.

Пошаговый отладчик

Если скрипт очень длинный, то просмотр сотен строк трассировочных сообщений может стать утомительным и занять много времени. Поэтому в некоторых случаях вы можете воспользоваться пошаговым отладчиком. Пошаговый отладчик позволяет выполнять скрипт поэтапно – одна строка за один раз. Перед выполнением строки вы можете сделать паузу и просмотреть содержимое переменных, объектов, свойств и.т.д. Пошаговый отладчик запускается с помощью команды:

```
Set-PSDebug –step
```

После того, как вы завершили работу, отключите отладчик с помощью команды:

```
Set-PSDebug –off
```

Во время выполнения скрипта вы можете дать пошаговому отладчику команду пойти вперед и выполнить следующую строку или сделать паузу. Во время паузы оболочка показывает различные подсказки, которые напоминают о том, что вы находитесь внутри скрипта. Вы можете выполнять команды как обычно и одновременно просматривать информацию об объектах и переменных. Чтобы продолжить выполнение скрипта, запустите Exit.

Контрольные точки

Одним из недостатков пошагового отладчика является отсутствие возможности пропускать те части скрипта, в корректной работе которых вы не сомневаетесь. Если ваш скрипт состоит из 200 строк, и вы знаете, что проблема находится где-то в конце, нажимать Yes столько раз подряд довольно утомительно.

Чтобы облегчить ситуацию, Windows PowerShell предлагает возможность создания контрольных точек. С контрольными точками скрипт выполняется как обычно. Однако когда оболочка сталкивается с условием контрольной точки, которое вы задали, она автоматически приостанавливает скрипт так же, как и пошаговый

отладчик. Так вы можете изучить значения свойств или переменные, после чего возобновить выполнение сценария.

Условия контрольных точек могут быть следующими:

• Остановить выполнение сценария при достижении определенной строки.

• Остановить выполнение сценария при чтении определенной переменной.

• Остановить выполнение сценария, когда определенная переменная изменяется или пишется.

• Остановить выполнение сценария, когда определенная переменная читается или пишется.

• Остановить выполнение сценария при выполнении опрделенной команды или функции.

Также вы можете указать конкретное действие, представляющее собой набор команд Windows PowerShell, которое вы хотели бы выполнить при достижении контрольной точки.

Управление контрольными точками осуществляется с помощью командлетов:

• Set-PSBreakpoint – создание нового условия контрольной точки.

• Remove-PSBreakpoint – удаление условия контрольной точки.

• Disable-PSBreakpoint - прекратить действие условия контрольной точки без его удаления.

• Enable-PSBreakpoint – возобновить действие остановленного условия контрольной точки.

• Get-PSBreakpoint – извлечь одно или несколько условий контрольной точки.

Контрольные точки позволяют вам задавать условия, при которых выполнение сценария будет приостановлено (или будет произведено другое действие).

Отладка в ISE

Отладчик Windows PowerShell обеспечивает базовую визуальную поддержку для отладки, в первую очередь, в сочетании с контрольными точками. Визуальных средств для создания контрольной точки нет. Однако когда вы создаете контрольную точку по номеру строки для сценария (используя параметр –script командлета Set-PSBreakpoint), ISE отображает эту контрольную точку внутри скрипта, подчеркивая соответствующую строку. Когда вы запускаете этот скрипт, ISE позволяет наводить курсор мышки на имена переменных, чтобы увидеть их содержимое.

Модуляризация

По мере того, как вы будете создавать все более сложные и полезные сценарии, вам все чаще будет требоваться повторное использование одних и тех же фрагментов. Модуляризация – это технология запаковки фрагментов кода в более простые и легко читаемые единицы. Windows PowerShell предлагает несколько уровней модуляризации, каждый из которых сложнее предыдущего, и в то же время обеспечивает больше гибкости и функциональности.

Модуляризация предназначена для создания полезных, независимых компонентов для многократного использования. Цель модуляризации – ускорить процесс написания скриптов, позволяя повторно использовать готовые фрагменты из предыдущих проектов. В Windows PowerShell базовой формой модуляризации является функция. Вы уже встречались с понятием функции ранее – сейчас мы рассмотрим их назначение более подробно.

Функция должна имеет настолько узкое предназначение и быть настолько независимой, насколько это возможно. Создавая функцию для выполнения одной конкретной задачи, вы делаете возможным ее повторное использование там, где выполнение этой же задачи снова потребуется.

Задачи могут включать такие вещи как написание базы данных, проверка возможности соединения с удаленным компьютером, валидация имени сервера и.т.д. В идеале функции должны настраивать себя на выполнение одной задачи так же, как это делает большинство командлетов.

Вы можете воспринимать командлеты как особую форму модуляризации, хотя настоящие командлеты требуют знания.NET Framework.

Для выполнения своей задачи функция должна получить особенный вид входящих данных. Функция никогда не должна пытаться получить доступ к информации, находящейся за пределами себя самой – это во многом снизит возможность ее повторного использования. Например, предположите, что вы имеете скрипт, содержащий функцию Get-ServerName. Если предполагается, что эта функция может читать переменные, содержащиеся в родительской оболочке, любой скрипт, в котором используется эта функция, должен содержать соответствующие переменные. Такая зависимость усложняет многократное использование функции в разных сценариях. Однако, поддерживая автономность функции, вы сможете с легкостью использовать ее в самых разных ситуациях.

Функции должны выдавать данные таким образом, чтобы их можно было использовать в разных ситуациях. В общем, это означает возможность передачи выходных данных по конвейеру. Например, функция, которая выводит данные напрямую в CSV файл, может использоваться только тогда, когда вам нужен результат в виде CSV файла. Функция, которая выводит выходные данные в конвейер, может использоваться в различных ситуациях, так как вы можете передать эти данные таким командлетам как Export- CSV, Export-CliXML, ConvertTo-HTML и другим.

Базовые функции

Базовая функция – это простейшая форма модуляризации. Она не принимает никаких входящих данных, что означает, что для ее выполнения не требуется никакой дополнительной информации. Она производит выходные данные, которые могут передаваться по конвейеру другим командлетам. Базовая функция указывается с функцией ключевых слов, а содержимое этой функции помещается в фигурные скобки:

```
Function Do-Something {
# function code goes here
}
```

Функции должны быть определены до того, как они могут быть использованы, а значит зачастую функции необходимо указывать в самом начале сценария. Чтобы запустить функцию, которая была определена, используйте ее имя, так же, как если бы это был командлет:

```
Do-Something
```

Командлет Write-Output – это верный способ извлечения выходных данных из функции. Этот командлет с псевдонимом Write отправляет объекты в конвейер, где они могут быть использованы другими командлетами. Существует три способа использования Write-Output. Предположим, вы имеете выходные данные в виде переменной с именем $var. Все три примера будут идентичными с функциональной точки зрения:

```
Write-Output $var
Write $var
$var
```

Помимо этого, функция может возвращать выходные данные, используя ключевое слово Return:

```
Return $var
```

Return является особенным в том плане, что он передает все, что получает в конвейер, после чего незамедлительно закрывает функцию.

Не используйте Write-Host для извлечения выходных данных из функции. Write-Host не передает данные в конвейер, напротив, он отправляет текст напрямую в окно консоли. Текст нельзя передать по конвейеру другому командлету, а значит, возможность повторного использования функции ограничивается.

Параметризованные функции

Параметризованная функция принимает базовую функцию и добавляет возможность передавать информацию в функцию. Как правило, функция не должна иметь жестко запрограммированных параметров, таких как имена компьютеров или имена пользователей. Вместо этого функции должна принимать эту информацию в качестве входящих данных – таким образом функция подойдет для использования в большем количестве ситуаций.

Существует два способа определить входящие параметры функции. Первый – это часть указание их как части функции:

```
Function Do-Something ($computername,$domainname) {
# function code goes here
}
```

На заметку: Старайтесь указывать имя функции в виде «глагол-существительное в единственном числе», так же, как это делается с командлетами. Однако следите за тем, чтобы случайно не присвоить функции имя существующего командлета.

Второй, и более предпочтительный способ определения входящих параметров функции выглядит так:

```
Function Do-Something {
```

```
Param(
$computername,
$domainname
)
# function code goes here
}
```

Этот метод предпочтительнее потому, что он легче читается, а также потому, что здесь используется условное обозначение, применяемое оболочкой для сохранения функции в памяти. Обратите внимание, что жестких правил форматирования здесь нет – вы также можете использовать следующую форму:

```
Function Do-Something {
Param($computername,$domainname)
# Function code goes here
}
```

Вынесение каждого параметра в отдельную строку упрощает чтение. Особенно, если вы следуете рекомендациям по оформлению входящих данных и значений по умолчанию для ваших параметров:

```
Function Do-Something {
Param(
[string]$computername = 'localhost',
[string]$domainname = 'contoso.com'
)
# function code goes here
}
```

Так же, как и в случае с параметризованными скриптами, вы можете обозначить параметры, значения по умолчанию которых будут выдавать пользователю подсказки или сообщения об ошибке, если значение не обеспечивается:

```
Function Do-Something {
Param(
[string]$computername = $(Read-Host 'Computer name'),
[string]$domainname = $(throw 'Domain name required.')
)
# function code goes here
}
```

Когда вы запускаете параметризованную функцию, вы можете передавать входящие значения с помощью:

```
Do-Something localhost 'contoso.com'
```

Обратите внимание, что параметры не разделяются запятой, как в других языках программирования; функции, как и командлеты, разделяют параметры пробелами. Также вы можете передать входящие значения, используя имена параметров, что позволяет располагать параметры в любом порядке:

```
Do-Something –domainname contoso –computername localhost
```

Передача данных конвейером

Итак, вы узнали, как передать входящие значения функции с помощью параметров. Но функции также могут принимать данные по конвейеру. В нормальной параметризованной функции оболочка автоматически входящие по конвейеру данные в специальную переменную, которая называется $input. Указывать $input в качестве параметра не нужно. Однако имеет смысл перечислить элементы в $input, так как зачастую она содержит несколько объектов. Для этого лучше всего использовать конструкцию ForEach. Например:

```
function Do-Something {
param (
$domain = 'contoso.com'
)
foreach ($computer in $input) {
write "Computer $computer is in domain $domain"
}
}
'localhost','server1' | do-something -domain adatum
```

Обратите внимание, что входящие по конвейеру данные о двух именах компьютера помещаются в $input, тогда как "adatum" помещается в стандартный параметр $domain.

Это лишь один из способов принятия функцией входящих данных по конвейеру. Более эффективным и легким способом является использование фильтров.

Функции-фильтры

Функции-фильтры предназначены специально для приема входящих данных из конвейера. Такая функция включает в себя три именованных скриптовых блока:

• BEGIN: этот блок выполняется один раз при обращении к функции. Вы можете использовать его для выполнения любой задачи по настройке, которую требует функция, например, для соединения с базой данных.

• PROCESS: этот блок выполняется один раз для каждого объекта, входящего по конвейеру. Внутри блока специальная переменная $_ содержит текущий объект.

• END: этот блок выполняется один раз, после того как все входящие объекты были обработаны. Его можно использовать для завершения работы, например, для закрытия базы данных.

Необязательно использовать все три скриптовых блока – включайте в ценарий только те, которые вам необходимы. Некоторые администраторы предпочитают включать все три блока, даже если некоторые из них не используются – это тоже допустимо. Например:

```
function Do-Something {
param (
$domain = 'contoso.com'
)
BEGIN {}
```

```
PROCESS {
$computer = $_
write "Computer $computer is in domain $domain"
}
END {}
}
'localhost','server1' | do-something -domain adatum
```

Скриптовый блок PROCESS обычно работает так же, как встроенная конструкция ForEach, автоматически перечисляя входящие объекты, и помещая каждый объект по очереди в переменную $_. Обратите внимание, что обычные параметры тоже поддерживаются.

самостоятельное конструирование выходных данных

Функции-фильтры, с которыми вы только что познакомились, запускали команды Windows PowerShell и позволяли этим командам помещать выходные данные в конвейер. Таким образом, выходные данные этих команд превращались в выходные данные функций. Например, в предыдущих примерах сюда входили одно или два обращения к Get-WmiObject.

Windows PowerShell не всегда хорошо справляется с отображением выходных данных, когда в конвейере содержится несколько разных типов объектов, например, Win32_OperatingSystem и Win32_BIOS одновременно. Одним из способов исправления ситуации может стать самостоятельное конструирование выходных данных. Например, вы можете вывести текст и сформировать заголовки столбца, а затем заполнить каждую строку выходными данными. Windows PowerShell даже предлагает специального оператора форматирования –f, который упрощает процесс создания данных такого типа.

В двух предыдущих примерах выходными данными являлся текст, точнее, объекты String. Проблема здесь заключается в том, что текст сложно использовать повторно. Например, вам может понадобиться вывести данные в файл CSV, а не в текстовую экранную таблицу. Используя в качестве отправной точки два вышеописанных подхода, вы будете вынуждены переписывать часть функции, чтобы вывести данные в файл CSV. А если позже вам понадобятся выходные данные в формате XML, вы будете переписывать ее снова.

Как вы уже знаете, Windows PowerShell – это объектоориентированная оболочка, которая работает с объектами гораздо лучше, чем с текстом. В двух предыдущих примерах извлекались два класса WMI: Win32_OperatingSystem и Win32_BIOS. Поскольку это два разных класса, оболочка не может просто отобразить их в одной строчке таблицы – по крайней мере, с использованием средств форматирования, предлагаемых по умолчанию. Решением может стать вывод данных в виде текста, но, как мы знаем, это не лучший способ для Windows PowerShell.

Еще одним способом скомбинировать информацию разных типов может стать создание пустого пользовательского объекта:

```
$obj = New-Object PSObject
```

"PSObject" – это очень простой объект, который выступает в роли пустого, чистого холста или полотна. Ему можно придавать любые свойства. Например, свойство ComputerName добавляется следующим образом:

```
$obj | Add-Member NoteProperty ComputerName $computername
```

Таким образом добавляется новое свойство типа NoteProperty. NoteProperty – это статическое значение, придаваемое объекту. Новое свойство NoteProperty называется ComputerName, а его значением может быть все, что вы поместите в переменную $computername.

Вы можете добавлять свойства до тех пор, пока объект не примет всю необходимую вам информацию. Затем вы отправляете этот пользовательский объект в конвейер:

```
Write $obj
```

Так как вы отправляете в конвейер всего один объект, Windows PowerShell может с легкостью сформировать выходные данные. Также вы можете передать этот объект другим командлетам для того, чтобы сформировать CSV-файл, затем HTML, и.т.д. Используя выходные данные функции в виде объектов, вы упрощаете использование этой функции в дальнейшем в различных ситуациях.

Разные способы решения

По мере того, как вы будете изучать примеры использования Windows PowerShell из разных источников, в том числе, из Интернета, вы все чаще будете осознавать, что одну и ту же задачу можно выполнить разными способами.

Так, в предыдущем примере вы видели, как с помощью New-Object и Add-Member можно создать новый пользовательский объект. Еще одним подходом является использование Select-Object для придания пользовательских свойств новому пустому объекту:

```
function Get-Inventory {
PROCESS {
$computer = $_
$os = gwmi win32_operatingsystem -comp $computer
$bios = gwmi win32_bios -comp $computer
$obj = new-object psobject
$obj | select @{Label='ComputerName';Expression={$computer}},
@{Label='SPVersion';Expression={$os.servicepackmajorversion}},
@{Label='BIOSSerial';Expression={$bios.serialnumber}},
@{Label='BuildNo';Expression={$os.buildnumber}}
}
}
gc names.txt | get-inventory
```

Нельзя сказать, что какой-то один из этих подходов является верным – они оба работают и оба обеспечивают одни и те же результаты. Так что можете смело использовать тот, который кажется вам более удобным. Когда вы лучше познакомитесь с работой Windows PowerShell, вы поймете, что многие задачи можно выполнить с помощью одной-единственной команды, хотя эта команда

может быть довольно сложной. Например, всю функцию Get-Inventory, показанную ранее, можно заменить одной сложной командой:

```
gwmi win32_operatingsystem -computer (gc names.txt) |
select @{Label='ComputerName';Expression={$_.__SERVER}},
@{Label='BuildNo';Expression={$_.BuildNumber}},
@{Label='SPVersion';Expression={$_.ServicePackMajorVersion}},
@{Label='BIOSSerial';Expression={
(gwmi win32_bios -comp $_.__SERVER).serialnumber
}}
```

Здесь первым запускается командлет Get-WmiObject. Его параметр –computerName получает выходные данные Get-Content, который читает текстовый файл, содержащий по одному имени компьютера в каждой строчке. WMI объект передается командлету Select-Object.

Первые три элемента в Select-Object создают пользовательские свойства объекта, которые используют свойства из Win32_OperatingSystem WMI объекта. Смысл этого – обеспечить пользовательские имена свойств, такие как ComputerName и BuildNo вместо того, чтобы использовать родные имена свойств класса WMI. Последний элемент в Select-Object создает пользовательское свойство, которое называется BIOSSerial. Это выражение свойства в действительности выполняет второй командлет Get-WmiObject для извлечения класса Win32_BIOS. Имя компьютера, передаваемое в этот Get-WmiObject, будет свойством __SERVER первого WMI объекта – это свойство содержит имя компьютера. Обратите внимание, что вся команда Get-WmiObject помещена в скобки – это заставляет оболочку выполнять команду, а скобки обозначают объект, который получится на выходе. Вслед за скобками идет точка, которая указывает на то, что мы хотим получить доступ к одному из элементов объекта, а затем имя этого элемента SerialNumber. Конечным результатом является то, что свойство SerialNumber объекта Win32_BIOS помещается в пользовательское свойство BIOSSerial в Select-Object.

Конечно, это сложная команда, но она наглядно демонстрирует, что Windows PowerShell позволяет выполнять сложные задачи без формального написания скриптов и программирования. Для прочтения и, тем более, написания такой команды, однако, требуется некоторый опыт. Поэтому вы вправе выбрать для себя такой способ работы, который вам кажется проще.

Расширенные функции

Расширенная функция является чуть более «продвинутой» по сравнению с функцией-фильтром. Точнее говоря, расширенная функция – это функция-фильтр, которая имеет дополнительные атрибуты для параметров, которые принимают входящие данные. За счет этих атрибутов расширенные функции выглядят и ведут себя практически так же, как командлеты, написанные на языке .NET Framework. Так указывается функция и параметр для очень простой расширенной функции:

```
function Get-ComputerDetails {
[CmdletBinding()]
param (
```

```
[parameter(Mandatory=$true,ValueFromPipeline=$true)]
[string[]]$computername
)
BEGIN {}
PROCESS {
```

. Вы заметите, что главное различие между расширенной функцией и фильтром заключается в наличии дополнительных атрибутов, таких как [CmdletBinding()], [parameter] и так далее. В данном примере параметр $computername был создан для приема одной или нескольких строк в качестве входных данных из конвейера, и является обязательным. При использовании этой технологии переменная $_ внутри блока PROCESS не является обязательной. Вместо нее входные данные могут помещаться в параметр $computername. Скриптовый блок PROCESS помещает один входящий объект в параметр $computername за один раз и выполняется один раз для каждого объекта.

Расширенные функции являются завершающим элементом модуляризации в Windows PowerShell: использование атрибутов параметров, которые выглядят как командлеты, позволит вам написать структуру, которая будет действовать практически так же, как и настоящие командлеты оболчки. При желании вы даже можете поместить справочник в функцию – как это сделать, вы узнаете в следующих разделах.

На заметку: Подробное изучение расширенных функций выходит за рамки данного курса. Однако вы можете найти более подробную информацию о них в интернете, и попытаться изучить ее самостоятельно.

Список литературы

1. Lee Holmes. Windows PowerShell Cookbook: The Complete Guide to Scripting Microsoft's New Command Shell, O'Reilly, 2010, ISBN-10: 0596801505,ISBN-13: 978-0596801502
2. Ed Wilson. Windows PowerShell 2.0 Best Practices, Best Practices (Microsoft), 2010, ISBN-10: 0735626464, ISBN-13: 978-0735626461
3. Попов А., Введение в Windows PowerShell, БХВ-Петербург,2009, ISBN5-9775-0283-4
4. MS Official course 10325A Automating Administration with Windows PowerShell® 2.0, Microsoft, 2010