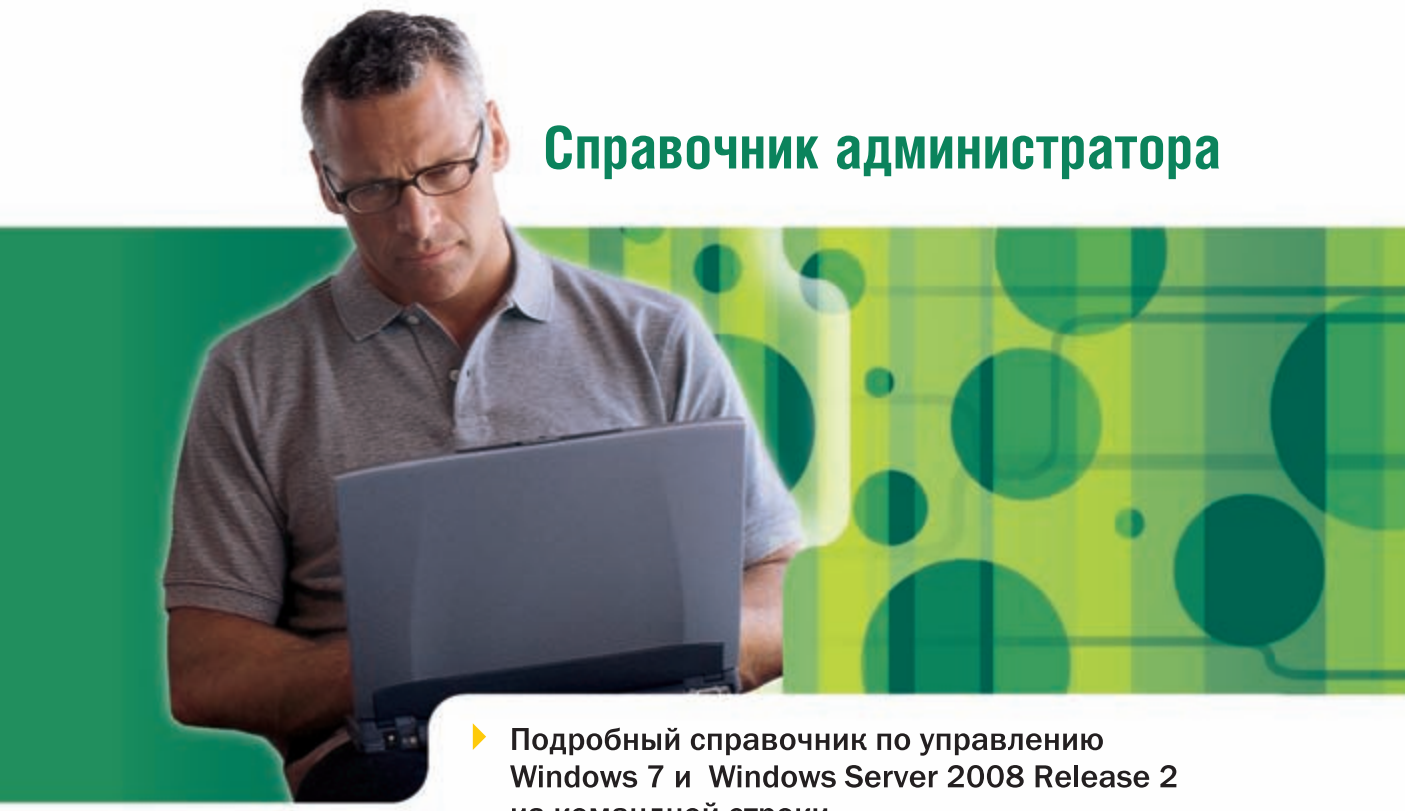
Уильям Р. Станек

Windows® PowerShell™ 2.0

Справочник администратора



- Подробный справочник по управлению Windows 7 и Windows Server 2008 Release 2 из командной строки
- Таблицы, пошаговые инструкции, списки параметров

IT Professional

РУССКАЯ РЕДАКЦИЯ Microsoft bhv®

William R. Stanek

Windows® PowerShell™ 2.0

Administrator's Pocket Consultant

Microsoft® Press

Уильям Р. Станек

Windows® PowerShell™ 2.0

Справочник администратора

РУССКАЯ РЕДАКЦИЯ

bhv®

2010

УДК 681.3.06
ББК 32.973.26–018.2
С76

Станек Уильям Р.

С76 Windows PowerShell 2.0. Справочник администратора / Пер. с англ. — М. : Издательство «Русская редакция» ; СПб. : БХВ-Петербург, 2010. — 416 стр. : ил.

ISBN 978-5-7502-0396-3 («Русская Редакция»)
ISBN 978-5-9775-0534-5 («БХВ-Петербург»)

Этот краткий справочник содержит ответы на любые вопросы, связанные с администрированием Windows из командной строки. Он отражает революционные нововведения, которые отличают Windows PowerShell 2.0 от предыдущих версий этой командной оболочки. За короткое время PowerShell стала полноценной средой командной строки с развитым языком программирования и мощнейшими возможностями. В этой книге системные администраторы найдут множество рецептов, которые позволят им задействовать богатый арсенал средств PowerShell 2.0 для решения своих повседневных задач, включая управление компьютерами и сетями, а также диагностику и устранение неполадок.

В Windows 7, Windows Server 2008 Release 2 и более поздние выпуски операционных систем Windows оболочка PowerShell 2.0 встроена, а в Windows XP, Windows Server 2003, Windows Vista и Windows Server 2008 ее можно установить. Для каждой версии Windows, включая 32- и 64-разрядные редакции, на сайте Microsoft Download Center (http://download.microsoft.com) доступны соответствующие сборки PowerShell.

Книга адресована системным администраторам Windows, разработчикам и всем, кому требуется управлять компьютерами с помощью командной оболочки PowerShell 2.0.

Издание состоит из 14 глав.

УДК 681.3.06
ББК 32.973.26–018.2

Оглавление

Введение

Третье издание книги, посвященной командной строке Windows из серии Справочник администратора задумано как краткий и удобный источник информации для администраторов Windows, разработчиков и всех, кому требуется управлять компьютерами с помощью командной оболочки *PowerShell 2.0*. В этой книге есть все, что нужно для успешного решения базовых административных задач с использованием оболочки *PowerShell* 2.0.

Говоря коротко, эта книга — источник, к которому вы обращаетесь за разрешением любых вопросов, связанных с администрированием Windows из командной строки. В ней рассказано об основных процедурах и часто решаемых задачах, описаны типичные примеры и приведены списки параметров, пусть не всегда полные, но достаточно представительные. Надеюсь, что мой справочник поможет вам легко и быстро справляться с повседневной работой, возникающими проблемами и с реализацией более сложных задач администрирования, таких как автоматизация мониторинга, анализ утечек памяти, разметка дисков на разделы, управление Active Directory и устранение неполадок в сетях.

Структура книги

Эта книга — справочник, а не учебник, поэтому ее основу составляют решения конкретных задач, а не описание компонентов Windows. Важная черта справочника — легкость, с которой в нем удается отыскать нужную информацию. Книга содержит развернутое оглавление. Выполняемые задачи расписаны по пунктам и снабжены списками параметров, таблицами и ссылками на другие разделы книги. Книга состоит из 14 глав.

В гл. 1 дается обзор концепций, инструментов и методик администрирования из командной строки *PowerShell*. Windows *PowerShell* предоставляет множество утилит командной строки, помогающих в управлении повседневными операциями. Гл. 2 поможет вам эффективно использовать командную оболочку. В ней подробно рассматриваются процедуры запуска командной оболочки с параметрами, форматирование вывода, способы перенаправления вывода и конвейеризация команд. В гл. 3 обсуждаются профили и способы загрузки рабочей среды. Вы научитесь расширять функциональность *PowerShell* с помощью дополнительных оснасток и модулей. В гл. 4 рассказывается об удаленном исполнении команд, удаленных сеансах и фоновых заданиях. При удаленной работе команды, которые вы вводите в Windows PowerShell на своем компьютере, исполняются на удаленных компьютерах. В гл. 5

рассматриваются многие из важнейших средств администрирования. Вы научитесь устанавливать переменные, работать с выражениями, управлять строками, массивами и наборами. В главе 6 изучаются псевдонимы, функции и объекты, которые повышают эффективность труда и позволяют решать практически любые административные задачи. В гл. 7 вы узнаете, как максимально эффективно использовать сценарии, профили и команды, научитесь работать с циклами и ветвлением по условию. Гл. 8 посвящена управлению серверными ролями, службами ролей и функциями *PowerShell*. В 9 гл. рассказывается о способах инвентаризации компьютеров и оборудования, а также обнаружения неполадок, требующих вмешательства администратора. Гл. 10 освещает методы управления файловыми системами, средствами защиты и аудита. В *PowerShell* управление большими наборами файлов и папок не сложнее, чем манипулирование отдельными файлами. В гл. 11 рассказывается о конфигурировании служб печати и TCP/IP-сетей, а также управления ими и устранения проблем. В гл. 12 рассматриваются средства управления реестром и его защиты. Вы научитесь читать и записывать параметры реестра, просматривать и устанавливать списки управления доступом, а также настраивать аудит реестра. В гл. 13 изучаются средства и методы мониторинга компьютеров и повышения производительности. Заключительная гл. 14 содержит подробности об оптимизации производительности. Изучив ее, вы освоите способы выявления и устранения неполадок в системах.

Условные обозначения

Чтобы текст было удобнее читать, я ввел в него несколько дополнительных элементов. Код и листинги набраны `моноширинным` (фиксированным) шрифтом. Команда или текст, которые нужно ввести с клавиатуры, выделены **полужирным** начертанием. Сетевые адреса и новые термины выделяются *курсивом*. Дополнительные сведения приводятся в следующих разделах.

Примечание комментарий к описываемой процедуре или операции.

Совет подсказка, дополнительная информация или рекомендация, связанная с описываемой процедурой или концепцией.

Внимание! предупреждение о потенциальных проблемах.

Поддержка

Обновления, исправления и дополнительные материалы для этой книги (на английском языке) публикуются на сайте издательства Microsoft Press по адресу *http://microsoftpresssrv.libredigital.com/serverclient*. Принять участие в обсуждении книги можно на сайте автора (*www.williamstanek.com*) либо использовать сервис Twitter (в этом случае ищите пользователя WilliamStanek). Издательский коллектив приложил все усилия, чтобы обеспечить точность

информации в книге. Список исправлений, если таковые понадобятся, вы найдете по адресу *http://www.microsoft.com/learning/support*.

Ваши замечания, вопросы и предложения по этой книге направляйте в Microsoft Press. Наш **почтовый адрес**:

Microsoft Press
Attn: Editor, Microsoft Windows Command Line Administrator's Pocket Consultant, Second Edition
One Microsoft Way
Redmond, WA 98052-6399

Электронная почта:
mspinput@microsoft.com

Пожалуйста, обратите внимание, что техническая поддержка продуктов по указанным адресам не осуществляется. Сведения о технической поддержке Windows Server или Windows Vista вы найдете по адресу *http://www.microsoft.com/support*.

Об авторе

За плечами Уильяма Р. Станека (William R. Stanek) 20-летний опыт программирования и разработки приложений, и сейчас он считается одним из ведущих экспертов в области компьютерных технологий, а также отличным писателем и преподавателем. Его советы помогли миллионам программистов, разработчиков и сетевых инженеров всего мира. На счету Уильяма немало наград за книги — он написал более 100 книг о компьютерах. В том числе «Active Directory Administrator's Pocket Consultant», «Windows Group Policy Administrator's Pocket Consultant», «Windows 7 Administrator's Pocket Consultant», «Microsoft Windows Server 2008. Справочник администратора» (Русская Редакция, БХВ-Петербург, 2009) и «Windows Server 2008 Inside Out». Об интересах и деятельности Уильяма можно узнать на его сайте: *http://www.williamstanek.com/*.

С 1991 г. Станек участвовал в разработке коммерческих Интернет-проектов. Его бизнес-качества и профессиональный опыт сформировались за 11 лет армейской службы. Уильям обладает большим опытом разработки в области серверных технологий, шифрования и Интернет-решений. Он написал массу технических статей и курсов лекций по широкому спектру проблем и весьма известен как компьютерный эксперт, обладающий опытом практической работы.

Станек получил степень магистра информационных систем с отличием и степень бакалавра компьютерных наук. Он гордится участием в боевых действиях в Персидском заливе в составе экипажа самолета радиоэлектронной борьбы. В его послужном списке несколько боевых вылетов в Ирак и девять медалей за воинскую службу, включая одну из высших авиационных наград США — Крест Военно-Воздушных Сил с отличием. В настоящее время Станек с женой и детьми живет на северо-западе Тихоокеанского побережья США.

Глава 1

Введение в Windows PowerShell

Если вы — специалист в области информационных технологий, то почти наверняка слышали о Windows PowerShell, читали книги об этой командной оболочке или даже работали с ней. Возможно также, что у вас осталось много вопросов о PowerShell либо вам просто любопытно, что нового в PowerShell 2.0 по сравнению с предыдущей версией. Если это так, то здесь вы найдете все ответы.

В каждой версии Windows была командная строка, которая использовался для запуска встроенных команд, утилит и сценариев. Установка Windows PowerShell наделяет встроенную командную строку впечатляющими новыми возможностями и позволяет работать с функциями операционной системы, для доступа к которым раньше приходилось писать сложный код. В этой книге я научу администраторов, разработчиков и ИТ-специалистов других профилей управлять Windows-компьютерами и конфигурировать их.

В первой главе речь рассказывается об основах Windows PowerShell. Вы научитесь работать с PowerShell, исполнять команды и пользоваться сопутствующими функциями. Искусные администраторы, квалифицированные специалисты по технической поддержке и опытные пользователи Windows-систем все чаще обращаются к возможностям PowerShell. Умение работать с PowerShell поможет вам добиться бессбойной работы вашей системы с меньшими затратами времени и усилий. Если же вы администрируете большой парк компьютеров, то владение приемами PowerShell, позволяющими экономить время, не просто важно, а жизненно необходимо для повседневной работы.

мечание В целом, показанные в этой книге методы работают во все х версиях Windows, поддерживающих PowerShell 2.0. Однако отдельные возможности, такие как remoting и фоновые задания, недос тупны на нек оторых платформах. Поэтому всег да проверяйте к оманды, параметры и сценарии на изо лированных тес товых машинах, прежде чем применять их в рабочей сети.

Азы Windows PowerShell

Все, кто работал с UNIX, знают, что такое командная оболочка. В большинстве операционных систем, основанных на UNIX, доступно несколько

полнофункциональных командных оболочек, включая Korn Shell (KSH). C Shell (CSH) и Bourne Shell (SH). Хотя в операционных системах из семейства Windows всегда была командная строка, но не было полнофункциональной командной оболочки, Windows PowerShell восполняет этот недостаток.

Как и менее мощная командная строка Windows, командные оболочки UNIX позволяют запускать встроенные и внешние команды, утилиты командной строки, возвращая результаты в виде потока текстовых данных. Над потоком выходных данных можно выполнять разные манипуляции, например перенаправлять его, чтобы сделать выходные данные одной команды входными данными другой. Передача выходных данных одной команды другой команде называется *конвейеризацией* (piping), этот прием широко используется в сценариях командной оболочки.

C Shell — одна из мощных командных оболочек для UNIX. Во многих аспектах C Shell представляет собой «сплав» лучших возможностей языка программирования C и среды полнофункциональной командной оболочки UNIX. Windows PowerShell — дальнейшее развитие идеи полнофункциональной командной оболочки, базирующейся на языке программирования. Она поддерживает язык сценариев, основанный на C#, и модель объектов на основе Microsoft .NET Framework.

Поскольку язык сценариев Windows PowerShell базируется на C#, его без труда освоят те, кто уже пишет на C#, другим разработчикам использование PowerShell поможет перейти на этот язык. Использование модели объектов на основе .NET Framework позволяет передавать командам Windows PowerShell целые объекты со всеми их свойствам. Перенаправление объектов — чрезвычайно мощная функция, значительно расширяющая возможности динамического манипулирования наборами результатов. Например, вы можете получить не только имя пользователя, но и представляющий этого пользователя объект, а затем работать с его свойствами, ссылаясь на них по имени.

Запуск Windows PowerShell

PowerShell 2.0 — это улучшенная и дополненная версия PowerShell. Изменения по сравнению с предыдущей версией впечатляют, они касаются и гибкости, и быстродействия командной оболочки. PowerShell 2.0 позволяет делать то, что в первой версии было просто невозможно, и намного повышает эффективность исполнения стандартных операций. Ниже мы познакомимся с параметрами PowerShell, конфигурированием командной оболочки и хронологией команд.

Работа с консолью PowerShell

В Windows 7, Windows Server 2008 Release 2 и более поздние выпуски операционных систем Windows оболочка PowerShell 2.0 встроена, а в Windows XP, Windows Server 2003, Windows Vista и Windows Server 2008 ее можно

установить. Для каждой версии Windows, включая 32- и 64-разрядные редакции, на сайте Microsoft Download Center (*http://download.microsoft.com*) доступны соответствующие сборки PowerShell.

Windows PowerShell 2.0 поддерживает среду командной строки и графический интерфейс для исполнения команд и сценариев. Консоль PowerShell (powershell.exe) — это 32- или 64-разрядная среда командной строки PowerShell. В 32-разрядных версиях Windows 32-разрядный исполняемый файл находится в папке %SystemRoot%\System32\WindowsPowerShell\v1.0. В 64-разрядных версиях Windows 32-разрядный исполняемый файл PowerShell располагается в каталоге %SystemRoot%\SysWow64\WindowsPowerShell\v1.0, а 64-разрядный — в каталоге %SystemRoot%\System32\WindowsPowerShell\v1.0.

Примечание *%SystemRoot%* — это ссылка на переменную окружения SystemRoot. В операционной системе Windows определены много численные переменные окружения для получения значений, специфичных для пользователя или системы. Я буду часто обращаться к переменным окружения по синтаксису *%Имя_переменной%*.

Совет Для работы PowerShell 2.0 требуется .NET Framework как минимум версии 2.0. В этом случае вам бу дут дос тупны лишь основные возмо жности PowerShell; чтобы получить полный набор функций PowerShell, необх одимо установить .NET Framework версии не ниже 3.5.1 (только в Windows Vista и выше).

Запустить консоль PowerShell можно при помощи поля **Выполнить (Run)**, доступного в меню **Start (Пуск)**. Щелкните **Пуск | Выполнить (Start | Run)**, введите **powershell** в поле **Открыть (Search box)** и нажмите Enter. Альтернативный способ: щелкните **Пуск | Все программы|Стандартные|Windows PowerShell|Windows PowerShell 2.0 (Start|All Programs | Accessories | Windows PowerShell | Windows PowerShell 2.0)**. В 64-разрядных системах по умолчанию запускается 64-разрядная версия PowerShell, чтобы запустить 32-разрядную консоль PowerShell в 64-разрядной системе, необходимо выбрать **Windows PowerShell 2.0 (x86)**.

Чтобы запустить PowerShell из командной строки Windows (cmd.exe), выполните в ней следующую команду:

```
powershell
```

На рис. 1-1 показано окно PowerShell. По умолчанию в окне умещается 50 строк длиной в 120 символов. Если после заполнения окна вывод данных продолжается, содержимое окна прокручивается вверх. Чтобы приостановить вывод данных, нажмите Ctrl+S. Повторное нажатие Ctrl+S возобновляет вывод, а нажатие Ctrl+C прекращает выполнение команды.

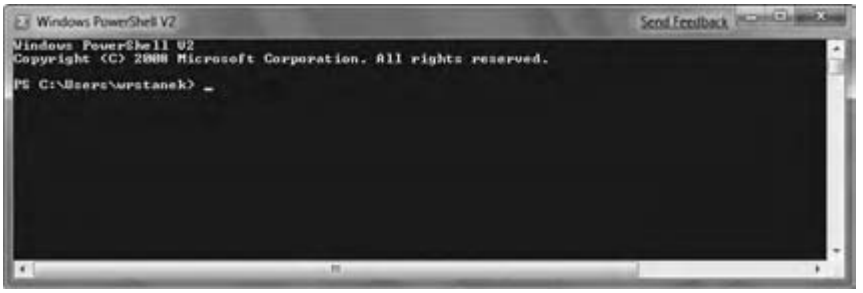


Рис. 1-1. При работе с **PowerShell** часто используется среда командной строки

В Windows 7 после запуска PowerShell выводится следующий текст:

```
Windows PowerShell 2.0
Copyright (c) 2008 Microsoft Corporation. All rights reserved.

PS C:\Users\wrstanek>
```

В данном случае приглашение командной строки (command prompt) показывает текущий рабочий каталог. По умолчанию это *%UserProfile%*, т. е. каталог профиля текущего пользователя. Мигающий курсор за приглашением означает, что командная строка находится в интерактивном режиме. В этом режиме вы можете вводить команды и нажимать Enter, чтобы их выполнить. Например, чтобы увидеть содержимое текущего каталога, введите **get-childitem** и нажмите Enter.

Кроме того, командная строка может работать в пакетном режиме для выполнения набора команд. В пакетном режиме командная строка считывает и выполняет команды одну за другой. Обычно команды пакетного режима считываются из файла сценария (script file), но можно и сразу запустить консоль в пакетном режиме.

Для выхода из PowerShell введите **exit**. Если PowerShell запущена из командной строки, исполнение команды **exit** вернет вас в командную строку Windows. Чтобы запустить новый экземпляр PowerShell из ранее запущенного, введите **powershell** в командной строке PowerShell. Так можно начать новый сеанс PowerShell, инициализировав среду командной оболочки с особыми параметрами.

Графическая среда Windows PowerShell ISE

Официальное название интегрированной графической среды PowerShell для работы со сценариями — Windows PowerShell Integrated Scripting Environment (ISE). Соответствующее приложение, powershell_ise.exe, предоставляет единый интерфейс для написания, исполнения и отладки сценариев. Существуют 32- и 64-разрядные версии графической среды PowerShell, соответствующие файлы располагаются там же, где и консольные версии командной оболочки.

Запустить графическую среду PowerShell можно через меню Start, как показано выше, либо командой **Пуск | Все программы | Стандартные | Windows PowerShell | Windows PowerShell 2.0 ISE (Start | All Programs | Accessories | Windows PowerShell | Windows PowerShell 2.0 ISE)**. В 64-разрядных системах по умолчанию запускается 64-разрядная версия PowerShell ISE, чтобы запустить 32-разрядную консоль PowerShell в 64-разрядной системе, необходимо выбрать Windows PowerShell 2.0 ISE (x86).

Чтобы запустить графическую среду PowerShell из командной строки Windows (cmd.exe), выполните в ней следующую команду:

```
powershell_ise
```

На рис. 1-2 показано главное окно PowerShell ISE. По умолчанию в нем отображаются панели **Script (Сценарий)**, **Command (Команда)** и **Output (Вывод)**. На панели Script вводятся команда и текст сценариев PowerShell. Панель Command содержит командную строку, похожую на таковую консольной версии PowerShell. На панели Output выводятся результаты исполнения сценариев и команд.

Элементы текста, вводимого на панелях Script и Command, такие как командлеты (cmdlets), функции, переменные и другие, подсвечиваются разными цветами. Для управления отображением служит пункт **View (Вид)** в окне ISE. Команда **Show Script Pane (Показать панель Script)** отображает панель Script, если она была скрыта, либо скрывает ее, если она отображалась на экране. Если панель Script видна на экране, командой **Script Pane Right (Переместить панель Script вправо)** можно переместить эту панель с ее стандартного места (вверху) в правую часть окна ISE. Повторный вызов этой команды возвращает панель Script на исходное место. Если вы предпочитаете располагать панель Command над панелью Output, щелкните **Command Pane Up (Переместить панель Command вверх)**. Я предпочитаю размещать панель Command наверху, а панель Script — справа (рис. 1-3), так мне проще работать с окнами. При желании можно изменять размеры панелей, перетаскивая мышью их границы.

Среда PowerShell ISE позволяет запускать сразу несколько сред для исполнения. Экземпляры среды называются сеансами, у каждого сеанса своя рабочая среда. ПО умолчанию запускается единственный сеанс PowerShell ISE, в дальнейшем можно запустить до семи дополнительных сеансов (максимальное общее число сеансов — восемь). Чтобы запустить новый сеанс, нажмите Ctrl+T или щелкните **New PowerShell Tab (Новый экземпляр PowerShell)** в меню **File (Файл)**.

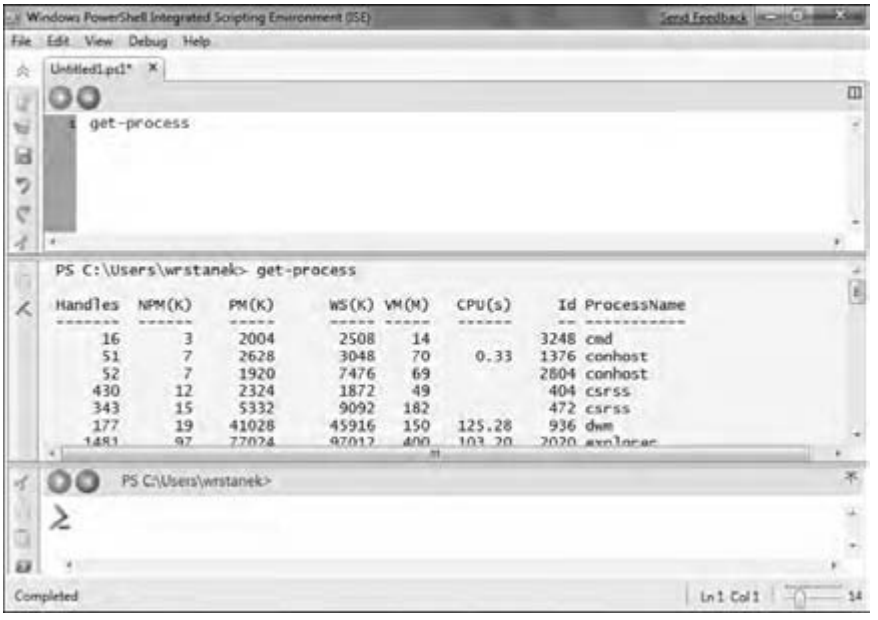


Рис. 1-2. В PowerShell ISE также часто используется командная строка

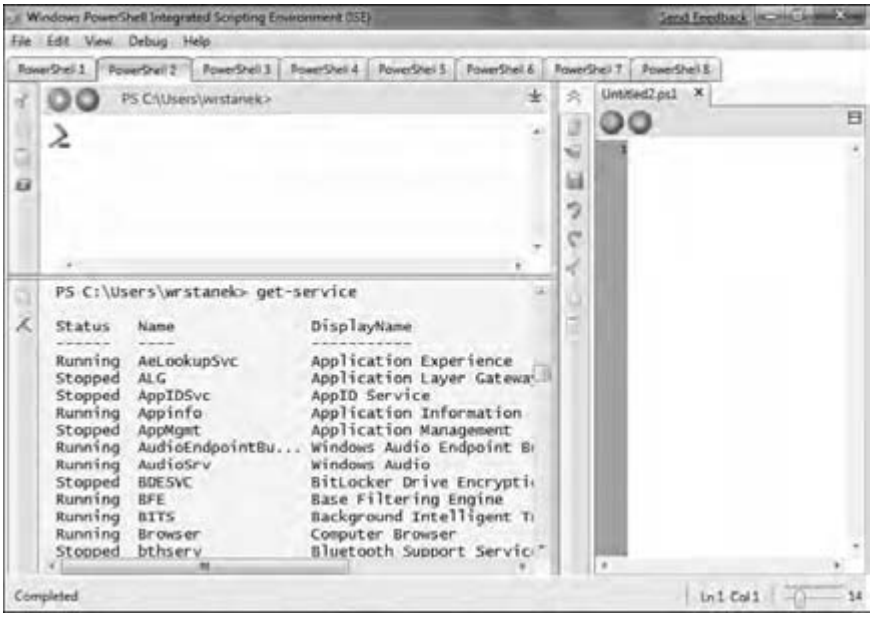


Рис. 1-3. Для переключения между сеансами служат вкладки

По умолчанию задан довольно мелкий размер текста. Изменять его можно, перетаскивая ползунок **Zoom (Масштаб)** в правом нижнем углу главного окна либо нажимая Ctrl и + (чтобы увеличить размер текста) или Ctrl и — (чтобы уменьшить его).

Чтобы закрыть сеанс, нажмите Ctrl+W или щелкните **Close (Закрыть)** в меню **File (Файл)**. Для выхода из PowerShell ISE нажмите Alt+F4 либо щелкните **File | Exit (Файл | Выход)**. Выйти из приложения также можно путем ввода команды **exit** в командной строке PowerShell на панели Command.

Настройка свойств консоли Windows PowerShell

Конечно, если вы часто работаете с командной оболочкой, имеет смысл настроить ее свойства. Например, добавить буферы, чтобы просматривать текст, который при прокрутке покинул область просмотра, изменить размер окна командной оболочки, сменить шрифты и т. д.

Чтобы приступить к настройке свойств, щелкните значок командной строки в верхнем левом углу окна командной оболочки или щелкните правой кнопкой мыши строку заголовка консоли и выберите **Properties (Свойства)**. Как показано на рис. 1-4, в диалоговом окне **Command Prompt Properties (Свойства: «Командная строка»)** четыре вкладки.

- **Options (Общие)** Настройка размера курсора, параметров отображения и редактирования, а также хронологии команд. Установите флажок **QuickEdit Mode (Выделение мышью)**, чтобы с помощью мыши выделять и вставлять текст в окне командной строки. Сбросьте флажок Insert Mode (Быстрая вставка), если вы хотите по умолчанию вести редактирование в режиме замещения. В разделе хронологии команд настраивается то, как ранее введенные команды буферизуются в памяти. (Подробнее о хронологии команд см. в разделе «Хронология команд» далее в этой главе.)
- **Font (Шрифт)** Настройка размера и начертания шрифта в окне командной строки. Для растровых шрифтов задается высота и ширина в пикселах. Например, размер 8 x 12 означает 8 экранных пикселов ширину и 12 в высоту. Для других шрифтов указывается размер в пунктах, например, вы можете выбрать шрифт Lucida Console размером 10 пт. Интересно, что если выбран размер *n* пт, шрифт будет иметь высоту *n* пикселов; например, у шрифта размером 10 пт высота равна 10 экранным пикселам. Кроме того, можно выбрать полужирное начертание, и тогда ширина шрифта в пикселах увеличится.
- **Layout (Расположение)** Задание размера буфера экрана, размера и позиции окна. Указывайте высоту буфера, достаточную для того, чтобы можно было просмотреть вывод предыдущих команд и сценариев. Для этого параметра подойдет значение в диапазоне 1000–2000. Высоту окна выбирайте такой, чтобы можно было одновременно видеть более одного окна командной оболочки. При разрешении экрана 1280 x 1024 и шрифте размером 12 пт подойдет высота в 60 строк. Если вы хотите, чтобы окно командной строки появлялось в заданном месте экрана, сбросьте флажок **Let System Position Window (Автоматический выбор)** и укажите в полях **Left (Левый край)** и **Top (Верхний край)** координаты левого верхнего угла окна командной строки.

- **Colors (Цвета)** Настройка цветов текста и фона в окне командной строки. Параметры **Screen Text (Текст на экране)** и **Screen Background (Фон экрана)** задают соответственно цвет текста и цвет фона. Параметры **Popup Text (Текст всплывающего окна)** и **Popup Background (Фон всплывающего окна)** указывают цвет текста и цвет фона в диалоговых окнах, генерируемых при выполнении команд в окне командной строки.

Рис. 1-4. Настройка свойств консоли PowerShell

Завершив настройку свойств командной оболочки, щелкните **OK**. Изменения можно применить только к текущему окну или ко всем окнам командных строк, с которыми вы будете работать в дальнейшем. Кроме того, возможно, будет предложено изменить ярлык, с помощью которого вы запустили текущее окно. В этом случае всякий раз, когда вы запускаете командную оболочку двойным щелчком этого ярлыка, будут использоваться заданные вами параметры.

Хронология команд

Буфер хронологии команд — функциональность командной оболочки Windows PowerShell. В нем запоминаются команды, выполненные в текущей командной оболочке, и он позволяет повторно обращаться к этим командам, не вводя их текст заново. Максимальное число команд в буфере задается в диалоговом окне **Properties (Свойства)** командной оболочки, о котором шла речь в предыдущем разделе. По умолчанию запоминается до 50 команд.

Для изменения размера хронологии выполните следующие операции.

1. Щелкните правой кнопкой мыши строку заголовка окна командной оболочки, выберите **Properties (Свойства)** и откройте вкладку **Options (Общие)**.
2. В поле **Buffer Size (Размер буфера)** задайте максимальное количество команд, запоминаемых в хронологии, а затем щелкните **OK**.

Эти настройки будут связаны с ярлыком, с помощью которого было открыто текущее окно PowerShell. Они будут применяться каждый раз при вызове PowerShell с помощью этого ярлыка.

К командам в хронологии можно обращаться несколькими способами.

- **Выбор с помощью клавиш-стрелок** С помощью клавиш «стрелка вверх» и «стрелка вниз» можно перемещаться вверх и вниз по хранящемуся в буфере списку команд. Когда вы найдете команду, которую нужно выполнить, нажмите Enter — она будет выполнена так же, как если бы ее ввели вручную. Кроме того, вы можете отредактировать показанный текст команды, например добавить или изменить параметры, а потом нажать Enter.
- **Просмотр хронологии команд во всплывающем окне** Нажмите F7, чтобы открыть всплывающее окно со списком команд в буфере. Затем с помощью клавиш-стрелок выберите команду. (Также можно нажать F9, набрать на клавиатуре номер команды и нажать Enter.) Нажмите Enter, чтобы выполнить выбранную команду, или Esc, чтобы закрыть всплывающее окно, не выполняя команду.
- **Поиск в хронологии команд** Введите несколько первых букв нужной команды и нажмите F8. Командная оболочка попытается найти в хронологии первую команду, которая начинается с введенных вами символов. Нажмите Enter, чтобы выполнить эту команду, или F8, чтобы найти в буфере следующую команду, начинающуюся с тех же символов.

Работая с хронологией команд, учтите, что у каждого экземпляра PowerShell свой набор буферов команд. Таким образом, буферы видны только в контексте соответствующей командной оболочки.

Работа с командлетами и сценариями

В Windows PowerShell появились так называемые *командлеты* (cmdlets). Командлет — это минимальная единица функциональности PowerShell. Командлет напоминает встроенную команду. Почти все командлеты просты и обладают небольшим набором свойств.

Работа с командлетами

С командлетами работают так же, как с обычными командами и утилитами. Имена командлетов нечувствительны к регистру символов. Это означает, что при вызове командлетов символы в нижнем и верхнем регистре можно использовать вперемешку. После запуска Windows PowerShell введите имя командлета в командной строке и нажмите Enter — он будет исполнен почти так же, как обычная команда.

Для именования командлетов используются пары «глагол-существительное». Глагол дает ориентировочную информацию о том, что делает данный командлет, а существительное — с чем он это делает (см. табл. 1-1). Например,

командлет Get-Variable получает информацию о заданной переменной PowerShell и возвращает ее значение. Если имя переменной не задано, Get-Variable возвращает полный список переменных PowerShell с указанием их значений.

Табл. 1-1. «Глагольные» части общих командлетов

Часть	Описание
Add	Создает экземпляр элемента, такого как запись в журнале или оснастка
Clear	Удаляет содержимое элемента, например, очищает журнал событий или сбрасывает значение переменной
ConvertFrom	Преобразует элемент от одного формата к другому, например, из списка, разделенного запятыми, в свойства объекта
ConvertTo	Преобразует элемент к заданному формату, например, сохраняет свойства объекта в виде списка, разделенного запятыми
Disable	Отключает функции, такие как удаленные подключения
Enable	Включает функции, такие как удаленные подключения
Export	Экспортирует свойства элемента, например свойства консоли, в заданном формате, таком как XML
Get	Запрашивает заданный объект или подмножество типов объектов, пример — получение списка работающих процессов
Import	Импортирует свойства элемента в заданном формате, например свойства консоли в виде сериализованных XML-данных
Invoke	Обрабатывает элемент, такой как выражение
New	Создает экземпляр элемента, такого как переменная или событие
Remove	Удаляет экземпляр элемента, такого как переменная или событие
Set	Изменяет заданный параметр объекта
Start	Запускает экземпляр элемента, такого как служба или процесс
Stop	Останавливает экземпляр элемента, такого как служба или процесс
Test	Проверяет состояние или значение экземпляра элемента, такого как подключение
Write	Записывает данные в экземпляр элемента, такого как журнал событий

В табл. 1-2 приводится список командлетов, которые чаще всего используются администраторами при решении рутинных задач.

Табл. 1-2. Командлеты, наиболее востребованные администраторами

Командлеты	Описание
Add-Computer, Remove-Computer	Добавляет или удаляет компьютер из рабочей группы или домена, соответственно
Checkpoint-Computer, Restore-Computer	Создает контрольную точку для восстановления или восстанавливает заданный компьютер до указанной контрольной точки, соответственно
Compare-Object, Group-Object, Sort-Object, Select-Object, New-Object	Командлеты для сравнения, группировки, сортировки, выборки и создания объектов, соответственно
ConvertFrom-SecureString, ConvertTo-SecureString	Создает или экспортирует защищенную строку, соответственно
Debug-Process	Отлаживает работающий на компьютере процесс
Get-Alias, New-Alias, Set-Alias, Export-Alias, Import-Alias	Командлеты для запроса, создания, установки, экспорта и импорта псевдонимов, соответственно
Get-AuthenticodeSignature, Set-AuthenticodeSignature	Получает или устанавливает цифровую подпись файла, соответственно
Get-Command, Invoke-Command, Measure-Command, Trace-Command	Выводит информацию о командлетах, исполняет команды, измеряет время выполнения команды, выполняет трассировку, соответственно
Get-Counter	Запрашивает значение счетчика производительности
Get-Credential	Получает объект-удостоверение по заданному паролю
Get-Date, Set-Date	Задает или устанавливает время и дату, соответственно
Get-EventLog, Write-EventLog, Clear-EventLog	Получает, записывает или сбрасывает зарегистрированные в журнале события, соответственно
Get-ExecutionPolicy, Set-ExecutionPolicy	Получает или устанавливает действующую политику исполнения для текущего экземпляра оболочки, соответственно
Get-Host	Получает информацию о хост-приложении PowerShell
Get-HotFix	Получает информацию об исправлениях и обновлениях, установленных на компьютере
Get-Location, Set-Location	Выводит или устанавливает текущий рабочий каталог, соответственно
Get-Process, Start-Process, Stop-Process	Получает, запускает или останавливает процессы на компьютере, соответственно
Get-PSDrive, New-PSDrive, Remove-PSDrive	Получает, создает или удаляет заданный диск –PowerShell, соответственно
Get-Service, New-Service, Set-Service	Получает, создает и устанавливает системные службы, соответственно

(*см. след. стр.*)

Табл. 1-2. Командлеты, наиболее востребованные администраторами

Командлеты	Описание
Get-Variable, New-Variable, Set-Variable, Remove-Variable, Clear-Variable	Получает, создает, устанавливает, удаляет переменные или сбрасывает их, соответственно
Import-Counter, Export-Counter	Импортирует либо экспортирует файлы журнала счетчика производительности, соответственно
Limit-EventLog	Задает предельный размер и давность журнала событий
New-EventLog, Remove-EventLog	Создает или удаляет пользовательский журнал и источник событий
Ping-Computer	Отправляет пакеты с ICMP-запросами заданному компьютеру
Pop-Location	Выталкивает и получает адрес из стека
Push-Location	Заталкивает в стек заданный адрес
Read-Host, Write-Host, Clear-Host	Читает входные данные, записывает выходные данные в хост-окно либо очищает его содержимое, соответственно
Rename-Computer, Stop-Computer, Restart-Computer	Переименовывает, останавливает или перезагружает компьютер, соответственно
Reset-ComputerMachinePassword	Изменяет и сбрасывает пароль учетной записи компьютера в домене
Show-EventLog	Открывает журналы событий компьютера в оснастке Event Viewer (Просмотр событий)
Show-Service	Открывает список служб на компьютере с помощью утилиты Services (Службы)
Start-Sleep	Приостанавливает работу оболочки или сценария на заданный период
Stop-Service, Start-Service, Suspend-Service, Resume-Service, Restart-Service	Останавливает, запускает, приостанавливает, возобновляет работу службы или перезапускает ее, соответственно
Wait-Process	Ожидает остановки заданного процесса перед приемом ввода
Write-Output	Записывает объект в конвейер
Write-Warning	Отображает предупреждение

Можно работать с командлетами непосредственно, вводя их в командной строке PowerShell, либо использовать их в сценариях. Чтобы получить полный список командлетов, наберите в командной строке PowerShell **get-command** — оболочка выведет перечень командлетов с указанием их имен и синтаксиса, который часто не умещается в одной строке. Однако чаще всего

требуется просто узнать, существует ли тот или иной командлет. Получить форматированный список командлетов позволяет следующая команда:

```
get-command | format-wide –column 3
```

В полученном списке много команд, которые будут часто требоваться вам при работе с PowerShell. Знак «|» называется символом конвейера. В этом примере вывод командлета Get-Command передается через конвейер командлету Format-Wide, тот принимает данные, разбивает на колонки и выводит. По умолчание данные выводятся в две колонки, но параметр –Column parameter задает вывод в три колонки, как в этом примере:

```
A: Add-Computer   Add-Content
Add-History Add-Member  Add-PSSnapin
Add-Type B: C:
cd.. cd\    Checkpoint-Computer
Clear-Content Clear-EventLog  Clear-History
Clear-Host Clear-Item Clear-ItemProperty
Clear-Variable Compare-Object Complete-Transaction
Connect-WSMan ConvertFrom-Csv    ConvertFrom-SecureString
```

Удобнее получать сведения о командлетах с помощью Get-Help. Команда **get-help *-*** выводит список командлетов с кратким описанием функций каждого из них, что много полезнее простого списка командлетов.

Чтобы ограничить длину списка, можно указать нужные имена полностью либо с использованием подстановочных знаков. Например, если имя нужного вам командлета начинается с «Get», используйте команду **get-help get*** — она покажет все команды, начинающиеся с «Get». Если известно, что в команде есть слово «*computer*», можно использовать команду **get-help *computer***, которая покажет все команды, в которых встречается искомое слово. Наконец, для поиска команд по ключевому слову применяется команда **get-help *** *ключевое_слово*. Например, список команд для работы с псевдонимами можно получить с помощью команды **get-command *alias**.

При работе с командлетами возникает два стандартных типа ошибок:

- **ошибки, вызывающие аварийный останов** (terminating errors) — прерывают исполнение сценария или команды;
- **ошибки, не вызывающие останов** — не прерывают исполнение, но генерируют сообщение об ошибке в выходных данных.

Возникновение обоих типов ошибок, как правило, сопровождается выводом сообщения с описанием их причин, помогающих в устранении ошибок. Подобные ошибки возникают, например, при отсутствии заданного файла или недостаточном уровне разрешений для выполнения заданного действия.

В освоении синтаксиса и применения командлетов вам помогут три уровня справочной документации, доступной в Windows PowerShell: стандартный, подробный и полный. Для просмотра стандартной справки по некоторому командлету введите **get-help** *имя_командлета*, например:

```
get-help new-variable
```

Стандартная справочная документация содержит полный синтаксис командлета, включая полный перечень его параметров, а также примеры их использования. Для получения подробной справочной информации необходимо указать параметр –Detailed; полная техническая информация о командлетах доступна при использовании параметра –Full. Подробная и полная справка нужны для углубленного изучения командлетов, как правило, в них удается найти все необходимые вам сведения.

Работа с параметрами командлетов

Перед параметрами командлетов всегда стоит дефис (-), например –Detailed или –Full (см. предыдущий раздел). Ради сокращения объема текста, который требуется ввести при определении команды, некоторые параметры сделали чувствительными к позиции, то есть, можно не указывать имена параметров, если они передаются в заданном порядке. Например, синтаксис командлета Get-Service позволяет опускать параметр -Name, поэтому он заключен в квадратные скобки:

```
Get-Service [-ComputerName <строка[]>] [-DependentServices]
[-Exclude < строка[]>] [-Include < строка[]>] [-ServicesDependedOn]
[[-Name] < строка[]>] [<Общие_параметры>]
```

В итоге можно вызывать Get-Service следующим образом:

```
get-service Имя_службы
```

где *Имя_службы* — имя нужной вам службы, например

```
get-service winrm
```

Эта команда возвращает состояние службы Windows Remote Management. Поскольку разрешается использовать подстановочные знаки, такие как *, допустима также команда **get-service win***, которая вернет список служб, имена которых начинаются с «*win*». Обычно это Windows Defender, Windows Management Instrumentation и Windows Remote Management, например:

```
Status Name    DisplayName
------ ----    -----------
Running WinDefend    Windows Defender
Stopped  WinHttpAutoProx   WinHTTP Web Proxy Auto-Discovery
Running  Winmgmt  Windows Management Instrumentation
Stopped  WinRM Windows Remote Management
```

Все командлеты поддерживают общий набор параметров. Многие командлеты поддерживают параметры обработки ошибок, такие как -Confirm и -WhatIf (табл. 1-3). Общие параметры можно использовать с любыми командлетами, но это не означает, что они всегда дают видимые результаты.

Так, использование параметра -Verbose с командлетами, не поддерживающими подробный вывод (verbose output), не даст никакого результата.

Табл. 1-3. Общие параметры обработки ошибок

Параметр	Описание
–Confirm	Приостанавливает исполнение до получения подтверждения от пользователя
–Debug	Предоставляет отладочную информацию об операции на уровне программирования
–ErrorAction	Определяет реакцию команды на возникновение ошибки. Допустимые значения: SilentlyContinue (продолжить исполнение, игнорируя ошибку), Continue (вывести сообщение об ошибке и продолжить; задано по умолчанию), Inquire (вывести сообщение об ошибке и продолжить после получения подтверждения от пользователя) и Stop (вывести сообщение об ошибке и прервать исполнение)
–ErrorVariable	Задает имя переменной для хранения сведений об ошибках (в дополнение к стандартным ошибкам)
–OutBuffer	Задает буфер для выходных данных командлета
–OutVariable	Задает имя переменной для хранения выходных объектов
–Verbose	Предоставляет подробные сведения об операции
–WarningAction	Определяет реакцию команды на предупреждения. Допустимые значения: SilentlyContinue (продолжить исполнение, игнорируя предупреждение), Continue (вывести сообщение и продолжить; задано по умолчанию), Inquire (вывести сообщение и продолжить после получения подтверждения от пользователя) и Stop (вывести сообщение и прервать исполнение)
–WarningVariable	Задает имя переменной для хранения предупреждений (в дополнение к стандартным ошибкам)
–WhatIf	Показывает пользователю, каков будет результат запуска командлета с заданными параметрами

Работа с внешними командами

Поскольку Windows PowerShell работает в контексте командной строки Windows, в командной строке и сценариях PowerShell доступны любые команды, утилиты командной строки и GUI-приложения Windows. Однако важно помнить, что интерпретатор Windows PowerShell анализирует все команды прежде, чем передать их в среду командной строки. Если в PowerShell существует одноименная команда, ключевое слово или псевдоним с тем же самым именем, будет исполнена команда из PowerShell, а не из Windows (о псевдонимах и функциях см. в главе 2).

Внешние команды и программы PowerShell должны находиться в каталоге, на который указывает переменная окружения PATH. Если заданная про-

грамма найдена в таком каталоге, она будет запущена. Переменная PATH также определяет места, в которых PowerShell ищет приложения, утилиты и сценарии. Для ссылки на переменные окружения Windows в PowerShell служит директива $env. Для просмотра текущего значения переменной PATH введите **$env:path**, для добавления каталогов к PATH используйте следующий синтаксис:

```
$env:path += «;Путь_к_добавляемому_каталогу»
```

где *Путь_к_добавляемому_каталогу* — путь к каталогу, который требуется добавить к PATH, например:

```
$env:path += «;C:\Scripts»
```

Чтобы этот каталог добавлялся к PATH при каждом запуске PowerShell, можно добавить соответствующую команду к профилю PowerShell. Профиль — это особый сценарий, который используется для настройки рабочей среды PowerShell. Помните, что командлеты больше похожи на встроенные команды, а не на отдельные утилиты, поэтому переменная PATH на них не действует.

Совет Компьютеры с Windows Vista и более поздними версиями Windows поддерживают утилиту SETX, которая позволяет записывать изменения переменных окружения непосредственно в системный реестр Windows. Так удается сохранить модификации переменных, тогда как команды вроде $env:path не позволяют это сделать. Кроме того, с помощью SETX можно получать текущие значения параметров реестра и записывать их в текстовый файл.

Работа со сценариями

Сценарии Windows PowerShell — это текстовые файлы с расширением .ps1. Любые командлеты и команды, вводимые в командной строке PowerShell, можно записать как сценарий, скопировав их в файл и сохранив его с расширением .ps1. После этого сценарий можно запускать как обычную команду или командлет. Однако при работе со сценариями PowerShell текущий каталог может отсутствовать в переменной окружения PATH. В этом случае следует предварить имя сценария, запускаемого из текущего каталога, строкой «./». Например, запустить сценарий run_all.ps1 из текущего каталога можно следующей командой:

```
./run_all
```

Примечание Оболочка PowerShell рассчитана на пользователей с опытом работы как в UNIX, так и в Windows, поэтому в качестве разделителей каталогов можно использовать как прямой, так и обратный слеш. Таким образом, команды **./run_all** и **.\run_all** равноценны.

Работая со сценариями, необходимо учитывать текущую политику исполнения, а также требования к подписыванию сценариев.

Введение в политики исполнения

Текущая политика исполнения Windows PowerShell контролирует запуск сценариев и обработку конфигурационных файлов. Политика исполнения — это встроенная защитная функция PowerShell, для каждого пользователя устанавливается своя политика, которая хранится в системном реестре Windows. Политика по умолчанию зависит от версии и редакции установленной операционной системы, чтобы определить текущую политику, введите в командной строке **get-executionpolicy**.

Доступны следующие политики (по убыванию степени защиты):

- **Restricted** — запрещает загрузку конфигурационных файлов и сценариев независимо от наличия у них цифровой подписи. Поскольку профиль PowerShell также является сценарием, эта политика запрещает и загрузку профилей;
- **AllSigned** — требует наличия у конфигурационных файлов и сценариев, полученных из любых источников, локальных и удаленных, доверенной цифровой подписи. Перед загрузкой файла с цифровой подписью PowerShell запрашивает у пользователя подтверждение;
- **RemoteSigned** — требует наличия доверенной цифровой подписи у конфигурационных файлов и сценариев, полученных из удаленных источников, для локальных конфигурационных файлов и сценариев это ограничение не действует. Подписанные файлы и сценарии загружаются без предупреждения;
- **Unrestricted** — разрешает запуск любых конфигурационных файлов и сценариев независимо от источника и наличия цифровой подписи, но перед обработкой файла из удаленного источника система запрашивает подтверждение у пользователя.

Как видите, политика исполнения определяет возможность загрузки того или иного сценария или конфигурационного файла, а также требования к наличию у них цифровой подписи. Если политика блокирует загрузку файла или сценария, отображается предупреждение с разъяснением соответствующих ограничений.

Командлет Set-ExecutionPolicy задает политику исполнения, изменения политики записываются в реестр. Тем не менее, если в групповой политике компьютера или пользователя активен параметр Turn On Script Execution, изменения политики исполнения записываются в реестр, но не вступают в силу, а PowerShell выводит сообщение о причинах конфликта. Командлет Set-ExecutionPolicy не переопределяет настройки групповой политики, даже если заданная с его помощью политика исполнения вводит более жесткие ограничения.

Чтобы установить политику, требующую наличия доверенной цифровой подписи у всех сценариев и конфигурационных файлов, введите

```
set-executionpolicy allsigned
```

Чтобы установить политику, требующую наличия подписи у всех сценариев и конфигурационных файлов, загруженных из веба, используйте команду

```
set-executionpolicy remotesigned
```

Отменить все ограничения позволяет следующая команда:

```
set-executionpolicy unrestricted
```

Изменения вступают в силу немедленно и действуют на локальный сеанс консоли PowerShell или ISE. Поскольку изменения политики записываются в реестр, новая политика будет действовать на все сеансы PowerShell.

мечание Поскольку в Vista и более поздних версиях Windows изменять политику исполнения разрешено только администраторам, для смены политики следует запускать PowerShell с помощью функции Run As Administrator (Запуск от имени).

Добавление цифровых подписей к сценариям

Подписать сценарий проще, чем вы думаете. Это делается с помощью командлета Set-AuthenticodeSignature. Этот командлет создает цифровые подписи на основе цифрового сертификата. Цифровой сертификат можно получить в центре сертификации, ЦС (certificate authority, CA) либо создать его самостоятельно. Сертификаты, выданные ЦС, можно использовать на любом компьютере, доверяющем этому ЦС. Самоподписанные сертификаты действительны только на локальном компьютере, на котором он были созданы.

Для создания ЦС, выдающих сертификаты в доменах Windows, применяются службы сертификации Active Directory (Active Directory Certificate Services). Поскольку в большинстве организаций имеются ЦС и система информационной безопасности на основе цифровых сертификатов, возможно, вам уже доступен сертификат, пригодный для подписания кода. Чтобы проверить это, введите

```
get-childitem cert:\CurrentUser\My -codesigningcert
```

либо изучите хранилище сертификатов, в котором на Windows-компьютерах хранится информация о доверенных ЦС. В хранилище сертификатов можно просматривать следующие сведения:

- **Personal** — ваши личные сертификаты, хранимые на локальном компьютере;
- **Other People** — сертификаты, принадлежащие другим людям, хранимые на локальном компьютере;
- **Trusted Root Certification Authorities** — корневые ЦС, которым доверяет ваш компьютер. Он также будет доверять любым ЦС, получившим сертификаты от доверенных корневых ЦС;
- **Trusted Publishers** — издатели, обладающие доверенной цифровой подписью;

- **Untrusted Publisher** — издатели, чья цифровая подпись не пользуется доверием.

Хранилище сертификатов доступно через диалоговое окно Internet Properties. Чтобы открыть его, на **Панели управления (Control Panel)** щелкните **Network And Internet | Internet Options (Сеть и Интернет | Свойства обозревателя)**. В окне **Internet Properties (Свойства: Интернет)** перейдите на вкладку **Content (Содержание)** и щелкните **Certificates (Сертификаты)** — откроется одноименное окно, в котором можно получить различные сведения о доверенных сертификатах и ЦС.

В PowerShell для доступа к сертификатам используется поставщик (provider) Cert. Компоненты-поставщики данных выглядят в системе как диски, содержимое поставщиков можно просматривать как файлы и папки на жестком диске (о поставщиках данных см. в главе 5). Команда **cd cert:** открывает доступ к сертификатам, хранящимся на компьютере. Чтобы получить список доступных каталогов, введите **dir** (это псевдоним командлета Get-ChildItem). Как правило, доступны каталоги CurrentUser и LocalMachine, представляющие хранилища сертификатов пользователя, вошедшего в систему, и локального компьютера, соответственно:

```
cd cert:
dir
```

```
Location   : CurrentUser
StoreNames : {SmartCardRoot, UserDS, AuthRoot, CA...}

Location   : LocalMachine
StoreNames : {SmartCardRoot, AuthRoot, CA, Trust...}
```

Если при работе с поставщиком Cert ввести **cd currentuser** и **dir**, вы увидите каталоги хранилища сертификатов текущего пользователя. Один из них — My — содержит личные сертификаты:

```
cd currentuser
dir
```

```
Name : SmartCardRoot
Name : UserDS
Name : AuthRoot
Name : CA
Name : ADDRESSBOOK
Name : Trust
Name : Disallowed
Name : My
Name : Root
Name : TrustedPeople
Name : TrustedPublisher
```

Для просмотра личных сертификатов для подписания кода в хранилище My введите **cd my**, затем **dir –codesigningcert**:

```
cd my
dir -codesigningcert
```

```
   Directory: Microsoft.PowerShell.Security\Certificate::currentuser\my
   Thumbprint                                Subject
   ----------                                -------
   D3828283483483882433482382423BE2828282833  CN=WRSTANEK
   AED3828283833834848348348348348BAC39839  CN=WRSTANEK
```

Команды Get-ChildItem –Codesigningcert и Dir –Codesigningcert дают одинаковый результат — список (массив) сертификатов. В PowerShell можно ссылаться на элементы массива по их номеру (индексу в массиве). Индекс первого элемента равен 0, второго — 1, и т.д.

Если личный сертификат для подписания кода вам вы получили от ЦС, то для добавления цифровой подписи к сценариям можете использовать следующие команды:

```
$cert = @(Get-ChildItem cert:\CurrentUser\My -codesigningcert)[0]
Set-AuthenticodeSignature Имя_сценария.ps1 $cert
```

Заметьте, что здесь используются две отдельные команды, а *Имя_сценария* — это имя сценария, который требуется подписать, вот пример:

```
$cert = @(Get-ChildItem cert:\CurrentUser\My -codesigningcert)[0]
Set-AuthenticodeSignature run_all.ps1 $cert
```

Здесь команда Get-ChildItem используется для получения из хранилища сертификатов My первого личного сертификата для подписания кода, а следующая команда использует этот сертификат для подписания сценария run_all.ps1. Если этот сертификат подписан корпоративным ЦС, то подписанный таким образом сценарий run_all.ps1 можно будет запускать на любом из компьютеров организации, доверяющих этому ЦС.

Создание и использование самоподписанных сертификатов

Для создания самоподписанных (self-signed) сертификатов служит утилита makecert.exe, которая имеется в Microsoft .NET Framework SDK 1.1 и выше, а также в Windows SDK. Загрузив и установив соответствующий SDK, вы должны:

1. Запустить командную строку (cmd.exe) с администраторскими полномочиями.
2. С ее помощью создать локальный ЦС для своего компьютера.
3. Использовать командную строку и новый ЦС для выдачи личного сертификата.

   Чтобы создать локальный ЦС, введите:

```
makecert -n «CN=PowerShell Local Certificate Root» -a sha1
-eku 1.3.6.1.5.5.7.3.3 -r -sv root.pvk root.cer
-ss Root -sr localMachine
```

Этап команда выполняется в командной строке, запущенной с администраторскими полномочиями, в каталоге, где находится утилита makecert. Заметьте, что это одна команда со следующими параметрами:

- -n — задает имя сертификата, которое предваряется строкой CN=;
- -a — задает для подписи алгоритм SHA1 вместо MD5 (алгоритма по умолчанию);
- -eku — задает идентификатор объекта enhanced key usage (1.3.6.1.5.5.7.3.3);
- -r — определяет создание самоподписанного сертификата;
- -sv — задает файл закрытого ключа и выходной файл, в который makecert запишет готовый сертификат;
- -ss — задает имя хранилища, в которое будет помещен готовый сертификат, как корневой ЦС (Root) для хранилища Trusted Root Certificate Authorities.
- -sr — задает хранилище сертификатов (хранилище для локального компьютера вместо хранилища для текущего пользователя, заданного по умолчанию).

Для создания личного сертификата с использованием нового ЦС введите:

```
makecert -pe -n «CN=PowerShell User» -ss MY -a sha1
-eku 1.3.6.1.5.5.7.3.3 -iv root.pvk -ic root.cer
```

И это одна команда, в которой использованы следующие параметры:

- -pe — помечает закрытый ключ сертификата как доступный для экспорта;
- -n — задает имя сертификата, которое также предваряется строкой CN=.
- -ss — задает имя хранилища для размещения нового сертификата, MY (хранилище личных сертификатов);
- -a — задает алгоритм цифровой подписи (SHA1);
- -eku — задает идентификатор объекта enhanced key usage (1.3.6.1.5.5.7.3.3);
- -iv — задает имя файла закрытого ключа ЦС;
- -ic — задает имя файла сертификата ЦС.

Первая команда генерируется два временных файла: root.pvk и root.cer, а вторая создает из них сертификат и записывает его в хранилище личных сертификатов на локальном компьютере.

Утилита MakeCert запрашивает новый пароль для доступа к закрытому ключу. Этот пароль защищает сертификат от несанкционированного использования. Рекомендуется выбирать легко запоминающийся пароль, поскольку его потребуется вводить в дальнейшем при использовании сертификата.

Чтобы проверить созданный сертификат, попробуйте найти его в хранилище личных сертификатов с помощью следующей команды:

```
get-childitem cert:\CurrentUser\My -codesigningcert
```

Эта команда обращается к поставщику данных о сертификатах PowerShell за информацией о сертификатах в хранилище My. Если сертификат был успешно создан, то он будет присутствовать в списке сертификатов, сгенерированном этой командой.

Созданный самоподписанный сертификат можно использовать для подписания сценариев, как показано выше. Подписанные таким образом сценарии можно будет запускать на локальном компьютере, но они не будут работать на компьютерах, у которых политика исполнения требует наличия у сценариев подписи доверенного ЦС. В этом случае при попытке запуска сценария PowerShell выводит сообщение об ошибке из-за невозможности проверки подписи сертификата.

Глава 2

Эффективная работа с PowerShell

Среда Windows PowerShell обеспечивает эффективную работу с командами и сценариями. Как сказано в главе 1, командная строка PowerShell поддерживает множество разнообразных команд и командлетов, утилит Windows и приложений с расширениями командной строки. Синтаксис любых команд, независимо от их источника, построен по единым правилам. Согласно этим правилам, команда включает имя команды, а также набор обязательных и необязательных аргументов. В число аргументов входят параметры, значения параметров и текстовые строки. Для объединения команд могут использоваться знаки конвейера, а операторы перенаправления позволяют манипулировать потоками входных и выходных данных, а также сообщениями об ошибках.

При исполнении команды в PowerShell происходит следующий ряд событий:

1. Набор команд, объединенных в цепочку или группу и переданных как единая строка, разбивается на отдельные единицы исполнения, а те — на серии *маркеров* (*tokens*).
2. Каждый маркер подвергается синтаксическому разбору, в ходе которого команды отделяются от значений и сопоставляются объектам того или иного типа, например String или Boolean, а переменные в тексте команды заменяются подходящими значениями.
3. После этого обрабатываются отдельные команды. Если имя команды содержит путь к файлу, PowerShell использует этот путь для поиска программы, представляющей команду. Если найти нужную команду по заданному пути не удается, PowerShell возвращает ошибку.
4. Если имя команды не содержит пути, PowerShell пытается найти соответствующую встроенную команду. В случае успеха заданное имя (либо псевдоним или функция) считается ссылкой на внутреннюю команду, которая может быть обработана немедленно. В противном случае PowerShell ищет внешнюю программу, имя которой совпадает с именем команды, используя переменную PATH. Если команду не удалось обнаружить ни одним из вышеописанных способов, PowerShell возвращает

ошибку. Поскольку PowerShell по умолчанию не ищет в текущем каталоге, этот каталог необходимо указать явно.

5. Обнаруженная команда исполняется с заданными аргументами, включая те, что определяют входные данные. Выходные данные и сообщения об ошибках отображаются в окне PowerShell либо записываются в заданные приемники.

Как видите, исполнение команды зависит от множества факторов, включая значение PATH, методы перенаправления, объединение команд в группы и цепочки. В этой главе я продемонстрирую на примерах способы построения команд, обеспечивающие максимально эффективное использование возможностей PowerShell. Но прежде, чем перейти к этому предмету, мы разберем тонкости запуска PowerShell, познакомимся с профилями и консольными файлами.

Инициализация среды

Windows PowerShell предоставляет динамическую расширяемую среду для исполнения команд и сценариев. Эту среду можно инициализировать разными способами, включая запуск Powershell.exe с параметрами, использование пользовательского профиля, консольного файла либо комбинацию всех трех способов. Также доступно несколько способов расширения среды PowerShell, таких как установка поставщиков и регистрация оснасток (подробнее об этом — в главе 3).

Передача параметров при запуске PowerShell

Если вы работали с прежней версией PowerShell, то, вероятно, открывали окно консоли через меню **Start | All Programs | Accessories | Windows PowerShell | Windows PowerShell (Пуск | Все программы | Стандартные | Windows PowerShell | Windows PowerShell)**. Так PowerShell запускается со стандартными пользовательскими привилегиями и не позволяет решать задачи, требующие администраторских полномочий. Чтобы запустить PowerShell с привилегиями администратора, необходимо щелкнуть **Start | All Programs | Accessories | Windows PowerShell | Windows PowerShell (Пуск | Все программы | Стандартные | Windows PowerShell)**, затем выбрать пункт Windows PowerShell, щелкнуть его правой кнопкой мыши и выбрать **Run As (Запуск от имени)** и ввести учетные данные администратора.

Есть и другие способы запуска консоли PowerShell, в том числе поле **Search (Найти)** и окно **Run (Выполнить)**, в этом случае в командной строке Windows следует ввести powershell. Данный метод позволяет передавать PowerShell параметры, управляющие работой командной оболочки и позволяющие выполнить дополнительные команды. Например, так можно отключить стандартное сообщение при запуске PowerShell (т.н. режим «no-

logo»), для этого служит команда powershell -nologo. По умолчанию при запуске через командную строку Windows работа PowerShell завершается после исполнения команды. Чтобы не закрывать PowerShell после исполнения команды, введите перед командой powershell /noexit.

На листинге 2-1 показан базовый синтаксис вызова консоли PowerShell со всеми параметрами (см. табл. 2-1). По умолчанию при запуске консоли PowerShell загружается профиль запуска (startup profile). Выйти из консоли можно в любое время с помощью команды **exit**.

Листинг 2-1. Синтак сис PowerShell

```
powershell[.exe] [-PSConsoleFile Имя_файла | -Version Номер_версии]
  [-NoLogo] [-NoExit] [-NoProfile] [-NonInteractive] [-Sta]
  [-InputFormat {Text | XML}] [-OutputFormat {Text | XML}]
  [-WindowsStyle Стиль] [-EncodedCommand Команда_закодированная_в _ Base64]
  [-File Путь_к_файлу_сценария] [-ExecutionPolicy Параметры_политики]
  [-Command Текст_команды]
```

Табл. 2-1. Параметры запуска PowerShell

Параметр	Описание
-Command	Задает команду «открытым текстом», то есть так, как она вводится в командной строке PowerShell
-EncodedCommand	Задает текст команды, закодированный по алгоритму Base64
-ExecutionPolicy	Задает политику исполнения по умолчанию для сеанса консоли
-File	Задает имя файла сценария для исполнения
-InputFormat	Задает формат входных данных для PowerShell: текстовый или сериализованный XML; доступные значения — text и XML (задано по умолчанию)
-NoExit	Предотвращает завершение работы оболочки после исполнения команд, удобен при запуске команд и сценариев PowerShell через командную строку Windows (cmd.exe)
-NoLogo	Отключает вывод стандартного сообщения при запуске
-Noninteractive	Запускает консоль в пакетном режиме, в котором отключено приглашение к вводу команд PowerShell
-NoProfile	Отключает загрузку текущего пользовательского профиля PowerShell
-OutputFormat	Задает формат входных данных для PowerShell: текстовый или сериализованный XML; доступные значения — text (задано по умолчанию) и XML
-PSConsoleFile	Загружает заданный консольный файл PowerShell. Консольные файлы имеют расширение .psc1 и используются для загрузки нужных оснасток. Консольные файлы создаются с помощью командлета Export-Console

(*см. след. стр.*)

Табл. 2-1. Параметры запуска PowerShell

Параметр	Описание
-Sta	Запускает PowerShell в однопоточном режиме
-Version	Задает версию PowerShell (например, 1.0) для использования в целях обеспечения совместимости
-WindowStyle	Задает стиль окна PowerShell: Normal (обычное, задано по умолчанию), Minimized (свернутое), Maximized (развернутое на весь экран) или Hidden (скрытое)

Запуск Windows PowerShell

Чаще всего администратору приходится работать с консолью либо графической средой PowerShell ISE, но время от времени требуется вызывать PowerShell из командной строки Windows (cmd.exe) или пакетного файла (.bat-файла) для исполнения отдельных команд. Для этого служит параметр -Command. Кроме того, при вызове PowerShell таким способом обычно отключают вывод стандартного логотипа (с помощью параметра -NoLogo) и загрузку профилей (параметром -NoProfile). Например, следующая команда, введенная в командной строке Windows или вставленная в пакетный файл, позволяет получить с помощью PowerShell список работающих процессов:

```
powershell -nologo -noprofile -command get-process
```

При исполнении этой команды Windows запускает PowerShell как обычную внешнюю программу и передает ей указанные параметры, после исполнения команды работа PowerShell завершается. Параметр -NoExit предотвращает завершение работы PowerShell после исполнения команды:

```
powershell -noexit -command get-process
```

Исполнение команд с помощью -Command

Параметр -Command чаще всего используется при вызове PowerShell из командной строки Windows и пакетных файлов, поэтому давайте разберем его использование поподробнее. Знак «-» в командной строке заставляет командную оболочку читать текст команды из стандартного ввода, наряду с ним используются конвейеры и перенаправление. Учтите, однако, что любые символы, которые идут после команды, интерпретируются как ее аргументы. По этой причине часть команды, включающая конвейеры и перенаправления ввода-вывода, должна заключаться в двойные кавычки. Следующая команда генерирует список работающих в данный момент процессов, упорядоченный по pid (идентификатору процесса):

```
powershell -nologo -noprofile -command «get-process | sort-object Id»
```

Совет Данные, генерируемые большинством команд, можно передавать для обработки другим командам. Для этого используются *конвейеры* (pipes), передающие вывод одной команды как ввод другой команде. Общий синтаксис конвейера выглядит следующим образом:

```
Команда1 | Команда2
```

Здесь конвейер передает вывод команды *Команда1* на ввод команды *Команда2*, но в конвейер можно объединить больше двух команд, например:

```
Команда1 | Команда2 | Команда3
```

Как правило, командлет, способный принимать через конвейеры ввод от других командлетов, поддерживает параметр -InputObject.

PowerShell также поддерживает *блоки сценариев* (script blocks). Блоком сценария называется серия последовательно исполняемых команд. Блоки сценария заключаются в фигурные скобки, {}, команды в блоках разделяются точкой с запятой (;). Вводить блоки сценариев в фигурных скобках непосредственно можно только при вызове Powershell.exe через командную строку. Такая команда возвращает объекты в виде десериализованных XML-данных, а не стандартные объекты. Например, чтобы выполнить несколько команд в отдельном экземпляре PowerShell, введите в командной строке ранее открытого экземпляра PowerShell нужные команды, разделенные точкой с запятой (взяв их в фигурные скобки):

```
powershell -command {get-service; get-process}
```

Такая команда сработает в командной строке PowerShell, но выполнить ее в командной строке Windows (cmd.exe) не удастся. Обойти это ограничение позволяет следующий синтаксис:

```
«& {Текст_команды}»
```

В этом примере кавычки и знак & представляют собой оператор вызова, обеспечивающий исполнение команды. В таком формате команды можно исполнять через командную строку как PowerShell, так и Windows. Так, команда **powershell -command {get-service; get-process}** не сработает в командной строке Windows, а эта команда будет исполнена без проблем:

```
powershell -command «& {get-service; get-process}»
```

Здесь PowerShell получает в виде строки блок кода и пытается его исполнить. В результате PowerShell вызывает Get-Service, отображает результаты вызова, а затем вызывает Get-Process и также отображает результаты. Этот синтаксис универсален: он позволяет исполнять отдельные команды и группы команд как в командной строке PowerShell, так и в командной строке Windows.

Исполнение сценариев с помощью параметра -File

При вызове сценариев PowerShell из командной строки Windows также разрешается использовать конвейеры и перенаправление ввода-вывода. Однако при этом вместо -Command применяют параметр -File, который задает сценарий для исполнения. После параметра -File указывают путь к файлу сценария:

```
powershell -nologo -noprofile -file c:\scripts\run_all.ps1
```

Если сценарий находится в текущем каталоге, достаточно указать имя сценария:

```
powershell -nologo -noprofile -file run_all.ps1
```

Если путь содержит пробелы, его необходимо взять в двойные кавычки, как в этом примере:

```
powershell -nologo -noprofile -file «c:\data\current scripts\run_all.ps1»
```

Совет Параметры можно передавать PowerShell независимо от способа запуска консоли (через командную строку или через меню). При запуске через меню для этого необходимо отредактировать ярлык консоли PowerShell следующим образом:

1. Щелкните ярлык консоли в меню правой кнопкой и выберите **Properties (Свойства)**. В окне свойств ярлыка по умолчанию выделено поле **Target (Объект)** на вкладке **Shortcut (Ярлык)**.
2. Нажмите клавишу — правую стрелку: в конце строки пути к PowerShell появится курсор. Добавьте пробел и введите необходимые параметры и значения.
3. Щелкните **OK**, чтобы сохранить внесенные изменения. Если вы допустили опечатку, либо введенные параметры больше не нужны, повторите вышеописанные действия, чтобы удалить лишние параметры.

Вложенные консоли

Иногда требуется временно открыть консоль с особыми параметрами среды, а затем вернуться в среду с исходными настройками, не закрывая окно консоли. В таких случаях используют *вложенные консоли*, запуская новую консоль из ранее открытой.

В отличие от командной строки Windows, вложенная консоль открывается со своими параметрами рабочей среды, а не наследует их от консоли, из которой она была открыта. В новой консоли можно исполнять команды и сценарии как обычно. Набрав exit, вы закроете новую консоль и вернетесь в прежний экземпляр консоли с исходными параметрами среды.

Ввод команд, синтаксический разбор и вывод результатов

Как видно из примеров, показанных в главе 1, ввод команд в командной строке PowerShell не представляет ничего сложного. Проще всего вводить

команды с клавиатуры, завершая ввод нажатием Enter, после чего PowerShell выполняет синтаксический разбор и обработку текста команды.

Основы ввода команд

Консоль PowerShell поддерживает базовые функции редактирования команд с помощью клавиатуры (см. табл. 2-2). Также можно получить список ранее введенных команд с помощью **get-history** (команда **clear-history** очищает этот список). В списке, который выдает Get-History, командам присвоены номера. Чтобы повторно выполнить ранее введенную команду, передайте командлету Invoke-History ее номер (в следующем примере так исполняется команда под номером 35):

```
invoke-history 35
```

Табл. 2-2. Ввод команд с помощью клавиатуры и мыши

Клавиша, комбинация или щелчок	Описание
`	Разрыв строки или escape-символ для ввода литерала; для разрыва строки также используют знак конвейера (|)
Alt+Space+E	Открывает контекстное меню с командами Mark (Выделить), Copy (Копировать), Paste (вставить), Select All (Выделить все), Scroll (Прокрутка) и Find (Найти). Соответствующие команды вызываются нажатием клавиш K, Y, P, S, L и F. Для копирования данных из экранного буфера в буфер обмена следует нажать Alt+Space+E+S, затем Alt+Space+E+Y
Alt+F7	Очистка журнала команд
Ctrl+C	Выход из подменю либо прерывание исполнение
Ctrl+End	Удаление всех символов в строке после курсора
Ctrl+Стрелка влево / Ctrl+Стрелка вправо	Перемещение к началу / концу текущего слова
Ctrl+S	Приостанавливает и возобновляет вывод на экран
Delete / Backspace	Удаляет символ справа / слева от курсора
Esc	Очищает текущую строку
F1	Перемещает курсор на один символ вправо; в конце строки вставляет один символ из последней команды
F2	Создает новую команду копированием части последней введенной команды от начала строки до курсора
F3	Вставляет в текущую команду часть последней введенной команды от курсора до конца строки
F4	Удаляет из текущей строки символы, начиная с текущего положения курсора
F5	Просмотр хронологии команд

(*см. след. стр.*)

Табл. 2-2. Ввод команд с помощью клавиатуры и мыши

Клавиша, комбинация или щелчок	Описание
F7	Открывает окно со списком ранее введенных команд. Для прокрутки списка служат клавиши – стрелки вверх и вниз, для запуска команды — Enter, стрелка вправо копирует команду из списка в командную строку
F8	Поиск ранее введенной команды, совпадающей текущей командой или фрагментом команды
F9	Повторное исполнение ранее введенной команды, заданной по ее номеру в списке, доступному по нажатию F7
Home / End	Перемещает курсор в начало / конец строки
Insert	Переключение между режимами вставки и замены символов
Стрелки влево / вправо	Перемещает курсор влево / вправо в текущей строке
Page Up / Page Down	Переход к первой / последней команде в хронологии
Щелчок	Если режим QuickEdit отключен, открывает контекстное меню с командами Mark, Copy, Paste, Select All, Scroll и Find. Для копирования содержимого экранного буфера в буфер обмена выберите в контекстном меню Select и нажмите Enter
Tab / Shift+Tab	Доступ к функциям расширения (см. главу 6)
Стрелки вверх / вниз	Перемещение по списку ранее введенных команд (см. раздел о хронологии команд в главе 1)
Windows+R и ввод **powershell**	Запуск PowerShell (при наличии нескольких версий оболочки, например в 64-разрядной системе, запускается первая попавшаяся версия, причем не обязательно нужная вам)

Совет Способ копирования-вставки текста в консоли PowerShell зависит от того, активен ли режим QuickEdit. Если он активен, текст копируется так: сначала копируемый фрагмент выделяют мышью и нажимают Enter, для вс тавки ск опированного ну жно щелкнуть мышью в месте вставки. Выделяя текст мышью, не задерживайте указатель на одном месте сразу после начала выделения, иначе PowerShell вставит содержимое буфера обмена. Если же режим QuickEdit отключен, для копирования нужно щелкнуть правой кнопкой, выбрать команду **Mark (Выделить)**, выделить мышью ну жный фрагмент и нажать Enter. Чтобы вставить скопированный фрагмент, необходимо щелкнуть правой кнопкой и выбрать к оманду **Paste (Вставить)**. Для включения и выключения режима QuickEdit служит окно Properties (подробнее об этом — в главе 1).

Синтаксический разбор команд

Наряду с различными режимами обработки команд PowerShell поддерживает несколько режимов синтаксического разбора. Не путайте режимы обработки (processing modes) и режимы синтаксического разбора (parsing modes): режимы обработки контролируют обработку команд (она может вы-

полняться интерактивно либо в пакетном режиме), а режимы синтаксического разбора управляют алгоритмами анализа командной строки.

PowerShell разбивает команды на единицы исполнения и маркеры. В единицу исполнения входит все, начиная с первого символа строки до точки с запятой либо конца строки. Маркер представляет значение в составе единицы исполнения. С учетом этого возможно:

- вводить в командной строке сразу несколько команд, разделенных точкой с запятой;
- помечать конец единицы исполнения нажатием Enter.

Режим синтаксического разбора значений определяется первым маркером, выявленным в начале разбора текущей единицы исполнения. PowerShell поддерживает следующие режимы синтаксического разбора:

- **Режим выражений** Этот режим используется, если первый маркер единицы исполнения *не* является именем командлета, ключевым словом, псевдонимом, функцией или внешней утилитой. При обработке выражений PowerShell получает числовые либо строковые значения. Значения — строки необходимо заключать в кавычки; цифры, не взятые в кавычки, интерпретируются как числа (а не как символы);
- **Режим команд** Этот режим используется, если первый маркер единицы исполнения *является* именем командлета, ключевым словом, псевдонимом, функцией или внешней утилитой PowerShell. PowerShell запускает команды, представленные маркерами. Значения, идущие после маркера команды, интерпретируются как строки, требующие подстановки значений. Исключение — случаи, когда эти значения начинаются со специального символа, начинающего переменную, массив, строку или вложенное выражение. Эти символы включают $, @, ‘, “ и (. Обнаружив их, PowerShell обрабатывает соответствующие значения как выражения.

С учетом этих правил очевидно следующее:

- если ввести 5+5 в командной строке PowerShell, эти символы интерпретируются как выражение, при вычислении которого получается (и отображается) значение 10;
- если ввести Write-Host 5+5, PowerShell интерпретирует 5+5 как аргумент командлета Write-Host и выводит 5+5;
- если же ввести в командной строке Write-Host (5+5), PowerShell интерпретирует (5+5) как выражение, вычисляет его значение и передает результат Write-Host. В итоге PowerShell выводит 10.

Разбор команд с присваиванием значений

В PowerShell определение переменной начинается со знака доллара ($), за которым следует имя определяемой переменной. Для присваивания переменным значений используется знак равенства (=), после которого идет

присваиваемое значение. После объявления переменной на нее можно ссылаться по имени, а также выводить ее значение.

Таким образом, если ввести $a = 5+5 в командной строке PowerShell, строка 5+5 будет интерпретирована как выражение, а его значение будет присвоено переменной a. В итоге исполнение команды

```
$a
```

или

```
Write-Host $a
```

даст следующий результат:

```
10
```

Далее, можно объявить переменную $a и присвоить ей строковое значение, например:

```
$a = «This is a string.»
```

В этом случае значение $a обрабатывается как строка-литерал в режиме выражений. В этом можно убедиться, введя команду

```
$a
```

или

```
Write-Host $a
```

Получится следующий результат:

```
This is a string.
```

Иногда требуется заставить PowerShell интерпретировать строку в режиме команд. Рассмотрим такие ситуации на следующем примере:

```
$a = «Get-Process»
```

Если вывести значение переменной командой

```
$a
```

получится следующий результат:

```
Get-Process
```

Причина в том, что присваиваемое переменной $a значение обрабатывается как строка, то есть в режиме выражений, но нам требуется исполнить командлет Get-Process. Для этого необходимо заставить PowerShell интерпретировать строковое значение переменной как маркер, который должен обрабатываться в режиме команд. Это делается путем добавления оператора & к ссылке на переменную $a:

```
&$a
```

Поскольку PowerShell теперь обрабатывает строковое значение переменной в режиме команд, Get-Process интерпретируется как маркер команды и вызывается одноименный командлет, который выводит список работающих процессов. Этот метод можно использовать при назначении имен любых командлетов, ключевых слов, псевдонимов, функций и внешних утилит строковым переменным. Иногда в значение переменной, помимо имени команды, нужно записать другие значения, например, параметры команды, либо несколько команд, объединенных в конвейер. В этих случаях такую строку необходимо заключить не в кавычки, а в фигурные скобки (как блок сценария). Вот пример:

```
$a = {get-eventlog -newest 25 -logname application}
```

Присвоенное переменной $a значение обрабатывается как специальная строка в режиме выражений. Вот что получится, если вывести значение $a в командной строке PowerShell:

```
get-eventlog -newest 25 -logname system
```

Чтобы заставить PowerShell интерпретировать содержимое блока сценария как команду, введите

```
&$a
```

В результате PowerShell будет анализировать каждый маркер в составе блока сценария. Результат будет такой же, как при вводе содержимого блока сценария в командной строке.

Обработка незаконченного ввода

Если ввести в командной строке незаконченное выражение, выводится т.н. вложенное приглашение (subprompt) >>, свидетельствующее о том, что PowerShell ожидает завершения ввода. Так, если ввести **Write-Host (** и нажать Enter, PowerShell отобразит приглашение к завершению выражения, >>. Введите окончание выражения, например **5+5)**, и нажмите Enter, затем нажмите Enter еще раз (не вводя никакого текста) — PowerShell интерпретирует дополненную команду как завершенную единицу исполнения и вложенное приглашение исчезнет.

Для разбиения команды на строки используется знак обратного апострофа (`). Это удобно при копировании и вставке длинных команд в консоли PowerShell. Вот как это работает:

1. Введите часть команды, знак `и нажмите Enter — PowerShell откроет приглашение >>.
2. Введите следующую часть команды. Если ввод команды будет продолжен, снова введите `, в противном случае нажмите Enter, чтобы отметить конец команды.
3. Закончив ввод, отметив конец команды нажатием Enter (и закончив все выражения), вы позволите PowerShell выполнить синтаксический разбор команды.

Вот пример такой команды и часть ее вывода:

```
 get-eventlog -newest 25 `
>> -logname system
>>
```

```
   Index Time  EntryType   Source   InstanceID Message
   ----- ----  ---------   ------   ---------- -------
   258248 Feb 28 16:12  Information Service Control M... 1073748860 The de-
   scription for Event ID '1073748860' in So...
   258247 Feb 28 14:27  Information Service Control M... 1073748860 The
   description for Event ID '1073748860' in So...
```

Если в команде используется знак конвейера (|), он тоже может служить знаком разрыва строки, как в следующем примере:

```
 get-process |
>>  sort-object Id
>>
```

```
   Handles  NPM(K)   PM(K) WS(K) VM(M) CPU(s)   Id ProcessName
   -------  ------   ----- ----- ----- ------   -- -----------
    0     0      0       24    0     0      Idle
    710     0    0       12904   20  4      System
    28    1      360       816   4   516      smss
    666     6    1872      5212  94  592      csrss
```

Вывод результатов команды после разбора

После синтаксического разбора команд и значений PowerShell возвращает вывод. В отличие от командной оболочки Windows (Cmd.exe), встроенные команды PowerShell возвращают объекты. Объект — это набор данных, представляющих некоторую сущность. Объекты принадлежат к определенному типу данных, например String, Boolean или Numeric, у них имеются методы и свойства. Методы объекта позволяют выполнять различные действия над сущностью, которую он представляет, а свойства объекта хранят информацию о ней. Используя свойства и методы, в PowerShell можно выполнять определенные действия над объектами и манипулировать данными.

Команды, объединенные в конвейер, передают друг другу информацию в виде объектов. Первая команда конвейера отправляет один или несколько объектов второй команде конвейера. Вторая команда получает объекты от первой, обрабатывает их и выводит результаты либо передает обработанные объекты следующей команде конвейера. Так продолжается до исполнения последней команды и вывода ее результатов. Но мы с вами не можем читать информацию из объектов, поэтому PowerShell преобразует их в текст, который отображается на экране. Этим текстом можно манипулировать различными способами.

Запись и форматирование вывода

Хотя PowerShell читает и записывает объекты, в конце исполнения командлетов все свойства объектов преобразуются в текстовый вид. Говорят, что при выводе на консоль данные записываются в *поток стандартного вывода*. PowerShell поддерживает и другие типы выходных потоков, но сначала давайте разберем форматирование выходных данных по умолчанию.

Форматирование с помощью командлетов

При работе с внешними утилитами и программами эти программы определяют формат выходных данных. В случае командлетов PowerShell для форматирования их вывода используются другие командлеты, предназначенные для этой цели. Форматирующий командлет определяет, какие свойства отображаются, а также форму, в которой они отображаются (список или таблица). Отбор данных для отображения производится по их типу: строки и объекты обрабатываются по-разному.

Примечание Форматирующие командлеты готовят данные для отображ ения, но не отображают их, для этого слу жат специальные к омандлеты, о к оторых пойдет речь ниже.

Формат выходных данных можно задать явно с помощью одного из следующих командлетов:

- **Format-List** — форматирует вывод в виде списка свойств. Всем свойствам объектов назначается формат по умолчанию, согласно которому каждое свойство выводится на отдельной строке. Параметр -Properties определяет выводимые свойства по именам, имена указывают в виде списка, разделенного запятыми, разрешается использование подстановочных знаков (*).

```
Format-List [-DisplayError] [-ShowError] [-Expand Строка] [-Force]
[-GroupBy Объект] [-InputObject Объект] [-View Строка]
[[-Property] Имена_свойств]
```

- **Format-Table** — форматирует выходные данные в виде таблицы, отображая заданные свойства объектов в столбцах. Вид таблицы и свойств по умолчанию определяется типом объекта. Параметр -AutoSize автоматически настраивает ширину и число столбцов по длине значений. Параметр -HideTableHeaders отключает вывод «шапки» таблицы, а параметр -Wrap позволяет переносить текст в столбцах.

```
Format-Table [-DisplayError] [-ShowError] [-Expand Строка] [-Force]
[-GroupBy Объект] [-InputObject Объект] [-View Строка]
[-AutoSize] [-HideTableHeaders] [-Wrap] [[-Property] Имена_свойств]
```

- **Format-Wide** — форматирует вывод в виде таблицы, но выводится только одно свойства каждого объекта. Параметр -AutoSize автоматически

настраивает ширину и число столбцов по длине значений, параметр -Columns задает число столбцов.

```
Format-Wide [-DisplayError] [-ShowError] [-Expand Строка] [-Force]
[-GroupBy Объект] [-InputObject Объект] [-View Строка]
[-AutoSize] [-Column Число_столбцов] [[-Property] Имя_свойства]
```

- **Format-Custom** — форматирует вывод по шаблону, заданному путем редактирования файлов *format.PS1XML в каталоге PowerShell. Новые шаблоны (.PS1XML-файлы) создают с помощью командлета Update-FormatData. Параметр -Depth задает число отображаемых столбцов.

```
Format-Custom [-DisplayError] [-ShowError] [-Expand Строка]
[-Force] [-GroupBy Объект] [-InputObject Объект] [-View Строка]
[-Depth Число] [[-Property] Имя_свойства]
```

Перечисленными выше средствами удобно пользоваться в сочетании с другими командлетами:

- **Group-Object** — группирует объекты с идентичными значениями заданных свойств. Объекты располагаются в группе последовательно, поэтому для получения нужного результат значения необходимо отсортировать. Параметр -CaseSensitive задает группирование с учетом регистра (по умолчанию регистр символов не учитывается). Параметр -NoElement заставляет игнорировать имена элементов группы, например имена файлов при группировании файлов по их расширению.

```
Group-Object [-CaseSensitive] [-Culture Строка] [-NoElement]
[-InputObject Объект] [[-Property] Имя_свойства]
```

- **Sort-Object** — сортирует объекты по возрастанию значения заданного свойства. Параметр -Descending обращает порядок сортировки, параметр -CaseSensitive задает сортировку с учетом регистра (по умолчанию регистр символов не учитывается). Параметр -Unique исключает при сортировке дублирующиеся элементы и возвращает только уникальные.

```
Sort-Object [-Culture Строка] [-CaseSensitive] [-Descending]
[-InputObject Объект] [-Unique] [[-Property] Имя свойства]
```

Формат вывода любого из командлетов можно изменить, направив его через конвейер (|) форматирующему командлету. Например, командлет Get-Service по умолчанию выводит данные в виде таблицы со столбцами Status, Name и DisplayName, отражающими свойства объектов служб:

```
get-service
```

```
Status Name    DisplayName
------ ----    -----------
Stopped  Adobe LM Service  Adobe LM Service
Running  Adobe Version C... Adobe Version Cue CS2
```

```
Stopped  Adobe Version C... Adobe Version Cue CS3
Running  AeLookupSvc Application Experience
Running  AlertService   Intel(R) Alert Service
Stopped  ALG   Application Layer Gateway Service
```

Командлет Format-Wide может отформатировать этот вывод в виде таблицы с множеством столбцов, но покажет при этом только одно свойство для каждого из объектов. Следующая команда направляет вывод Get-Service командлету Format-Wide:

```
get-service | format-wide -column 3
```

```
Adobe LM Service  Adobe Version Cue CS2   Adobe Version Cue CS3
AeLookupSvc AlertService   ALG
AOL ACS  Appinfo  Apple Mobile Device
AppMgmt AudioEndpointBuilder  Audiosrv
BFE BITS    Bonjour Service
Browser CertPropSvc  clr_optimization_v2.0
COMSysApp CryptSvc   CscService
DcomLaunch DFSR   Dhcp
```

В результате выводятся только имена служб в виде таблицы, в которой несколько столбцов.

Получив имя службы, можно вывести сведения о ее свойствах в виде списка. Например, следующая команда получает подробные сведения о службе WinRM:

```
get-service winrm | format-list
```

```
Name :  WinRM
DisplayName : Windows Remote Management (WS-Management)
Status :   Stopped
DependentServices : {}
ServicesDependedOn  : {RPCSS, HTTP}
CanPauseAndContinue : False
CanShutdown : False
CanStop :  False
ServiceType : Win32ShareProcess
```

В этом случае данные выводятся в виде списка, а не таблицы, и включают дополнительные сведения, которые не отображались в предыдущем примере.

Все форматирующие командлеты поддерживают параметр -Properties, которым задают имена свойств. В его значении разрешается использовать подстановочные знаки (*). Например, для вывода всех свойств процесса winlogon введите

```
get-process winlogon | format-list -property *
```

```
__NounName     : Process
Name    :     winlogon
Handles    :     147
VM    :       58609664
WS    :       6696960
PM    :       2437120
NPM     :     3832
Id    :       808
PriorityClass     :
HandleCount    : 147
WorkingSet     : 6696960
PagedMemorySize :   2437120
PrivateMemorySize : 2437120
VirtualMemorySize : 58609664
```

Чтобы получить полный список свойств объекта, следует передать их командлету Get-Member. Так, следующая команда показывает все свойства объекта службы:

```
get-service | get-member -membertype *property
```

```
TypeName: System.ServiceProcess.ServiceController
Name MemberType   Definition
---- ----------   ----------
Name  AliasProperty Name = ServiceName
CanPauseAndContinue Property  System.Boolean CanPauseAndContinue {get;}
CanShutdown Property System.Boolean CanShutdown {get;}
CanStop  Property System.Boolean CanStop {get;}
Container Property   System.ComponentModel.IContainer
DependentServices Property System.ServiceProcess.ServiceController
DisplayName Property System.String DisplayName {get;set;}
MachineName Property System.String MachineName {get;set;}
ServiceHandle Property  System.Runtime.InteropServices.SafeHand
ServiceName Property System.String ServiceName {get;set;}
ServicesDependedOn  Property  System.ServiceProcess.ServiceController
ServiceType Property System.ServiceProcess.ServiceType
Site Property  System.ComponentModel.ISite Site
Status Property   System.ServiceProcess.ServiceController
```

Все эти свойства принадлежат объекту службы, полученному командлетом Get-Service, для их отображения используется параметр -Property. Например, в следующей команде используется Format-Table для вывода ограниченного набора свойств каждой из служб (Name, Status, ServiceType и ServicesDependedOn):

```
get-service | format-table Name, Status, ServiceType, ServicesDependedOn
```

```
Name Status    ServiceType ServicesDependedOn
----- ------- ------------ ------------------
Adobe LM Service  Stopped  Win32OwnProcess {}
Adobe Version C...   Running  Win32OwnProcess {}
Adobe Version C...   Stopped  Win32OwnProcess {}
AeLookupSvc Running Win32ShareProcess {}
AlertService   Running ...ractiveProcess {}
ALG Stopped    Win32OwnProcess {}
AOL ACS  Running ...ractiveProcess {}
Appinfo  Stopped Win32ShareProcess {ProfSvc, RpcSs}
Apple Mobile De...   Running  Win32OwnProcess {Tcpip}
AppMgmt  Stopped Win32ShareProcess {}
AudioEndpointBu...   Running Win32ShareProcess {PlugPlay}
Audiosrv Running Win32ShareProcess {AudioEndpoint...
BFE   Running Win32ShareProcess {RpcSs}
BITS  Running Win32ShareProcess {EventSystem, ...
Bonjour Service   Running  Win32OwnProcess {Tcpip}
Browser  Running Win32ShareProcess {LanmanServer,...
CertPropSvc Running Win32ShareProcess {RpcSs}
clr_optimizatio... Stopped    Win32OwnProcess {}
COMSysApp   Stopped  Win32OwnProcess {SENS, EventSy...
```

Помимо форматирования отображаемых данных можно сгруппировать и отсортировать объекты. Все форматирующие командлеты поддерживают параметр -GroupBy, объединяющий в группы объекты с идентичным значением заданного свойства.

Параметр -GroupBy дает те же результаты, что и направление вывода командлету Group-Object с последующей передачей его вывода форматирующему командлету. Но вряд ли этот прием позволит отформатировать данные как вам нужно, поскольку для каждого значения заданного свойства такая команда генерирует отдельный заголовок. Например, так можно сгруппировать службы по их состоянию (Running или Stopped):

```
get-service | format-list -groupby status
```

```
Status: Stopped
Name :  WinRM
DisplayName : Windows Remote Management (WS-Management)
Status :   Stopped
DependentServices : {}
ServicesDependedOn  : {RPCSS, HTTP}
CanPauseAndContinue : False
CanShutdown : False
CanStop :  False
ServiceType : Win32ShareProcess
```

```
Status: Running
Name :  Wlansvc
DisplayName : WLAN AutoConfig
Status :   Running
DependentServices : {}
ServicesDependedOn  : {Eaphost, RpcSs, Ndisuio, nativewifip}
CanPauseAndContinue : False
CanShutdown : True
CanStop :  True
ServiceType : Win32ShareProcess
```

При группировании служб по состоянию с помощью Group-Object получаются совершенно другие результаты:

```
get-service | group-object status
```

```
Count Name  Group
----- ----  -----
 68   Stopped  {System.ServiceProcess.ServiceControll
 89   Running  {System.ServiceProcess.ServiceControll
```

Можно пользоваться и такими командами, но ни одна из них не выводит сначала список всех остановленных, а затем список всех работающих служб. Решение в том, чтобы сначала отсортировать объекты и только потом группировать их. Для сортировки объектов используется командлет Sort-Object, поддерживающий сортировку по значению одного или нескольких свойств. Свойства – критерии сортировки задают с помощью параметра –Property; если свойств несколько, их разделяют запятыми. Так, следующая команда сортирует список служб сначала по состоянию, а затем по имени:

```
get-service | sort-object status, name | format-table -groupby status
```

```
Status: Stopped
Status Name    DisplayName
------ ----    -----------
Stopped  Adobe LM Service  Adobe LM Service
Stopped  Adobe Version C... Adobe Version Cue CS3
Stopped  ALG   Application Layer Gateway Service
Stopped Appinfo Application   Information
Stopped AppMgmt Application   Management
Stopped  clr_optimizatio... Microsoft .NET Framework NGEN v2.0
Stopped  COMSysApp   COM+ System Application
Stopped DFSR   DFS Replication
Stopped dot3svc Wired   AutoConfig

 Status:   Running
Status Name    DisplayName
------ ----    -----------
```

```
Running  Adobe Version C... Adobe Version Cue CS2
Running AeLookupSvc  Application Experience
Running  AlertService   Intel(R) Alert Service
Running  AOL ACS  AOL Connectivity Service
Running  Apple Mobile De... Apple Mobile Device
Running  AudioEndpointBu... Windows Audio Endpoint Builder
Running Audiosrv  Windows Audio
Running  BFE   Base Filtering Engine
Running  BITS  Background Intelligent Transfer Ser...
```

По умолчанию сортировка осуществляется по возрастанию, отсортировать список по убыванию позволяет параметры -Descending. Например, сортировка вывода Get-Process по убыванию позволяет выявить наиболее «ресурсоемкий» процесс на компьютере. Вот как это делается:

```
get-process | sort-object ws -descending
```

```
Handles  NPM(K)   PM(K) WS(K) VM(M) CPU(s)   Id ProcessName
------- ------    ----- ----- ----- ------   -- -----------
   481   22 97236 79804 371   4  31 5276  powershell_ise
   1057  39 84208 78912 279   17 94 3664  iexplore
   743   16 69532 74688 188   1120  svchost
   1377  44 62880 73172 218   1132  svchost
   763 14   89156 66536 347 784 VersionCueCS2
   459   12 58148 64140 201   7.88  5520 powershell
   596   23 39208 63676 412   110.25   5400 WINWORD
   614   27 33284 56512 421   5.22  5776  OUTLOOK
   739   0 0 55780   68 4 System
   1113  12 45712 43988 154   2560  SearchIndexer
   588   19 26768 34592 193   5.33  1704 explorer
   378   13 54952 34056 132   1004  svchost
```

По умолчанию сортировка ведется без учета регистра символов. Задать сортировку с учетом регистра позволяет параметр -CaseSensitive, а параметр -Unique включает отображение лишь объектов с уникальными значениями заданного свойства, объекты с дублирующимися значениями этого свойства исключаются из списка результатов. Это удобно, например, при сортировке по бизнес-имени, но неприемлемо, скажем, при сортировке по имени процесса, почему — рассмотрим на примере следующей команды, которая показывает список процессов, упорядоченный по имени:

```
get-process | sort-object name
```

```
Handles  NPM(K)   PM(K) WS(K) VM(M) CPU(s)   Id ProcessName
-------  ------   ----- ----- ----- ------   -- -----------
   52 3  1292 3856 51   0.00 2204 acrotray
   139   4  2556 7356 75   1380 AlertService
   586 8    22100 19900 132 652 csrss
```

```
    658   6  1720 5160 95   592   csrss
    57 2  884   2860  23 2100  svchost
    306   24 17576 21816 86 1828  svchost
    680   20 20688 24184 119   1524  svchost
    135   5  2764 6036 52   2112 svchost
    105   4  1436 4076 46   640   wininit
    147   4  2380 6540 56   808   winlogon
    598   23 39240 63696 413   111.77   5400  WINWORD
```

В этом выводе некоторые имена процессов, например powershell и svchost, повторяются. Если ввести команду

```
get-process | sort-object name -unique
```

```
Handles  NPM(K)   PM(K) WS(K) VM(M) CPU(s)   Id ProcessName
-------  ------   ----- ----- ----- ------   -- -----------
   52 3  1292 3856 51   0.00 2204 acrotray
   139   4  2556 7356 75   1380 AlertService
   586 8    22100 19900 132 652 csrss
   57 2  884   2860  23 2100  svchost
   105   4  1436 4076 46   640   wininit
   147   4  2380 6540 56   808   winlogon
   598   23 39240 63696 413   111.77   5400 WINWORD
```

будут показаны только первые копии повторяющихся процессов. По такому выводу невозможно понять, сколько процессов работает в системе и сколько ресурсов они используют.

Запись в выходные потоки

Windows PowerShell поддерживает несколько командлетов Write для записи данных в различные выходные потоки. Прежде всего, нужно знать, что эти командлеты не отображают выходные данные, а просто направляют их (по конвейеру) в заданный выходной поток. Некоторые выходные потоки изменяют формат выходных данных, но за подготовку и завершение вывода на самом деле отвечают командлеты Output (о них — в следующем разделе).

Поддерживаются следующие типы выходных потоков:

- поток стандартного вывода;
- поток подробного (verbose) вывода;
- поток предупреждений;
- поток отладочных сообщений;
- поток сообщений об ошибках.

Явный вывод

Для явного вывода служат следующие командлеты:

- **Write-Host** — записывает данные в поток стандартного вывода, позволяет задавать цвета для текста и фона. По умолчанию выводимый текст

завершается символом конца строки, параметр -NoNewLine позволяет записывать выходные данные без этого символа. Параметр -Separator задает строку, которая разделяет выводимые объекты, а параметр -Object — объект или литерал для вывода.

```
Write-Host [-BackgroundColor Цвет] [-ForegroundColor Цвет]
[-NoNewline] [-Separator Объект] [[-Object] Объект]
```

- **Write-Output** — отправляет заданный объект по конвейеру следующей команде либо выводит его на консоли. В отличие от предыдущего командлета, принимает объекты.

```
Write-Output [[-InputObject] Объект]
```

Командлет Write-Host используют в основном ради возможностей форматирования, которые он предоставляет, включая назначение цветов тексту и фону. Параметр -BackgroundColor задает цвет для фона, а -ForegroundColor — цвет для текста. Поддерживаются следующие цвета:

- Black (черный), DarkBlue (темно-синий), DarkGreen (темно-зеленый), DarkCyan (темно-голубой);
- DarkRed (темно-красный), DarkMagenta (темно-пурпурный), DarkYellow (темно-желтый), Gray (серый);
- DarkGray (темно-серый), Blue (синий), Green (зеленый), Cyan (голубой);
- Red (красный), Magenta (пурпурный), Yellow (желтый), White (белый).

Следующая команда задает вывод черного текста на желтом фоне:

```
write-host -backgroundcolor yellow -foregroundcolor black «This is text!»

This is text!
```

Примечание Командлет Write-Host передает вывод хост-приложению PowerShell. Как правило, это консоль (powershell.exe) или графическая среда PowerShell (powershell_ise.exe). Хостом для исполняющей среды PowerShell (PowerShell engine) могут быть другие приложения, иначе обрабатывающие вывод, переданный Write-Host. Это означает, что Write-Host следует использовать, только если хост приложение и его алгоритм обработки вывода известны.

Командлет Write-Output также записывает данные в поток стандартного вывода, но, в отличие от Write-Host, принимает объекты в качестве входных данных. Однако Write-Output служит просто для передачи заданных объектов следующим командлетам конвейера. Последняя команда конвейера выводит объект на консоли.

Так, Write-Output используют, когда требуется явно задать запись данных в поток вывода, например:

```
get-process | write-output
```

```
Handles  NPM(K)   PM(K) WS(K)  VM(M)   CPU(s)   Id    ProcessName
-------- -----    ----- -----  -----   --       --    ---------
 52    3      1292      3856   51    0.00     2204    acrotray
 139     4    2556      7356   75       1380          AlertService
```

При исполнении командлета Get-Process его вывод передается по конвейеру Write-Output, который выводит данные на консоль.

При работе с переменным Write-Output также удобен для явного определения вывода. Рассмотрим следующий пример:

```
$p = get-process; $p
```

```
Handles  NPM(K)   PM(K) WS(K) VM(M) CPU(s)   Id    ProcessName
-------  ------   ----- ----- ----- ------   --    -----------
52   3        1292      3856  51  0.00     2204    acrotray
139    4      2556      7356  75     1380          AlertService
```

Здесь объявляется переменная $p, в которой сохраняется набор объектов-процессов, после чего эти содержимое переменной выводится на консоли. Чтобы объявить вывод явно, нужно переписать предыдущую команду следующим образом:

```
$p = get-process; write-output $p
```

```
Handles  NPM(K) PM(K)   WS(K) VM(M) CPU(s)   Id    ProcessName
-------  ------ -----   ----- ----- ------   --    -----------
52   3        1292      3856  51  0.00     2204    acrotray
139    4      2556      7356  75     1380          AlertService
```

Работа с другими потоками вывода

Для работы с потоками, отличными от потока стандартного вывода, используются следующие командлеты:

- **Write-Debug** — выводит на консоли сообщения сценария или команды. По умолчанию отладочные сообщения не выводятся и не прерывают исполнение. Все командлеты поддерживают параметр -Debug для вывода отладочных сообщений, той же цели служит и переменная $DebugPreference variable. Параметр -Debug переопределяет значение переменной $DebugPreference для текущей команды.

  `Write-Debug [-message]` *Отладочное_сообщение*

- **Write-Error** — выводит на консоли сообщения об ошибках из сценариев и команд. По умолчанию сообщения об ошибках выводятся, но не прерывают исполнение. Вывод этих сообщений настраивают с помощью параметра -ErrorAction, который поддерживают все командлеты; той же цели служит и переменная $ErrorActionPreference variable. Параметр -ErrorAction переопределяет значение переменной $ErrorActionPreference для текущей команды.

```
Write-Error -ErrorRecord ErrorRecord [Доп._параметры]

Write-Error [-TargetObject Object] [-Message] Строка
[-ErrorId Строка] [Доп._параметры]

Write-Error -Exception Исключение [-Category Строка] [Доп._параметры]

AddtlParams=
[-CategoryTargetName Строка] [-CategoryTargetType  Строка]
[-CategoryReason Строка] [-CategoryActivity Строка]
[-RecommendedAction Строка]
```

- **Write-Warning** — выводит на консоли предупреждения из сценариев и команд. По умолчанию предупреждения выводятся, но не прерывают исполнение. Вывод предупреждений настраивают с помощью параметра -WarningAction, который поддерживают все командлеты; той же цели служит и переменная $WarningPreference variable. Параметр -WarningAction переопределяет значение переменной $WarningPreference для текущей команды:

  ```
  Write-Warning [-message] Предупреждение
  ```

- **Write-Verbose** — записывает на консоль подробный (verbose) вывод сценария или команды. По умолчанию подробные сообщения не выводятся и не прерывают исполнение. Все командлеты поддерживают параметр -Verbose для вывода подробных сообщений, той же цели служит и переменная $VerbosePreference variable. Параметр -Verbose переопределяет значение переменной $VerbosePreference для текущей команды:

  ```
  Write-Verbose [-message] Подробное_сообщение
  ```

Командлетами Write-Debug, Write-Error, Write-Warning и Write-Verbose управляют с помощью общих параметров либо переменных (см. выше). Соответствующие параметры принимают значения $true или $false, а переменные — следующие значения:

- Stop
- Inquire
- Continue
- SilentlyContinue

Переменную $DebugPreference, управляющую обработкой отладочных сообщений, можно определить одним из следующих способов:

- $DebugPreference=Stop — выводит отладочного сообщения и останов исполнения;
- $DebugPreference=Inquire — вывод отладочного сообщения и запрос у пользователя разрешения продолжить исполнение;
- $DebugPreference=Continue — вывод отладочного сообщения и продолжение исполнения;

- $DebugPreference=SilentlyContinue — продолжение исполнения без вывода отладочных сообщений.

Учтите, что параметр -Debug переопределяет значение переменной $DebugPreference при исполнении команды, для которой он задан. Чтобы включить отладку, укажите -Debug:$true или -Debug; параметр -Debug:$false отключает отладку (если переменной $DebugPreference задано значение, отличное от SilentlyContinue).

Подготовка и завершение вывода

Включает ли командная строка единственный командлет или конвейер, используется ли явное форматирование вывода или нет, в любом случае в завершение ее синтаксического разбора и вывода результатов выполняется скрытый фоновый вызов командлета для вывода (как правило, Out-Host).

Также разрешается явно указывать один из следующих командлетов для вывода:

- **Out-File** — направляет вывод в файл, расположенный по заданному пути. Если выходной файл существует, можно перезаписать его (указав параметр -Force) или дописать к нему новые данные (с помощью параметра -Append). Возможно также использовать Out-File вместо стандартных методов перенаправления (см. следующий раздел).

```
Out-File [-InputObject Объект] [-NoClobber] [-Width Число_знаков]
[-Force] [-Append] [-FilePath] Строка [[-Encoding] Строка]
```

- **Out-GridView** — направляет вывод в интерактивную таблицу, которая открывается в отдельном окне и поддерживает сортировку, копирование и фильтрацию результатов, а также объединение.

```
Out-GridView [-InputObject Объект]
```

- **Out-Host** — направляет вывод в командную строку. Параметр -Paging включает постраничный вывод (аналогично команде More в командной строке Windows).

```
Out-Host [-InputObject Объект] [-Paging]
```

- **Out-Null** — направляет вывода на null-порт, удаляя его без отображения. Это удобно, когда требуется избавиться от ненужного вывода.

```
Out-Null [-InputObject Объект]
```

- **Out-Printer** — направляет вывод на принтер по умолчанию либо на принтер, заданный параметром -Name. Этот параметр принимает UNC-путь к принтеру, например -Name “\\PrintServer85\LaserP45”/.

```
Out-Printer [-InputObject Объект] [[-Name] Строка]
```

- **Out-String** — преобразует все объекты в выводе в единую строку и направляет результат на консоль. Параметр -Stream заставляет направлять

строки для каждого из объектов по отдельности. Параметр -Width задает длину строки в знаках (значение по умолчанию — 80 знаков); строки, превышающие заданный предел, усекаются.

```
Out-String [-InputObject Объект] [-Width Число_знаков] [-Stream]
```

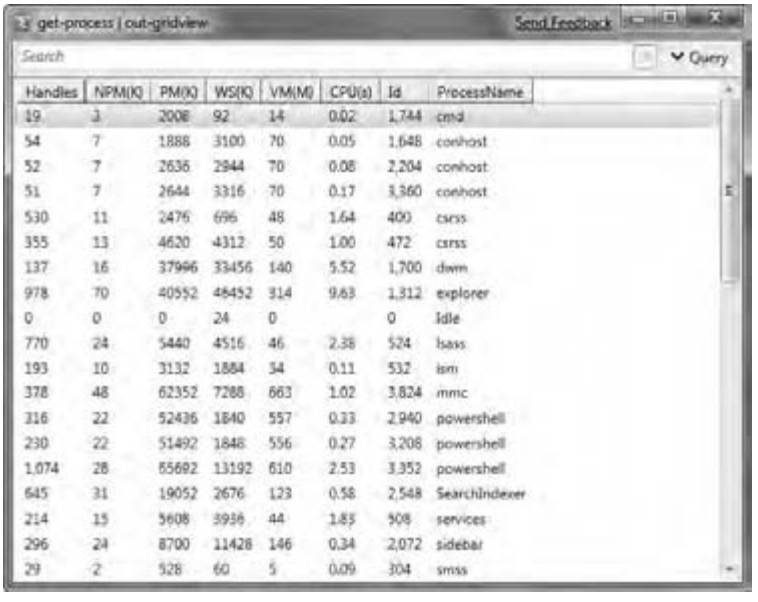
Все перечисленные выше командлеты принимают объекты, которые можно передавать им через конвейер. Следующая команда переписывает сведения о событиях из журнала Application в файл C:\logs\app\current.txt:

```
get-eventlog -newest 10 -logname application | out-file -filepath c:\
logs\app\current.txt
```

Все эти командлеты также поддерживают параметр -InputObject, позволяющий задавать входной объект. Вот как с его помощью вывести сведения о текущем процессе в интерактивной таблице (см. рис. 2-1):

```
$p = get-process; out-gridview -inputobject $p
```

Поскольку такие команды не интерпретирую аргументы в зависимости от их порядка, необходимо объявлять параметр -InputObject явно.



Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
19	3	2008	92	14	0.02	1,744	cmd
54	7	1888	3100	70	0.05	1,648	conhost
52	7	2636	2944	70	0.08	2,204	conhost
51	7	2644	3316	70	0.17	3,360	conhost
530	11	2476	696	48	1.64	400	csrss
355	13	4620	4312	50	1.00	472	csrss
137	16	37996	33456	140	5.52	1,700	dwm
978	70	40552	46452	314	9.63	1,312	explorer
0	0	0	24	0		0	Idle
770	24	5440	4516	46	2.38	524	lsass
193	10	3132	1884	34	0.11	532	lsm
378	48	62352	7288	663	1.02	3,824	mmc
316	22	52436	1840	557	0.33	2,940	powershell
230	22	51492	1848	556	0.27	3,208	powershell
1,074	28	65692	13192	610	2.53	3,352	powershell
645	31	19052	2676	123	0.58	2,548	SearchIndexer
214	15	5608	3936	44	1.85	908	services
296	24	8700	11428	146	0.34	2,072	sidebar
29	2	528	60	5	0.09	304	smss

Рис. 2-1. Вывод результатов исполнения команды в интерактивной таблице

Еще о перенаправлении

По умолчанию извлекают входные данные из значений параметров, заданных при вызове команды, а затем направляют свой вывод, включая сообщения об ошибках, в стандартное окно консоли. Однако иногда требуется передавать командам входные данные из других источников, а также направлять

вывод в файлы и другие устройства ввода-вывода, например на принтер. Кроме того, сообщения об ошибках в некоторых случаях необходимо записывать в файл, а не выводить на консоли. В дополнение к командлетам для вывода (см. выше), для перенаправления ввода-вывода используются приемы, описанные в табл. 2-3 и проиллюстрированные примерами ниже.

Табл. 2-3. Методы перенаправления ввода-вывода и сообщений об ошибках

Метод	Описание
команда1 | команда2	Передача вывода первой команды на ввод второй
команда > [путь] файл	Направление вывода в заданный файл. Если файл не существует, он создается; существующий файл может быть перезаписан или дописан
command >> [путь] файл	Дописывает вывод к заданному файлу; если файл не существует, он создается
команда 2> [путь] файл	Создает заданный файл и записывает в него сообщения об ошибках; если файл существует, он перезаписывается
command 2>> [путь] файл	Дописывает сообщения об ошибках к заданному файлу; если файл не существует, он создается
команда 2>&1	Направляет сообщения об ошибках туда же, куда и стандартный вывод

Основной метод перенаправления ввода-вывода — использование конвейеров (см. примеры к этой главе). Другой распространенный метод — направление вывода в файл, например с помощью командлета Out-File, оператора > или >>. Оператор > создает заданный файл либо перезаписывает его, если файл существует, а оператор >> создает заданный файл либо дописывает к нему данные, если этот файл существует. Так, чтобы записать состояние текущего процесса в файл, введите следующую команду:

```
get-process > processes.txt
```

Только учтите: если в текущем каталоге окажется файл, имя которого совпадает с именем файла, заданного для команды, этот файл будет перезаписан. Чтобы дописать новые данные, сохранив существующий файл, команду нужно изменить следующим образом:

```
get-process >> processes.txt
```

По умолчанию сообщения об ошибках выводятся в командной строке. Как сказано выше, потоком сообщений об ошибках управляют с помощью командлета Write-Error, параметра -ErrorAction (этот параметр — общий для всех командлетов) и переменной $ErrorActionPreference. Директива 2>&1 позволяет направлять стандартные сообщения об ошибках вместе со стандартным выводом в заданный приемник, как показано ниже:

```
chkdsk /r > diskerrors.txt 2>&1
```

Здесь стандартные сообщения об ошибках направляются вместе со стандартным выводом в файл Diskerrors.txt. Чтобы записывать в этот файл только сообщения об ошибках, следует направить в него только поток ошибок. В этом примере стандартный вывод направляется в командную строку, а стандартные ошибки — в файл Diskerrors.txt:

```
chkdsk /r 2> diskerrors.txt
```

Если этот файл существует, он будет автоматически перезаписан. Чтобы вместо этого дописать данные к файлу, воспользуйтесь командой следующего вида:

```
chkdsk /r 2>> diskerrors.txt
```

Глава 3

Управление средой PowerShell

При запуске Windows PowerShell рабочая среда загружается автоматически. Многие аспекты рабочей среды определяются профилями, которые представляют собой особые сценарии, исполняемые при запуске PowerShell. Параметры рабочей среды также определяются установленными модулями, оснастками, поставщиками, значением PATH и файловыми сопоставлениями. Все эти функции рассматриваются в данной главе.

Кроме того, среда для удаленной работы отличается от среды для локальной работы. По этой причине для работы на удаленных и локальных компьютерах используются разные методики. Впрочем, PowerShell 2.0 поддерживает не только удаленное исполнение команд, но и удаленные сеансы, и фоновые задания (подробнее об этом — в главе 4).

Работа с профилями

Файлы профилей можно отличить по расширению .ps1. В общем случае профили загружаются при каждом запуске PowerShell, но есть несколько исключений. Например, для отладки сценариев иногда требуется среда, запущенная без загрузки профиля. Так удается проверить корректность сценария и убедиться в отсутствии в нем параметров, зависимых от профиля.

В профилях хранятся часто используемые элементы, а именно:

- **псевдонимы** — альтернативные имена для команд, функций, сценариев, файлов (в том числе исполняемых) и других элементов команд. Псевдонимы используются для ускоренного вызова соответствующего элемента команды. Так, gsv — псевдоним для командлета Get-Service. Соответственно, вместо **get-service winrm** для получения информации о службе WinRM достаточно ввести **gsv winrm**. Для вывода полного списка псевдонимов используйте команду **get-alias**.

  ```
  Get-Alias [-Exclude Строка] [[-Name Строки] | [-Definition Строки]]
  [-Scope Строка]
  ```

- **функции** — это именованные наборы команд PowerShell. При вызове функции соответствующие команды поочередно исполняются, как если

бы они были отданы через командную строку. Например, можно создать функцию для генерации отчета о работе критически важных процессов и служб на компьютере. Добавив эту функцию в профиль, вы сможете ее вызывать ее в любое время, вводя имя этой функции в командной строке PowerShell. Для вывода списка всех функций введите **get-childitem function:**

```
Get-ChildItem [[-Filter] Строки] [-LiteralPath] Строки [AddtlParams]
Get-ChildItem [[-Path] Строки] [[-Filter] Строки] [AddtlParams]
AddtlParams=
[-Exclude Строки] [-Force] [-Include Строки] [-Name] [-Recurse]
```

- **переменные** — это строки, вместо которых подставляются реальные значения. В дополнение к переменным среды операционной системы PowerShell поддерживает переменные, объявленные автоматически, управляющие переменные (preference) и переменные, объявленные пользователем. Для ссылки на переменную в командах и сценариях необходимо ввести ее имя, предварив его знаком доллара ($), например **$home**. Для получения списка переменных введите **get-variable** в командной строке PowerShell.

```
Get-Variable [-Scope Строки] [-Exclude Строки] [-Include Строки]
[-ValueOnly] [[-Name] Strings]
```

Примечание В командах и сценариях разрешается использовать любые доступные переменные. Автоматически объявленные переменные (automatic variables) являются фиксированными и служат для хранения информации о состоянии. Управляющие переменные (preference variables) могут устанавливаться пользователем, в них хранятся рабочие значения параметров конфигурации PowerShell. Переменные, объявленные пользователем, по умолчанию существуют только в текущем сеансе и пропадают после выхода из командной оболочки либо завершения сеанса. Чтобы сохранить переменные, объявленные пользователем, необходимо поместить их в профиль (подробнее о переменных — в главе 5).

Совет Для просмотра значения автоматически объявленной или управляющей переменной достаточно ввести ее имя в командной строке PowerShell. Например, для просмотра значения переменной $home введите **$home**. Доступ к переменным окружения осуществляется немного иначе: перед именем такой переменной необходимо ввести $env:. Например, для просмотра переменной %ComputerName% следует ввести **$env:computername**.

Создание профилей

Профили можно создавать в стандартном текстовом редакторе. Просто введите определения псевдонимов, функций, переменных и других нужных вам элементов, а затем сохраните файл в соответствующем каталоге — вот и все. Разберем теперь эту процедуру поподробнее.

1. В Блокноте или любом другом текстовом редакторе введите необходимые команды, определения псевдонимов, функций, переменных и других нужных вам элементов.

2. Сохраните файл со стандартным расширением профилей, например Profile.ps1.
3. Скопируйте файл в соответствующий каталог, например в $pshome.

При работе с консолью и графической средой PowerShell следует знать о шести типах профилей (табл. 3-1). $home и $pshome — автоматически объявленные переменные, в $home хранится путь к домашней папке текущего пользователя, а в $pshome — путь к установочному каталогу PowerShell.

Табл. 3-1. Общие типы профилей PowerShell

Типы	Описание	Размещение
Current User, PowerShell Console	Профиль, связанный с контекстом учетной записи текущего пользователя (только для консоли PowerShell)	Каталог: $home\[My]Documents\WindowsPowerShell Имя: profile.ps1
Current User, PowerShell ISE	Профиль, связанный с контекстом учетной записи текущего пользователя (только для графической среды PowerShell ISE)	Каталог: $home\[My]Documents\WindowsPowerShell Имя: Microsoft.PowerShellISE_profile.ps1
Current User, All Hosts	Профиль, связанный с контекстом учетной записи текущего пользователя (для консоли и графической среды PowerShell ISE)	Каталог: $home\[My]Documents Имя: profile.ps1
All Users, PowerShell Console	Профиль, управляющий консолью PowerShell (для всех пользователей)	Каталог: $pshome Имя: Microsoft.PowerShell_profile.ps1
All Users, PowerShell ISE	Профиль, управляющий средой PowerShell ISE (для всех пользователей)	Каталог: $pshome Имя: Microsoft.PowerShellISE_profile.ps1
All Users, All Hosts	Профиль, управляющий консолью и средой PowerShell ISE (для всех пользователей)	Каталог: $pshome Имя: profile.ps1

При запуске PowerShell ищет профили в вышеперечисленных каталогах и загружает их в следующем порядке:

1. Общий профиль для всех пользователей и сред.
2. Профиль для всех пользователей и загруженной среды (консоли либо ISE).
3. Профиль для текущего пользователя и всех сред.
4. Профиль для текущего пользователя и загруженной среды (консоли либо ISE).

Порядок обработки профилей определяет алгоритм разрешения возможных конфликтов: при конфликте приоритет имеет значение, установленное последним. Таким образом, псевдоним, объявленный в профиле Current User, PowerShell либо Current User, PowerShell ISE, переопределит любые конфликтующие объявления в других профилях.

Порядок исполнения элементов сценариев и профилей

При работе с командами и сценариями, а также с профилями PowerShell важно учитывать порядок поиска элементов и значение переменной окружения PATH. Последовательность поиска и исполнения команд в PowerShell такова:

1. **Псевдонимы** Сначала PowerShell просматривает встроенные и объявленные в профиле определения в поисках команд, сопоставленных псевдониму. Если такая команда будет обнаружена, она запустится.
2. **Функции** Далее PowerShell ищет имя команды среди встроенных или объявленных в профиле функций. Если команда будет найдена среди функций, будет исполнена соответствующая функция.
3. **Командлеты и ключевые слова** Затем PowerShell ищет команду среди встроенных командлетов и ключевых слов языка программирования. Если команда совпадет с именем командлета или ключевым словом, будет выполнено соответствующее действие.
4. **Сценарии** Далее поиск ведется среди сценариев (файлов с расширением .ps1). Если будет обнаружен соответствующий сценарий, она запустится.
5. **Внешние команды и утилиты** В завершение PowerShell просматривает внешние команды, другие сценарии и утилиты. Обнаружив (в заданном PATH каталоге) внешнюю команду или файл соответствующую заданному имени, PowerShell выполняет соответствующее действие. Если найден файл, PowerShell запускает сопоставленное ему приложение.

Вводя команду **dir** и нажимая Enter чтобы получить список файлов в текущем каталоге, вы ожидаете, что будет запущенная встроенная команда командной оболочки Windows (cmd.exe). Однако из-за вышеописанного порядка исполнения при вводе **dir** в командной строке PowerShell запускается команда Get-ChildItem из PowerShell, а не встроенная команда Windows. Почему? А потому, что PowerShell вызывает команды командной оболочки Windows, только если среди команд или псевдонимов PowerShell не удалось найти соответствующую команду. Поскольку PowerShell находит строку «dir» в объявлении псевдонима для Get-ChildItem, при вводе **dir** запускается командлет Get-ChildItem.

Путь к командам и переменная PATH

Для поиска исполняемых файлов в операционной системе Windows используется переменная PATH. Исполняемым в Windows считаются файлы

с определенными расширениями. Кроме того, с помощью файловых сопоставлений можно привязывать файлы с различными расширениями к приложениям, которые будут открывать и обрабатывать их.

Настройка пути к командам

Для просмотра текущего пути к исполняемым файлам следует вывести значение переменной окружения PATH. Для этого откройте консоль PowerShell, введите **$env:path** и нажмите Enter. Вы должны получить примерно такой результат:

```
C:\Windows\System32;C:\Windows;C:\Windows\System32\Wbem;
C:\Windows\System32\WindowsPowerShell\v1.0\
```

Примечание Обратите внимание на то чку с запятой (;), к оторой разделяются отдельные пути. Этим знаком отмечают конец одного пути и начало другого.

Путь к командам устанавливается во время входа в систему с использованием системных и пользовательских переменных окружений %PATH%. Порядок каталогов в значении этой переменной определяет порядок поиска исполняемых файлов средой PowerShell:

1. C:\Windows\System32
2. C:\Windows
3. C:\Windows\System32\Wbem
4. C:\Windows\System32\PowerShell\v1.0

Команда SETX позволяет изменить путь к командам, заданный переменными среды. Например, если вы используете определенные каталоги для хранения сценариев, имеет смысл внести их в PATH. Это делается с помощью команды SETX, которая добавляет заданный путь к текущему значению PATH, например: **setx PATH "%PATH%;C:\Scripts"**.

Примечание Заметьте, что пути заключаются в кавычки и разделяются точкой с запятой. Кавычки гарантируют, что с трока %PATH%;C:\Scripts будет интерпретирована как второй аргумент команды SETX. Поскольку путь к командам устанавливается при запуске к онсоли PowerShell, необх одимо перезапус тить к онсоль, чтобы изменения пути вступили в силу. Впрочем, проверить новый путь к командам можно и без выхода из системы. Для этого нужно открыть окно **System Properties (Свойства системы)**: в **Control Panel** | **System (Панель управления** | **Система)** щелкните на панели **Tasks (Задачи)** пункт **Advanced System Settings (Дополнительные параметры системы)**, затем выберите **Environment Variables (Переменные окружения)** на вкладке **Advanced (Дополнительно)**.

В этом примере каталог C:\Scripts добавляется к пути, показанному в предыдущем примере; в результате путь он принимает следующий вид:

```
C:\Windows\System32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\
System32\PowerShell\v1.0;C:\Scripts
```

Не забывайте о порядке, в котором Windows ищет команды. Поскольку поиск ведется в том порядке, в каком каталоги перечислены в пути к командам, поиск в каталоге C:\Scripts выполняется в последнюю очередь. Иногда это может замедлить выполнение ваших сценариев. Чтобы Windows быстрее находила сценарии, можно сделать C:\Scripts первым каталогом, просматриваемым при поиске. Для этого путь к командам следует задать следующим образом:

```
setx PATH «C:\Scripts;%PATH%»
```

Указывая путь к командам, соблюдайте осторожность. Очень легко нечаянно перезаписать всю информацию о путях. Например, если при задании пути к командам не указать переменную окружения *%PATH%*, потеряется вся остальная информация о путях к командам. Один из способов обеспечить возможность воссоздать путь к командам — создать файл, содержащий копию пути к командам. Чтобы записать в файл текущий путь к командам, введите **$env:path > orig_path.txt**. Здесь тоже следует учесть, что при использовании оболочки с правами обычного пользователя (а не администратора) вы не сможете записывать данные в защищенные системные каталоги. Для записи вам будут доступны папки личного профиля, а также каталоги, доступ к которым вам явно разрешен.

Расширения файлов и файловые сопоставления

Именно благодаря расширениям файлов становится возможным запуск команд простым вводом ее имени в командной строке. Используется два типа расширений файлов.

- **Расширения исполняемых файлов** Задаются переменной окружения *%PATHEXT%*. Чтобы посмотреть ее текущее значение, введите **$env:pathext** в командной строке. По умолчанию PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC;.PSC1. По значению этой переменной командная строка определяет, какие файлы являются исполняемыми, а какие — нет, поэтому для запуска исполняемого файла не обязательно указывать в командной строке его расширение.
- **Расширения файлов для приложений** Расширения файлов, открываемых приложениями, используются в файловых сопоставлениях. *Файловые сопоставления* (*file associations*) — то, что позволяет передавать аргументы исполняемым файлам и открывать документы, электронные таблицы и другие файлы двойным щелчком значка файла. Для каждого известного системе расширения имеется файловое сопоставление, которое можно увидеть, введя **cmd /c assoc**, а затем расширение, например **cmd /c assoc .exe**. Каждое сопоставление задает тип файла для данного расширения файла. Информацию о типе файла можно выяснить, введя команду FTYPE и тип файла, например **cmd /c ftype exefile**.

Примечание Заметьте, что команды ASSOC и FTYPE вызываются через командную оболочку Windows, поскольку это ее встроенные команды.

При поиске исполняемых файлов в каталоге командная оболочка ищет файлы в порядке, в котором перечислены их расширения. Таким образом, если в данном каталоге содержится несколько исполняемых файлов с одинаковыми именами, .com-файл имеет приоритет перед .exe-файлом и т. д.

Для каждого известного системе расширения файла и даже для расширений исполняемых файлов существуют сопоставление и тип файла. В большинстве случаев тип файла — это расширение файла, после которого идет ключевое слово *file* (без точки), например, cmdfile, exefile или batfile, а файловое сопоставление задает, что первый параметр — это имя команды, остальные параметры передаются приложению.

Если известно расширение, можно выяснить файловое сопоставление и тип файла с помощью команд ASSOC и FTYPE. Чтобы найти сопоставление, введите **cmd /c assoc**, затем расширение файла вместе с точкой. Команда ASSOC выведет тип файла. Если вы введете type **cmd /c ftype *mun*** (где *mun* — вывод команды ASSOC), то увидите команду, соответствующую типу файла. Например, чтобы увидеть команду, соответствующую расширению исполняемого файла .exe, сначала введите **cmd /c assoc .exe,** затем **cmd /c ftype exefile**.

Вы получите следующую информацию о команде, соответствующей типу exefile:

```
exefile="%1" %*
```

Таким образом, когда вы запускаете .exe-файл, Windows рассматривает первое значение в строке как команду, а остальные значения — как передаваемые ей параметры.

Совет Сопоставления и типы файлов хранятся в реес тре Windows и задаются командами A SSOC и FTYPE. Чтобы создать сопос тавление файла, введите **cmd /c assoc**, затем расширение и сопос тавляемый ему тип файла, например **cmd /c assoc .pl=perlfile**. Чтобы определить команду для типа файла, введите **cmd /c ftype** и укажите тип файла, соответс твующую ему к оманду и передаваемые параметры, например **cmd /c ftype perlfile=C:\Perl\Bin\Perl.exe «%1» %***.

Расширения PowerShell

Доступно несколько способов расширения функциональности PowerShell. Обычно для этого устанавливают оснастки (snap-ins) PowerShell, которые добавляют к рабочей среде компоненты-поставщики данных (providers). Данные, доступные через поставщики, выглядят и просматриваются как содержимое обычного жесткого диска. В PowerShell 2.0 добавлена поддержка модульных расширений, которые перед использованием необходимо импортировать.

Работа с расширениями PowerShell

Для работы с оснастками, поставщиками и дисками в PowerShell используются следующие командлеты:

- **Add-PSSnapin** — подключает одну или несколько зарегистрированных оснасток к текущему сеансу. После подключения оснастки поддерживаемые ей командлеты и поставщики становятся доступными для использования в текущем сеансе.

  ```
  Add-PSSnapin [-PassThru] [-Name] Строки
  ```

- **Export-Console** — экспортирует имена оснасток PowerShell, подключенных к текущему сеансу, в файл консоли PowerShell (.psc1). Этот файл можно будет использовать для подключения тех же оснасток к другим сеансов, вызвая PowerShell.exe с параметром с -PSConsoleFile.

  ```
  Export-Console [-NoClobber] [-Force] [[-Path] Строка]
  ```

- **Get-Module** — получает сведения о модулях расширений. Первая команд из приведенного ниже примера получает сведения об импортированных модулях, доступных в текущем сеансе. Вторая команда получает информацию обо всех доступных модулях.

  ```
  Get-Module [[-Name] Строки] [-All]
  Get-Module [-ListAvailable [-Name] Строки] -Recurse
  ```

- **Get-PSProvider** — получает информацию о всех или о заданных поставщиках, установленных на компьютере и доступных в текущем сеансе.

  ```
  Get-PSProvider [[-PSProvider] Строки]
  ```

- **Get-PSSnapin** — получает объекты, представляющие оснастки, подключенные к текущему сеансу или зарегистрированные в системе. Оснастки выводятся в порядке их обнаружения. Для регистрации оснасток используется утилита InstallUtil из Microsoft .NET Framework 2.0.

  ```
  Get-PSSnapin [-Registered] [[-Name] Строки]
  ```

- **Import-Module** — импортирует модули в текущий сеанс. Командлеты и функции, поддерживаемые подключенным модулем, становятся доступными в текущем сеансе.

  ```
  Import-Module [-Name] Строки [AddtlParams]
  Import-Module [-Assembly] Сборки [AddtlParams]
  Import-Module [-ModuleInfo] GUID_модулей [AddtlParams]

  AddtlParams=
  [-Prefix String] [-Function Строки] [-Cmdlet Строки] [-Variable Строки]
  [-Alias Строки] [-Force] [-PassThru] [-AsCustomObject] [-Version Номер_
  версия] [-ArgumentList Объекты]
  ```

- **New-Module** — создает модули из заданных блоков сценария, функций и командлетов. Для этого также служат командлеты New-ModuleManifest и Test-ModuleManifest.

```
New-Module [-ScriptBlock] Блок_сценария [-Function Строки] [-Cmdlet
Строки] [-ReturnResult] [-AsCustomObject] [-ArgumentList Объекты]

New-Module [-Name] Строки [-ScriptBlock] Блоки_сценария [-Function
Строки] [-Cmdlet Строки] [-ReturnResult] [-AsCustomObject]
[-ArgumentList Объекты]
```

- **Remove-Module** — удаляет модуль, подключенный к текущему сеансу.

```
Remove-Module [-Name] Строки [-Force]
Remove-Module [-ModuleInfo] GUID_модулей [-Force]
```

- **Remove-PSSnapin** — отключает оснастку PowerShell от текущего сеанса. Оснастки, установленные вместе с PowerShell, отключить нельзя.

```
Remove-PSSnapin [-PassThru] [-Name] Строки
```

Работа с оснастками

Оснастки PowerShell являются программами, использующими .NET и скомпилированными в DLL-файлы. В оснастках могут содержаться поставщики, командлеты и функции. Поставщики PowerShell — это .NET-программы, обеспечивающие доступ к специализированным хранилищам данных из командной строки. Перед использованием поставщика необходимо установить соответствующую оснастку и подключить ее к сеансу PowerShell.

Вместе с PowerShell устанавливается набор основных оснасток, который можно расширять, устанавливая оснастки, содержащие дополнительные поставщики и командлеты. Например, серверные приложения, такие как Microsoft Exchange Server 2007 SP1 или SQL Server 2008, содержат расширения среды PowerShell. Так, Exchange Server 2007 SP1 и выше поддерживает Exchange Management Shell. Эта оболочка — просто консоль PowerShell, настроенная для управления Exchange Server и снабженная соответствующими оснастками. SQL Server 2008 и выше поддерживает оболочку SQL Server PowerShell, которая также является консолью PowerShell с оснастками и параметрами, необходимыми для работы с SQL Server.

Сразу после подключения оснастки поддерживаемые ею поставщики и командлеты становятся доступными для использования в текущем сеансе. Чтобы оснастка была доступна не только в текущем, но и в других сеансах, ее нужно подключать через профиль. Кроме того, с помощью командлета Export-Console можно сохранить подключенные оснастки в файл консоли. Если открыть новый сеанс PowerShell с эти файлом консоли, сохраненные в нем оснастки будут доступны в этом сеансе.

Чтобы сохранить список оснасток, подключенных к сеансу, в файл консоли (.psc1-файл), используйте командлет Export-Console. Например, чтобы сохранить список оснасток в файле MyConsole.psc1, расположенном в текущем каталоге, введите следующую команду:

```
export-console MyConsole
```

Следующая команда запускает PowerShell с файлом консоли MyConsole.psc1:

```
powershell.exe -psconsolefile MyConsole.psc1
```

Чтобы получить список доступных оснасток, введите **get-pssnapin**. Найти оснастки, соответствующие поставщикам PowerShell, позволит следующая команда:

```
get-psprovider | format-list name, pssnapin
```

а вывести список командлетов, поддерживаемых заданной оснасткой, — команда следующего вида:

```
get-command -module Имя_оснастки
```

где *Имя_оснастки* — имя оснастки. Встроенные оснастки PowerShell перечислены в табл. 3-2.

Табл. 3-2. Встроенные оснастки PowerShell

Имя	Что содержит
Microsoft.PowerShell.Core	Поставщики и командлеты для работы с ключевыми функциями PowerShell, включая поставщики FileSystem, Registry, Alias, Environment, Function, Variable и базовые командлеты, такие как Get-Help, Get-Command и Get-History
Microsoft.PowerShell.Diagnostics	Командлеты для чтения журналов событий и конфигурации Windows, например Get-WinEvent
Microsoft.PowerShell.Host	Командлеты, которые использует хост PowerShell, например Start-Transcript и Stop-Transcript
Microsoft.PowerShell.Management	Командлеты для управления компонентами Windows, такие как Get-Service и Get-ChildItem
Microsoft.PowerShell.Security	Командлеты для управления защитой PowerShell, такие как Get-Acl, Get-AuthenticodeSignature и ConvertTo-SecureString
Microsoft.PowerShell.Utility	Командлеты для управления объектами и данными, такие как Get-Member, Write-Host и Format-List
Microsoft.PowerShell.WSMan.Management	Командлеты для управления операциями WSMan, такие как Get-WSManInstance и Set-WSManInstance

Встроенные оснастки регистрируются в операционной системе и добавляются к сеансу по умолчанию при каждом запуске PowerShell. Оснастки, создан-

ные самостоятельно или полученные из других источников, необходимо явно зарегистрировать и подключить к сеансу консоли. Чтобы узнать, какие оснастки (за исключением встроенных) зарегистрированы в системе, либо проверить, зарегистрирована ли некоторая оснастка, введите следующую команду:

```
get-pssnapin -registered
```

Для подключения зарегистрированных оснасток к текущему сеансу используется командлет Add-PSSnapin. Например, для подключения оснастки SQL Server введите команду **add-pssnapin sqlserver**. После подключения оснастки к сеансу становятся доступными поддерживаемые ей поставщики и командлеты. Чтобы нужные расширения загружались во всех сеансах консоли, добавьте к профилю соответствующие вызовы Add-PSSnapin.

Для отключения оснасток от текущего сеанса PowerShell служит командлет Remove-PSSnapin. Например, для отключения оснастки SQL Server от текущего сеанса введите **remove-pssnapin sqlserver**. После отключения оснастка не выгружается, но содержащиеся в ней поставщики и командлеты становятся недоступными.

При администрировании и написании сценариев часто требуется проверить доступность некоторой оснастки перед использованием содержащихся в ней функций и командлетов. Рассмотрим следующий пример:

```
if (get-pssnapin -name ADRMS.PS.Admin -erroraction silentlycontinue)
 {

  Здесь должен быть код, исполняемый, если нужная оснастка доступна…

} else {

  …а здесь – код, исполняемый при отсутствии нужной оснастки.

}
```

Если оснастка ADRMS.PS.Admin доступна, то выражение в скобках дает True и выполняется соответствующий блок сценария. В противном случае выражение в скобках дает False и выполняется код из блока Else. Заметьте также, что параметру -ErrorAction присвоено значение SilentlyContinue для подавления вывода сообщения об ошибке при отсутствии нужной консоли.

Совет Аналогичный прием применяется с поставщиками и модулями.

Работа с поставщиками

Работа с данными, доступными через компоненты-поставщики, мало чем отличается от работы с содержимым жестких дисков: поставщики поддерживают просмотр, поиск и манипулирование данными. Чтобы получить список доступных поставщиков, введите **Get-PSProvider**. Встроенные поставщики перечислены в табл. 3-3. Обратите внимание, что с каждым поставщиком связано имя диска.

Табл. 3-3. Встроенные поставщики данных PowerShell

Поставщик	К чему открывает доступ	Имя диска
Alias	Псевдонимы PowerShell	{Alias}
Certificate	Сертификаты стандарта X509 для цифровых подписей	{Cert}
Environment	Переменные окружения Windows	{Env}
FileSystem	Диски, файлы и каталого файловой системы	{C, D, E, ...}
Function	Функции PowerShell	{Function}
Registry	Системный реестр Windows	{HKLM, HKCU}
Variable	Переменные PowerShell	{Variable}
WSMan	WS-Management	{WSMan}

PowerShell поддерживает специальные командлеты, предназначенные для управления хранилищами данными (табл. 3-4). Эти командлеты одинаково работают с любыми доступными поставщиками.

Табл. 3-4. Командлеты для работы с хранилищами данных

Командлет	Описание
Get-PSDrive	Получает все или заданные диски, подключенные к текущей консоли, включая логические диски компьютера, сетевые диски и «диски» поставщиков PowerShell. Get-PSDrive не видит диски, подключенные после запуска PowerShell, чтобы сделать такие диски доступны-ми, используйте командлет New-PSDrive Get-PSDrive [-PSProvider *Строки*] [-Scope *Строка*] [[-LiteralName] | [-Name]] *Строки*
New-PSDrive	Создает диск PowerShell и сопоставляет его каталогу хранилища данных, которым может быть локальная или сетевая папка либо раздел реестра. Этот диск доступен только в текущем сеансе New-PSDrive [-Credential *Удостоверения*] [-Description *Строка*] [-Scope *Строка*] [-Name] *Строка* [-PSProvider] *Строка* [-Root] *Строка*
Remove-PSDrive	Отключает диски (кроме дисков Windows и сетевых дисков, подключенных другими методами) от текущего сеанса консоли Remove-PSDrive [-Force] [-PSProvider *Строки*] [-Scope *Строка*] [[-LiteralName] | [-Name]] *Строки*
Get-ChildItem	Получает элементы, включая дочерние, для заданных каталогов Get-ChildItem [[-Path] *Строки*] [[-Filter] *Строки*] [AddtlParams] Get-ChildItem [[-Filter] *Строка*] [-LiteralPath] *Строки* [AddtlParams] AddtlParams=[-Exclude *Strings*] [-Force] [-Include *Strings*] [-Name] [-Recurse]

(*см. след. стр.*)

Табл. 3-4. Командлеты для работы с хранилищами данных

Командлет	Описание
Get-Item	Получает элемент заданного каталога Get-Item [[-LiteralPath] | [-Path]] *Строки* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Exclude *Строки*] [-Filter *Строка*] [-Force] [-Include *Строки*]
New-Item	Создает новый элемент New-Item [-Credential *Удостоверения*] [-Force] [-ItemType *Строка*] [-Path *Строки*] [-Value *Объект*] -Name *Строка*
Set-Item	Присваивает элементу заданное значение Set-Item [-Value] *Объект*] [[-LiteralPath] | [-Path] *Строки* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Exclude *Строки*] [-Filter *Строка*] [-Force] [-Include *Строки*] [-PassThru]
Remove-Item	Удаляет заданный элемент Remove-Item [[-LiteralPath] | [-Path]] *Строки* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Exclude *Строки*] [-Filter *Строки*] [-Force] [-Include *Строки*] [-Recurse]
Move-Item	Перемещает элемент из одного каталога в другой Move-Item [[-Destination] *Строка*] [[-LiteralPath] | [-Path]] *Строки*[AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Exclude *Строки*] [-Filter *Строка*] [-Force] [-Include *Строки*] [-PassThru]
Rename-Item	Переименовывает элемент в пространстве имен поставщика данных PowerShell Rename-Item [-Credential *Удостоверения*] [-Force] [-PassThru] [-Path] *Строка* [-NewName] *Строка*
Copy-Item	Копирует элемент из одного каталога в другой Copy-Item [[-Destination] *Строка*] [[-LiteralPath] | [-Path]] *Строки* [AddtlParams] AddtlParams= [-Container] [-Credential *Удостоверения*] [-Exclude *Строки*] [-Filter *Строка*] [-Force] [-Include *Строки*] [-PassThru] [-Recurse]

Табл. 3-4. Командлеты для работы с хранилищами данных

Командлет	Описание
Clear-Item	Удаляет содержимое элемента, сохраняя сам элемент Clear-Item [[-LiteralPath] | [-Path]] *Строки* [AddtlParams] AddtlParams= [-Credential *Удостоверение*] [-Exclude *Строки*] [-Filter *Строка*] [-Force] [-Include *Строки*]
Invoke-Item	Выполняет над элементом операцию по умолчанию Invoke-Item [[-LiteralPath] | [-Path]] *Строки* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Exclude *Строки*] [-Filter *Строка*] [-Include *Строки*]
Clear-Item-Property	Deletes the value of a property but does not delete the property. Clear-ItemProperty [[-LiteralPath] | [-Path]] *Strings* [-Name] *String* [AddtlParams] AddtlParams= [-Credential *Credential*] [-Exclude *Strings*] [-Filter *String*] [-Force] [-Include *Strings*] [-PassThru]
Copy-Item-Property	Копирует свойства вместе с их значениями Copy-ItemProperty [[-LiteralPath] | [-Path]] *Строки* [-Destination] *Строка* [-Name] *Строка* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Exclude *Строки*] [-Filter *Строка*] [-Force] [-Include *Строки*] [-PassThru]
Get-Item-Property	Получает свойства заданного элемента Get-ItemProperty [-Name] *Строка* [[-LiteralPath] | [-Path]] *Строки* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Exclude *Строки*] [-Filter *Строки*] [-Include *Строки*]
Move-Item-Property	Перемещает свойства Move-ItemProperty [[-LiteralPath] | [-Path]] *Строки* [-Destination] *Строки* [-Name] *Строка* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Exclude *Строки*] [-Filter *Строка*] [-Force] [-Include *Строки*] [-PassThru]

(*см. след. стр.*)

Табл. 3-4. Командлеты для работы с хранилищами данных

Командлет	Описание
New-Item-Property	Создает свойство элемента и устанавливает его значение New-ItemProperty [-PropertyType *Строка*] [-Value *Объект*] [[-LiteralPath] | [-Path]] *Строки* [-Name] *Строка* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Exclude *Строки*] [-Filter *Строка*] [-Force] [-Include *Строки*]
Remove-ItemProperty	Удаляет заданное свойство и его значение Remove-ItemProperty [[-LiteralPath] | [-Path]] *Строки* [-Name] *Строка* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Exclude *Строки*] [-Filter *Строка*] [-Force] [-Include *Строки*]
Rename-ItemProperty	Удаляет заданное свойство элемента Rename-ItemProperty [[-LiteralPath] | [-Path]] *Строки* [-Name] *Строка* [-NewName] *Строка* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Exclude *Строки*] [-Filter *Строка*] [-Force] [-Include *Строки*] [-PassThru]
Set-ItemProperty	Создает свойство или, если оно есть, изменяет его значение Set-ItemProperty -InputObject *Объект* [-LiteralPath] *Строки* [AddtlParams] Set-ItemProperty [-Name] *Строка* [-Value] *Объект* [AddtlParams] Set-ItemProperty [-Path] *Строки* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Exclude *Строки*] [-Filter *Строка*] [-Force] [-Include *Строки*] [-PassThru]

Наряду с поддержкой встроенных командлетов поставщики могут:

- предоставлять собственные командлеты, разработанные для управления их данными;
- добавлять к встроенным командлетам т. н. динамические параметры, доступные только при использовании командлетов с данными соответствующего поставщика.

По умолчанию Get-PSProvider выводит список имен дисков, сопоставленных поставщикам. Получить дополнительную информацию о таком «диске» позволяет командлет Get-PSDrive. Например, поставщик Registry открывает доступ к корневому разделу реестра HKEY_LOCAL_MACHINE через диск HKLM. Чтобы вывести все свойства это диска, введите следующую команду:

```
get-psdrive hklm | format-list *
```

С данными поставщика работают совершенно так же, как с содержимым обычного жесткого диска. Для просмотра содержимого «диска» поставщика служат командлеты Get-Item и Get-ChildItem, после которых необходимо ввести имя диска и двоеточие (:). Например, для просмотра диска Function нужно ввести:

```
get-childitem function:
```

Разрешается просматривать данные любых дисков и управлять им, находясь в каталоге другого диска. Для этого необходимо включить в команду имя диска и полный путь к нужному каталогу. Например, для просмотра раздела реестра HKLM\Software из другого диска введите

```
get-childitem hklm:\software
```

Для перехода на другой диск служит командлет Set-Location (не забывайте ставить двоеточие после имени диска). Например, чтобы перейти в корневой каталог диска Function drive, введите **set-location function:**, а для просмотра содержимого диска Function — **get-childitem**.

Содержимое диска поставщика просматривают так же, как содержимое обычного жесткого диска. Для ссылки на вложенные элементы используется обратный слеш (\):

```
Set-location drive:\location\child-location\...
```

Например, для перехода в раздел реестра HKLM\Software с помощью Set-Location введите следующую команду:

```
set-location hklm:\software
```

Также можно использовать относительные ссылки на каталоги. Точка (.) представляет текущий каталог. Например, если вы находитесь в каталоге C:\Windows\System32 и хотите вывести список находящихся в нем файлов и папок, введите следующую команду:

```
get-childitem .\
```

Поставщики могут добавлять к стандартным командлетам динамические параметры, доступные только при использовании командлета с соответствующим поставщиком. Например, поставщик Certificate добавляет к командлетам Get-Item и Get-ChildItem параметр -CodeSigningCert. Этот параметр можно использовать только при вызове Get-Item или Get-ChildItem для диска Cert.

Хотя удалить поставщик из текущего сеанса нельзя, можно отключить представляющую его оснастку PowerShell от текущего сеанса с помощью командлета Remove-PSSnapin. Он не удаляет поставщики, а отключает оснастки со всем содержимым, включая поставщики и командлеты, после чего они становятся недоступными в текущем сеансе.

Другой способ отключения функций, добавленных путем подключения оснасток, — отключение диска от текущего сеанса с помощью командлета Remove-PSDrive. При отключении диска его содержимое сохраняется, но диск становится недоступным в текущем сеансе.

Часто требуется проверить, доступен ли определенный поставщик PowerShell или диск (PSDrive) перед использованием его возможностей. Проще всего сделать это в сценарии следующим образом:

```
iIf (get-psprovider -psprovider wsman -erroraction silentlycontinue)
{

  Здесь должен быть код, который исполняется, если поставщик доступен…

} else {

  …а здесь – код, исполняемый при отсутствии поставщика.

}
```

Если в этом примере поставщик WSMan доступен, выражение в скобках дает True и соответствующий блок сценария исполняется. В противном случае выражение в скобках дает False и выполняется код в блоке Else. Заметьте, что параметру -ErrorAction присвоено значение SilentlyContinue, чтобы подавить вывод сообщений об ошибках при отсутствии нужного поставщика.

Работа с дисками поставщиков данных

При работе с поставщиками разрешается манипулировать данными и каталогами их дисков. Для этих целей используются командлеты, перечисленные в табл. 3-5.

Табл. 3-5. Командлеты для работы с поставщиками данных

Командлет	Описание
Add-Content	Добавляет содержимое к заданному элементу, например текст к текстовому файлу Add-Content [[-LiteralPath] | [-Path]] *Строки* [-Value] *Объекты* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Encoding {<Unknown> | String | <Unicode> | <Byte> | <BigEndianUnicode> | <UTF8> | <UTF7> | <Ascii>}] [-Exclude *Строки*] [-Filter String] [-Force] [-Include *Строки*] [-PassThru]

Табл. 3-5. Командлеты для работы с поставщиками данных

Командлет	Описание
Clear-Content	Удаляет содержимое элемента, например текст из текстового файла, но сохраняет сам элемент Clear-Content [[-LiteralPath] | [-Path]] *Строки* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Exclude Strings] [-Filter *Строка*] [-Force] [-Include *Строки*]
Get-Content	Получает содержимое элемента, расположенного в заданном каталоге Get-Content [[-LiteralPath] | [-Path]] *Строки* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Delimiter *Строка*] [-Encoding {<Unknown> | *Строка*| <Unicode> | <Byte> | <BigEndianUnicode> | <UTF8> | <UTF7> | <Ascii>}] [-Exclude *Строки*] [-Filter *Строка*] [-Force] [-Include *Строки*] [-ReadCount <Int64>] [-TotalCount <Int64>] [-Wait]
Set-Content	Добавляет содержимое в элемент, заменяя его текущее содержимое Set-Content [[-LiteralPath] | [-Path]] *Строки* [-Value] *Объекты* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Encoding {<Unknown> | String | <Unicode> | <Byte> | <BigEndianUnicode> | <UTF8> | <UTF7> | <Ascii>}] [-Exclude *Строки*] [-Filter *Строки*] [-Force] [-Include *Строки*] [-PassThru]
Get-Location	Получает информацию о текущем рабочем каталоге Get-Location [-PSDrive *Строки*] [-PSProvider *Строки*] Get-Location [-Stack] [-StackName *Строки*]
Set-Location	Задает текущий рабочий каталог Set-Location [-PassThru] [[-Path] *Строки*] Set-Location [-PassThru] [-StackName *Строки*]
Push-Location	Помещает текущий каталог в начало списка (стека) каталогов Push-Location [-PassThru] [-StackName *Строка*] [[-LiteralPath] | [-Path] *Строка*]

(*см. след. стр.*)

Табл. 3-5. Командлеты для работы с поставщиками данных

Командлет	Описание
Pop-Location	Переходит в каталог, добавленный в стек каталогов последним Pop-Location [-PassThru] [-StackName *Строка*]
Join-Path	Объединяет пути к родительскому и вложенному каталогу, используя предоставленный поставщиком разделитель Join-Path [-Credential *Удостоверения*] [-Resolve] [-Path] *Строки* [-ChildPath] *Строка*
Convert-Path	Преобразует путь PowerShell в путь к каталогу поставщика данных Convert-Path [[-LiteralPath] | [-Path]] *Строки*
Split-Path	Возвращает заданную часть пути Split-Path [-LiteralPath *Строки*] [-Path] *Строки* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-IsAbsolute | -Leaf | -Parent | -NoQualifier | -Qualifier] [-Resolve]
Test-Path	Проверяет существование всех элементов пути Test-Path [[-LiteralPath] | [-Path]] *Строки* [AddtlParams] AddtlParams= [-Credential *Удостоверения*] [-Exclude *Строки*] [-Filter *Строка*] [-Include *Строки*] [-IsValid] [-PathType {<Any> | <Container> | <Leaf>}]
Resolve-Path	Заменяет подстановочные знаки в пути реальными значениями и выводит результат Resolve-Path [-Credential *Удостоверения*] [[-LiteralPath] | [-Path] *Строки*]

Доступное в данный момент хранилище данных определяется выбором поставщика. Хранилищем данных по умолчанию является файловая система. Путь по умолчанию — это (в большинстве случаев) путь к каталогу профиля текущего пользователя.

Текущий рабочий каталог — это каталог, который PowerShell использует в отсутствие явно указанного пути к элементу или каталогу, обрабатываемому командой. Как правило, это каталог жесткого диска, доступный через поставщик FileSystem. В отсутствие явно заданного пути все команды обрабатываются в рабочем каталоге.

PowerShell отслеживает текущий рабочий каталог для каждого диска, даже если диск в данный момент не используется. Это позволяет обращаться из те-

кущего рабочего каталога к элементам, расположенным на других дисках, указывая только имя диска. Например, если вы находитесь в каталоге C:\Scripts\PowerShell, то следующая команда позволит перейти на диск HKLM:

```
Set-Location HKLM:
```

Теперь вы находитесь на диске HKLM, но по-прежнему можете обращаться к элементами в каталоге C:\Scripts\PowerShell, указывая только имя диска — C:, например:

```
Get-ChildItem C:
```

PowerShell запоминает рабочий каталог на диске C (C:\Scripts\PowerShell), поэтому по умолчанию используются элементы из этого каталога. Таким образом, следующая команда даст те же результаты, что и предыдущая:

```
Get-ChildItem C:\Scripts\PowerShell
```

Определить текущий рабочий каталог можно при помощи командлета Get-Location, а изменить его — с помощью Set-Location. Например, следующая команда позволяет перейти в каталог Scripts на диске C:

```
Set-Location c:\scripts
```

Даже сменив рабочий каталог, можно обращаться из текущего рабочего каталога к элементам, расположенным на других дисках, указывая только имя диска, например, так:

```
Get-ChildItem HKLM:\software
```

Эта команда получает список элементов раздела Software из куста HKEY Local Machine системного реестра.

Для представления родительского и текущего рабочего каталога используются специальные символы. Текущий рабочий каталог представляет точка, а родительский каталог — две точки. Например, следующая команда обрабатывает каталог PowerShell в текущем рабочем каталоге:

```
Get-ChildItem .\PowerShell
```

Если текущий рабочий каталог — C:\Scripts, эта команда вернет список элементов в каталоге C:\Scripts\PowerShell. Если же указать вместо одной две точки, будет обработан родительский каталог рабочего каталога, как показано в следующем примере:

```
Get-ChildItem ..\Data
```

В этом случае PowerShell интерпретирует две точки как корневой каталог диска C, поэтому команда возвращает список элементов из каталогов C:\Data.

Путь, в начале которого стоит слеш, ведет от корневого каталога текущего диска. Например, если C:\Scripts\PowerShell — текущий рабочий каталог,

то корневой каталог — C:\, и следующая команда вернет список элементов в каталоге C:\Data:

```
Get-ChildItem \Data
```

Если в начале пути не стоит слеш, имя диска или точка, предполагается, что заданный контейнер находится ткущем рабочем каталоге. Например, если C:\Scripts — текущий рабочий каталог, то следующая команда вернет список элементов в каталоге C:\Scripts\PowerShell:

```
Get-ChildItem PowerShell
```

Если в этой команде указан файл, а не каталог, PowerShell вернет сведения об этом файле, если же он не будет найден в текущем рабочем каталоге, PowerShell вернет ошибку.

Работа с модулями

Модули Windows PowerShell — это автономные, пригодные для многократного использования единицы исполнения, которые могут содержать:

- сценарии (в виде .PSM1-файлов);
- .NET-сборки, скомпилированные в .DLL-библиотеки (объявленные в .PSD1-файлах);
- оснастки PowerShell (в виде .DLL-файлов);
- пользовательские представления и типы данных, объявленные в .PS1-XML-файлах.

Большинство модулей содержит оснастки, .NET-сборки, пользовательские представления и типы данных. В .PSD1-файлах с определением сборок, содержащихся в модуле, содержится массив ассоциативных элементов (см. листинг ниже и табл. 3-6).

```
@{
GUID=»{8FA5064B-8479-4c5c-86EA-0D311FE48875}»
Author=»Microsoft Corporation»
CompanyName=»Microsoft Corporation»
Copyright=»© Microsoft Corporation. All rights reserved.»
ModuleVersion=»1.0.0.0»
Description=»Powershell File Transfer Module»
PowerShellVersion=»2.0»
CLRVersion=»2.0»
NestedModules=»Microsoft.BackgroundIntelligentTransfer.Management»
FormatsToProcess=»FileTransfer.Format.ps1xml»
RequiredAssemblies=Join-Path $psScriptRoot «Microsoft.
BackgroundIntelligentTransfer.Management.Interop.dll»
}
```

Табл. 3-6. Общие свойства модулей

Свойство	Описание
Author, CompanyName, Copyright	Содержит сведения об авторе модуля и его правах
CLRVersion	Версия общеязыковой исполняющей среды (CLR) .NET Framework, требуемая модулем
Description	Понятное имя модуля
FormatsToProcess	Список FORMAT.PS1XML-файлов, загружаемых модулем с целью создания пользовательских представлений для поддерживаемых им командлетов
GUID	Глобально уникальный идентификатор (GUID) модуля
ModuleVersion	Номера версии и ревизии модуля
NestedModules	список оснасток и .NET-сборок, загружаемых модулем
PowerShellVersion	Минимальная версия PowerShell, требуемая модулем
RequiredAssemblies	Список .NET-сборок, необходимых для работы модуля
TypesToProcess	Список TYPES.PS1XML-файлов, загружаемых модулем с целью объявления пользовательских типов данных для поддерживаемых им командлетов

Для создания модулей в PowerShell предусмотрен командлет New-Module, но чаще используются командлеты для работы с существующими модулями (Get-Module, Import-Module и Remove-Module). Получить список доступных модулей позволяет команда **get-module -listavailable**, но она выводит слишком много информации. Удобнее искать модули по имени, пути и описанию:

```
get-module -listavailable | format-list name, path, description
```

либо только по имени и описанию:

```
get-module -listavailable | format-table name, description
```

Чтобы проверить, существует ли некоторый модуль, введите

```
get-module -listavailable [-name] Имена_модулей
```

где *Имена_модулей* — список имен модулей, разделенных запятыми. Разрешается указывать только имена модулей, без файловых расширений, а также использовать подстановочные знаки. Помните, что при использовании параметра -ListAvailable параметр -Name интерпретируется в зависимости от его места в списке параметров. Таким образом, можно указывать модули так:

```
get-module -listavailable -name Имена_модулей
```

или так:

```
get-module -listavailable Имена_модулей
```

Вот пример:

```
get-module -listavailable -name networkloadbalancingclusters
```

Набор ключевых модулей PowerShell зависит от версии Windows и набора установленных компонентов (см. табл. 3-7).

Табл. 3-7. Общие модули PowerShell

Имя	Что содержит	Версия ОС
ActiveDirectory	Полный набор команделтов для работы со службами Active Directory Domain Services (AD DS)	Windows Server 2008 Release 2 и выше
ADRMS	Командлеты для обновления, установки и удаления служб Active Directory Rights Management Services (AD RMS): Update-ADRMS, Uninstall-ADRMS и Install-ADRMS	Windows Server 2008 Release 2 и выше
BestPractices	Командлеты для тестирования сценариев, автоматизирующих рекомендованные приемы (best-practices scenarios): Get-BPAModel, Invoke-BPAModel, Get-BPAResult и Set-BPAResult	Windows Server 2008 Release 2 и выше
FailoverClusters	Командлеты для работы с кластерными службами Microsoft Cluster Service	Windows Server 2008 Release 2 и выше
FileTransfer	Командлеты для работы с фоновой интеллектуальной службой передачи (Background Intelligent Transfer Service, BITS): Add-FileTransfer, Clear-FileTransfer, Complete-FileTransfer, Get-FileTransfer, New-FileTransfer, Resume-FileTransfer, Set-FileTransfer и Suspend-FileTransfer	Windows Vista и выше
GroupPolicy	Командлеты для работы с объектами групповой политики (Group Policy objects, GPO).	Windows Server 2008 Release 2 и выше
NetworkLoad-Balancing-Clusters	Командлеты для работы с кластерами NLB (Network Load Balancing)	Windows Server 2008 Release 2 и выше
PSDiagnostics	Функции для трассировки событий: Disable-PSTrace, Disable-PSWSManCombinedTrace, Disable-WSManTrace, Enable-PSTrace, Enable-PSWSManCombinedTrace, Enable-WSManTrace, Get-LogProperties, Set-LogProperties, Start-Trace и Stop-Trace	Windows Vista и выше
RemoteDesktop-Services	Командлеты для работы со службами терминалов (Terminal Services) в режиме Remote Desktop	Windows Server 2008 Release 2 и выше

Табл. 3-7. Общие модули PowerShell

Имя	Что содержит	Версия ОС
ServerManager	Командлеты для просмотра, добавления и удаления Windows-функций: Get-WindowsFeature, Add-WindowsFeature и Remove-WindowsFeature	Windows Server 2008 Release 2 и выше
TroubleshootingPack	Командлеты для получения информации об установленных пакетах troubleshooting pack: Get-TroubleshootingPack и Invoke-TroubleshootingPack	Windows 7 и выше
WebAdministration	Командлеты для работы со службами Internet Information Services (IIS)	Windows Server 2008 Release 2 и выше

Компоненты модулей регистрируются в операционной системе по мере необходимости, но (в большинстве случаев) зарегистрированные модули не импортируются в сеансы PowerShell по умолчанию. Для использования функций, командлетов и других возможностей модуля его сначала требуется импортировать. Модули, которые содержат объявления функций (например, PSDiagnostics) и включают .PSM1-файлы, требуют, чтобы была уставлена политика, разрешающая исполнение подписанных сценариев.

Для импорта зарегистрированных модулей в текущий сеанс служит командлет Import-Module. Например, для импорта модулей WebAdminstration введите **import-module webadministration**. Поставщики, командлеты и другие функции импортированного модуля становятся доступными в текущем сеансе консоли.

Чтобы нужные модули импортировались при каждом запуске консоли, следует добавить необходимые вызовы в соответствующий профиль. Чтобы найти импортированный модуль либо проверить, импортирован ли заданный модуль, введите следующую команду:

```
get-module | format-table name, description
```

Для отключения модулей от текущего сеанса служит командлет Remove-Module. Например, для отключения модуля WebAdministration следует ввести **remove-module webadministration**. При отключении модули не выгружаются, но поддерживаемые ими поставщики, командлеты и функции становятся недоступными.

Часто требуется проверить, импортирован ли некоторый модуль, прежде чем использовать его функции. Если в следующем примере модуль WebAdministration доступен, выражение в скобках дает True и выполняется соответствующий блок кода:

```
if (get-module -name WebAdministration -erroraction silentlycontinue)
{

 Здесь должен быть код, который исполняется, если модуль импортирован,…
```

```
} else {

 ...а здесь – код, исполняемый при отсутствии модуля.

}
```

Как показано выше, запрограммировать альтернативные варианты действий можно при помощи конструкции Else. В этом примере параметру -ErrorAction также назначено значение SilentlyContinue для подавления вывода сообщения об ошибке при отсутствии заданного модуля.

Совет Показанный выше способ проверки наличия модуля удобен, но случай SQL Server заслуживает отдельного рассмотрения. PowerShell-консоль SQL Server загружает расширенную среду, предназначенную для работы с SQL Server. В принципе, можно самостоятельно создать подобную среду, написав для этого длинный сценарий, загружающий все необходимые расширения, но намного удобнее просто запускать готовую PowerShell-консоль.

В общем случае, если существует переменная $sqlpsreg с дочерними элементами, PowerShell-консоль SQL Server запущена. Зная это, можно проверить доступность консоли следующим образом:

```
if (Get-ChildItem $sqlpsreg -ErrorAction «SilentlyContinue»)
{ throw «SQL Server PowerShell is not installed.»
} else {
  $item = Get-ItemProperty $sqlpsreg
  $sqlpsPath = [System.IO.Path]::GetDirectoryName($item.Path)
}
```

Этот сценарий проверяет, существует ли переменная $sqlpsreg с дочерними элементами. Если переменной нет, PowerShell возвращает ошибку, в противном случае — свойства соответствующего объекта, включая свойство Path, которое содержит путь для использования с SQL Server.

Расширения PowerShell для Exchange Server и SQL Server

Доступны расширения PowerShell для Exchange Server 2007 с Service Pack 1 и выше, а также для SQL Server 2008. Расширения для Exchange Server реализованы в виде нестандартной консоли Exchange Management Console, а расширения для SQL Server 2008 — в виде мини-оболочки (minishell) SQL Server PowerShell console.

Использование нестандартных консолей и мини-оболочек представляет собой альтернативные подходы к созданию специализированных сред PowerShell. Нестандартная консоль автоматически загружает все оснастки, составляющие рабочую среду, но не полностью сохраняет ее состояние. Мини-оболочка предоставляет полнофункциональную рабочую среду, которая сохраняет все аспекты состояния, включая оснастки, поставщики и ранее загруженные расширения типов. Однако мини-оболочка — закрытая

среда с собственными параметрами защиты. По этим причинам их невозможно расширять, а управление их защитой осуществляется отдельно от защиты PowerShell.

Совет С помощью PowerShell SDK разработчики смогут создавать мини-оболочки, используя make-shell. В PowerShell 1.0 нестандартные среды можно было создавать только в виде нестандартных консолей и мини-оболочек, во второй версии PowerShell к этим способам добавились модули, поддерживающие добавление функций сценариев и .NET-сборок. Модули более гибкие, их легко подключать к консоли, поэтому Microsoft использовала именно эту технологию для реализации новых расширений PowerShell для Exchange Server и SQL Server.

Загрузив нестандартную консоль для Exchange на сервере с Exchange Server 2007 SP1 и выше, вы получите командлеты для управления Exchange, которые будут доступны как в командной строке, так и в сценариях. Чтобы загрузить консоль для Exchange Server 2007, введите следующую команду:

```
powershell.exe -noexit -psconsolefile «C:\Program Files\Microsoft\Exchange
Server\Bin\exshell.psc1»
```

При работе с запланированными заданиями не забывайте загружать нестандартную консоль для Exchange, как показано в следующем примере:

```
at 05:30 /interactive powershell.exe -noexit -psconsolefile «C:\Program
Files\Microsoft\Exchange Server\Bin\exshell.psc1» -command «Move-Mailbox
-identity <wrstanek@cpandl.com> -TargetDatabase EngMailboxDB»
```

Примечание Обе показанных выше команды состоят из одной строки. Если Exchange Server 2007 установлен в другой каталог, необходимо указать путь к папке bin в каталоге Exchange Server на вашем сервере.

Загрузив мини-оболочку SQL Server PowerShell вручную или с помощью сценария, вы получите командлеты для управления SQL Server, которые будут доступны как в командной строке, так и в сценариях. Исполняемый файл мини-оболочки называется SQLPS.exe, путь к нему настраивается автоматически при установке SQL Server. Чтобы загрузить мини-оболочку через командную строку, введите **sqlps**. Чтобы сделать это в сценарии, необходимо также загрузить оснастки SQL Server, установить ряд глобальных переменных и загрузить объекты управления SQL Server. Вот пример сценария, инициализирующего среду для работы с SQL Server.

```
#
# Подключаем поставщик SQL Server, если он доступен
$ErrorActionPreference = «Stop»
$sqlpsreg=»HKLM:\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.
SqlServer.Management.PowerShell.sqlps»

if (Get-ChildItem $sqlpsreg -ErrorAction «SilentlyContinue»)
{ throw «SQL Server PowerShell Provider is not installed.»
} else {
```

```
  $item = Get-ItemProperty $sqlpsreg
  $sqlpsPath = [System.IO.Path]::GetDirectoryName($item.Path)
}

#
# Устанавлвиаем глобальные переменные
Set-Variable SqlServerMaximumChildItems 0 -scope Global
Set-Variable SqlServerConnectionTimeout 30 -scope Global
Set-Variable SqlServerIncludeSystemObjects $false -scope Global
Set-Variable SqlServerMaximumTabCompletion 1000 -scope Global

#
# Загружаем объекты управления SQL Server
$assemblylist = «Microsoft.SqlServer.Smo»,
«Microsoft.SqlServer.Dmf »,
«Microsoft.SqlServer.SqlWmiManagement »,
«Microsoft.SqlServer.ConnectionInfo »,
«Microsoft.SqlServer.SmoExtended »,
«Microsoft.SqlServer.Management.RegisteredServers »,
«Microsoft.SqlServer.Management.Sdk.Sfc »,
«Microsoft.SqlServer.SqlEnum »,
«Microsoft.SqlServer.RegSvrEnum »,
«Microsoft.SqlServer.WmiEnum »,
«Microsoft.SqlServer.ServiceBrokerEnum »,
«Microsoft.SqlServer.ConnectionInfoExtended »,
«Microsoft.SqlServer.Management.Collector »,
«Microsoft.SqlServer.Management.CollectorEnum»

foreach ($asm in $assemblylist)
{ $asm = [Reflection.Assembly]::LoadWithPartialName($asm) }

#
# Подключаем оснастки SQL Server,загружаем типы и форматы
Push-Location
cd $sqlpsPath
Add-PSSnapin SqlServerCmdletSnapin100
Add-PSSnapin SqlServerProviderSnapin100
Update-TypeData -PrependPath SQLProvider.Types.ps1xml
update-FormatData -prependpath SQLProvider.Format.ps1xml
Pop-Location
```

Перед использованием подобного сценария необходимо проверить в документации номер версии и пакета исправлений SQL Server, чтобы определить набор необходимых компонентов. Примеры сценариев инициализации для вашей комбинации SQL Server и пакета исправлений вы, скорее всего, найдете на сайте технической поддержки Microsoft (support.microsoft.com).

Глава 4

Сеансы, задания и удаленная работа

Windows PowerShell 2.0 поддерживает удаленное исполнение команд, удаленные сеансы и удаленные фоновые задания. Во время удаленной работы вы вводите команды в PowerShell на своем компьютере, но исполняются эти команды на удаленных компьютерах. Для удаленной работы на локальном и удаленном компьютерах должно быть установлено следующее программное обеспечение: PowerShell V2, .NET Framework 2.0 и Windows Remote Management 2.0.

Подготовка к удаленной работе

Поддержка удаленной работы в PowerShell обеспечивается протоколом WS-Management и службой удаленного управления Windows (Windows Remote Management, WinRM) — реализацией этого протокола в Windows. На компьютерах под управлением Windows 7 и выше служба WinRM 2.0 установлена по умолчанию. На компьютеры под управлением прежних версий Windows потребуется установить WinRM 2.0 или другую версию WinRM, поддерживаемую системой. В настоящее время удаленная работа поддерживается только Windows Vista с Service Pack 1 или выше, Windows 7, Windows Server 2008 и Windows Server 2008 Release 2.

Чтобы проверить доступность службы WinRM и настроить PowerShell для удаленной работы, выполните следующие действия:

1. Запустите Windows PowerShell от имени администратора, щелкнув ярлык PowerShell правой кнопкой и выбрав команду **Run As Administrator (Запуск от имени администратора)**.
2. По умолчанию служба WinRM настроена для запуска вручную. Необходимо изменить ее тип запуска на **Automatic (Автоматически)** и запустить службу на каждом из компьютеров, с которым вы хотите работать удаленно. В командной строке PowerShell выполните следующую команду, чтобы проверить, работает ли служба:

```
get-service winrm
```

В столбце Status у работающей службы должно быть значение Running:

```
Status   Name               DisplayName
------   ----               -----------
Running  WinRM              Windows Remote Management
```

3. Чтобы настроить PowerShell для удаленной работы, введите

```
Enable-PSRemoting -force
```

Во многих случаях у вас будет возможность работать с удаленными компьютерами, расположенными в других доменах. Но если удаленный компьютер домен находится в домене, который не является доваренным, проверка ваших удостоверений этим компьютером может окончиться неудачей. Чтобы проверка прошла успешно, необходимо добавить удаленный компьютер в список доверенных узлов локального компьютера в WinRM. Чтобы сделать это, введите

```
winrm s winrm/config/client '@{TrustedHosts=«Удаленный_компьютер»}'
```

где *Удаленный_компьютер* — имя удаленного компьютера, например:

```
winrm s winrm/config/client '@{TrustedHosts=«CorpServer56»}'
```

Для работы с компьютерами, объединенными в рабочую группу или домашнюю группу (homegroup), необходимо использовать HTTPS либо добавить эти компьютеры в список доверенных узлов (с помощью параметра TrustedHosts). Если подключиться к удаленному компьютеру не удается, проверьте, включен ли этот компьютер, и запущена ли на нем служба WinRM, выполнив следующую команду на компьютере, с которого вы пытаетесь подключиться:

```
winrm quickconfig
```

Эта команда анализирует и настраивает конфигурацию службы WinRM. Если служба WinRM настроена правильно, вывод команды должен быть следующим:

```
WinRM already is set up to receive requests on this machine.
WinRM already is set up for remote management on this machine.
```

Если служба WinRM настроена неверно, вывод будет другим (см. ниже) и вам придется утвердительно ответвить на несколько вопросов, после чего служба будет настроена верно.

```
WinRM is not set up to receive requests on this machine.
The following changes must be made:

Set the WinRM service type to delayed auto start.
Start the WinRM service.
Configure LocalAccountTokenFilterPolicy to grant administrative rights
remotely to local users.Make these changes [y/n]? y
```

```
WinRM has been updated to receive requests.
WinRM service type changed successfully.
WinRM service started.
Configured LocalAccountTokenFilterPolicy to grant administrative rights
remotely to local users.
WinRM is not set up to allow remote access to this machine for
management. The following changes must be made:

Create a WinRM listener on HTTP://* to accept WS-Man requests to any IP
on this machine.

Make these changes [y/n]? y

WinRM has been updated for remote management.
Created a WinRM listener on HTTP://* to accept WS-Man requests to any IP
on this machine.
```

Для удаленной работы необходимо запустить PowerShell с администраторскими правами, щелкнув ярлык PowerShell и выбрав команду **Run As Administrator (Запуск от имени администратора)**. Если PowerShell вызывается из другой программы, эта программа должна быть запущена от имени администратора.

Удаленное исполнение команд

PowerShell поддерживает удаленное исполнение командлетов и внешних программ. Например, можно запускать на удаленных компьютерах любые встроенные командлеты и внешние программы, доступные через путь, заданный переменной окружения PATH ($env:path). Но, поскольку PowerShell работает как сетевая или локальная служба, она не позволяет открывать пользовательский интерфейс программ, запущенных на удаленных компьютерах. Запустить таким образом программу с графическим интерфейсом можно, но команда не будет завершена и PowerShell не вернет управление, пока не завершится запущенная программа либо вы не нажмете Ctrl+C.

Введение в удаленное исполнение

Когда вы отдаете команду для исполнения на удаленном компьютере, эта команда передается через сеть на удаленный компьютер, где ее исполняет клиент PowerShell. Результаты исполнения команды возвращаются на локальный компьютер и выводятся в окне работающего на нем сеанса PowerShell. Учтите, что вводимая команда отправляются на удаленный компьютер только после завершения ее ввода, а вывод возвращается по мере его генерации.

При удаленной работе в PowerShell необходимо иметь в виду следующее:

- PowerShell необходимо запускать с администраторскими полномочиями. Для этого нужно щелкнуть правой кнопкой ярлык PowerShell и выбрать

команду **Run As Administrator (Запуск от имени администратора)**. Если PowerShell запускается из другой программы, например из командной строки Windows (cmd.exe), эта программа должна быть запущена с администраторскими полномочиями;

- текущий пользователь должен быть членом группы Administrators (Администраторы) на удаленном компьютере либо ему придется ввести учетные данные администратора. При подключении к удаленному компьютеру PowerShell использует для входа на него учетные данные текущего пользователя, которые передаются по сети в зашифрованном виде;
- при удаленной работе используется несколько экземпляров PowerShell: один на локальном компьютере и как минимум один на удаленном. Обычно на локальном компьютере действуют локальные политики и профили. то означает, что не все командлеты, псевдонимы, функции, переменные и другие элементы локального профиля будут доступны удаленным командам. Чтобы использовать все необходимые локальные элементы на удаленных компьютерах, необходимо скопировать локальные профили на все удаленные компьютеры;
- на удаленном компьютере должны существовать все файлы, каталоги и другие ресурсы, с которыми работают удаленные команды. Кроме того, у вашей учетной записи должны быть разрешения на подключение к удаленному компьютеру, запуск Windows PowerShell и доступ к нужным файлам, папкам и другим ресурсам.

Команды для удаленной работы

Для работы с удаленными компьютерами используются следующие командлеты:

- **Invoke-Command** — запускает команды на локальном или удаленных компьютерах, возвращает весь вывод, включая сообщение об ошибках. Для исполнения одной команды на удаленном компьютере используют параметр -ComputerName. Чтобы выполнить серию команд, использующих одни и те же данные, следует открыть сеанс PowerShell (создать объект PSSession) на удаленном компьютере, а затем вызвать командлет Invoke-Command с параметром -Session, в качестве значения которого указан объект удаленного сеанса PSSession.

```
Invoke-Command [-ArgumentList Аргументы] [-InputObject Объект]
[-ScriptBlock] Блок_сценария

Invoke-Command [[-ComputerName] Компьютеры] [-ApplicationName
String] [-FilePath] String [-Port PortNum] [-UseSSL]
[BasicParams] [SecurityParams]

Invoke-Command [[-Session] Сеансы] [-FilePath] Строка
```

```
[BasicParams]

Invoke-Command [[-ConnectionURI] Список_URI] [-AllowRedirection]
[-FilePath] Строка [BasicParams] [SecurityParams]

Invoke-Command [[-Session] Сеансы] [-ScriptBlock] Блок_сценария
[BasicParams]

Invoke-Command [[-ConnectionURI] Список_URI] [-AllowRedirection]
[-ScriptBlock] Блок_сценария [BasicParams] [SecurityParams]

BasicParams=
[-ArgumentList Аргументы] [-AsJob] [-HideComputerName] [-InputObject
Объект] [-JobName Строки] [-ThrottleLimit Предел]

SecurityParams=
[-Authentication {<Default> | <Basic> | <Negotiate> |
<NegotiateWithImplicitCredential> | <Credssp>}]
[-CertificateThumbprint Строка] [-ConfigurationName Строка]
[-Credential Удостоверения] [-NoCompression]
[-SessionOption Параметр_сеанса]
```

- **New-PSSession** — создает сеанс PowerShell (объект PSSession) на локальном или удаленном компьютере. При создании объекта PSSession PowerShell устанавливает постоянное подключение к удаленному компьютеру, в результате этот объект можно использовать для прямого взаимодействия с удаленным компьютером.

```
New-PSSession [[-Session] Сеансы] [BasicParams]

New-PSSession [[-ComputerName] Компьютеры] [-ApplicationName
Приложение] [-UseSSL] [BasicParams] [SecurityParams]

New-PSSession [-ConnectionURI] Список_URI [-AllowRedirection]
[BasicParams] [SecurityParams]

BasicParams=
[-Noprofile] [-ThrottleLimit Предел] [-TimeOut Число]

SecurityParams=
[-Authentication {<Default> | <Basic> | <Negotiate> |
 <NegotiateWithImplicitCredential> | <Credssp>}]
[-CertificateThumbprint Строка] [-ConfigurationName Строка]
[-Credential Учетные_данные] [-Name Имена] [-NoCompression]
[-Port Номер_порта] [-SessionOption Параметр_сеанса]
```

- **Get-PSSession** — получает объекты PSSession, представляющие сеансы PowerShell, созданные в течение текущего сеанса. При вызове без па-

раметров этот командлет возвращает все объекты PSSession, созданные в текущем сеансе. Параметры командлета Get-PSSession позволяют получить сеансы, подключенные к определенным компьютерам; нужные сеансы задают по имени, идентификатору (ID) или идентификатору экземпляра (instance ID). Компьютеры задают по их NetBIOS-именам, IP-адресам или полным доменным именам. Чтобы указать локальный компьютер, введите его имя, **localhost** или просто точку (**.**). Идентификаторами являются целочисленные уникальные идентификаторы объектов PSSession в текущем сеансе. Объектам PSSession можно назначать понятные имена, используя параметр -Name. При работе с понятными именами разрешается применять подстановочные знаки. Чтобы получить список имен и идентификаторов объектов PSSession вызовите Get-PSSession без параметров. ID экземпляра — это GUID (глобально уникальный идентификатор), уникально идентифицирующий объект PSSession не только в текущем, но и в любых других сеансах PowerShell. ID экземпляра хранится как значение свойства RemoteRunspaceID объекта RemoteRunspaceInfo, представляющего PSSession. Чтобы получить значение InstanceID для объектов PSSession в текущем сеансе, введите **get-pssession | Format-Table Name, ComputerName, RemoteRunspaceId**.

```
Get-PSSession [[-ComputerName] Компьютеры] | [-InstanceID Список_GUID] |
[-Name имена] | [-ID Список_ID]
```

- **Enter-PSSession** — запускает интерактивный сеанс, взаимодействующий с одним удаленным компьютером. Во время такого сеанса можно исполнять команды так, как если бы они вводились в командной строке удаленного компьютера. Разрешается открывать не более одного интерактивного сеанса одновременно. Как правило, удаленный компьютер, на котором следует открыть интерактивный сеанс, задают параметром -ComputerName. Для открытия интерактивного сеанса также можно использовать сеанс, ранее созданный командлетом New-PSSession.

```
Enter-PSSession [[-Session] Сеанс] | [-InstanceID GUID] |
[-Name Имя] | [-ID ID]

Enter-PSSession [-ComputerName] Компьютер [-ApplicationName Строка]
[-UseSSL] [SecurityParams]

Enter-PSSession [[-ConnectionURI] URI][-AllowRedirection]
[SecurityParams]

SecurityParams=
[-Authentication {<Default> | <Basic> | <Negotiate> |
<NegotiateWithImplicitCredential> | <Credssp>}]
[-CertificateThumbprint Строка] [-ConfigurationName Строка]
[-Credential Учетные_данные] [-NoCompression]
```

```
[-SessionOption Параметр_сеанса]
```

- **Exit-PSSession** — завершает интерактивный сеанс и отключается от удаленного компьютера, то же самое можно сделать вводом команды **exit**.

```
Exit-PSSession
```

- **Import-PSSession** — импортирует командлеты, псевдонимы, функции и другие команды из сеанса, открытого на локальном компьютере, в удаленный сеанс. Разрешается импортировать любые команды, которые Get-Command сможет обнаружить в других сеансах. Чтобы импортировать команды, сначала подключитесь к нужному сеансу с помощью New-PSSession, затем вызовите Import-PSSession. По умолчанию Import-PSSession импортирует все команды, за исключением команд, существующих в текущем сеансе. Чтобы «перезаписать» команды, укажите параметр -CommandName. PowerShell добавляет импортированные команды во временный модуль, существующий только в текущем сеансе, и возвращает объект, представляющий этот модуль. Импортированные команды можно использовать, как любые другие команды, но исполняются такие команды в сеансе, из которого они были импортированы. Поскольку исполнение импортированных команд может занимать больше времени, чем локальных команд, поэтому Import-PSSession добавляет к каждой импортированной команде параметр -AsJob. Этот параметр заставляет командную оболочку исполнять команду как фоновое задание PowerShell.

```
Import-PSSession [-CommandType {Alias | Function | Filter | Cmdlet
| ExternalScript | Application | Script | All}]
[-FormatTypeName Типы] [-PSSnapin Оснастки] [[-CommandName]
Команды] [[-ArgumentList] Аргументы] [-Session] Сеанс
```

- **Export-PSSession** — экспортирует командлеты, псевдонимы, функции и другие команды из сеанса, открытого на локальном или удаленном компьютере, в файл модуля сценария PowerShell (.psm1-файл). Чтобы использовать команды, сохраненные в .psm1-файле, сначала загрузите этот модуль в локальный сеанс с помощью командлета Add-Module. Чтобы экспортировать команды из загруженного модуля, сначала подключитесь к нужному сеансу с помощью New-PSSession, затем вызовите Export-PSSession. По умолчанию Export-PSSession экспортирует все команды, за исключением команд, существующих в текущем сеансе. Чтобы «перезаписать» команды, укажите параметры -PSSnapin, -CommandName и -CommandType.

```
Export-PSSession [-CommandType {Alias | Function | Filter | Cmdlet
| ExternalScript | Application | Script | All}]
[-Encoding String] [-Force] [-NoClobber] [-PSSnapin Оснастки]
[[-CommandName] Команды] [[-ArgumentList] Аргументы] [-Session] Сеансы
```

Удаленный вызов команд

Один из способов исполнения команд на удаленных компьютерах — использование командлета Invoke-Command, позволяющего:

- указывать (с помощью параметра -ComputerName) нужные удаленные компьютеры по их DNS- и NetBIOS-именам либо IP-адресам;
- указывать несколько удаленных компьютеров в виде списка имен или IP-адресов, разделенных запятыми;
- задавать блоки сценария, заключая команды в фигурные скобки, а также указывать команды для исполнения с помощью параметра ScriptBlock.

Определив необходимые параметры Get-Process, введите команду следующего вида (как единую строку) — эта команда будет исполнена на удаленных компьютерах:

```
invoke-command -computername Server43, Server27, Server82
-scriptblock {get-process}
```

Совет По умолчанию Invoke-Command использует ваши учетные данные. Параметр -Credential позволяет указать другие учетные данные в формате *Имя_пользователя* или Имя_домена\Имя_пользователя; после чего система предложит ввести пароль.

Совет При подключении к удаленному компьютеру с Windows Vista, Windows Server 2003 и более низких версий этих ОС начальным каталогом по умолчанию будет домашний каталог текущего пользователя, заданный переменной окружения *%HomePath%* ($env:homepath) и PowerShell-переменной $home, а при подключении к компьютеру с Windows XP — домашний каталог пользователя по умолчанию, заданный теми же переменными.

Invoke-Command возвращает объект, содержащий имя компьютера, на котором был сгенерирован вывод, в свойстве PSComputerName. По умолчанию это свойство отображается, скрыть его можно параметром -HideComputerName.

Если свойство PSComputerName скрыто, но требуется узнать имя компьютера, вернувшего вывод, используйте командлет Format-Table, чтобы добавить свойство PSComputerName к выводу следующим образом:

```
$procs = invoke-command -script {get-process |
sort-object -property Name} -computername Server56, Server42, Server27
```

```
&$procs | format-table Name, Handles, WS, CPU, PSComputerName -auto

 Name                    Handles       WS        CPU PSComputerName
----                    -------       --        --- --------------
acrotray                     52  3948544          0 Server56
AlertService                139  7532544            Server56
csrss                       594 20463616            Server56
csrss                       655  5283840            Server56
```

```
CtHelper                    96  6705152   0.078125 Server56
 . . .
acrotray                    43  3948234          0 Server42
AlertService               136  7532244            Server42
csrss                      528 20463755            Server42
csrss                      644  5283567            Server42
CtHelper                    95  6705576   0.067885 Server42

acrotray                    55  3967544          0 Server27
AlertService               141  7566662            Server27
csrss                      590 20434342            Server27
csrss                      654  5242340            Server27
CtHelper                    92  6705231   0.055522 Server27
```

PowerShell поддерживает индивидуальное ограничение числа одновременных подключений для команд. По умолчанию разрешено от 32 или 50 подключений, в зависимости от команды. Задавать пользовательский лимит числа подключений для команд позволяет параметр -ThrottleLimit. Учтите, что этот параметр действует на отдельные команды, а не на сеансы или компьютеры. Для команд, исполняемых в нескольких сеансах одновременно, общее число подключений рассчитывается как сумма подключений, установленных во всех сеансах.

Несмотря на то, что PowerShell способна обслуживать сотни одновременных подключений, реальное их число зависит от производительности компьютера и числа сетевых подключений, которое он способен обрабатывать одновременно. Чтобы сделать удаленную работу безопаснее, можно вызвать Invoke-Command с параметром -UseSSL. При этом вместо HTTP через Port 80 для обмена данными используется протокол HTTPS через порт 443. Как и при локальной работе, можно приостановить или прервать исполнение удаленно вызванных команд нажатием Ctrl+S или Ctrl+C, соответственно.

Совет PowerShell поддерживает удаленную работу даже с компьютеров, не включенных в домен. Для нужд тестирования и разработки можно открывать дополнительные «удаленные» сеансы на том же компьютере. С ними можно использовать те же самые средства PowerShell, что применяют для работы с настоящими удаленными компьютерами.

Для удаленного исполнения команд на компьютерах в составе рабочей группы может потребоваться изменить настройки Windows на этих компьютерах. В Windows XP с Service Pack 2 (SP2) используйте оснастку консоли Local Security Settings (локальные параметры безопасности, файл secpol.msc), чтобы назначить параметру Network Access: Sharing And Security Model For Local Accounts (Сетевой доступ: модель совместного доступа и безопасности для локальных учетных записей) политики Security Settings\Local Policies\Security Options значение Classic. В Windows Vista следует создать в разделе реестра HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System параметр LocalAccountTokenFilterPolicy и присвоить ему значение 1.

В Windows 2003 обычно ничего изменять не требуется, поскольку параметру политики Network Access: Sharing And Security Model For Local Accounts policy по умолчанию присвоено значение Classic. Впрочем, лучше все же проверить, не изменил ли кто это значение.

Работа с удаленными сеансами

Windows PowerShell 2.0 поддерживает как локальные, так и удаленные сеансы. *Сеансом* (session) называют пространство исполнения, которое образует общую рабочую среду для команд. Команды, исполняемые в одном сеансе, могут обмениваться данными. Подробнее о сеансах рассказывается в следующих главах, а в этом разделе мы рассмотрим работу с удаленными сеансами.

Инициирование сеансов

Чтобы открыть локальный или удаленный сеанс с постоянным подключением, вызовите командлет New-PSSession. Если при вызове не указаны удаленные компьютеры (параметром -ComputerName), PowerShell откроет сеанс на локальном компьютере. В дальнейшем при исполнении команд с помощью Invoke-Command потребуется указывать имя сеанса в параметре –Session. Имя сеанса возвращает командлет New-PSSession при создании сеанса, сохранить его можно следующим образом:

```
$s = new-PSSession -computername Server24
```

Здесь *$s* — переменная, в которой хранится имя сеанса. Поскольку в этом примере задан параметр -ComputerName, PowerShell откроет сеанс не на локальном, а на заданном удаленном компьютере. Далее можно использовать командлет Invoke-Command с параметром -Session для исполнения команд в новом сеансе, как показано ниже:

```
invoke-command -session $s -scriptblock {get-process}
```

В этом примере Invoke-Command используется для вызова Get-Process в сеансе $s. Поскольку данный сеанс подключен к удаленному компьютеру, вызванная команда будет исполнена на этом удаленному компьютере.

Открыть несколько сеансов на разных компьютерах так же просто, достаточно указать нужные компьютеры. В некоторых случаях вам также придется ввести учетные данные, что можно сделать с помощью параметра -Credential. Необходимо указать данные учетной записи, обладающей разрешениями на выполнение заданного действия. По умолчанию используется учетная запись текущего пользователя, предоставить альтернативные учетные данные можно двумя способами:

- передать объект Credential с учетными данными. У этого объекта имеются свойства UserName и Password, имя пользователя хранится «открытым текстом», а пароль — в зашифрованном виде; подробнее про объекты Credential — в гл. 9;
- задать учетную запись, обладающую необходимыми разрешениями. Имя пользователя вводится в формате *Имя_домена\Имя_пользователя* либо *Имя_пользователя*. Если задано только имя пользователя, PowerShell попросит ввести пароль. По запросу введите пароль и щелкните **OK**.

Рассмотрим использование параметра -Credential на следующем примере:

```
$t = new-PSSession -computername Server24, Server45, Server36
-Credential Cpandl\WilliamS
```

Эта команда открывает сеанс на компьютерах Server24, Server45 и Server36 от имени заданного пользователя, для которого указано имя и домен. В результате при запуске команд с помощью Invoke-Command в сеансе $t заданные команды будут исполнены на этих Server24, Server45 и Server36 от имени учетной записи Cpandl\WilliamS. Заметьте, что команда открывает единственный сеанс, но на каждом из компьютеров создается собственное пространство исполнения.

Список удаленных компьютеров можно без труда взять и из текстового файла. В этом примере файл servers.txt содержит список компьютеров, разделенный запятыми:

```
$ses = get-content c:\test\servers.txt | new-PSSession
-Credential Cpandl\WilliamS
```

В этой команде содержимое текстового файла передается New-PSSession через конвейер. В результате сеанс $ses открывается на всех компьютерах, перечисленных в файле.

Иногда требуется исполнить приложение или внешнюю утилиту на удаленных компьютерах, что делается следующим образом:

```
$comp = get-content c:\computers.txt
$s = new-pssession -computername $comp
invoke-command -session $s { powercfg.exe -energy }
```

Здесь C:\Computers.txt — путь к файлу со списком подлежащих проверке компьютеров. На каждом из них команда запускает утилиту PowerCfg с параметром –Energy. В результате генерируется файл отчета Energy-Report.html в каталоге по умолчанию учетной записи пользователя, под которой был открыт сеанс. Отчет содержит параметры конфигурации электропитания и список неполадок, препятствующих корректной работе управления электропитанием. Чтобы не собирать отчеты со всех компьютеров, можно задать запись отчета на сетевой диск и добавить имя компьютера в имя отчета:

```
$comp = get-content c:\computers.txt
$s = new-pssession -computername $comp
invoke-command -session $s { powercfg.exe -energy -output
«\\fileserver72\reports\$env:computername.html»}
```

Эта команда записывает отчет в сетевую папку \\fileserver72\reports и добавляет в имя файла отчета значение переменной окружения ComputerName. Заметьте, что при запуске внешних программ необходимо указывать расширение .exe после имени файла программы.

При выполнении группы команд на нескольких удаленных компьютерах иногда требуется, чтобы управление возвращалось только после исполне-

ния всех команд. Чтобы не ожидать завершения команд, вызывайте Invoke-Command с параметром –AsJob, чтобы выполнить нужные действия как фоновые задания:

```
invoke-command -session $s -scriptblock {get-process moddr |
stop-process -force } -AsJob
```

В этом примере Invoke-Command используется для получения и остановки заданного процесса в сеансе $s. Поскольку данная команда исполняется как фоновое задание, командная строка возвращает управление немедленно после ввода команды, не дожидаясь ее завершения на всех компьютерах.

Хотя возможность запуска сеансов на нескольких компьютерах очень удобна, нередко требуется возможность интерактивной работы с отдельным удаленным компьютером. Для этого используется командлет Enter-PSSession, запускающий интерактивный сеанс на удаленном компьютере. Вызывают его вводом в командной строке Powershell команды **Enter-PSSession** Имя_компьютера, где *Имя_компьютера* — имя удаленного компьютера. В результате командная строка изменяется следующим образом, показывая, что вы подключены к удаленному компьютеру:

```
[Server49]: PS C:\Users\wrstanek.cpandl\Documents>
```

Теперь все команды будут исполняться на этом удаленном компьютере, как если бы вы отдавали их, сидя за его клавиатурой. Для защиты обмена данными Enter-PSSession поддерживает параметры -Credential и -UseSSL. Завершить интерактивный сеанс можно с помощью команды Exit-PSSession или **exit**.

Удаленное исполнение и сериализация объектов

При работе с удаленными компьютерами необходимо знать:

- как исполняются команды;
- как происходит сериализация объектов.

При работе с удаленными компьютерами с помощью Invoke-Command и Enter-PSSession при исполнении каждой команды открывается временное подключение, которое используется для ее исполнения, а затем закрывается. Это эффективно при вызове отдельной команды или нескольких несвязанных команд даже на большой группе удаленных компьютеров.

Альтернативный способ — использование командлета New-PSSession для создания сеанса с постоянным подключением. После вызова New-PSSession открывается постоянное подключение, которое используется для исполнения всех команд, вводимых в данном сеансе. Поскольку эти команды работают в одном и том же пространстве исполнения, они могут использовать одни и те же данные, включая значения переменных, определения псевдонимов и функций. Командлет New-PSSession также поддерживает параметр -UseSSL.

Работая в PowerShell локально, вы взаимодействуете с активными объектами (live object) .NET, связанными с реальными программами и компонентами, которые изменяются при вызове методов и изменении свойств активных объектов. Соответственно, изменения свойств программы или компонента отражаются на состоянии представляющих их объектов.

Активные объекты невозможно передавать по сети, поэтому PowerShell сериализует объекты перед тем, как передать их удаленно вызванным командам. Под сериализацией понимают преобразование объекта в серию пригодных для передачи элементов данных на языке Constraint Language in XML (CLiXML). Получив сериализованный файл, PowerShell преобразует (десериализует) его XML-содержимое, получая в результате объект. Десериализованный объект представляет собой точное описание свойств программы или компонента на момент исполнения команды. Однако он теряет прямую связь с исходным компонентом и все методы, поскольку они перестают действовать. Кроме того, сериализованный объект, который возвращает Invoke-Command, получает дополнительные свойства, позволяющие выяснить происхождение команды.

Примечание Командлет Export-Clixml создает сериализованные предс тавления объектов в виде CLiXML-данных и со храняет их в файл. Для импорта файла с CLiXML-данными и воссоздания десериализованных объектов служит командлет Import-CLixml.

Создание удаленных фоновых заданий

Windows PowerShell 2.0 поддерживает локальные и удаленные фоновые задания. Фоновое задание — это команда, которая исполняется асинхронно и не требует взаимодействия с пользователем. При создании фонового задания командная строка возвращает управление немедленно, позволяя продолжить работу сразу после ввода команды, при этом отданная команда исполняется в фоновом режиме.

Работа с фоновыми заданиями

По умолчанию PowerShell запускает фоновые задания на локальном компьютере. Запустить их на удаленном компьютере можно разными способами:

- открыв интерактивный сеанс на удаленном компьютере и запустив задание в этом сеансе. При таком подходе с фоновыми заданиями работают так же, как на локальном компьютере;
- запустив фоновое задание на удаленном компьютере и направив его результаты на локальный компьютер. Этот подход позволяет собирать обрабатывать результаты удаленных фоновых заданий на компьютере администратора;
- запустив фоновое задание на удаленном компьютере и сохранив его результаты на том же компьютере. Этот подход позволяет надежнее защитить результаты удаленных фоновых заданий.

PowerShell поддерживает следующие командлеты для работы с фоновыми заданиями:

- **Get-Job** — получает объекты, представляющие фоновые задания, запущенные в текущем сеансе. При вызове без параметров Get-Job возвращает список всех заданий в текущем сеансе, при этом возвращается объект-задание, не содержащий результатов исполнения заданий. Чтобы получить их, пользуйтесь командлетом Receive-Job. Параметры Get-Job позволяют получать отдельные задания, которые задают по тексту команд, именам, ID или ID экземпляра. При поиске задания по тексту команды можно ввести полный текст команды либо его часть, разрешается использовать подстановочные знаки. При поиске по ID следует ввести целочисленное значение, уникально идентифицирующее задание в текущем сеансе. При поиске по имени следует ввести понятное имя, ранее назначенное заданию. ID экземпляра — это GUID, уникально идентифицирующий задание среди всех заданий, запущенных в PowerShell. Чтобы получить список имен, ID и ID экземпляров заданий, вызовите Get-Jobs без параметров. Параметр -State позволяет получать задания с заданным состоянием. Допустимые значения этого параметра — NotStarted, Running, Completed, Stopped, Failed и Blocked;

```
Get-Job [-Command Commands] | [[-InstanceId] GUIDs] | [[-Name]
Names] | [[-Id] IDs] | [-State JobState]
```

- **Receive-Job** — получает вывод и сообщения об ошибках фоновых заданий PowerShell, запущенных в текущем сеансе. Можно получать вывод всех заданий либо указать нужные задания с помощью имени, ID, ID экземпляра, имя и адрес компьютера, сеанс либо объекта-задания. По умолчанию результаты задания удаляются после их получения, но параметр -Keep позволяет сохранить их для повторного получения. Чтобы удалить результаты задания, следует получить их еще раз, но без параметра -Keep, закрыть сеанс или вызвать командлет Remove-Job, чтобы удалить задание из сеанса.

```
Receive-Job [[-ComputerName Компьютеры] | [-Location Адреса] |
[-Session Сеансы]] [-Job] Задания [BasicParams]

Receive-Job [[-Id] Список_ID] | [[-InstanceId] Список_GUID] | [[-Name] Имена] |
[-State Состояния] [BasicParams]

BasicParams=
[-Error] [-Keep] [-NoRecurse]
```

- **Remove-Job** — удаляет фоновые задания PowerShell, запущенные командлетом Start-Job либо другими командлетами, вызванными с параметром -AsJob. Вызов Remove-Job без параметров не дает эффекта. Можно удалить все задания либо отдельные задания, указанные с помощью текс-

та команды, имени, ID, ID экземпляра, состояния или объекта-задания. Перед удалением активного задания следует остановить его вызовом Stop-Job, в противном случае удаление активного задания оканчивается неудачей. Чтобы удалить активное задание, используйте параметр -Force. Задания, не удаленные с помощью Remove-Job, остаются в глобальном кэше заданий до закрытия сеанса, в котором они были созданы.

```
Remove-Job [-Force] [-Command Команды] | [[-Job] Задания] |
[[-Id] Список_ID] | [[-InstanceId] Список_GUID] | [[-Name] Имена] |
[-State Состояния]
```

- **Start-Job** — запускает фоновое задание на локальном компьютере. Чтобы запустить фоновое задание на удаленном компьютере, используйте параметр –AsJob и командлеты, поддерживающие этот параметр, либо вызывайте Start-Job на удаленном компьютере с помощью Invoke-Command. Фоновое задание возвращает результаты не сразу после запуска, вместо них команда возвращает представляющий фоновое задание объект, который содержит информацию о задании, но не содержит его результатов. Такой подход позволяет продолжить работу, не дожидаясь завершения задания.

```
Start-Job [-FilePath Путь] [AddtlParams]
Start-Job [-ScriptBlock] Блок_сценария [AddtlParams]

AddtlParams=
[-ArgumentList Аргументы] [-Authentication {<Default> | <Basic> |
<Negotiate> | <NegotiateWithImplicitCredential> | <Credssp>}]
[-ConfigurationName Строка] [-Credential Удостоверения]
[-InputObject Объект] [-Name Строка] [-NoCompression]
```

- **Stop-Job** — останавливает активное фоновое задание. Можно остановить все задания либо отдельные задания, указанные с помощью текста команды, имени, ID, ID экземпляра, состояния или объекта-задания. При остановке фонового задания PowerShell завершает все отложенные операции в очереди этого задания и завершает само задание, добавление новых операций в очередь остановленного задания невозможно. Stop-Job не удаляет фоновые задания, для этого необходимо вызвать Remove-Job.

```
Stop-Job [-PassThru] [[-Job] Задания] | [[-Id] Список_ID] |
[[-InstanceId] Список_GUID] | [[-Name] Имена] | [-State Состояния]
```

- **Wait-Job** — ожидает завершения фоновых заданий PowerShell перед выводом командной строки. Можно задать ожидание завершения всех или отдельных заданий. Параметр -Timeout задает предельное время ожидания. После завершения последней команды задания Wait-Job отображает командную строку и возвращает объект задания, который можно передать через конвейер другой команде. Параметр -Any отображает команд-

ную строку по завершении любого задания (по умолчанию Wait-Job ожидает завершения всех заданий).

```
Stop-Job [-Any] [-TimeOut Время_ожидания] [[-Job] Задания] | [[-Id]
Список_ID] |
[[-InstanceId] Список_GUID] | [[-Name] Имена] | [-State Состояния]
```

Некоторые командлеты автоматически запускаются как фоновые задания при добавлении параметра -AsJob. Чтобы получить полный список командлетов, поддерживающих параметр –AsJob, введите **get-help * -parameter AsJob**. Список этих командлетов включает:

- **Invoke-Command** — исполняет команды на локальном и удаленных компьютерах;
- **Invoke-WmiMethod** — вызывает методы Windows Management Instrumentation (WMI);
- **Test-Connection** — отправляет компьютерам ICMP-пакеты с запросом эхо-отклика (аналогично команде *pings*);
- **Restart-Computer** — перезагружает операционную систему локального и удаленных компьютеров;
- **Stop-Computer** — завершает работу локального и удаленных компьютеров.

В простейшем случае с фоновыми заданиями работают так:

1. Вы запускаете фоновое задание с помощью Start-Job или командлета с параметром –AsJob.
2. Задание запускается, но вместо результатов возвращает объект, представляющий фоновое задание.
3. Этот объект можно использовать для получения полезных сведений о задании, но результатов он не содержит; возврат этого объекта просто позволяет продолжить работу, не дожидаясь завершения задания.
4. Для просмотра результатов фонового задания, запущенного в текущем сеансе, вы вызываете Receive-Job. Можно получать вывод всех заданий либо указать нужные задания с помощью имени, ID, ID экземпляра, имя и адрес компьютера, сеанс либо объекта-задания. После получения результаты задания удаляются (если при вызове Receive-Job не указан параметр -Keep).

Запуск заданий в интерактивных сеансах

Процедура запуска фоновых заданий в интерактивных сеансах на локальном и удаленном компьютерах практически не отличается. Командлет Enter-PSSession запускает интерактивный сеанс на удаленном компьютере, при этом параметр -ComputerName задает имя удаленного компьютера, например так:

```
enter-pssession -computername filesvr32
```

Примечание Для завершения интерактивного сеанса служит команда **exit-pssession**.

Для запуска фоновых заданий в локальных и удаленных сеансах используется командлет Start-Job. Его параметр -ScriptBlock позволяет задать блок сценария, а параметр -FilePath — локальный файл со сценарием.

Следующая команда (ее нужно вводить как единую строку) запускает фоновое задание для сбора событий из журналов System, Application и Security. Start-Job возвращает объект, представляющий задание, который сохраняется в переменной $job.

```
$job = start-job -scriptblock {$share = «\\FileServer85\logs»;
$logs = «system»,»application»,»security»;
foreach ($log in $logs) {
$filename = «$env:computername».ToUpper() + «$log» + «log» +
(get-date -format yyyyMMdd) + «.log»;
Get-EventLog $log | set-content $share\$filename; }
}
```

Чтобы разбить командную строку на фрагменты, используйте знак апострофа:

```
$job = start-job -scriptblock {$share = «\\FileServer85\logs»; `
$logs = «system»,»application»,»security»; `
foreach ($log in $logs) { `
$filename = «$env:computername».ToUpper() + «$log» + «log» + `
(get-date -format yyyyMMdd) + «.log»; `
Get-EventLog $log | set-content $share\$filename; } `
}
```

Кроме того, можно сохранить команды сценария в файл на локальном компьютере и загрузить его с помощью параметра –FilePath, как показано ниже.

```
Команда

$job = start-job -filepath c:\scripts\eventlogs.ps1

Файл Eventlogs.ps1

$share = «\\FileServer85\logs»
$logs = «system»,»application»,»security»

foreach ($log in $logs) {
  $filename = «$env:computername».ToUpper() + «$log» + «log» + `
(get-date -format yyyyMMdd) + «.log»
  Get-EventLog $log | set-content $share\$filename
}
```

Файл сценария должен располагаться на локальном компьютере либо в каталоге, доступном с локального компьютера. Если указан параметр FilePath, PowerShell преобразует содержимое заданного файла в блок сценария и запускает его как фоновое задание.

Можно запускать другие команды и задания, не дожидаясь завершения этого задания, только не закрывайте сеанс, пока фоновое задание не завершится, в противном случае его исполнение будет прервано, а результаты — потеряны.

мечание Сохранять объек ты-задания в переменных не обязательно, но этот прием облегчает работу с такими объектами. Если вы запускаете несколько фоновых заданий, непременно сохраняйте их объек ты в разных переменных, например $job1, $job2 и $job3.

Командлет Get-Job позволяет:

- узнать, завершилось ли задание;
- вывести команду, переданную заданию;
- получить информацию, необходимую для работы с заданиями.

Можно получить сведения о заданиях либо указать нужные задания по имени, ID, ID экземпляра, имени компьютера, адресу, сеансу или объекту задания. PowerShell присваивает заданиями последовательные номера (ID) и имена. Первому заданию назначается ID, равный 1, и имя Job1, второму — ID 2 и имя Job2, и т.д. Чтобы назначить заданию собственное имя, используйте при запуске задания параметр -Name. Например, так запускается задание с именем Logs:

```
start-job -filepath c:\scripts\eventlogs.ps1 -name Logs
```

Получить информацию об этом задании можно вызовом Get-Job с параметром –Name:

```
get-job -name Logs
```

```
Id  Name  State   HasMoreData   Location   Command
--  ----  -----   -----------   --------   -------
1   Logs  Failed  False         filesvr32  $share = “\\FileServer...
```

Поскольку это задание закончилось неудачей, вы можете не получить его вывод или сообщения об ошибках, зато доступны более подробные сведения о задании. Их можно получить, отформатировав вывод команды следующим образом:

```
get-job -name logs | format-list
```

```
HasMoreData   : False
StatusMessage :
Location      : filesvr32
Command       : $share = «\\FileServer85\logs»; $logs = «system»,
«application»,»security»; foreach ($log in $logs) { $f
```

```
ilename = «$env:computername».ToUpper() + «$log» + «log» + (get-date
-format yyyyMMdd) + «.log»; Get-Ev
entLog $log | set-content $share\$filename; }
JobStateInfo  : Failed
Finished      : System.Threading.ManualResetEvent
InstanceId    : 679ed475-4edd-4ba5-ae79-e1e9b3aa590e
Id            : 3
Name          : Logs
ChildJobs     : {Job4}
Output        : {}
Error         : {}
Progress      : {}
Verbose       : {}
Debug         : {}
Warning       : {}
```

Если было запущено несколько заданий, введите **get-job**, чтобы проверить состояние всех заданий:

```
get-job
```

```
Id Name   State      HasMoreData  Location   Command
-- ----   -----      -----------  --------   -------
1  Job1   Completed  False        localhost  $share = "\\FileServer...
3  Job3   Running    True         localhost  $logs = "system","appl...
```

После завершения задания можно вызвать Receive-Job, чтобы получить результаты заданий, но учтите, что в случае заданий, которые не выводят данные и сообщения об ошибках на консоль PowerShell, получать будет нечего.

Команда **receive-job** позволяет получить результаты всех заданий; чтобы получить результаты отдельных заданий, укажите их с помощью имени, ID, ID экземпляра, имени компьютера, адреса, сеанса или объекта-задания. Вот пример получения заданий, указанных поименно:

```
receive-job -name Job1, Job3
```

А эта команда получает результаты заданий с заданными ID:

```
receive-job -id 1, 3
```

Результаты заданий удаляются автоматически после получения, чтобы сохранить их для повторного получения, используйте параметр -Keep. Удалить результаты заданий можно тремя способами: получите их снова, не указывая параметр -Keep; закройте сеанс, в котором было запущено задание; вызовите Remove-Job, чтобы удалить задание из сеанса.

Альтернативный способ — запись результатов задания в файл. Вот как можно записать результаты задания в файл C:\logs\mylog.txt:

```
receive-job -name Job1 > c:\logs\mylog.txt
```

При удаленной работе учтите, что эта команда будет исполнена на удаленном компьютере, соответственно, файл будет создан на удаленном компьютере. Если вы записываете в один и тот же файл результаты нескольких заданий, не забывайте использовать оператор, позволяющий дописывать данные к файлу:

```
receive-job -name Job1 >> c:\logs\mylog.txt
receive-job -name Job2 >> c:\logs\mylog.txt
receive-job -name Job3 >> c:\logs\mylog.txt
```

Для просмотра содержимого файла с результатами из текущего сеанса можно использовать следующую команду:

```
get-content c:\logs\mylog.txt
```

Если же сеанс, в котором был записан файл на удаленном компьютере, закрыт, можно воспользоваться для той же цели командлетом Invoke-Command:

```
$ms = new-pssession -computername fileserver84
invoke-command -session $ms -scriptblock {get-content c:\logs\mylog.txt}
```

Неинтерактивный запуск заданий

Вместо интерактивных сеансов можно использовать командлет Invoke-Command с параметром -AsJob. В этом случае объект-задание создается на локальном компьютере, даже если само задание запускается на удаленном компьютере. По завершении задания его результаты вернутся на локальный компьютер.

В следующем примере открывается неинтерактивный сеанс на трех удаленных компьютерах, затем с помощью Invoke-Command запускается фоновое задание, собирающее события из журналов System, Application и Security. Аналогичное задание было показано выше, но теперь оно запускается на компьютерах, заданных параметром -ComputerName. Следующая команда должна вводиться как единая строка:

```
$s = new-pssession -computername fileserver34, dataserver18, dcserver65
Invoke-command -session $s
-asjob -scriptblock {$share = "\\FileServer85\logs";
$logs = "system","application","security";
foreach ($log in $logs) {
$filename = "$env:computername".ToUpper() + "$log" + "log" +
(get-date -format yyyyMMdd) + ".log";
Get-EventLog $log | set-content $share\$filename; }
}
```

либо по частям, разделенным апострофом:

```
$s = new-pssession -computername fileserver34, dataserver18, dcserver65 `
Invoke-command -session $s `
```

```
-asjob -scriptblock {$share = «\\FileServer85\logs»; `
$logs = «system»,»application»,»security»; `
foreach ($log in $logs) { `
$filename = «$env:computername».ToUpper() + «$log» + «log» + `
(get-date -format yyyyMMdd) + «.log»; `
Get-EventLog $log | set-content $share\$filename; } `
}
```

Другой способ заключается в записи текста команды в файл на локальном компьютере и его загрузке с помощью параметра –FilePath, как показано ниже.

```
Команда

$s = new-pssession -computername fileserver34, dataserver18, dcserver65
Invoke-command -session $s -asjob -filepath c:\scripts\eventlogs.ps1

Файл Eventlogs.ps1

$share = «\\FileServer85\logs»
$logs = «system»,»application»,»security»

foreach ($log in $logs) {
  $filename = «$env:computername».ToUpper() + «$log» + «log» + `
(get-date -format yyyyMMdd) + «.log»
  Get-EventLog $log | set-content $share\$filename
}
```

Файл сценария должен располагаться на локальном компьютере либо в каталоге, доступном с локального компьютера. И в этом случае PowerShell преобразует содержимое файла в блок сценария, который запускается как фоновое задание.

В неинтерактивном сеансе вызывать Invoke-Command не обязательно, но такой прием позволяет работать с объектами заданий, запущенных в этом сеансе. Например, чтобы получить сведения о всех заданиях, введите:

```
get-job
```

Для получения результатов введите такую команду:

```
receive-job -keep
```

или, чтобы сохранить результаты в файл на локальном компьютере, такую:

```
receive-job > c:\logs\mylog.txt
```

Другая разновидность этого приема предполагает вызов Invoke-Command для запуска Start-Job. Этот метод позволяет запускать фоновые задания на нескольких удаленных компьютерах и сохранять их результаты на тех же компьютерах. Вот как это работает:

1. Вы вызываете Invoke-Command без параметра –AsJob, чтобы запустить командлет Start-Job.
2. На заданных удаленных компьютерах создаются объекты заданий.
3. На каждом из удаленных компьютеров выполняются команды фонового задания.
4. Результаты задания сохраняются отдельно на каждом из удаленных компьютеров.
5. Объекты и результаты заданий обрабатываются отдельно и на каждом из компьютеров.

Ниже командлет Invoke-Command используется для запуска заданий на трех компьютерах и сохранения объектов-заданий в переменной $j:

```
$s = new-pssession -computername fileserver34, dataserver18, dcserver65
$j = invoke-command -session $s {start-job -filepath c:\scripts\elogs.ps1}
```

Повторю, что использовать Invoke-Command в неинтерактивных сеансах не обязательно, но такой прием позволяет работать в сеансе с объектами заданий, запущенных на всех компьютерах. Например, чтобы получить информацию о заданиях, работающих на всех трех компьютерах, введите

```
invoke-command -session $s -scriptblock {get-job}
```

или, поскольку объекты-задания хранятся в переменной $j,

```
$j
```

Чтобы получить результаты заданий, введите такую команду:

```
invoke-command -session $s -scriptblock { param($j) receive-job -job $j
-keep} -argumentlist $j
```

или, чтобы сохранить результаты в файлах на удаленных компьютерах, такую:

```
invoke-command -session $s -command {param($j) receive-job -job $j > c:\
logs\mylog.txt} -argumentlist $j
```

В обоих примерах Invoke-Command используется для запуска командлета Receive-Job в каждом из сеансов, сохраненных в переменной $s. Поскольку $j — локальная переменная, в блоке сценария используется ключевое слово «param» для объявления этой переменной, а также параметр ArgumentList — для передачи ее значения.

Удаленная работа без WinRM

Некоторые командлеты поддерживают параметр -ComputerName, который позволяет обойтись без использования специальных функций Windows PowerShell remoting при удаленной работе. Это означает, что такие командлеты можно использовать на любых компьютерах с PowerShell, даже если

эти компьютеры не настроены для удаленной работы с командной оболочкой. Вот список этих командлетов:

Get-WinEvent	Get-Counter	Get-EventLog
Clear-EventLog	Write-EventLog	Limit-EventLog
Show-EventLog	New-EventLog	Remove-EventLog
Get-WmiObject	Get-Process	Get-Service
Set-Service	Get-HotFix	Restart-Computer
Stop-Computer	Add-Computer	Remove-Computer
Rename-Computer	Reset-Computer	MachinePassword

Чтобы использовать их на любом удаленном компьютере в домене, достаточно указать имена нужных компьютеров как значение параметра -ComputerName. Естественно, при этом политики и параметры конфигурации Windows должны разрешать удаленные подключения, а у текущего пользователя должны быть соответствующие разрешения.

Следующая команда запускает командлет Get-WinEvent на компьютерах PrintServer35 и FileServer17:

```
get-winevent -computername printserver35, fileserver17
```

Командлеты, вызванные с параметром ComputerName, возвращают объекты, которые содержат имя компьютера, на котором был сгенерирован вывод команды, в свойстве MachineName. По умолчанию свойство MachineName не отображается, но его можно добавить к выводу с помощью командлета Format-Table:

```
$procs = {get-process -computername Server56, Server42, Server27 |
sort-object -property Name}

&$procs | format-table Name, Handles, WS, CPU, MachineName -auto
```

```
Name                    Handles       WS        CPU MachineName
----                    -------       --        --- -----------
acrotray                     52  3948544          0 Server56
AlertService                139  7532544            Server56
csrss                       594 20463616            Server56
csrss                       655  5283840            Server56
CtHelper                     96  6705152   0.078125 Server56

. . .
acrotray                     43  3948234          0 Server42
AlertService                136  7532244            Server42
csrss                       528 20463755            Server42
csrss                       644  5283567            Server42
CtHelper                     95  6705576   0.067885 Server42

acrotray                     55  3967544          0 Server27
AlertService                141  7566662            Server27
```

```
csrss                          590 20434342            Server27
csrss                          654  5242340            Server27
CtHelper                        92  6705231   0.055522 Server27
```

Чтобы получить полный список командлетов, поддерживающих параметр –ComputerName, введите **get-help * -parameter ComputerName**. Чтобы узнать, поддерживает ли некоторый командлет параметр -ComputerName, введите **get-help** Имя_командлета **-parameter ComputerName**, например:

```
get-help Reset-ComputerMachinePassword -parameter ComputerName
```

Если параметр не требует специальной поддержки для удаленной работы, об этом сказано в его описании, например:

```
-ComputerName <string[]>

Resets the password for the specified computers. Specify one or more re-
mote computers. The default is the local computer.
```

Для работы с удаленными компьютерами достаточно указать их NetBIOS-имена, IP-адреса или полные доменные имена; чтобы указать локальный компьютер, следует вести его имя, точку (**.**) или **localhost**. Как видно из описания, этот параметр не требует поддержки Windows PowerShell remoting, значит его можно использовать даже на компьютерах, не настроенных для удаленного исполнения команд.

Глава 5

Основные структуры языка PowerShell

В любом языке программирования основные структуры определяют возможности языка и способы их использования. Сердцем Windows PowerShell является язык программирования C#, поэтому основные структуры PowerShell включают:

- выражения и операторы;
- переменные, значения и типы данных;
- строки, массивы и наборы.

Работая с PowerShell, вы каждый раз будете использовать основные структуры. Рекомендую вам внимательно изучить эту главу, чтобы освоить базовые «механизмы» PowerShell — это необходимо для эффективной работы с командной оболочкой. В этой главе мы подробно разберем азы языка PowerShell, чтобы не возвращаться к ним при разборе более сложных примеров, простроенных на базовых элементах.

Работа с выражениями и операторами

В Windows PowerShell выражением называет формулу, при вычислении по которой получается некоторый результат (значение). PowerShell поддерживает различные типы выражений: арифметические выражения (возвращающие численные значения), присваивание (устанавливающие значение) и сравнение (выражения, которые сравнивают значения).

Оператор — это элемент выражения, который «подсказывает» PowerShell, какую операцию нужно выполнить. Операторы используют в выражения для выполнения математических действий, присваивания и сравнения значений. Операторы делятся на три общих типа: арифметические, операторы присваивания и сравнения. PowerShell также поддерживает расширенные операторы, которые используются в регулярных выражениях, логические операторы и операторы для работы с типами.

Операторы для арифметических действий, группирования и присваивания

PowerShell поддерживает стандартные арифметические операторы (см. табл. 5-1).

Табл. 5-1. Арифметические операторы в PowerShell

Оператор	Действие	Результат обработки…			Пример
		…чисел	…строк	…массивов	
+	Сложение	Сумма	Объединенная строка	Объединенный массив	3 + 4
/	Деление	Частное	—	—	5 / 2
%	Целочисленное деление	Остаток от деления	—	—	5 % 2
*	Умножение	Произведение	Строка прибавляется сама к себе заданное число раз	Массив прибавляется сам к себе заданное число раз	6 * 3
–	Деление, отрицание	Разность	—	—	3 – 2

В табл. 5-2 перечислены операторы присваивания, поддерживаемые PowerShell. Такие операторы присваивают переменной одно или несколько значений и могут выполнять арифметические действия над значениями перед присваиванием. PowerShell поддерживает два специальных оператора присваивания, которые могут быть знакомы не всем: оператор приращения (++) и оператор декремента (--). Эти операторы позволяет быстро увеличить или уменьшить на единицу значение переменной, свойства или элемента массива.

Табл. 5-2. Операторы присваивания в PowerShell

Оператор	Действие	Пример	Значение
=	Присваивание значение	$a = 7	$a = 7
+=	Сложение с текущим значением переменной	$g += $h	$g = $g + $h
–=	Вычитание из текущего значения переменной	$g –= $h	$g = $g – $h
*=	Умножение на текущее значение переменной	$g *= $h	$g = $g * $h
/=	Деление на текущее значение переменной	$g /= $h	$g = $g / $h
%=	Целочисленное деление на текущее значение	$g %= $h	$g = $g % $h

Табл. 5-2. Операторы присваивания в PowerShell

Оператор	Действие	Пример	Значение
++	Приращение на 1	$g++ or ++$g	$g = $g + 1
--	Уменьшение на 1	$g-- or --$g	$g = $g – 1

Как видно из табл. 5-1 и 5-2, PowerShell поддерживает вполне обычный набор арифметических операторов и операторов присваивания, но несколько моментов все же заслуживают упоминания. В PowerShell функция целочисленного деления возвращает не частное, а остаток от деления. В следующем примере переменной $Remainder присваивается значение 0 (ноль):

```
$Remainder = 12 % 4
```

При вычислении этого выражения переменная $Remainder получает значение 1:

```
$Remainder = 10 % 3
```

Сделать значение отрицательным позволяет оператор –. Ниже переменной $Answer присваивается значение –15:

```
$Answer = -5 * 3
```

Операторы для расширения и исполнения команд позволяют объединять наборы элементов. Список группирующих операторов приводится с примерами ниже.

- **&** — используется для вызова блока сценария, команды или функции.

```
$a = {Get-Process -id 0}
&$a
```

```
Handles  NPM(K)   PM(K)    WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------   -----    ----- -----   ------     -- -----------
      0       0       0       24     0               0 Idle
```

- **( )** — используется для объединения операторов в группу, возвращает результат объединенного выражения.

```
$a = (5 + 4) * 2; $a
```

```
18
```

- **$( )** — служит для объединения команд в группы. После исполнения объединенной команды возвращается ее результат.

```
$($p= «win*»; get-process $p)
```

```
Handles  NPM(K)  PM(K)    WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------  -----    ----- -----   ------     -- -----------
    106       4   1448     4092    46              636 wininit
    145       4   2360     6536    56              780 winlogon
```

- **@( )** — объединяет операторы в группу, исполняет их и помещает результаты в массив.

```
@(get-date;$env:computername;$env:logonserver)
```

```
Friday, February 12, 2010 9:01:49 AM
CORSERVER34
\\CORDC92
```

мечание Массив — это просто структура данных для хранения серий значений. Подробнее о массивах — ниже в разделе «Работа с массивами и наборами».

Если в ваших выражениях используются разные операторы, то при определении порядка действий PowerShell следует правилам, знакомым вам со школьной скамьи. Так, умножение и деление выполняются до сложения и вычитания, то есть:

```
5 + 4 * 2 = 13
```

и

```
10 / 10 + 6 = 7
```

В табл. 5-3 описан порядок действий для арифметических операторов. Как видно из таблицы, сначала всегда вычисляются сгруппированные операторы, за ними — операторы инкремента и декремента. Далее устанавливаются положительные и отрицательные значения. После этого выполняются операции умножения и деления, затем целочисленное деление. Последними выполняются сложение и вычитание, после чего результат присваивается, как задано выражением.

Табл. 5-3. Порядок действий в PowerShell

Очередность	Действие
1	Группирование ( ) { }
2	Инкремент ++, декремент --
3	Унарные операторы + -
4	Умножение *, Деление /
5	Деление с остатком %
6	Сложение +, Вычитание –
7	Присваивание =

Интересно, что PowerShell может выполнять операции над единицами объема информации, такими как:

- килобайты (KB);
- мегабайты (MB);
- гигабайты (GB);

- терабайты (TB);
- петабайты (PB).

Вооружившись этим знанием, можно выполнять кое-какие простые расчеты прямо в командной строке. Например, чтобы при копировании содержимого диска емкостью 1 TB на диски Blu-Ray узнать, сколько двухслойных дисков потребуется, просто введите:

```
1TB / 50GB
```

и получите ответ:

```
20.48
```

В следующем примере выводится список файлов из каталога C:\Data, размер которых превышает 100 Кб:

```
get-item c:\data\* | where-object {$_.length -gt 100kb}
```

Операторы сравнения

Во время сравнения проверяется некоторое условие, например A больше B или A равно C. Операторы сравнения используются, прежде всего, в условных конструкциях, таких как If и If Else.

В табл. 5-4 приводится список операторов сравнения, поддерживаемых PowerShell, большинство из которых не нуждается в пояснениях. По умолчанию PowerShell не учитывает регистр символов при сравнении. Чтобы явно указать, нужно или нет учитывать регистр символов при сравнении, к оператору сравнения добавляют букву C или I, соответственно: -ceq или -ieq.

Табл. 5-4. Операторы сравнения в PowerShell

Оператор	Операция	Пример	Условие	Тип вывода
–eq	Проверка равенства	$g –eq $h	Равны ли g и h?	Boolean
	Проверка вхождения	$g –eq $h	Входит ли строка h в строку g?	Boolean
–ne	Проверка неравенства	$g –ne $h	Являются ли g и h неравными?	Boolean
	Проверка вхождения других строк	$g –ne $h	Входит ли в строку g строки, отличные от h?	Boolean
–lt	Меньше	$g –lt $h	g меньше h?	Boolean
–gt	Больше	$g –gt $h	g больше h?	Boolean
–le	Меньше или равно	$g –le $h	g меньше или равно h?	Boolean
–ge	Больше или равно	$g –ge $h	g больше или равно h?	Boolean

(*см. след. стр.*)

Табл. 5-4. Операторы сравнения в PowerShell

Оператор	Операция	Пример	Условие	Тип вывода
–contains	Вхождение	$g –contains $h	Входит ли строка h в строку g?	Boolean
–notcontains	Отрицание вхождения	$g –notcontains $h	Действительно ли строка h не входит в строку g?	Boolean
–like	Вхождение подобных строк	$g –like $h	Входит ли в строку g строка, подобная h?	Boolean
–notlike	Вхождение строк, отличных от подобных	$g –notlike $h	Действительно ли строка, подобная h, не входит в строку g?	Boolean
–match	Поиск соответствий	$g –match $h	Есть ли в строке g подстроки, соответствующие выражению h?	Boolean
–notmatch	Поиск несоответствий	$g –notmatch $h	Действительно ли в строку g не входят подстроки, соответствующие выражению h?	Boolean
–replace	Замена	$g –replace $h, $i	Замена всех вхождений подстроки h в строке g подстрокой i	String

Заметьте, что эти операторы можно использовать для сравнения не только чисел, но и строк, а порядок действий для этих операторов не установлен, то есть операции сравнения всегда выполняются поочередно, слева направо.

Большинство операторов сравнения возвращают булевы (Boolean) значения, которые указывают, соответствуют ли друг другу сравниваемые значения. В этом примере вычисляются и сравниваются значения $a и $b; в результате получается False, поскольку эти значения различаются:

```
$a = 5; $b = 6
$a -eq $b
```

```
False
```

А здесь проверяется неравенство значений $a и $b, поэтому результат — True:

```
$a = 5; $b = 6
$a -ne $b
```

```
True
```

Здесь мы проверяем, действительно ли $a меньше $b и получаем True:

```
$a = 5; $b = 6
$a -lt $b
```

```
True
```

При работе с массивами и наборами операторы –eq, –ne, –lt, –gt, –le и –ge возвращают все элементы массива, соответствующие заданному выражению. Например, оператор –eq проверяет, есть ли в массиве значения, соответствующие заданному, и выводит все обнаруженные соответствия:

```
$a = «iexplorer», «iexplorer», «powershell»; $b = «iexplorer»
$a -eq $b
```

```
iexplorer
iexplorer
```

Аналогичным образом оператор –ne проверяет наличие в массиве значений, отличных от заданного. Обнаружив такие значения, оператор выводит их, как показано ниже:

```
$a = «svchost», «iexplorer», «powershell»; $b = «iexplorer»
$a -ne $b
```

```
svchost
powershell
```

Операторы –contains и –notcontains применяют для поиска в строках, массивах и наборах значений, соответствующих либо не соответствующих заданным. В следующем примере проверяется наличие строки winlogon в переменной $a:

```
$a = «svchost», «iexplorer», «powershell»
$a -contains «winlogon»
```

```
False
```

Операторы –like, –notlike, –match и –notmatch используются для поиска по шаблону. Операторы –like и –notlike допускают использование подстановочных знаков для поиска строк, сходных либо несходных с заданной. В следующем примере показан поиск в значении $a строк, соответствующих *host:

```
$a = «svchost»
$a -like «*host»
```

```
True
```

В примерах из других глав я пользовался подстановочным знаком *, соответствующим любой подстроке; PowerShell поддерживает и другие подстановочные знаки, включая ? и [ ] (табл. 5-5).

Табл. 5-5. Подстановочные знаки в PowerShell

Подстановочный знак	Чему соответствует	Пример	Результат примера
*	Любой строке длиной 0 и более символов	help *-alias	export-alias, import-alias, get-alias, new-alias, set-alias
?	Одному символу в заданной позиции	help ??port-alias	export-alias, import-alias
[ ]	Одному символу из заданного диапазона	help [a-z]e[t-w]-alias	get-alias, new-alias, set-alias
[ ]	Одному символу из перечисленных	help [efg]et-alias	get-alias

При работе с массивами и наборами оператор –like возвращает все элементы массива, соответствующие либо не соответствующие заданному выражению, например:

```
1,1,2,3,4,5,5,6,3,4 -like 4
```

```
4
4
```

Напротив, оператор –notlike возвращает элементы, не соответствующие заданному:

```
2,3,4,5,5,6,4 -notlike 4
```

```
2
3
5
5
6
```

Операторы –match и –notmatch определяют, соответствует ли значение заданному регулярному выражению. Регулярное выражение можно назвать расширенным набором подстановочных знаков. В следующем примере мы проверяем, содержит ли значение переменной $a буквы s, t или u:

```
$a = «svchost»
$a -match «[stu]»
```

```
True
```

В табл. 5-6 перечислены знаки, которые используются в регулярных выражениях. При использовании регулярных выражений в стиле .NET Framework, к этим знакам добавляются квантификаторы, принятые в .NET, позволяющие точнее задавать соответствие.

Табл. 5-6. Знаки, применяемые в регулярных выражениях

Знак	Чему соответствует	Пример, возвращающий True
[подстрока]	Подстроке в любой позиции заданной строки	“powershell” –match “er”
.	Любому символу	“svchost” –match “s.....t”
[список]	Любому символу из заданного списка	“get” –match “g[aeiou]t”
[диапазон]	Любому значению из заданного диапазона; непрерывный диапазон задают с помощью дефиса (-)	“out” –match “[o-r]ut”
[^]	Любому символу, кроме заданных в скобках	“bell” –match “[^tuv]ell”
^	Начальной части строки	“powershell” –match “^po”
$	Окончанию строки	“powershell” –match “ell$”
*	Любой подстроке	“powershell” –match “p*”
?	Любому символу	“powershell” –match “powershel?”
+	Повторам заданной подстроки	“zbzbzb” –match “zb+”
\	Литералу, который идет после \. Например, двойные кавычки, перед которыми стоит знак \, интерпретируются как литерал, а не разделительный знак	“Shell$” –match “Shell\$”
Регулярные выражения в стиле .NET		
\p{имя}	Любому символу в именованном классе символов, объявленном как {имя}. Поддерживаются группы и диапазоны Unicode-символов, такие как Ll, Nd, Z, IsGreek и IsBoxDrawing	“abcd” –match “\p{Ll}+”
\P{имя}	Любой строке, кроме знаков из диапазона, объявленного как {имя}	1234 –match “\P{Ll}+”
\w	Любому знаку, образующему слово (эквивалент категорий Unicode-символов [\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}), при включенном параметре ECMAScript — эквивалент [a-zA-Z_0-9]	“abcd defg” –match “\w+”
\W	Любому знаку, не образующему слово (эквивалент Unicode-категорий [^\p{Ll}\p{Lu}\p{Lt} \p{Lo}\p{Nd}\p{Pc}])	“abcd defg” –match “\W+”
\s	Любому пробельному символу (эквивалент Unicode-категорий [\f\n\r\t\v\x85\p{Z}])	“abcd defg” –match “\s+”

(*см. след. стр.*)

Табл. 5-6. Знаки, применяемые в регулярных выражениях

Знак	Чему соответствует	Пример, возвращающий True
\S	Любому непробельному символу (эквивалент Unicode-категорий [^\f\n\r\t\v\x85\p{Z}].	“abcd defg” –match “\S+”
\d	Любой десятичной цифре; эквивалент \p{Nd} (Unicode) и [0–9] (не-Unicode)	12345 –match “\d+”
\D	Любому символу кроме цифр; эквивалент \P{Nd} (Unicode) и [^0-9] (не-Unicode)	“abcd” –match “\D+”
Квантификаторы .NET Framework		
*	Любому числу вхождений подстроки, заданной регулярным выражением в стиле .NET Framework	“abc” –match “\w*”
?	Нулевому или единственному вхождению подстроки, заданной регулярным выражением в стиле .NET Framework	“abc” –match “\w?”
{n}	Указанному числу вхождений подстроки, заданной регулярным выражением в стиле .NET Framework	“abc” –match “\w{2}”
{n,}	Минимум *n* вхождениям подстроки, заданной регулярным выражением в стиле .NET Framework	“abc” –match “\w{2,}”
{n,m}	Не меньше *n* и не больше *m* вхождений подстроки, заданной регулярным выражением в стиле .NET Framework	“abc” –match “\w{2,3}”

Регулярные выражения можно использовать не только с операторами –match и –notmatch. Так, разрешается сравнивать значения с регулярными выражениями, объявленными явно. В следующем примере объявлено регулярное выражение, которое проверяет наличие в строке букв из диапазонов a–z и A–Z:

```
[regex]$regex=«^([a-zA-Z]*)$»
```

В регулярных выражениях знак ^ представляет начало строки, а $ — конец строки. С помощью квадратных скобок задают знаки и диапазоны знаков, соответствие которым проверяет регулярное выражение. Так, выражение [a-zA-Z]* задает поиск любого числа прописных и строчных букв латинского алфавита.

Определив регулярное выражение, можно использовать его метод IsMatch() для проверки соответствия значений этому выражению, как показано в примере ниже:

```
$a =«Tuesday»
```

```
[regex]$regex=«^([a-zA-Z]*)$»
$regex.ismatch($a)
```

```
True
```

Если для сравнения используется строка, содержащая цифры, пробелы, знаки препинания и другие спецсимволы, IsMatch вернет False:

```
$days =«Monday Tuesday»
[regex]$regex=«^([a-zA-Z]*)$»
$regex.ismatch($days)
```

```
False
```

Примечание PowerShell также поддерживает операторы для разделения, объединения и форматирования строк (см. раздел «Работа со строками» ниже).

Оператор –replace позволяет заменить все вхождения заданной подстроки другой подстрокой. Так, в следующем примере все вхождения «host» в значении $a заменяются строкой «console»:

```
$a = «svchost», «iexplorer», «loghost»
$a -replace «host», «console»
```

```
svcconsole
iexplorer
logconsole
```

Другие операторы

В дополнение к рассмотренным выше PowerShell поддерживает логические операторы и операторы для работы с типами (табл. 5-7).

Табл. 5-7. Операторы PowerShell для логических действий и работы с типами

Оператор	Действие	Описание	Пример
–and	Логическое И	Дает True, только если оба выражения дают True	(5 –eq 5) –and (3 –eq 6) False
–or	Логическое ИЛИ	Дает True, если одно из выражений дает True	(5 –eq 5) –or (3 –eq 6) True
–xor	Исключающее ИЛИ	Дает True, только если одно из выражений дает True, другое — False	(5 –eq 5) –xor (3 –eq 6) True
–not, !	Логическое НЕ	Меняет значение выражения на противоположное; дает True, если выражение дает False и наоборот	–not (5 –eq 5) False ! (5 –eq 5) False

(*см. след. стр.*)

Табл. 5-7. Операторы PowerShell для логических действий и работы с типами

Оператор	Действие	Описание	Пример
–is	Проверка равенства объектов	Возвращает True, если заданный объект соответствует указанному типу .NET Framework	(get-date) –is [datetime] True
–isnot	Проверка неравенства объектов	Возвращает True, если заданный объект не соответствует указанному типу .NET Framework	(get-date) –isnot [datetime] False
–as	Преобразование типов	Преобразует объект в указанный тип .NET Framework	“3/31/10” –as [datetime] Wednesday, March 31, 2010 12:00:00 AM
,	Создание массивов	Создает массив значений, разделенных запятыми	$a = 1,2,4,6,4,2
..	Определение диапазона	Задает диапазон значений	$a = 2..24
–band	Двоичное И	Выполняет двоичную операцию AND	–
–bor	Двоичное ИЛИ	Выполняет двоичную операцию OR	–
–bnot	Двоичное НЕ	Выполняет двоичное отрицание	–
–bxor	Двоичное исключающее ИЛИ	Выполняет операцию двоичное XOR	–

Еще один оператор, с которым вам необходимо познакомиться, — оператор Is. Они используется для сравнения объектов по типу .NET Framework. Если сравниваемые объекты одного типа, в результате получается True, в противном случае — False.

В таб. 5-8 приводится расширенный порядок действий для всех поддерживаемых операторов, учитывающий все возможные комбинации операторов и порядок их обработки.

Табл. 5-8. Расширенный порядок действий в PowerShell

Очередность	Действие
1	Объединение в группы ( ) { }
2	Расширение команды @ $
3	Отрицание !
4	Расширение через подстановочные знаки [ ]
5	Подстановка вместо точки .

Табл. 5-8. Расширенный порядок действий в PowerShell

Очередность	Действие
6	Вызов через &
7	Инкремент ++, декремент --
8	Унарные операторы + –
9	Умножение *, деление /
10	Деление с остатком %
11	Сложение +, вычитание –
12	Операторы сравнения
13	–and, –or
14	Конвейер |
15	Перенаправление > >>
16	Присваивание =

Работа с переменными и значениями

Переменные — это символы, которые заменяются реальным значением переменной, которым может быть строка, число или объект. Чтобы обратиться к значению переменной, достаточно сослаться на нее по имени. Переменные, хранящие объекты (сгенерированные объектами), позволяют передавать их другим командам через конвейер.

В сценариях и командах разрешается использовать любые поддерживаемые переменные. По умолчанию, объявленные переменные существуют только в текущем сеансе и теряются после завершения сеанса. Чтобы сохранить переменные после завершения сеанса, необходимо добавить и объявления в профиль (подробнее о профилях см. в главе 3).

Если информацию о конфигурации не требуется жестко «зашивать» в код сценария, имеет смысл хранить ее в переменных — так проще обновлять и обслуживать сценарии. Также рекомендуется собирать объявления переменных в одном месте, например в начале сценария, — это облегчит поиск переменных и работу с ними.

PowerShell поддерживает четыре класса переменных: автоматические переменные (automatic variables), управляющие переменные (preference variables), переменные окружения (environment variables) и переменные, объявленные пользователем (user-created variables). В отличие от командной строки Windows, хранящей переменные исключительно в виде текстовых строк, в PowerShell переменные хранятся как строки либо как объекты. Впрочем, с технической точки зрения строка также является объектом, подробнее об этом — ниже, в разделе «работа со строками».

Основы работы с переменными

Для работы с переменными используются следующие командлеты:

- **Get-Variable** — выводит имена и значения всех или только заданных переменных, установленных в текущем сеансе.

```
Get-Variable [[-Name] Имена_переменных] [AddtlParams]

AddtlParams=
[-Scope Строки] [-Exclude Строки] [-Include Строки] [-ValueOnly]
```

- **New-Variable** — объявляет новую переменную.

```
New-Variable [[-Value] Объект] [-Name] Имя_переменных [AddtlParams]

AddtlParams=
[-Description Строка] [-Force] [-Option None | ReadOnly | Constant
| Private | AllScope] [-PassThru] [-Scope Строка] [-Visibility
Public | Private]
```

- **Set-Variable** — объявляет новую или изменяет объявление существующей переменной.

```
Set-Variable [[-Value] Объект] [-Name] Имена_переменных [AddtlParams]

AddtlParams=
[-Description Строка] [-Exclude Строки] [-Force] [-Include
Строки] [-Option None | ReadOnly | Constant | Private | AllScope]
[-PassThru] [-Scope Строк] [-Visibility Public | Private]
```

- **Remove-Variable** — удаляет переменную и ее значение; параметр –Force позволяет удалять неизменяемые переменные.

```
Remove-Variable [-Name] VarNames[AddtlParams]

AddtlParams=
[-Scope Строка] [-Force] [-Exclude Строка] [-Include Строки]
```

- **Clear-Variable** — сбрасывает значение переменной в NULL.

```
Clear-Variable [-Force] [-PassThru] [-Scope String] [-Exclude
Strings] [-Include Strings] [-Name] VarNames
```

Независимо от класса, на переменные ссылаются по имени, перед которым ставится знак доллара ($). Так поступают как при объявлении новых переменных, так и при ссылке на уже объявленные и установленные. Знак доллара отличает имена переменных от псевдонимов, функций, командлетов и других элементов PowerShell. Возможности определения переменных описаны в табл. 5-9.

Табл. 5-9. Синтаксис переменных в PowerShell

Синтаксис	Описание
$myVar = “*Значение*”	Определяет переменную со стандартным именем, которое начинается с $ и содержит буквы (a–z, A–Z), цифры (0–9) и символ подчеркивания (_); имена переменных не чувствительны к регистру символов
${my.var!!!!} = “*Значение*”	Определяет переменную с нестандартным именем, в начале которого стоит знак $, а затем идут круглые скобки, в которых могут быть любые символы. Чтобы включить знак скобки в имя переменной, необходимо предварить этот знак апострофом (`)
[*type*] $myVar = “*Значение*”	Определяет строго типизированную переменную, принимающую только значения заданного типа. Если при установке этой переменной новое значение не удается привести к заданному типу, PowerShell генерирует ошибку
$*SCOPE*:$myVar = “*Значение*”	Определяет переменную с заданной областью действия (область действия — логические границы доступности переменной, подробнее см. в разделе «Управление областью действия переменных» ниже)
New-Item Variable:\myVar –Value *Значение*	Определяет новую переменную с использованием поставщика Variable (подробнее о поставщиках см. в главе 3)
Get-Item Variable:\myVar Get-Variable myVar	Получает переменную с помощью поставщика Variable или командлета Get-Variable
${*путь\имя.расширение*}	Определяет переменную с помощью Get-Content или Set-Content. Если *путь\имя.расширение* указывает на допустимый файл, можно получать и устанавливать значение такой переменной путем чтения и записи данных в нее

Чтобы определить переменную, необходимо назначить ей имя и присвоить значение с помощью оператора равенства (=). Стандартные имена переменных не чувствительны к регистру и могут включать любые комбинации букв (a–z, A–Z), цифр (0–9) и символа подчеркивания (_). Соответственно, следующие переменные допустимы:

```
$myString = «String 1»
$myVar = «String 2»
$myObjects = «String 3»
$s = «String 4»
```

Для доступа к значению переменной достаточно сослаться на нее по имени. Так, чтобы вывести значение ранее определенной переменной $myString, введите в командной строке **$myString** — PowerShell отобразит, например, следующее:

Получить список доступных переменных, введите **get-variable** в командной строке PowerShell. Этот вывод включает переменные, объявленные вами, а также текущие автоматические и управляющие переменные. Поскольку к переменным окружения обращаются через соответствующий поставщик, перед их именем следует писать $env:, а затем — имя нужной переменной. Например, для просмотра переменной %UserName% необходимо ввести **$env:username**.

У переменных могут быть и нестандартные имена, включающие спецсимволы, такие как дефис, точка, двоеточие и скобки, такие имена заключаются в фигурные скобки, которые заставляют PowerShell интерпретировать имя переменной как литерал. Таким образом, следующие имена допустимы:

```
${my.var} = «String 1»
${my-var} = «String 2»
${my:var} = «String 3»
${my(var)} = «String 4»
```

Чтобы использовать круглые скобки в именах переменных, необходимо предварить их знаком апострофа (`). Например, чтобы определить переменную “my{var}string” с значением “String 5”, введите

```
${my`{var`}string} = «String 5»
```

В предыдущих примерах переменным назначались строковые значения, так же просто присваивать в качестве значений числа, массивы и объекты. При этом числовые значения интерпретируются как числа, а не строки. Массивы — это структуры для хранения однотипных элементов данных. Значения в массиве разделяются запятой либо вводятся с помощью оператора, задающего диапазоны значений (..). Набор объектов можно получив, сохранив в переменной вывод какого-нибудь командлета, например:

```
$myFirstNumber = 10
$mySecondNumber = 500
$myFirstArray = 0,1,2,3,4,8,13,21
$mySecondArray = 1..9
$myString = «Hello!»
$myObjectCollection1 = get-service
$myObjectCollection2 = get-process -name svchost
```

Здесь создаются переменные, хранящие целочисленные значения, массивы, строки и наборы объектов. Со значениями, хранящимися в переменных, работают так же, как и с исходными значениями. Например, чтобы сосчитать число объектов-служб в переменной $myObjectCollection1, введите

```
$myObjectCollection1.Count
```

Хранящиеся в переменной объекты можно даже отсортировать по состоянию и вывести упорядоченный список с помощью следующей команды:

```
$myObjectCollection1 | sort-object -property status
```

Допускаются значения с любым типом данных, поддерживаемым .NET Framework (табл. 5-10).

Табл. 5-10. Типы данных и их псевдонимы

Псевдоним	Тип данных
[adsi]	Объект Active Directory Services Interface (ADSI)
[array]	Массив значений
[bool]	Булево значение (True или False)
[byte]	8-разрядное значение без знака из диапазона 0–255
[char]	16-разрядный символ Unicode
[datetime]	Время и дата
[decimal]	128-разрядное десятичное значение
[double]	64-разрядное число двойной точности с плавающей точкой
[float]	32-разрядное число одинарной точности с плавающей точкой
[hashtable]	Ассоциативный массив, определенный в объекте Hashtable
[int]	32-разрядное целое число со знаком
[long]	64-разрядное целое число со знаком
[psobject]	Объект общего типа, способный инкапсулировать объект любого базового типа
[regex]	Регулярное выражение
[scriptblock]	Серия команд
[single]	32-разрядное число одинарной точности с плавающей точкой
[string]	Unicode-строка фиксированной длины
[wmi]	объект WMI
[xml]	объект XML

Переменные также используются для хранения результатов вычислений. Рассмотрим следующую команду и ее результаты:

```
$theFirstResult = 10 + 10 + 10
$theSecondResult = $(if($theFirstResult -gt 25) {$true} else {$false})
write-host «$theFirstResult is greater than 25? `t $theSecondResult»
```

```
30 is greater than 25?       True
```

Вычислив значение первого выражения, PowerShell сохраняет его в первой переменной, значение второго выражения — во второй переменной. Второе выражение сравнивает значение первой переменной и 25. Если оно больше 25, PowerShell присваивает второй переменной значение True либо False в противном случае. В завершение PowerShell выводит результаты на консоли.

В этом примере первая переменная объявлена с типом данных Integer, а вторая —Boolean. В общем случае, если во время определения переменной присваивается целочисленное значение (например, 3 или 5), PowerShell создает переменную целочисленного типа (Integer). Если присваивается дробное значение, например 6,87 или 3,2, переменной назначается тип Double, представляющий с двойной точностью дробные числа с плавающей точкой (или плавающей запятой). Если же присваиваемое значение представляет собой смесь букв и цифр, как в Hello или S35, создается строковая переменная типа String.

В PowerShell довольно часто используются переменные типа Boolean. При вычислении выражений с булевыми операторами, например при обработке конструкции If, PowerShell получает значения типа Boolean. Булевы значения бывают лишь двух видов — True (Истина) или False (Ложь), для представления используются одноименные литералы $true и $false.

В табл. 5-11 приводятся примеры представлений различных элементов PowerShell в виде булевых значений. Заметьте, что для представления ссылок на любые объекты используется True, а для представления $null — False.

Табл. 5-11. Булевы представления различных элементов

Элемент или число	Булево представление
$true	True
$false	False
$null	False
Ссылка на объект	True
Число, отличное от ноля	True
Ноль	False
Непустая строка	True
Пустая строка	False
Непустой массив	True
Пустой массив	False
Пустой ассоциативный массив	True
Непустой ассоциативный массив	True

Преобразование типов данных и присваивание

В PowerShell разрешается присваивать объекты напрямую, не объявляя их тип данных, благодаря встроенной возможности автоматического определения типа данных. Впрочем, вы всегда можете узнать тип данных переменной, вызвав команду

Имя_переменной`.GetType()`

где *Имя_переменной* — реальное имя переменной, например:

```
$myFirstNumber.GetType()
```

```
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Int32                                    System.ValueType
```

Обратите внимание, что в выводе базовый тип переменной указан как System.ValueType, а имя — как Int32. Из этого видно, что общий тип переменной — из пространства имен System, а конкретный тип — Int32, то есть переменная имеет 32-разрядное целочисленное значение.

Проще всего объявить тип переменной с использованием псевдонима, но можно делать эти и с применением...

- полного имени класса, например [System.Array], [System.String] или [System.Diagnostics.Service] или
- имени класса в пространстве имен System, например [Array], [String] или [Diagnostics.Service].

Но будьте осторожны: если привести значение к указанному типу данных не удастся, PowerShell генерирует ошибку, именно поэтому определение типа данных лучше оставить PowerShell. Если тип данных переменной установлен, то каждый раз во время присваивания ей значения PowerShell пытается привести его к этому типу, и несоответствие типов приводит к ошибке. Рассмотрим следующий пример:

```
$myNumber = 52
$myNumber += «William»
```

Здесь создается переменная, которой присваивается целочисленное значение (52). Далее мы пытаемся прибавить к переменной строку с помощью оператора инкремента (+=), что приводит к ошибке из-за несоответствия типов:

```
Cannot convert value «William» to type «System.Int32». Error: «Input
string was not in a correct format.»
At line:1 char:13
+ $myNumber += <<<<  “William”
    + CategoryInfo          : NotSpecified: (:) [], RuntimeException
    + FullyQualifiedErrorId : RuntimeException
```

Причина ошибки — невозможность автоматического преобразования строковых значений в числовые. Однако в других случаях PowerShell выполняет необходимое преобразование типов автоматически, например:

```
$myNumber = «William»
$myNumber += 52
```

Мы создаем переменную, присваиваем ей строковое значение и пытаемся прибавить к нему число. В результате PowerShell автоматически преобразует число в строку, и переменная $myNumber получает значение “William52”.

При создании переменных можно использовать любые псевдонимы типов, перечисленные в табл. 5-1. Эти псевдонимы всегда используются одинаково при объявлении типа данных в PowerShell.

В следующем примере переменная объявляется как 32-разрядная целочисленная переменная с именем $myNumber и значением 10:

```
[int]$myNumber = 10
```

В результате создается строго типизированная переменная, которой можно присваивать только значения объявленного типа. Если PowerShell не сможет привести присваиваемое значение к этому типу, возникает ошибка. Подробнее о типах данных — в главе 6.

Чтобы явно задать для числа длинный целочисленный или десятичный формат, используйте суффиксы L и D, соответственно. Вот примеры:

```
$myLongInt = 52432424L
$myDecimal = 2.2425D
```

PowerShell также поддерживает научный формат чисел. Так, выражение

```
$mathPi = 3141592653e-9
```

устанавливает для переменной $mathPi значение 3,141592653.

Совет В PowerShell для ссылки на математические константы π и e используются статические свойства класса [System.Math]. Значение [System.Math]::Pi равно 3,14159265358979, а [System.Math]::E — 2,71828182845905.

Шестнадцатеричные числа вводятся с префиксом 0x, PowerShell хранит их как целочисленные значения. При воде следующего выражения

```
$ErrorCode = 0xAEB4
```

PowerShell преобразует шестнадцатеричное число AEB4 в десятичное и хранит его как 44724.

В PowerShell нет встроенной поддержки других систем счисления, но статические методы класса [Convert] из .NET Framework (табл. 5-12) позволяет выполнять самые разные преобразования типов данных.

Табл. 5-12. Наиболее востребованные статические методы класса [Convert]

Метод	Во что преобразует заданное значение
ToBase64CharArray()	Массив символов, закодированных по алгоритму base64
ToBase64String()	Строка, закодированная по алгоритму base64
ToBoolean()	Boolean
ToByte()	8-разрядное целое число без знака
ToChar()	16-разрядный Unicode-символ (Char)
ToDateTime()	Дата и время (DateTime)
ToDecimal()	Десятичное число

Табл. 5-12. Наиболее востребованные статические методы класса [Convert]

Метод	Во что преобразует заданное значение
ToDouble()	64-разрядное число двойной точности с плавающей точкой
ToInt16()	16-разрядное целое число
ToInt32()	32-разрядное целое число
ToInt64()	64-разрядное целое число
ToSByte()	8-разрядное число со знаком
ToSingle()	32-разрядное число одинарной точности с плавающей точкой
ToString()	Строка
ToUInt16()	16-разрядное целое число без знака
ToUInt32()	32-разрядное целое число без знака
ToUInt64()	64-разрядное целое число без знака

Методы класса [Convert] также позволяют преобразовывать двоичные, восьмеричные, десятичные и шестнадцатеричные методы. Например, так можно преобразовать значение в 32-разрядное целое:

```
$myBinary = [Convert]::ToInt32(«1011111011110001», 2)
```

В результате получается 48881. Следующий пример демонстрирует преобразование восьмеричного значения в 32-разрядное целое:

```
$myOctal = [Convert]::ToInt32(«7452», 8)
```

Результат — 3882. А так можно преобразовать шестнадцатеричное число в 32-разрядное целое:

```
$myHex = [Convert]::ToInt32(«FEA07E», 16)
```

Это выражение дает 16687230. Обратные преобразования выполняются так же просто. Следующий пример преобразует число 16687230 в строку, содержащую шестнадцатеричное число:

```
$myString = [Convert]::ToString(16687230, 16)

The result is “FEA07E”.
```

Как видите, возможны разные преобразования, но иногда требуется заставить PowerShell использовать значение переменную именно как строку. Для этого служит метод ToString(), преобразующий значение переменной в строку. Он работает со значениями типа Boolean, Byte, Char и Datetime, а также с любыми числами, но не работает с большинством других типов данных.

Чтобы преобразовать значение в строку, просто передайте его методу ToString():

```
$myNumber = 505
```

```
$myString = $myNumber.ToString()
$myString.GetType()
```

```
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object
```

У классов [System.Datetime] и [System.Math] есть и другие методы и свойства, самые полезные из них перечисленные в табл. 5-13.

Табл. 5-13. Полезные статические методы и свойства классов [DateTime] и [Math]

Имя класса или члена	Описание	Синтаксис
[Datetime]		
Compare()	Сравнивает объекты, представляющие дату. Возвращает 0, если они равны, либо 1 в противном случае	[Datetime]::Compare(d1,d2)
DaysInMonth()	Возвращает число дней в заданном месяце	[Datetime]::DaysInMonth (year, month)
Equals	Сравнивает объекты, представляющие дату. Возвращает True, если они равны	[Datetime]::Equals(d1,d2)
IsLeapYear	Возвращает True, если заданный год — високосный	[Datetime]::IsLeapYear(year)
Now	Возвращает текущую дату и время	[Datetime]::Now
Today	Возвращает текущую дату и время в формате 12:00 A.M.	[Datetime]::Today
[Math]		
Abs()	Возвращает модуль числа	[Math]::Abs(value1)
Acos()	Возвращает арккосинус	[Math]::Acos(value1)
Asin()	Возвращает арксинус	[Math]::Asin(value1)
Atan()	Возвращает арктангенс	[Math]::Atan(value1)
Atan2()	Возвращает обратный арккосинус	[Math]::Atan2(value1)
BigMul()	Returns the multiple as a 64-bit integer.	[Math]::BigMul(val1,val2)
Ceiling()	Возвращает наименьшее целое число, превосходящее заданное	[Math]::Ceiling(val1)
Cos()	Возвращает косинус	[Math]::Cos(val1)

Табл. 5-13. Полезные статические методы и свойства классов [DateTime] и [Math]

Имя класса или члена	Описание	Синтаксис
Cosh()	Возвращает обратный косинус	[Math]::Cosh(val1)
DivRem()	Возвращает остаток от деления двух чисел	[Math]::DivRem(val1,val2)
Equals()	Проверяет равенство объектов	[Math]::Equals(obj1,obj2)
Exp()	Возводит *e* в заданную степень	[Math]::Exp(value1)
Floor()	Возвращает наибольшее целое число, не превосходящее заданное	[Math]::Floor(val1)
Log()	Возвращает логарифм	[Math]::Log(value1)
Log10()	Возвращает десятичный логарифм	[Math]::Log10(value1)
Max()	Возвращает большее из заданных чисел	[Math]::Max(val1,val2)
Min()	Возвращает меньшее из заданных чисел	[Math]::Min(val1,val2)
Pow()	Возводит заданное число в заданную степень	[Math]::Pow(val1,val2)
Round()	Округляет число	[Math]::Round(val1)
Sign()	Возвращает 16-разрядное целое число со знаком	[Math]::Sign(val1)
Sin()	Возвращает синус	[Math]::Sin(value1)
Sinh()	Возвращает обратный синус	[Math]::Sinh(value1)
Sqrt()	Возвращает квадратный корень	[Math]::Sqrt(value1)
Tan()	Возвращает тангенс	[Math]::Tan(value1)
Tanh()	Возвращает обратный тангенс	[Math]::Tanh(value1)
Truncate()	Усекает десятичное число	[Math]::Truncate(value1)
E	Возвращает константу *e*	[Math]::E
Pi	Возвращает константу π	[Math]::Pi

В табл. 5-14 приводится обзор методов экземпляра и свойств объекта String, которым часто пользуются при работе со значениями в PowerShell.

Табл. 5-14. Востребованные методы и свойства объектов String

Имя	Описание	Пример
Length	Возвращает длину строки в символах	$s.Length
Contains()	Возвращает True, если строка string2 входит в строку, заданную переменной $s	$s.Contains(“string2”)
EndsWith()	Возвращает True, если строка $s оканчивается строкой string2	$s.EndsWith(“string2”)
Insert()	Вставляет строку string2 в строку $s после заданного символа	$s.Insert(0,“string2”) $s.Insert($s.Length,“string2”)
Remove()	Удаляет из строки $s указанное число символов в заданной позиции. Если число удаляемых символов не указано, удаляются все символы после заданного	$s.Remove(5,3) $s.Remove(5)
Replace()	Заменяет все вхождения заданной подстроки другой заданной подстрокой	$s.Replace(“this”,”that”)
StartsWith()	Возвращает True, если строка $s начинается строкой string2	$s.StartsWith(“string2”)
SubString()	Вырезает из строки $s подстроку заданной длины, начинающуюся в заданной позиции. Если длина не указана, извлекаются все символы после заданного	$s.Substring(3,5) $s.Substring(3)
ToLower()	Преобразует строку в нижний регистр	$s.ToLower()
ToString()	Преобразует объект в строку	$s.ToString()
ToUpper()	Преобразует строку в верхний регистр	$s.ToUpper()

Управление областью действия переменных

Область действия (scope) переменной определяет ее логические границы. Переменные могут быть глобальными (global), локальными (local), локальными для сценария (script) и закрытыми (private). По области действия (от самой широкой до самой узкой) переменные можно упорядочить следующим образом:

global > script/local > private

Информация с более широкой областью действия доступна с более низких уровней, но не наоборот.

Примечание Область действия имеется не только у переменных, но и у псевдонимов, функций и дисков PowerShell. Глобальную область действия можно представить как родительскую, а более узкие области действия — как дочерние или вложенные области действия. Технически, переменные, псевдонимы и функции родительской области действия не являются частью дочерней области действия: дочерняя область действия ничего не наследует от родительской, но переменные, псевдонимы и функции родительской области действия доступны из дочерней области действия. Таким образом, код из дочерней области действия может изменять переменные, псевдонимы и функции родительской области действия, явно указывая родительскую область действия.

Для сценария по умолчанию определена область действия script, т.е. все переменные, псевдонимы и функции, объявлены в сценарии, существуют только для этого сценария. Для функции по умолчанию определена область действия local. Соответственно, все объявленные в функции переменные существуют только для этой функции, несмотря на то, что сам функция объявлена в сценарии.

Для всех переменных, псевдонимов, функций и дисков PowerShell области действия работают одинаково: элемент, определенный в некоторой области действия, доступен в ней и ее дочерних областях действия (если только ему не назначена явно область действия private). Элемент, определенный в некоторой области действия, может быть изменен только кодом, которому она доступна, если не задана явно другая область действия. Если в дочерней области действия создается некоторый элемент, для которого существует одноименный элемент в родительской области действия, последний становится недоступным в дочерней области действия, но не изменяется и не переопределяется в своей области действия.

Область действия создается при запуске сценария или функции, определении блока сценария, запуске локального или удаленного сеанса либо нового экземпляра PowerShell. Новая область действия, созданная при запуске сценария, функции, блока сценария или вложенного экземпляра командной оболочки, становится вложенной (дочерней) по отношению к исходной (родительской) области действия. Напротив, при запуске нового сеанса его область действия становится глобальной и независимой от исходной области.

Области действия переменных задают явно либо неявно. В первом случае к имени переменной добавляют префикс Global, Local, Script или Private. Например, так объявляют глобальные переменные:

```
$Global:myVar = 55
```

Каждый раз, определяя переменную в командной строке, в сценарии или функции, вы неявно устанавливаете ее область действия. По умолчанию переменным назначаются следующие области действия:

- Global — переменным, интерактивно объявленным в командной строке PowerShell;
- Script — переменным, объявленным в сценарии вне функций и блоков;
- Local — переменным, объявленным в функциях, блоках и других элементах.

Глобальная область действия (global) охватывает экземпляр PowerShell. Глобальные элементы наследуются всеми дочерними областями действия.

Другими словами, глобальные переменные доступны любым командам, функциям и сценариям.

Глобальные области действия разных экземпляров PowerShell не перекрываются. Это означает, что переменная, объявленная в одном экземпляре консоли (или вкладке в окне графической среды) PowerShell, будет недоступной через другой экземпляр консоли (на другой вкладке графической среды), если не объявить ту же самую переменную в том же экземпляре консоли.

Каждый раз при вызове функции, сценария или фильтра создается локальная область действия. Элементы, объявленные с локальной областью действия, могут читать информацию из глобальной области действия, но изменять ее они могут, только явно указывая область действия. После завершения функции, сценария или фильтра вся информация из соответствующей локальной области действия отбрасывается.

Чтобы познакомимся с областями действия ближе, рассмотрим следующий пример:

```
function numservices {$serv = get-service}
numservices
write-host «The number of services is: `t» $serv.count
```

```
The number of services is:
```

Здесь переменная $serv объявлена внутри функции numservices, поэтому при вызове функции она создается как локальная переменная. В результате при ссылке на $serv из-за пределов функции оказывается, что значение этой переменной не установлено. Если же явно объявить эту переменную как глобальную, она станет доступной и вне функции, как показано в следующем примере:

```
function numservices {$Global:serv = get-service}
numservices
write-host «The number of services is: `t» $serv.count
```

```
The number of services is:        157
```

Область действия script создается при каждом запуске сценария. В этой области действия исполняются команды сценария, она также является локальной. Поскольку функции, вызываемые из сценария, исполняются в собственных локальных областях действия, объявленные в них переменные недоступны напрямую из сценария. Чтобы обойти это ограничение, необходимо явно устанавливать область действия переменных, объявляемых в функциях, например:

```
function numservices {$Script:serv1 = get-service}
numservices
write-host «The number of services is: `t» $serv1.count
```

Как правило, после завершения функции или сценария соответствующая область действия и связанная с ней информация отбрасывается. Однако можно заставить PowerShell не создавать новую локальную область действия для функции или сценария, а загрузить соответствующую информацию в родительскую область действия (область действия взывающего элемента). Для этого достаточно поставить точку (.) перед вызовом функции или сценария:

```
. c:\scripts\runtasks.ps1
```

Последний тип областей действия — закрытая (private). Элементы с такой областью действия необходимо объявлять явно. Информация в закрытой области действия доступна только элементам, объявленным в ней, и никаким другим (см. пример ниже).

```
function numservices {$Private:serv = get-service

write-host «The number of services is: `t» $serv.count

&{write-host «Again, the number of services is: `t» $serv.count}

}
numservices
```

```
The number of services is:       157
Again, the number of services is:
```

Здесь объявлена функция с закрытой переменной, а внутри — блок сценария. Закрытая переменная доступна внутри функции, поэтому ее вызов возвращает число служб. Для блока сценария автоматически создается отдельная локальная область действия, в которой закрытая переменная недоступна, поэтому второй раз определить число служб при вызове функций не удается.

Автоматические, управляющие переменные и переменные окружения

Наряду с пользовательскими переменными PowerShell поддерживает автоматические, управляющие переменные и переменные окружения. Автоматические переменные фиксированы и служат для хранения информации о состоянии. Управляющие переменные можно устанавливать, они используются для хранения рабочих параметров конфигурации PowerShell. Переменные окружения хранят параметры рабочей среды текущего пользователя и операционной системы.

В табл. 5-15 приводится список общих автоматических переменных PowerShell. Многие из них используются при работе с PowerShell, особенно $_, $Args, $Error, $Input и $MyInvocation.

Табл. 5-15. Общие автоматические переменные в PowerShell

Переменная	Что содержит
$$	Последний маркер последней команды, отданной в сеансе
$?	Состояние исполнения последней операции (TRUE, если она закончилась успешно, либо FALSE в противном случае)
$^	Первый маркер последней команды, отданной в сеансе
$_	Текущий объект, передаваемый по конвейеру; используется для манипуляций над объектами, передаваемыми по конвейеру
$Args	Массив необъявленных параметров и значений, передаваемый функции, сценарию или блоку сценария
$ConsoleFileName	Путь к последнему использованному в сценарии файлу консоли (.psc1-файлу)
$Error	Массив объектов, представляющих недавние ошибки, элемент $Error[0] представляет ошибку, возникшую последней
$ExecutionContext	Объект EngineIntrinsics, представляющий контекст исполнения хост-программы Windows PowerShell
$False	Значение FALSE, используется вместо строки “false”
$ForEach	Перечислитель цикла ForEach-Object
$Home	Полный путь к домашней папке пользователя
$Host	Объект, представляющий текущее хост-приложение
$Input	Ввод, переданный функции или блоку сценария. Переменная $Input чувствительна к регистру; при завершении блока Process она сбрасывается в NULL. Если у функции нет блока Process, значение $Input может быть прочитано блоком End
$LastExitCode	Код завершения последней запущенной Windows-программы
$Matches	Хэш-таблицу строк для поиска с помощью оператора –Match
$MyInvocation	Объект со сведениями о текущей команде, включая путь запуска и имя файла сценария
$NestedPromptLevel	Уровень текущей консоли. Значение 0 (ноль) означает исходную консоль; при открытии вложенной консоли это значение увеличивается, а при выходе из нее — уменьшается
$NULL	NULL, используется вместо строки «NULL»
$PID	Идентификатор (PID) хост-процесса текущего сеанса PowerShell

Табл. 5-15. Общие автоматические переменные в PowerShell

Переменная	Что содержит
$Profile	Полный путь к профилю PowerShell для текущего пользователя и хост-приложения
$PSBoundParameters	Хэш-таблицу активных параметров с их текущими значениями
$PsCulture	Имя текущих региональных параметров операционной системы
$PSDebugContext	Информацию о среде отладчика (если тот используется, и NULL, если отладчика нет)
$PsHome	Полный путь к установочному каталогу PowerShell
$PsUICulture	Имя текущей культуры (локализации) пользовательского интерфейсаоперационной системы
$PsVersionTable	Неизменяемую хэш-таблицу с элементами PSVersion (номер версии PowerShell), BuildVersion (номер сборки текущей версии), CLRVersion (версия CLR) и PSCompatibleVersions (версии PowerShell, совместимые с текущей)
$Pwd	Объект, представляющий полный путь к текущему рабочему каталогу
$ShellID	Идентификатор текущей оболочки
$This	Свойство или метод сценария внутри блока сценария; ссылка на расширяемый объект
$True	TRUE, используется вместо строки «true»

Переменная $_ используется вместе с командлетом Where-Object для выполнения операций над всеми или заданными объектами конвейера. Синтаксис ее использования следующий:

```
where-object {$_.Имя_свойства -Оператор_сравнения «Значение»}
```

где *Имя_свойства* — имя свойства объекта, *–Оператор_сравнения* — тип операции сравнения, а *Значение* — искомое значение, например:

```
get-process | where-object {$_.Name -match «svchost»}
```

Здесь переменная $_ используется для поиска объектов с заданным значением свойства Name. Если свойство Name объекта содержит строку «svchost», этот объект передается по конвейеру.

Можно дополнить этот пример регулярным выражением:

```
get-process | where-object {$_.Name -match «^s.*»}
```

Эта команда проверяет свойство Name у каждого объекта, передаваемого по конвейеру. Если значение свойства Name начинается с буквы S, объект передается по конвейеру. Аналогичным образом можно использовать любой оператор сравнения (см. табл. 5-4).

Другая полезная автоматическая переменная — $Error. С ее помощью можно получить список всех объектов, представляющих ошибки, возникшие в текущем сеансе, а элемент $Error[0] хранит объект, представляющий последнюю ошибку. Для просмотра этих сведений просто введите **$Error** или **$Error[0]**. Поскольку сведения об ошибках хранятся в виде объектов, их свойства можно форматировать как свойства любых других объектов, например так:

```
$error[0] | format-list -property * -force
```

Объекты ошибок поддерживают следующие свойства:

- **CategoryInfo** — содержит категорию ошибки, например InvalidArgument или PSArgumentException;
- **Exception** — содержит подробные сведения об ошибке;
- **FullyQualifiedErrorId** — содержит полное имя ошибки, например Argument;
- **InvocationInfo** — содержит подробности о команде, вызвавшей ошибку;
- **PipelineIterationInfo** — содержит подробности о работе конвейера;
- **PSMessageDetails** — содержит подробности о сообщениях PowerShell;
- **TargetObject** — указывает обрабатываемый объект.

Все эти свойства доступны, например, чтобы получить подробные сведения о команде, вызвавшей последнюю ошибку, вызовите свойству InvocationInfo:

```
$currError = $error[0]
$currError.InvocationInfo
```

```
MyCommand        : Sort-Object
BoundParameters  : {}
UnboundArguments : {}
ScriptLineNumber : 1
OffsetInLine     : 25
ScriptName       :
Line             : $p | sort-object -status
PositionMessage  :
                   At line:1 char:25
                   + $p | sort-object -status <<<<
InvocationName   : sort-object
PipelineLength   : 0
PipelinePosition : 0
ExpectingInput   : False
CommandOrigin    : Internal
```

Чтобы сбросить все ошибки в текущем сеансе, введите:

```
$error.clear()
```

Следующий представитель полезных автоматических переменных — $Args. Хранящийся в ней массив открывает доступ к аргументам командной строки:

```
$argCount = $args.Count
$firstArg = $args[0]
$secondArg = $args[1]
$lastArg = $args[$args.Count -1]
```

В этом примере с помощью $args.Count определяется число аргументов, переданных в командной строке. Первый аргумент извлекается из элемента $args[0], второй — из $args[1], а последний — из $args[$args.Count -1].

Данные, переданные функции или блоку сценария через конвейер, хранятся в переменной $Input, которая представляет собой перечислимое значение .NET Framework. Такой объект открывает доступ к входному потоку, но не позволяет обращаться к его элементам произвольным образом, как к элементам массива. Обработав один элемент входного потока, необходимо вызвать метод Reset() перечислимого $Input перед обработкой следующего элемента. Один из способов доступа к входному потоку предполагает использование цикла For Each:

```
foreach($element in $input) { «The input was: `t $element» }
```

либо загрузку элементов потока в массив:

```
$iArray = @($input)
```

Информация о контексте исполнения сценария доступна через переменную $MyInvocation. Для получения в сценариях подробных сведений о текущей команде используйте свойство $MyInvocation.MyCommand, а для получения имени и пути к файлу сценария — $MyInvocation.MyCommand.Path. Свойство $MyInvocation.MyCommand.Name содержит имя функции, а $MyInvocation.ScriptName — имя сценария.

В табл. 5-16 перечислены общие управляющие переменные. Эти переменные служат для настройки PowerShell, поэтому рекомендую вам изучить их; о некоторых них уже говорилось в главе 2.

Табл. 5-16. Общие управляющие переменные PowerShell

Переменная	Описание	Значение по умолчанию
$Confirm-Preference	Управляет запросом у пользователя подтверждения на выполнение командлетов. Принимает значения **High** (запрос подтверждений для рискованных операций), **Medium** (запрос подтверждений для операций, связанных с высоким и средним риском), **Low** (запрос подтверждений для операций с любым уровнем риска) и **None** (подтверждения не запрашиваются, если не используется параметр –Confirm)	High

(*см. след. стр.*)

Табл. 5-16. Общие управляющие переменные PowerShell

Переменная	Описание	Значение по умолчанию
$Debug-Preference	Управляет реакцией PowerShell на отладочные сообщения	Silently Continue
$ErrorAction Preference	Управляет реакцией PowerShell на ошибки, не прерывающие обработку командлетов	Continue
$ErrorView	Управляет форматом вывода сообщений Power-Shell об ошибках; принимает значения **Normal-View** (подробный вывод) и **CategoryView** (сокращенный вывод)	Normal View
$Format-Enumeration-Limit	Задает число элементов в группе для вывода; принимает целочисленные значения	4
$LogCommand HealthEvent	Управляет регистрацией ошибок при инициализации и исполнении команд в журнале событий PowerShell; принимает значения **$true** (ошибки регистрируются) и **$false** (не регистрируются)	$false
$LogCommand LifecycleEvent	Управляет регистрацией в журнале PowerShell запуска и остановки команд, конвейеров, а также исключений системы безопасности при обнаружении команд; принимает значения **$true** (регистрировать) и **$false** (не регистрировать)	$false
$LogEngine HealthEvent	Управляет регистрацией в журнале PowerShell ошибок сеансов; принимает значения **$true** (регистрировать) и **$false** (не регистрировать)	$true
$LogEngine Lifecycle-Event	Управляет регистрацией в журнале PowerShell запуска и завершения сеансов; принимает значения **$true** (регистрировать) и **$false** (не регистрировать)	$true
$LogProvider HealthEvent	Управляет регистрацией в журнале PowerShell ошибок поставщика (ошибок, чтения-записи, поиска и вызова); принимает значения **$true** (регистрировать) и **$false** (не регистрировать)	$true
$LogProvider LifecycleEvent	Управляет регистрацией в журнале PowerShell подключения и отключения поставщиков; принимает значения **$true** (регистрировать) и **$false** (не регистрировать)	$true
$Maximum-AliasCount	Задает максимальное число псевдонимов в сеансе PowerShell; допустимые значения — **1024–32768**. Для подсчета числа псевдонимов введите (get-alias).count	4096

Табл. 5-16. Общие управляющие переменные PowerShell

Переменная	Описание	Значение по умолчанию
$Maximum-DriveCount	Задает максимальное число дисков (включая диски файловой системы и «диски» поставщиков) в сеансе PowerShell; допустимые значения — **1024–32768**. Для подсчета числа дисков введите (get-psdrive).count	4096
$Maximum-ErrorCount	Задает максимальное число регистрируемых ошибок; объекты ошибок хранятся в автоматической переменной $Error; допустимые значения — **256–32768**. Для подсчета ошибок введите $Error.count	256
$Maximum-Function-Count	Задает максимальное число функций в сеансе; допустимые значения — **1024–32768**. Для подсчета числа функций введите (get-childitem function:).count	4096
$Maximum-History Count	Задает максимальное число команд в хронологии сеанса; допустимые значения — **1–32768**. Для подсчета числа команд введите (get-history) .count	64
$Maximum-Variable Count	Задает максимальное число переменных (пользовательских, автоматических и управляющих) в сеансе; допустимые значения — **1024–32768**. Для подсчета числа переменных введите (get-variable).count	4096
$OFS	Задает разделитель значений для преобразования массивов в строки; принимает любые строковые значения, например «+»	" "
$Output-Encoding	Задает метод кодировки символов при отправке текста из PowerShell другим программам; принимает значения **ASCIIEncoding**, **SBCSCode-PageEncoding**, **UTF7Encoding**, **UTF8Encoding**, **UTF32Encoding** и **UnicodeEncoding**	ASCII Encoding
$Progress-Preference	Управляет реакцией PowerShell на сообщения о ходе операции, генерируемые Write-Progress. Допустимые значения: **Stop** (вывести сообщение об ошибке и прервать исполнение), **Inquire** (запросить разрешение на продолжение), **Continue** (вывести индикатор хода и продолжить операцию) и **SilentlyContinue** (продолжить исполнение без вывода индикатора хода)	Continue
$PSMaximum-Received ObjectSizeMB	Задает предельный размер объекта, возвращаемого командой (только для команд, применяемых для удаленной работы)	10 Мб

(*см. след. стр.*)

Табл. 5-16. Общие управляющие переменные PowerShell

Переменная	Описание	Значение по умолчанию
$PSMaximum-Received DataSizePer-CommandMB	Задает предельный размер набора данных, возвращаемого командой (только для команд, применяемых для удаленной работы)	50 Мб
$PSSession-Application-Name	Задает приложение, применяемое по умолчанию при идентификации удаленного компьютера конечной точкой HTTP	WSMAN
$PSSession Configuration-Name	Задает конфигурацию по умолчанию для удаленных сеансов; принимает как значения имена из таблицы пользовательских удаленных сеансов WinRM, связанные с исполняемым файлом, который запускает сеанс, и URI ресурса сеанса	Microsoft. PowerShell
$Verbose-Preference	Управляет реакцией PowerShell на подробные сообщения	Silently Continue
$Warning-Preference	Управляет реакцией PowerShell на предупреждения	Continue
$WhatIf-Preference	Управляет автоматическим включением параметра –WhatIf у поддерживающих его командлетов. Если этот параметр включен, возвращается описание эффекта операции, которую должен выполнить командлет, но сама операция не выполняется. Допустимые значения: **0** (отключен) и **1** (включен)	0
$WsmanMax Redirection-Count	Задает максимальное число перенаправлений подключения на альтернативные URI при сбое; значение 0 запрещает перенаправление	5

Для просмотра значения автоматической или управляющей переменной достаточно ввести ее имя в командную строку PowerShell. Например, чтобы увидеть значение $pshome, введите $pshome в командной строке. В отличие от автоматических переменных, изменять значения управляющих переменных разрешено. Это делается так же, как с переменными, объявленными пользователем. Например, чтобы присвоить переменной $WarningPreference значение SilentlyContinue, введите:

```
$warningpreference = «silentlycontinue»
```

Для работы с переменными окружения используется особый поставщик. Для ссылки на переменные окружения этого перед их именами пишут имя диска $env:, например:

```
$env:logonserver
```

Можно также «перейти» на диск поставщика env: с помощью команды:

```
set-location env:
```

Приглашение PowerShell примет следующий вид:

```
PS Env:\>
```

После этого можно работать с переменными окружения напрямую. Например, чтобы вывести список переменных окружения, введите:

```
get-childitem
```

Чтобы вывести сведения о конкретной переменной или переменных, введите имя нужной переменной полностью либо используйте подстановочные знаки:

```
get-childitem userdomain
get-childitem user*
```

Закончив работу с поставщиком переменных окружения, можно вернуться на диск файловой системы путем ввода **set-location** и буквы нужного диска, например:

```
set-location c:
```

Другой способ работы с этим поставщиком основан на использовании командлета Get-Item, переходить на диск поставщика при этом не требуется. Например, следующая команда выводит полный список переменных окружения независимо от текущего диска:

```
get-item -path env:*
```

Получать переменные окружения можно, и ссылаясь на них по имени:

```
get-item -path env:username
```

либо с использованием подстановочных знаков:

```
get-item -path env:user*
```

Для установки переменных окружения использует командлет Set-Item в командах вида:

```
set-item -path env:Имя_переменной -value Новое_значение
```

где *Имя_переменной* — имя переменной окружения, значение которой нужно изменить, а *Новое_значение* — новое значение этой переменной, например:

```
set-item -path env:homedrive -value D:
```

Разрешается также использовать простое присваивание, например:

```
$env:homedrive = D:
```

Независимо от способа, изменения переменных окружения действуют только в текущем сеансе. Чтобы сохранить внесенные изменения, воспользуйтесь утилитой командной строки Setx.

Работа со строками

Хотя мы использовали строки в некоторых рассмотренных выше примерах, подробны разбор их свойств еще впереди. Строка — это последовательность букв, цифр и других символов. В PowerShell принят ряд правил синтаксического разбора строк, определяющих обработку строковых значений. Для успешной работы с командной строкой и сценариями PowerShell необходимо как следует знать эти правила.

Кавычки двойные и одинарные

Кавычками помечают начало и конец литеральных строковых выражений. Строки можно заключать в одинарные (‘ ’) либо двойные (“ ”) кавычки, но следует учитывать, что PowerShell разбирает строки с одинарными и двойными кавычками по-разному.

Строка, взятая в двойные кавычки, передается команде «как есть», без какой-либо подстановки. Рассмотрим следующий пример:

```
$varA = 200
Write-Host ‘The value of $varA is $varA.’
```

```
The output of this command is

The value $varA is $varA.
```

Аналогичным образом, строки в одинарных кавычках не обрабатываются, а интерпретируются как литералы. То есть, команда

```
‘The value of $(2+3) is 5.’
```

выводит

```
The value of $(2+3) is 5.
```

Если строка берется в двойные кавычки, перед именами переменных ставится знак доллара ($). Прежде, чем строка будет передана для обработки, в нее вместо переменных подставляются их реальные значения. Рассмотрим пример:

```
$varA = 200
Write-Host “The value of $varA is $varA.”
```

```
The value 200 is 200.
```

Чтобы запретить подстановку значения переменной в строке, взятой в двойные кавычки, следует предварить имя переменной обратным апострофом (`), который не помечает продолжающуюся строку, но и является спецсимволом. Рассмотрим пример:

```
$varA = 200
Write-Host “The value of `$varA is $varA.”
```

Здесь обратный апостроф, стоящий перед первым упоминанием переменной, запрещая PowerShell заменять ее значением. Вот что получается в результате:

```
The value $varA is 200.
```

Кроме того, при обработке строк в двойных кавычках вычисляются выражения и результаты подставляются в строку, как в следующем примере:

```
«The value of $(100+100) is 200.»
```

```
The value of 200 is 200.
```

Чтобы включить двойные кавычки в выводимую строку, необходимо взять всю строку вместе двойными кавычками в двойные или одинарные кавычки, например, так:

```
'He said, «Hello, Bob»'
```

или так:

```
“He said, ““Hello, Bob”””
```

Чтобы включить в выводимую строку одинарные кавычки, достаточно продублировать их:

```
'He won''t go to the store.'
```

Наконец, обратный апостроф (`) заставляет PowerShell интерпретировать любые кавычки как литералы:

```
«You use a double quotation mark (`») with expandable strings.»
```

Escape-символы и подстановочные знаки

В предыдущих примерах обратный апостроф (`) использовался как специальный или escape-символ, заставляющий PowerShell интерпретировать имена переменных, двойные и одинарные кавычки как литералы. Другие escape-символы служат для включения в строки специальных знаков и значений (табл. 5-17). Вот как, например, включить в строку знак табуляции:

```
$s = “Please specify the computer name `t []”
$c = read-host $s
write-host “You entered: `t $c”
```

```
Please specify the computer name        []: corpserver45
You entered:        corpserver45
```

Здесь создается строка со знаком табуляции, затем с помощью командлета Read-Host выводится приглашение к вводу. Введенная пользователем строка записывается в переменную и выводиться командлетом Write-Host после табулятора.

Табл. 5-17. Escape-коды в Windows PowerShell

Escape-код	Описание
`'	Одинарные кавычки
`"	Двойные кавычки
`0	Null-символ
`a	Звуковое оповещение
`b	Backspace
`f	Протяжка листа (при выводе на принтер)
`n	Новая строка
`r	Возврат каретки
`t	Горизонтальный табулятор (восемь знаков)
`v	Вертикальный табулятор (для вывода на принтер)

Часто при работе со строками требуется найти нужную подстроку. Для этого используются подстановочные знаки (см. табл. 5-5 выше) и следующие диапазоны:

- букв ([a–z], [A–Z]) и
- цифр [0-9].

В общем случае PowerShell не учитывает регистр при сравнении строк, поэтому, например, следующие диапазоны считаются идентичными:

- [a-c] и [A-C];
- [abc] и [ABC].

Однако в некоторых случаях при обработке регулярных выражений PowerShell различает заглавные и строчные буквы. Как сказано выше, для поиска строк по шаблону и передачи их командлетам удобны подстановочные знаки. Многие (но не все) командлеты поддерживают подстановочные знаки в строковых параметрах, проверить это можно, запросив справку с параметром –Full. Например, команда **get-help get-alias –full** выводит полную справку по командлету Get-Alias:

```
NAME
    Get-Alias

SYNOPSIS
    Gets the aliases for the current session.

SYNTAX
    Get-Alias [-Exclude <string[]>] [-Name <string[]>] [-Scope <string>]
[<CommonParameters>]

DETAILED DESCRIPTION
    The Get-Alias cmdlet gets the aliases (alternate names for commands
and executable files) in the current session.
```

```
PARAMETERS
    -Definition <string[]>
        Gets the aliases for the specified item. Enter the name of a cm-
dlet, function, script, file, or executable file.

        Required?                    false
        Position?                    named
        Default value
        Accept pipeline input?       false
        Accept wildcard characters?  true
```

Видно, что параметр –Definition принимает значения с подстановочными знаками (Accept WildCard Characters = True, в противном случае мы бы увидели здесь False).

Многострочные string-значения

Иногда требуется разбить string-переменную на несколько фрагментов, выводимых на отдельных строках. Для этого следует вставить в начале и конце каждого фрагмента знаки @. В результате получаются т.н. *однострочные фрагменты* (*here-string*s), в которых одинарные и двойные кавычки обрабатываются согласно общим правилам. Рассмотрим следующий пример:

```
$myString = @»
=========================
$env:computername
=========================
«@
write-host $myString
```

```
=========================
EngPC85
=========================
```

Строки данных (data strings) также относятся к многострочным string-значениям. Они используются в PowerShell, главным образом, для локализации сценариев: они отделяют код от текста, подлежащего переводу на другие языки. Впрочем, ничто не мешает использовать и для других целей, поскольку их они обладают куда более гибким функционалом по сравнению с другими типами строк. Основная причина в том, что строки данных включают в код в виде переменных.

Вот базовый синтаксис строк данных:

```
DATA Имя {
   Текст
}
```

где *Имя* — имя переменной, представляющей строку данных, а *Текст* может включать одинарные и двойные кавычки, а также однострочные фрагменты

в любых комбинациях. Учтите, что ключевое слово DATA нечувствительно к регистру, то есть слова DATA, data и Data интерпретируются как начало строки данных.

Вот пример объявления строки данных MyValues:

```
DATA MyValues {
  «This is a data string.»
  «You can use data strings in many different ways.»
}
```

Для вставки строки MyValues на нее ссылаются по имени, например, так:

```
Write-Host $MyValues
```

или просто

```
$MyValues
```

Расширенный синтаксис строк данных имеет следующий вид:

```
DATA Имя [-supportedCommand Командлет] {
Допустимое_содержимое
}
```

где *Командлет* — список командлетов, разделенный запятыми, а *Допустимое_содержимое* может включать:

- строки и строковые литералы;
- операторы PowerShell кроме –match;
- конструкции If, Else и ElseIf;
- некоторые автоматические переменные, включая $PsCulture, $PsUICulture, $True, $False и $Null;
- операторы, разделенные точкой с запятой, комментариями и знаками конвейера.

Строки данных, к которым добавлены командлеты и операторы, очень похожи на функции, в этих случаях строку данных удобнее заменить настоящей функцией (подробнее о функциях — в главе 6).

Строки, содержащие переменные и вложенные выражения, необходимо брать в одинарные кавычки либо помечать как однострочные фрагменты, чтобы запретить подстановку значений переменных и выражений.

Командлет ConvertFrom-StringData преобразует строки с парами «имя=значение» в ассоциативные массивы. Поскольку каждая такая пара должна занимать отдельную строку, входную строку для этого командлета обычно размечают на фрагменты, при этом разрешено использовать знаки @, одинарные и двойные кавычки. В примере ниже показано, как работает ConvertFrom-StringData.

```
DATA DisplayNotes {
  ConvertFrom-StringData -stringdata @'
```

```
  Note1 = This appears to be the wrong syntax.
  Note2 = There is a value missing.
  Note3 = Cannot connect at this time.
  '@
}
```

Здесь объявляется строка данных DisplayNotes, в которую включен командлет ConvertFrom-StringData, создающий ассоциативный массив, содержащий строки Note1, Note2 и Note3. Для ссылки на эти строки используется синтаксис с точками: первый элемент массива представляет строка $DisplayNotes.Note1; второй элемент —$DisplayNotes.Note2, а третий — $DisplayNotes.Note3. Чтобы вывести все элементы, просто вводит **$DisplayNotes**.

То же самое можно сделать без использования строк данных, например:

```
$string = @'
  Note1 = This appears to be the wrong syntax.
  Note2 = There is a value missing.
  Note3 = Cannot connect at this time.
  '@

$strArray = $string | convertfrom-stringdata
```

Как и в предыдущем примере, здесь определен фрагмент, который через конвейер (|) отправляется командлету ConvertFrom-StringData, а тот записывает в переменную $strArray массив, с элементами которого работают, как показано выше.

Операторы для работы со строками

Как сказано в разделе, посвященном операторам сравнения, операторы –eq, –ne, –lt, –gt, –le, –ge, –like, –notlike, –match, –notmatch, –contains, –notcontains и –replace применяют для работы со строками и массивами.

Наряду с ними для обработки строк используют следующие операторы:

- = — присваивает переменной строковое значение;
- + — служит для конкатенации строк;
- * — повторяет строку заданное число раз;
- –Join — объединяет строки с разделителями либо без них;
- –Split — разбивает строку по пробелами или заданным разделителям;
- –f — форматирует строку по заданному шаблону.

Чаще всего в работе со строками выполняется присваивание и конкатенация. Для присваивания, как и в других случаях, используется знак равенства:

```
$myString = «This is a String.»
```

Конкатенацией или сцеплением называется объединение строк. Обычно оно выполняется оператором +:

```
$streetAdd = «123 Main St.»
$cityState = «Anywhere, NY»
$zipCode = «12345»
$custAddress = $streetAdd + « « + $cityState + « « + $zipCode
$custAddress
```

```
123 Main St. Anywhere, NY 12345
```

Иногда требуется вывести строку со значением переменной. Для этого достаточно добавить имя переменной к строке и взять результат в двойные кавычки:

```
$company = «XYZ Company»
Write-Host «The company is: $company»
```

```
The company is: XYZ Company
```

Аналогично, для конкатенации массивов используется оператор +, например:

```
$array1 = «PC85», «PC25», «PC92»
$array2 = «SERVER41», «SERVER32», «SERVER87»
$joinedArray = $array1 + $array2
$joinedArray
```

```
PC85
PC25
PC92
SERVER41
SERVER32
SERVER87
```

Функция «умножения» строк и массивов удобна, когда требуется многократно повторить символ в строке или элемент в массиве. Например, можно без труда сделать из строки с единственным символом + строку из 80 таких символов, например, для разделения фрагментов вывода:

```
$separator = «+»
$sepLine = $separator * 60
$sepLine
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Операторы –Join и –Split объединяют и разбивают строки, соответственно. Общий синтаксис Join выглядит так:

```
-Join (Строка1, Строка2, Строка3 …)
```

```
Строка1, Строка2, Строка3 … -Join «Разделитель»
```

Первая команда просто объединяет несколько строк в одну, а вторая вставляет между строками разделители. Вот пример первой команды:

```
$a =  -join («abc», «def», «ghi»)
$a
```

```
abcdefghi
```

А это — пример второй команды:

```
$a = «abc», «def», «ghi» -join «:»
$a
```

```
abc:def:ghi
```

Общий синтаксис команды для разбиения строк имеет следующий вид:

```
-Split Строка

Строка -Split «Разделитель» [,MaxSubStrings]
```

Первая команда разбивает строку, используя в качестве разделителей пробелы:

```
$a = «abc def ghi»
-Split $a
```

```
abc
def
ghi
```

Можно задавать и собственные разделители:

```
$a = «jkl:mno:pqr»
$a -Split «:»
```

```
jkl
mno
pqr
```

Оператор –f форматирует строки по заданному шаблону. Синтаксис шаблонов позволяет задавать особое форматирование для букв, цифр и их комбинаций. Обычно параметры форматирования стоят до оператора –f, а за ним идет список форматируемых значений, разделенный запятыми:

```
'Инструкции' -f «Значение1», «Значение2», «Значение3», …
```

Здесь ссылка {0} представляет первое форматируемое значение, {1} — второе и т.д. Меняя очередность ссылок, можно изменять параметр значений и их формат. Так, например, можно вывести значения в обратном порядке:

```
'{2} {1} {0}' -f «Monday», «Tuesday», «Wednesday»
```

```
Wednesday Tuesday Monday
```

Порядок вывода может быть не только обратным, но и произвольным:

```
'{2} {0} {1}' -f «Cloudy», «Sunny», «Rainy»
```

```
Rainy Cloudy Sunny
```

Число инструкций должно совпадать с числом значений, иначе «лишние» значения игнорируются, что, впрочем, можно использовать для своих целей. В следующем примере преднамеренно пропущено второе значение:

```
'{0} {2}' -f «Server15», «Server16», «Server17»
```

```
Server15 Server17
```

К инструкциям можно добавлять спецификаторы формата (табл. 5-18).

Табл. 5-18. Спецификаторы формата

Спецификатор	Что и во что преобразует	Пример	Результат
:c или :C	Число в денежную сумму (в формате, заданном настройками Windows)	‘{0:c}’ –f 145.50	$145.50
:e или :E	Число в научный формат; числовой параметр задает точность (число разрядов после запятой)	‘{0:e4}’ –f [Math]::Pi	3.1416e+000
:f или :F	Число в формат с фиксированной точкой; числовой параметр задает точность (число разрядов после запятой)	‘{0:f4}’ –f [Math]::Pi	3.1426
:g или :G	Число в самый компактный формат (научный либо фиксированной точкой). Числовой параметр задает число значимых разрядов	‘{0:g3}’ –f [Math]::Pi	3.14
:n или :N	Число в формат с разделителями, заданными региональными параметрами; числовой параметр задает точность (число разрядов после запятой)	‘{0:n2}’ –f 1GB	1,073,741,824.00
:p или :P	Число в процентный формат; числовой параметр задает точность (число разрядов после запятой)	‘{0:p2}’ –f .112	11.20 %
:r или :R	Число с сохранением точности, при обратном преобразовании гарантирует получение исходного числа	‘{0:r}’ –f (1GB/2.0)	536870912Note: (536870912 * 2) = 1,073,741,824

Табл. 5-18. Спецификаторы формата

Спецификатор	Что и во что преобразует	Пример	Результат
:x или :X	Число в шестнадцатеричный вид	'{0:x}' –f 12345678	bc614e
{*N*:hh} : {*N*:mm} : {*N*:ss}	Объект DateTime в формат ЧЧ:ММ:СС, вывод часов, минут и (или) секунд можно отключать	'{0:hh:0:mm}' –f (get-date)	12:35
{*N*:ddd}	Объект DateTime в день недели; разрешается комбинировать с предыдущим спецификатором	'{0:ddd} {0:hh:0:mm}' –f (get-date)	Mon 12:57

Работа с массивами и наборами

В Windows PowerShell термины *массив* (*array*) и *набор* (*collection*) синонимичны. Массив — это структура данных для хранения набора значений. Массивы позволяют группировать логически связанные значения, при этом PowerShell не требует, чтобы у них был одинаковый тип. Например, можно помещать в один и тот же массив числа, строк и объекты.

Наиболее распространены одномерные массивы. Одномерный массив напоминает столбец из таблицы, двухмерный — таблицу из нескольких строк и столбцов, а трехмерный — куб, разбитый на ячейки.

Элементы массива проиндексированы, на них можно ссылаться по их индексу. Для доступа к элементу одномерного массива достаточно указать одно число в качестве индекса, индекс элементов многомерных массивов включает несколько чисел.

Примечание PowerShell также поддерживает ассоциативные массивы, в которых хранятся пары «имя–значение» (подробнее об этом — выше в этой главе).

Создание одномерных массивов и работа с ними

PowerShell предоставляет несколько операторов для работы с массивами, из которых чаще всего используется запятая. Запятая разделяет элементы одномерного массива. Вот общий синтаксис такого массива:

```
$Имя = Элемент1, Элемент2, Элемент3, …
```

где $*Имя* — переменная, в которой хранится массив; *Элемент1* — первый элемент массива, *Element2* — второй и т.д.

Вот пример создания массива чисел:

```
$myArray = 2, 4, 6, 8, 10, 12, 14
```

Объявление массива строк немногим сложнее:

```
$myStringArray = «This», «That», «Why», «When», «How», «Where»
```

Для доступа к элементам массива следует передать оператору [ ] индекс нужного элемента (нумерация элементов начинается с нуля, то есть индекс первого элемента — 0, второго — 1 и т.д.). Вот как выглядит общий синтаксис доступа к элементу:

```
$Имя[Индекс]
```

где $*Имя* — переменная, в которой хранится массив, а *Индекс* — индекс нужного элемента. Соответственно, ссылка на первый элемент массива $myStringArray примет следующий вид:

```
$myStringArray[0]
```

```
This
```

Есть несколько способов индексации. Так, индекс –1 представляет последний элемент массива, –2 — предпоследний и т.д. Так выглядит ссылка на последний элемент массива:

```
$myStringArray[-1]
```

```
Where
```

Для доступа к диапазону элементов массива служит оператор (..). Так, следующая команда возвращает элементы 1–3:

```
$myStringArray[0..2]
```

```
This
That
Why
```

Эти приемы можно по-разному комбинировать. Например, следующая команда возвращает последний, первый и второй элементы массива:

```
$myStringArray[-1..1]
```

```
Where
This
That
```

А эта команда перебирает массив в направлении от последнего к первому:

```
$myStringArray[-1..-3]
```

```
Where
How
When
```

В команде, возвращающей диапазон элементов вместе с избранными отдельными элементами, индексы отдельных элементов и ссылку на диапазон

необходимо объединить оператором +. Следующий пример возвращает элементы 1, 2 и 4–6:

```
$myStringArray[0,1+3..5]
```

```
This
That
When
How
Where
```

Свойство Length задает число элементов массива. Так, команда **$myArray.Length** объявляет массив из семи элементов. Это свойство можно использовать и для доступа к массиву. Следующая команда возвращает элементы 5 и 6:

```
$myStringArray[4..($myArray.Length-1)]
```

```
How
Where
```

Cast-объявление массивов

Массивы также можно создавать с помощью т.н. cast-структуры. Вот ее общий синтаксис:

```
$Имя = @(Элемент1, Элемент2, Элемент3, …)
```

где $*Имя* — имя переменной, в которой хранится массив; *Элемент1* — первый элемент массива, *Element2* — второй элемент и т.д.

Так, например, объявляется массив чисел:

```
$myArray = @(3, 6, 9, 12, 15, 18, 21)
```

Преимущество cast-массивов в том, что, заменив разделители-запятые точкой с запятой, вы заставите PowerShell интерпретировать каждый элемент массива как текст команды. Таким образом, PowerShell исполнит эти команды, как если бы они вводились в командной строке, и сохранит результат. Рассмотрим следующий пример:

```
$myArray = @(14; «This»; get-process)
```

Как показано ниже, первый элемент массива имеет целочисленный тип:

```
$myArray[0].gettype()
```

```
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Int32                                    System.ValueType
```

Второй — строковый тип:

```
$myArray[1].gettype()
```

```
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object
```

Третий же является набором объектов Process:
$myArray[2].gettype()

```
IsPublic IsSerial Name                 BaseType
-------- -------- ----                 --------
True     False    Process              System.ComponentModel.Component
```

Работа с элементами массива

Для установки значений элементов массива используется простое присваивание. Например, если объявлен массив:

```
$myArray = @(3, 6, 9, 12, 15, 18, 21)
```

можно изменить значение его второго элемента:

```
$myArray[1] = 27
```

То же самое можно сделать с помощью метода SetValue(), его синтаксис имеет следующий вид:

```
$Имя.SetValue(Новое_значение,Индекс)
```

где $*Имя* — имя переменной, в которой хранится массив; *Новое_значение* — присваиваемое значение; *Индекс* — индекс изменяемого элемента. Так можно изменить значение первого элемента массива, объявленного выше:

```
$myArray.SetValue(52,0)
```

Для добавления элементов к существующему массиву с одновременной установкой их значений служит оператор +=. Например, добавить к массиву $myArray элемент со значением 75, позволит следующая команда:

```
$myArray += 75
```

Удалять элементы из массивов PowerShell не так просто, зато можно создать новый массив, собрав в нем лишь нужные элементы прежнего массива. Например, чтобы создать массив $myNewArray, в котором будут все элементы массива $myArray кроме третьего, введите

```
$myNewArray = $myArray[0..2+4..($myArray.length - 1)]
```

Можно также объединить несколько массивов в один с помощью оператора + следующим образом:

```
$array1 = 1,2,3,4
$array2 = 5,6,7,8
$array3 = 9,10,11,12
$cArray = $array1 + $array2 + $array3
```

Большие массивы занимают много памяти, поэтому следует удалять ненужные массивы. Чтобы удалить массив, вызовите командлет Remove-Item с переменной, в которой хранится массив. Например, так можно удалить массив $myArray:

```
remove-item variable:myArray
```

Строго типизированные массивы

Иногда требуются массивы, способные хранить только объекты заданного типа. Чтобы получить такой массив, необходимо добавить объявление типа к объявлению массива с использованием оператора [], например:

- [int32[]]$myArray — объявляет массив целочисленных значений;
- [bool[]]$myArray — объявляет массив булевых значений;
- [object[]]$myArray — объявляет массив объектов;
- [string[]]$myArray — объявляет массив строковых значений.

Следующая команда создает массив целочисленных значений и устанавливает их:

```
[int32[]]$myArray = 5,10,15,20,25,30,35
```

Полученный массив строго типизирован, поэтому при работе с ним разрешено использовать только тот тип значений, с которым он объявлен. Например, при попытке установить для элемента этого массива строковое значение вы получите следующую ошибку:

```
$myArray[0]= «Kansas»
```

```
Array assignment to [0] failed: Cannot convert value «Kansas» to type
«System.Int32». Error: «Input string was not in a correct format.».
At line:1 char:10
+ $myArray[ <<<< 0]= «Kansas»
  + CategoryInfo : InvalidOperation: (Kansas:String) [],
RuntimeException
  + FullyQualifiedErrorId : ArrayAssignmentFailed
```

Работа с многомерными массивами

В многомерных массивах используются многомерные индексы. Так, для двухмерного массива, похожего на таблицу, индекс элемента включает два числа — номера строки и столбца. Такой индекс можно представить как координаты точки на плоскости.

Рассмотрим следующий пример.

Индекс	Столбец 0	Столбец 1	Столбец 2
СТРОКА 0	Red	Green	Blue
СТРОКА 1	Washington	Ohio	Florida
СТРОКА 2	Ocean	Lake	Stream
СТРОКА 3	Sky	Clouds	Rain

Это таблица значений, которые требуется сохранить в массиве. Согласно принятым правилам, первым идет номер строки, за ним — номер столбца. Ячейка с номером (индексом) 0, 0 содержит значение Red, ячейка 0, 1 — Green и т.д.

Для объявления одномерных массивов достаточно простых конструкторов, тогда как с многомерными массивами PowerShell обращается как с объектами. То есть, массив необходимо сначала объявить (создать), и только после этого его можно заполнять. Вот синтаксис для создания двухмерных массивов:

```
$Имя = new-object 'object[,]' Число_строк,Число_столбцов
```

где $*Имя* — имя переменной, в которой хранится массив; *object[,]* — спецификатор двухмерного массива; *Число_строк* — число строк и *Число_столбцов* — число столбцов в массиве.

Следующая команда объявляет массив их 4 строк и 3 столбцов:

```
$myArray = new-object 'object[,]' 4,3
```

Созданный массив можно заполнить, ссылаясь на ячейки массива по индексу:

```
$myArray[0,0] = «Red»
$myArray[0,1] = «Green»
$myArray[0,2] = «Blue»
$myArray[1,0] = «Washington»
$myArray[1,1] = «Ohio»
$myArray[1,2] = «Florida»
$myArray[2,0] = «Ocean»
$myArray[2,1] = «Lake»
$myArray[2,2] = «Stream»
$myArray[3,0] = «Sky»
$myArray[3,1] = «Clouds»
$myArray[3,2] = «Rain»
```

После заполнения ячейки массива доступны для чтения:

```
$myArray[0,1]
```

```
Green
```

Создание массивов с тремя и более измерениями мало отличается от создания двухмерных массивов, единственное различие — объявление дополнительного измерения. Вот синтаксис для создания трехмерного массива:

```
$Имя = new-object 'object[,,]' Число_X, Число_Y, Число_Z
```

где $*Имя* — переменная, в которой хранится массив; *object[,,]* — спецификатор трехмерного массива, а значения *ЧислоX*, *ЧислоY* и *ЧислоZ* задают число элементов в измерениях X, Y и Z.

Вот пример объявления массива размером 5 × 5 × 3 элементов:

```
$a = new-object 'object[,,]' 5,5,3
```

Такой массив напоминает 3 сложенные стопкой таблицы размером 5 × 5 ячеек каждая. Теперь можно заполнить, например, ячейку с координатами 0, 0, 0:

```
$a[0,0,0] = «Texas»
```

Во время работы над этой книгой PowerShell поддерживала до 17 измерений в массивах. Чтобы создать 17-мерный массив, следует добавить в скобки конструктора *object[]* 17 запятых, а затем определить размеры 17 измерений, например, так:

```
$myHugeArray = new-object 'object[,,,,,,,,,,,,,,,,]'
5,5,5,5,3,3,3,3,3,3,3,3,3,3,3,4
```

Закончив работу с таким огромным массивом, не забудьте удалить его, чтобы освободить занятую им память.

Глава 6

Псевдонимы, функции и объекты

Поимо основных структур, о которых рассказывалось в предыдущей главе, PowerShell поддерживает другие элементы, необходимые для работы с командной оболочкой. К ним относятся:

- псевдонимы;
- функции;
- объекты.

Эти элементы позволяют автоматизировать рутинные операции, и совершенно необходимы для решения любых административных задач в PowerShell. О них пойдет речь в этой главе, но для понимания данной главы необходимо прежде ознакомиться с материалом главы 6. Я также не планирую возвращаться к обсуждаемым здесь азам в следующих главах, посвященным практическому использованию функций, псевдонимов и объектов. Так, в разделах, посвященных работе с объектами WMI, я не буду объяснять, как с помощью командлета Get-WmiObject получить объект Win32_Processor, а сосредоточусь на новом материале, например, определении тактовой частоты процессора или инвентаризации компьютеров, работающих под управлением определенной версии Windows.

Создание псевдонимов и работа с ними

Псевдонимы — это альтернативные имена команд, функций, программ, сценариев и других элементов команд PowerShell. В PowerShell по умолчанию определено изрядное число псевдонимов, пользователю также разрешено определять собственные псевдонимы. Использование псевдонимов позволяет сократить объем ввода с клавиатуры; у одной команды может быть несколько псевдонимов. Например, ls — псевдоним командлета Get-ChildItem. В UNIX-системах команда ls используется для вывода списка содержимого каталога, и вывод Get-ChildItem больше напоминает вывод UNIX-команды ls, чем вывод команды dir в Windows.

Работа со встроенными псевдонимами

В табл. 6-1 перечислены наиболее востребованные из псевдонимов, объявленных по умолчанию.

Табл. 6-1. Наиболее востребованные псевдонимы

Имя	Связанный командлет
clear, cls	Clear-Host
diff	Compare-Object
cp, copy	Copy-Item
epal	Export-Alias
epcsv	Export-Csv
foreach	ForEach-Object
fl	Format-List
ft	Format-Table
fw	Format-Wide
gal	Get-Alias
ls, dir	Get-ChildItem
gcm	Get-Command
cat, type	Get-Content
h, history	Get-History
gl, pwd	Get-Location
gps, ps	Get-Process
gsv	Get-Service
gv	Get-Variable
group	Group-Object
ipal	Import-Alias
ipcsv	Import-Csv
r	Invoke-History
ni	New-Item
mount	New-MshDrive
nv	New-Variable
rd, rm, rmdir, del, erase	Remove-Item
rv	Remove-Variable
sal	Set-Alias
sl, cd, chdir	Set-Location
sv, set	Set-Variable

(*см. след. стр.*)

Табл. 6-1. Наиболее востребованные псевдонимы

Имя	Связанный командлет
sort	Sort-Object
sasv	Start-Service
sleep	Start-Sleep
spps, kill	Stop-Process
spsv	Stop-Service
write, echo	Write-Output

В табл. 6-2 содержится список команд, встроенных в командную строку Windows (cmd.exe), для которых нет одноименных исполняемых файлов. Поскольку командлеты работают иначе, чем встроенные в cmd.exe команды, необходимо точно знать, что именно вы вызываете. С этой целью в таблице указаны псевдонимы, использование которых исключает ошибочный вызов встроенной команды cmd.exe. Встроенные команды все же можно вызывать из командной строки и сценариев PowerShell. Для этого необходимо вызвать сначала cmd.exe с соответствующими параметрами, а затем — нужную команду. Как правило, используется параметр /c, который закрывает командную оболочку после исполнения заданной команды. Вот пример:

```
cmd /c dir
```

Табл. 6-2. Встроенные команды командной оболочки Windows (cmd.exe)

Имя	Описание	Чем переопределяется
assoc	Выводит или изменяет текущее файловое сопоставление	
break	Устанавливает точки прерывания для отладки	
call	Вызывает из сценария процедуру или другой сценарий	
cd (chdir)	Выводит имя текущего каталога или переходит в другой каталог	Set-Location
cls	Очищает окно cmd.exe и экранный буфер	Clear-Host
color	Задает цвета текста и фона в окне cmd.exe	
copy	Копирует или объединяет файлы	Copy-Item
date	Выводит или устанавливает системное время	
del (erase)	Удаляет заданные файлы и каталоги	Remove-Item
dir	Выводит список файлов и каталогов, расположенных в текущем каталоге	Get-ChildItem

Табл. 6-2. Встроенные команды командной оболочки Windows (cmd.exe)

Имя	Описание	Чем переопределяется
dpath	Позволяет программам открывать файлы из заданного каталога, как если бы они располагались в текущем каталоге	
echo	Отображает строки на экране и управляет эхо-выводом команд (on | off)	Write-Output
endlocal	Завершает локализацию переменных	
exit	Завершает работу командной оболочки	
for	Запускает заданную команду для каждого из файла из набора файлов	
ftype	Выводит текущий тип файлов или изменяет типы файлов, используемые в сопоставлениях	
goto	Переход интерпретатора команд к строке с заданной меткой в сценарии	
if	Условный вызов команд	
md (mkdir)	Создает каталог в текущей папке	*md вызывает команду mkdir через cmd.exe
mklink	Создает символическую или жесткую связь для файла или каталога	
move	Используется для перемещения и переименования файлов и каталогов	Move-Item
path	Выводит или устанавливает путь к командам, который ОС использует для поиска исполняемых файлов и сценариев	
pause	Приостанавливает обработку пакетного файла в ожидании ввода с клавиатуры	
popd	Переход в каталог, созданный PUSHD	Pop-Location
prompt	Задает текст приглашения к вводу команд	
pushd	Сохраняет текущий каталог и (если указано) переходит в заданный каталог	Push-Location
rd (rmdir)	Удаляет каталог вместе с содержимым	Remove-Item
rem	Помечает комментарий в пакетных файлах и Config.sys	
ren (rename)	Переименовывает файлы и каталоги	Rename-Item (только для ren)

(*см. след. стр.*)

Табл. 6-2. Встроенные команды командной оболочки Windows (cmd.exe)

Имя	Описание	Чем переопределяется
set	Выводит текущие переменные окружения, устанавливает временные переменные для текущего сеанса командной оболочки	Set-Variable
setlocal	Отмечает начало локализации переменной в пакетном файле	
shift	Сдвигает подставляемые параметры пакетного файла	
start	Запускает заданную программу или команду в отдельном окне	Start-Process
time	Выводит или устанавливает системное время	
title	Задает текст заголовка окна cmd.exe	
type	Выводит содержимое текстового файла	Get-Content
verify	Заставляет ОС проверять запись файлы после их записи на диск	
vol	Выводит метку и серийный номер дискового тома	

Создание псевдонимов

Для работы с псевдонимами служат следующие псевдонимы:

- **Get-Alias** — выводит список всех или указанных псевдонимов, установленных текущем сеансе, упорядоченный по имени и определению.

```
Get-Alias [[-Name | -Definition] Строки] [AddtlParams]

AddtlParams=
[-Exclude Строки] [-Scope Строка]
```

- **New-Alias** — создает псевдоним.

```
New-Alias [-Description Строка] [-Name] Строка [-Value] Строка
[AddtlParams]

AddtlParams=
[-Force] [-PassThru] [-Scope Строка] [-Option None | ReadOnly |
Constant | Private | AllScope]
```

- **Set-Alias** — создает новый псевдоним или изменяет определение существующего.

```
Set-Alias [-Description Строка] [-Name] Строка [-Value] Строка
[AddtlParams]
```

```
AddtlParams=
[-Force] [-PassThru] [-Scope Строка] [-Option None | ReadOnly |
Constant | Private | AllScope]
```

- **Export-Alias** — экспортирует все псевдонимы, которые используются в данный момент консолью PowerShell (включая встроенные и пользовательские), в файл псевдонимов.

```
Export-Alias [-Append] [-As Csv | Script] [-Path] Строка
[AddtlParams]

AddtlParams=
[-Description Строка] [-Force] [-NoClobber] [-PassThru]
[-Scope Строка] [[-Name] Строки]
```

- **Import-Alias** — импортирует файл псевдонимов в консоль PowerShell, импортированные псевдонимы становятся доступными в сеансах PowerShell. Нужные псевдонимы необходимо импортировать при каждом запуске консоли PowerShell.

```
Import-Alias [-Path] Строка [AddtlParams]

AddtlParams=
[-PassThru] [-Force] [-Scope Строка]
```

Как сказано выше, командлет Get-Alias позволяет вывести все доступные псевдонимы. Чтобы получить определенный псевдоним, используйте параметр –Name. например, так можно вывести все псевдонимы, начинающиеся на «a»:

```
get-alias -name a*
```

Параметр –Definition позволяет вывести псевдонимы для определенной команды. Так, следующая команда выводит псевдонимы Remove-Item:

```
get-alias -definition Remove-Item
```

Для создания псевдонимов служат командлеты New-Alias и Set-Alias. Главное различие между ними в том, что New-Alias создает псевдоним, только если он еще не существует, тогда как Set-Alias перезаписывает определения существующих псевдонимов. Это его общий синтаксис:

```
set-alias -name Имя -value Команда
```

где *Имя* — нужный псевдоним, а *Команда* — командлет, для которого создается псевдоним. Следующая команда создает псевдоним «cm» для оснастки Computer Management:

```
set-alias -name cm -value c:\windows\system32\compmgmt.msc
```

Параметры –Name и –Value интерпретируются по положению в списке параметров, поэтому достаточно указать их значения в нужном порядке, не указывая имен параметров:

```
set-alias cm c:\windows\system32\compmgmt.msc
```

Примечание Иногда требуется дополнительная настройка псевдонима с помощью параметра –Option, принимающего значения None, ReadOnly, Constant, Private и AllScope. По умолчанию задано значение None (нет настройки). Значение ReadOnly разрешает изменять псевдоним только при использовании параметра –Force, значение Constant полностью запрещает изменение псевдонима. Значение Private делает псевдоним доступным только в области действия, заданной параметром –Scope, а значение AllScope — в любых областях.

Псевдонимы используют как команды, для которых они созданы. Чтобы всегда вызвать некоторую команду с определенными параметрами, внешними программами и утилитами, следует включить их в псевдоним этой команды, взяв их в двойные кавычки, например, так:

```
set-alias cm «c:\windows\system32\compmgmt.msc /computer=engpc57»
```

Однако этот способ определения параметров не подходит для командлетов. Так, можно создать псевдоним *gs* для командлета Get-Service, но нельзя создать псевдоним с таким определением: get-service –name winrm. Обойти это ограничение позволяют функции, о которых рассказывается ниже в этой главе.

Импорт и экспорт псевдонимов

Как правило, псевдонимы, которые требуются в каждом сеансе, добавляют в профиль. Чтобы сохранить псевдонимы, созданные в сеансе PowerShell, воспользуйтесь командлетом Export-Alias. Этот командлет экспортирует все используемые в данный момент псевдонимы в файл. В дальнейшем этот файл, который представляет собой список значений, разделенных запятыми, можно импортировать в сценарии PowerShell с помощью командлета Set-Alias.

Общий синтаксис Export-Alias имеет следующий вид:

```
export-alias -path Файл_псевдонимов
```

где *Файл_псевдонимов* — имя и полный путь к файлу псевдонимов, например:

```
export-alias -path myaliases.csv
```

По умолчанию псевдонимы экспортируются в виде списка, разделенного запятыми, параметр –As включает экспорт в виде сценария, как в этом примере:

```
export-alias -as script -path myaliases.ps1
```

Поддерживаются и другие параметры:

- **–Append** — дописывает псевдонимы к концу существующего файла псевдонимов (по умолчанию новый экспорт перезаписывает содержимое существующего файла);

- **–Force** — разрешает перезаписывать даже файлы псевдонимов с атрибутом read-only;
- **–Noclobber** — запрещает автоматически перезаписывать файлы псевдонимов.

Командлет Import-Alias импортирует файл псевдонимов в консоль PowerShell. Вот его общий синтаксис:

```
import-alias -path Файл_псевдонимов
```

где *Файл_псевдонимов* — имя и полный путь к импортируемому файлу псевдонимов, например:

```
import-alias -path c:\powershell\myaliases.csv
```

Параметр –Force разрешает повторный импорт ранее импортированных псевдонимов с атрибутом read-only.

Создание функций и работа с ними

Функция PowerShell — это именованный набор команд, принимающий ввод через конвейер. При вызове функции по имени составляющие ее команды исполняются, как если бы они вводились в командной строке. Обычно функции, которыми пользуются часто, сохраняют в профиле. Функции также можно добавлять к сценариям.

Создание функций

Чтобы создать функцию, введите ключевое слово **function**, затем имя функции. Далее введите составляющие ее команды, взяв их в фигурные скобки, { }. Ниже показан пример функции getwinrm, вызывающей команду «get-service -name winrm»:

```
function getwinrm {get-service -name winrm}
```

Теперь для вызова этой команды достаточно ввести **getwinrm**, при желании для этой функции можно даже объявить псевдонимы, например псевдоним *gr*:

```
new-alias gr getwinrm
```

В код функций разрешается включать команды, конвейеры, операторы перенаправления и прочие элементы команд. Например, так можно использовать конвейер для форматирования вывода Get-Service:

```
function getwinrm {get-service -name winrm | format-list}
```

Функции — очень мощный инструмент еще и потому, что им можно передавать различные параметры. Вот общий синтаксис функции с параметрами:

```
function Функция {
  param ($Параметр1, $Параметр2, ...)
  Код }
```

где *Функция* — имя функции, *$Параметр1* — имя первого параметра, *$Параметр2* — имя второго параметра и т.д. Рассмотрим следующий пример:

```
function ss {param ($status) get-service | where { $_.status -eq $status} }
```

Этот код объявляет функцию *ss* с параметром *status*. Функция *ss* ищет в форматированном списке служб, полученном с помощью командлета Where-Object, службы в заданном состоянии. Так, чтобы получить список остановленных служб, введите:

```
ss -status stopped
```

а чтобы получить список работающих служб, введите:

```
ss -status running
```

Имена параметров указывать не обязательно, достаточно указать значения, например:

```
ss stopped
```

или

```
ss running
```

Учтите, что функции исполняются в контексте собственной локальной области действия. Соответственно, элементы, такие как переменные, объявленные в функции, доступны только внутри этой функции, а если функция объявлена в сценарии — то внутри сценария. Это означает, что объявленную в сценарии функцию по умолчанию невозможно вызвать из командной строки. Функции, объявленные как глобальные, доступны и в сценариях, и в командной строке.

Чтоб задать область действия функции, достаточно объявить ее перед именем функции. Например, так можно объявить глобальную функцию:

```
function global:getwinrm {get-service -name winrm}
```

Расширенные функции

Познакомившись с простейшими функциями, можно переходить к расширенным функциям. Вот как выглядит их синтаксис:

```
function $Функция {
  param ($Параметр1, $Параметр2, ...)
  Begin {
    <команды, выполняемые однократно перед обработкой>
  }
  Process{
    <команды, обрабатывающие каждый объект >
  }
  End{
```

```
    <команды, выполняемые многократно после обработки>
  }
}
```

Синтаксис расширенных функций предусматривает блоки Begin, Process и End, которые позволяют максимально приблизить функции к командлетам. Begin — необязательный блок, в который помещают команды, однократно исполняемые перед обработкой данных. Поскольку команды из блока Begin исполняются до обработки объектов, поступающих по конвейеру, в момент исполнения блока Begin эти объекты недоступны.

Блок Process необходим для обработки ввода. В общем синтаксисе этот блок определен неявно, но при использовании других необязательных блоков, а также при необходимости особой обработки ввода необходимо явно объявлять блок Process. Команды из этого блока исполняются при обработке каждого из переданных по конвейеру объектов.

Блок End также не является обязательным, в него помещают команды, однократно исполняемые после обработки всех объектов. Следовательно, им, как и командам из блока Begin, недоступны объекты, переданные по конвейеру.

Вот пример использования блоков Begin, Process и End:

```
function scheck {
param ($status)
 Begin {
  Write-Warning «################ Services on $env:computername»
 }
 Process {
  get-service | where { $_.status -eq $status}
 }
 End {
  Write-Warning «##############################################»
 }
}
```

Объявив эту функцию в командной строке или сценарии, можно получить список всех служб в состоянии Stopped с помощью команды:

```
scheck stopped
```

а чтобы получить список служб в состоянии Running, введите:

```
ss -status running
```

Использование фильтров

Фильтр — это функция особого типа, обрабатывающая каждый из переданных по конвейеру объектов. Можно назвать фильтром функцию, весь код который сосредоточен в блоке Process.

Общий синтаксис фильтров имеет следующий вид:

```
filter Фильтр {
  param ($Параметр1, $Параметр2, ...)
  Код }
```

где *Фильтр* — имя фильтра, *$Параметр1* — имя первого параметра, *$Параметр2* — имя второго параметра и т.д.

Сильная сторона фильтров в том, что они обрабатывают объекты поочередно. Поэтому фильтры идеальны для обработки больших объемов данных, поступающих по конвейеру, и выделения из них нужной информации. Как и для функций, для фильтров можно объявлять область действия, указывая ее перед именем фильтра.

При работе с фильтрами часто используется автоматическая переменная $_ (для манипулирования текущим объектом, поступившим через конвейер):

```
filter Name { $_.Name }
```

Этот фильтр выводит значение свойства Name каждого из переданных фильтру объектов. В следующем примере вывод командлета Get-PSDrive передается функции-фильтру через конвейер:

```
get-psdrive | name
```

Фильтр возвращает имя каждого из доступных объектов PSDrive. Поскольку фильтр — это не что иное, как функция, все команды которой сосредоточены в блоке Process, следующая функция равноценна показанному выше фильтру:

```
function Name {
   Process { $_.Name }
}
```

Подробнее о функциях

Описание функций в предыдущем разделе далеко от исчерпывающего, поэтому ниже мы остановимся на особенностях функций более подробно. Например, можно задать для параметра значение по умолчанию, как показано в следующем примере:

```
function ss {param ($status = «stopped») get-service |
where { $_.status -eq $status} }
```

После этого объявления параметр –Status получил значение по умолчанию «Stopped». Чтобы переопределить значение по умолчанию, укажите другое значение при вызове функции.

При объявлении функций можно указать тип для параметров с использованием ранее псевдонимов, перечисленных выше в табл. 6-1. Ниже объявлена функция, динамически настраивающая размер окна PowerShell:

```
function set-windowsize {

param([int]$width=$host.ui.rawui.windowsize.width,
[int]$height=$host.ui.rawui.windowsize.height)

$size=New-Object System.Management.Automation.Host.Size($width,$height);

$host.ui.rawui.WindowSize=$size

}
```

У этой функции два параметра: $width и $height, оба объявлены как 32-разрядные целочисленные значения. Соответственно, PowerShell ожидает, что при вызове функции для этих параметров будут указаны именно такие значения. Если для параметров заданы другие значения, PowerShell попытается привести их к заданному типу, а в случае неудачи PowerShell вернет сообщение об ошибке из-за невозможности привести значение параметра к заданному типу.

Командлет Get-Command выводит список определенных в данный момент командлетов и функций. Чтобы получить только список доступных функций, можно отфильтровать вывод с использованием конструкции Where:

```
get-command | where {$_.commandtype -eq «function»}
```

Эта команда отбирает объекты команд, у которых свойство CommandType содержит значение Function, составляет из них список функций. Добавив | **format-list** к предыдущей команде, можно дополнить список функций расширенными определениями:

```
get-command | where {$_.commandtype -eq «function»} | format-list
```

Передача параметра функции напрямую при ее вызове отлично работает, когда требуется:

- объявить функцию в командной строке и там же вызвать ее;
- объявить функцию в сценарии и вызвать ее из сценария.

Однако передать через командную строку параметры функции, объявленной в сценарии, не получится. Для создания сценария-библиотеки функций, которые можно вызвать из командной строки, придется написать функции так, чтобы они «понимали» аргументы, переданные сценарию, либо добавить к сценарию код, обрабатывающий аргументы. Одно из решений показано в следующем примере:

```
Файл CheckIt.ps1

function scheck {param ($status)
 Begin {
  Write-Warning «################ Services on $env:computername»
 }
```

```
 Process {
  get-service | where { $_.status -eq $status}
 }
 End {
  Write-Warning «##############################################»
 }
}
if ($args[0] = «scheck») {scheck $args[1]}
```

Команда в последней строке сценария проверяет, была ли вызвана функция Scheck. Если при запуске сценарию было передано значение *scheck* как первый аргумент и строка с названием состояния как второй аргумент, вызывается функция Scheck, которая выводит список служб в указанном состоянии. Альтернативный подход прост: он состоит в том, чтобы назначить параметру Status значение первого аргумента, переданному сценарию (см. пример ниже), и вызвать сценарий, передав ему искомое состояние как первый аргумент. Обратите внимание, что последняя строка сценария — это вызов функции.

файл CheckIt2.ps1

```
function scheck {param ($status = $args[0])
 Begin {
  Write-Warning «################ Services on $env:computername»
 }
 Process {
  get-service | where { $_.status -eq $status}
 }
 End {
  Write-Warning «##############################################»
 }
}
scheck
```

Просмотр определений функций

С функциями позволяет работать поставщик *function:*, содержимое которого выглядит как содержимое диска. Например, если перейти на диск поставщика function: командой:

```
set-location function:
```

приглашение PowerShell примет следующий вид:

```
PS Function:\>
```

Далее можно работать с любыми или только с заданными функциями. Чтобы вывести список имен и определений функций, введите:

```
get-childitem
```

Чтобы вывести имена и определения требуемых функций, введите после Get-ChildItem имена нужных функций полностью либо с подстановочными знаками:

```
get-childitem enable-psremoting
get-childitem *psremoting
```

Чтобы вернуться на диск файловой системы по окончании работы с функциями, введите **set-location** и букву нужного диска, например:

```
set-location c:
```

Получить определения функций с диска function:, не переходя на него, можно с помощью командлета Get-Item:

```
get-item -path function:*
```

Чтобы получить конкретные функции, введите их имена полностью, как в этом примере:

```
get-item -path function:prompt
```

либо с подстановочными знаками:

```
get-item -path function:pr*
```

Работа со встроенными функциями

В табл. 6-3 приводится список функций, определенных по умолчанию. Как видите, большинство их служит для доступа к различным дискам, папками и файлам.

Табл. 6-3. Функции, объявленные по умолчанию

Имя функции	Определение (какие команды заменяет функция)	Назначение
A:, B:, C:, D:, E:, F:, G:, H:, I:, J:, K:, L:, M:, N:, O:, P:, Q:, R:, S:, T:, U:, V:, W:, X:, Y:, Z:	Set-Location *Диск*, где *Диск* — буква нужного диска	Позволяет перейти на заданный диск командой **C:** вместо **Set-Location C:**
CD..	Set-Location ..	Позволяет вернуться в родительский каталог командой **CD..** вместо **Set-Location ..**
CD\	Set-Location \	Позволяет переходить в корневой каталог командой **CD** вместо **Set-Location **
Clear-Host	$space = New-Object System.Management.Automation.Host.BufferCell$space.Character = ‘ ‘	Очищает хронологию консоли PowerShell одной командой **Clear-Host** без вызова пространства исполнения

(*см. след. стр.*)

Табл. 6-3. Функции, объявленные по умолчанию

Имя функции	Определение (какие команды заменяет функция)	Назначение
Disable-PSRemoting	Disable-PSSession Configuration * –force:$force	Отключает поддержку удаленной работы в PowerShell одной командой **Disable-PSRemoting**
Enable-PSRemoting	Enable-PSSession Configuration * –force:$force	Включает поддержку удаленной работы в PowerShell одной командой **Enable-PSRemoting**
Help	Get-help | More	Постранично выводит справку по командлетам
Mkdir	New-Item –path *Путь* –name *Имя* –type directory	Создает заданный каталог
More	| More	Постранично отображает содержимое файла, заменяя команды вида **Команда | More**
Prompt	$(if (test-path variable:/PSDebug Context) { '[DBG]: ' } else { '' }) + 'PS ' + $(GetLocation) + $(if ($nestedpromptlevel –ge 1) { '>>' }) + '> '	Отображает приглашение PowerShell. По умолчанию оно содержит строку PS, пробел и текущий каталог. При вводе незаконченной команды приглашение принимает вид >> и остается таким до завершения ввода команды
TabExpansion	*	Позволяет завершать имена команд и параметров нажатием Tab

Рекомендую вам поскорее «набить руку» в использовании этих функций, поскольку они эффективно автоматизируют решение множества задач, особенно это касается функций Prompt и TabExpansion. По умолчанию приглашение PowerShell содержит строку PS, пробел и текущий каталог. При вводе незаконченной команды приглашение принимает вид >> и остается таким до завершения ввода команды. В режиме отладки приглашение имеет вид [DBG]:.

Вы можете объявить собственную функцию Prompt, заменив ею одноименную стандартную функцию. Например, чтобы заменить в приглашении каталог текущей датой, объявите функцию Prompt так:

```
function prompt {«$(get-date)> »}
```

А такое объявление включает в приглашение имя компьютера:

```
function prompt {«PS [$env:computername]> »}
```

Чтобы оставить в приглашении стандартную информацию, скопируйте исходное определение функции и отредактируйте его как требуется. Стандартное приглашение выводится с помощью такой функции:

```
function prompt {
$(if (test-path variable:/PSDebugContext) { '[DBG]: ' }
else { '' }) + 'PS ' + $(Get-Location)
+ $(if ($nestedpromptlevel -ge 1) { '>>' }) + '> '
}
```

При настройке функции код, выделенный **полужирным**, заменяют нужными переменными. Например, для включения в стандартное приглашение даты используйте следующее объявление:

```
function prompt {
$(if (test-path variable:/PSDebugContext) { '[DBG]: ' }
else { '' }) + «$(get-date)> »
+ $(if ($nestedpromptlevel -ge 1) { '>>' }) + '> '
}
```

Это объявление добавляет в приглашение PowerShell имя компьютера:

```
function prompt {
$(if (test-path variable:/PSDebugContext) { '[DBG]: ' }
else { '' }) + «PS [$env:computername]> »
+ $(if ($nestedpromptlevel -ge 1) { '>>' }) + '> '
}
```

Если функцию Prompt можно назвать полезным излишеством, то, однажды попробовав функцию TabExpansion, вы уже не сможете обойтись без нее. Эта функция позволяет завершать ввод имен командлетов, а также имен и даже значений параметров, нажатием клавиши Tab. Вот как это работает:

- вы вводите первые несколько букв имени командлета и нажатием Tab перебираете имена (в алфавитном порядке) командлетов, соответствующие введенным буквам. Предположим, что нужный вам командлет начинается с Get-. Вы вводите **Get-** и нажатием Tab перебираете команды, имена которых начинаются с этих букв. В этом примере PowerShell сначала предлагает Get-Acl, затем Get-Alias, Get-AuthenticateSignature и т.д. Аналогично, если известна первая части (Get) и первая буква второй части (C), вы вводите **Get-C** и нажатием Tab перебираете доступные варианты завершения ввода. Мало кто знает, что комбинация Shift+Tab позволяет перебирать варианты в обратном порядке — это удобно, если вы знаете, что нужный вариант находится ближе к концу алфавита;
- после ввода имени командлета можно воспользоваться Tab и для ввода параметров. Если вы не знаете имя параметра, введите дефис (**-**), после этого вам будет доступен список вариантов, который также можно перебирать нажатием Tab. Если вам известны первые буквы имени параметра, введите дефис (**-**), затем известные буквы и нажмите Tab. Как и в пре-

дыдущем случае, комбинация Shift+Tab прокручивает список вариантов в обратном порядке.

Примечание Нажатие Tab в пустой строке вставляет в нее знак табуляции, но нажатие этой клавиши в месте ожидаемого ввода параметра открывает доступный для перебора список имен файлов и папок в текущей папке.

Работа с объектами

Любая операция в PowerShell выполняется в контексте того или иного объекта. Объект является фундаментальной сущностью в объектно-ориентированном программировании (ООП). Программы на языках, ориентированных на ООК, описывают взаимодействие объектов, в PowerShell объекты используются совершенно так же.

Основы работы с объектами

Всю реальную работу в PowerShell выполняют объекты. Данные, которые передаются от одной команды другой, на самом деле путешествуют внутри объектов. В сущности, объекты — это просто наборы данных, представляющих элементы в определенных пространствах имен.

У каждого объекта есть тип, состояние и запрограммированное поведение. Тип — это информация о сущности, которую представляет объект. Например, объекты, представляющие работающие в системе процессы, относятся к типу Process. Под состоянием объекта понимают составляющие его элементы и их значения. Совокупное описание этих элементов и значений и является состоянием объекта. Данные, содержащиеся в объекте, хранятся в виде значений их свойств.

Поведение объекта — это совокупность действий, которые он может выполнять над сущностями, которые этот объект представляет, в ООП эти действия называются *методами*. Метод принадлежит классу объекта и является его членом, методы вызывают, когда требуется выполнить определенное действие над сущностью, с которой связан объект. Например, у объекта Process есть метод для остановки процессов. Вызвав это метод, вы остановите исполнение процесса, представленного объектом, чей метод вы вызвали. Резюмируем сказанное: состояние объекта — это содержащаяся в нем информация, а поведение объекта — это действия, которые он может выполнять. Объекты инкапсулируют логически связанные свойства и методы в целостные идентифицируемые единицы. Благодаря всем этим качествам с объектами просто работать в программе, их также легко многократно использовать и обновлять.

Классы объектов инкапсулируют отдельные экземпляры объектов. То есть, на основе класса можно создать множество объектов или, как еще говорят, экземпляров этого класса. Инкапсуляция объектов классами позволяет группировать объекты по типу. Например, после ввода команды **get-process**

PowerShell возвращает набор объектов, представляющих процессы, работающие на компьютере. Тем не менее, все объекты в этом наборе существуют по отдельности, у каждого из них — свое состояние и свойства.

PowerShell поддерживает несколько десятков типов объектов. Команды, объединенные в конвейер, обмениваются данными в виде объектов. Первая в конвейере команда передает сгенерированные ей объекты определенного класса следующей команде, которая принимает объекты, обрабатывает их и передает результаты по конвейеру, также в виде объектов. Так продолжается до завершения последней команды конвейера, после чего выводятся окончательные результаты.

Чтобы получить сведения о свойствах и методах объектов, следует направить вывод через командлет Get-Member. Так, системные процессы и службы представлены объектами Process и Service, соответственно. Чтобы узнать, какие свойства и методы есть у объектов Process, введите **get-process | get-member**, а чтобы выполнить эту операцию для объекта Service, введите **get-service | get-member**. В обоих примерах знак конвейера (|) служит для передачи вывода первого командлета другому командлету, Get-Member, который выводит имя класса объекта и полный список его членов:

```
get-service | get-member
```

```
   TypeName: System.ServiceProcess.ServiceController

Name                      MemberType    Definition
----                      ----------    ----------
Name                      AliasProperty Name = ServiceName
Disposed                  Event         System.EventHandler Disposed
Close                     Method        System.Void Close()
. . .
CanPauseAndContinue       Property      System.Boolean CanPauseAndCo
. . .
Status                    Property      System.ServiceProcess.Service
```

Совет По умолчанию командлет Get-Member не выводит статические методы и свойства классов. Чтобы включить их вывод, введите **get-member –static**. Например, чтобы получить список статических членов объекта ServiceProcess, следует ввести **get-service | get-member –static**.

Заметьте, что в выводе команды для каждого члена класса Service указано имя его типа. Эта информация проще воспринимается, если отфильтровать ее с помощью параметра –MemberType. Его допустимые значения включают:

- AliasProperty, CodeProperty, NoteProperty, ParameterizedProperty, Property, PropertySet, ScriptProperty, CodeMethod, MemberSet, Method и ScriptMethod — для анализа элементов определенного типа;
- Properties — для анализа элементов, связанных со свойствами;
- Methods — для анализа элементов, связанных с методами;

- All — для анализа любых свойств и методов (это значение по умолчанию).

При анализе объектов с помощью Get-Member следует обратить внимание на свойства-псевдонимы (alias properties), которые похожи на другие псевдонимы и предоставляют понятные имена объектов. У объекта Service имеется единственное свойство-псевдоним (Name), у большинства распространенных объектов таких свойств несколько, как у объекта Process:

```
get-process | get-member
```

```
   TypeName: System.Diagnostics.Process

Name                       MemberType     Definition
----                       ----------     ----------
Handles                    AliasProperty  Handles = Handlecount
Name                       AliasProperty  Name = ProcessName
NPM                        AliasProperty  NPM = NonpagedSystemMemory
PM                         AliasProperty  PM = PagedMemorySize
VM                         AliasProperty  VM = VirtualMemorySize
WS                         AliasProperty  WS = WorkingSet
```

Именно эти псевдонимы выводят командлеты, которые показывают список процессов, например:

```
get-process
```

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     52       3     1296       3844    51     0.00   3004 acrotray
    139       4     2560       7344    75            1292 AlertService
    573      14    14680      11028   120     7.41   2764 aolsoftware
     97       4     2872       4476    53            1512 AppleMobil...
```

Возможно, вам будет любопытно, куда делись остальные десятки свойств объекта Process и почему **get-process | get-member** не выводит их. Причина в том, что по умолчанию PowerShell показывает упрощенные представления распространенных объектов, включающие только их наиболее важные свойства.

PowerShell получает информацию о представлении объектов того или иного типа из XML-файлов *.format.ps1xml. Формат вывода информации о многих объектах определен в файле types.ps1xml file, расположенном в каталоге $pshome.

Отсутствие в выводе некоторых свойств вовсе не означает, что у объектов их нет: они есть и полностью доступны вам. Например, команда **get-process winlogon | format-list –property *** выводит имена и значения всех свойств объекта процесса Winlogon.

Методы и свойства объектов

Некоторые свойства и методы могут быть связаны только с экземплярами объекта, поэтому они называются *методами экземпляров* и *свойствами экземпляров*. В данном случае термины *экземпляр* и *объект* являются синонимами.

Нередко требуется обращаться к методам и свойствам объекта, ссылаясь на него по имени переменной, в которой этот объект хранится. Чтобы разобраться, как это работает, рассмотрим следующий пример:

```
$myString = «THIS IS A TEST!»
```

Здесь строка записывается в переменную $myString. В PowerShell для представления строк служат объекты String, у которых имеется свойство Length, в котором хранится длина строки. Таким образом, узнать длину строки позволяет следующая команда:

```
$myString.Length
```

В этом примере строка содержит 15 символов, поэтому команда выводит 15.

Объекты String обладают единственным свойством, но зато у них несколько методов, включая ToUpper() и ToLower(). Метод ToUpper() выводит все символы строки в верхнем регистре, а ToLower() — в нижнем. Например, чтобы привести строку из предыдущего примера к виду «this is a test!», введите

```
$myString.ToLower()
```

Из предыдущих примеров видно, что к свойствам объекта обращаются, указывая имя переменной с объектом, затем точку и имя нужного свойства:

```
$Объект.Свойство
$Объект.Свойство = Значение
```

Аналогично, для вызова метода после имени переменной с объектом и точки указывают имя метода:

```
$Объект.Метод()
```

Метод выполняют над объектами различные операции, при этом им нередко требуются параметры, которые передают следующим образом:

```
$Объект.Метод(параметр1, параметр2, …)
```

Выше рассматривались приемы работы с методами и свойствами экземпляров объектов, но администратору приходится работать не только с конкретными экземплярами объектов, но и со статическими методами и свойствами их классов.

Инфраструктура .NET Framework поддерживает множество классов или типов. Имена классов .NET Framework заключаются в квадратные скоб-

ки. Например, класс [System.Datetime] служит для работы с датой и временем, а [System.Diagnostics.Process] — для работы с системными процессами. PowerShell добавляет префикс *System.* к именам типов автоматически, поэтому на классы для работы с датами и процессами можно также ссылаться по именам [Datetime] и [Diagnostics.Process].

Статические методы и свойства классов .NET Framework всегда доступны при работе с классами объектов, но командлет Get-Member выводит их только при вызове с параметром –Static, например:

```
[System.Datetime] | get-member -Static
```

Чтобы обратиться к статическому свойству класса .NET Framework, следует после имени класса, взятого в квадратные скобки, поставить два двоеточия (::), затем имя свойства, как показано ниже:

```
[Класс]::Свойство
[Класс]::Свойство = Значение
```

В этом примере статическое свойство класса [System.Datetime] используется для вывода текущей даты и времени:

```
[System.Datetime]::Now
```

```
Monday, February 15, 2010 11:05:22 PM
```

Аналогично вызывают статические методы классов .NET Framework:

```
[Класс]::Метод()
```

Для передачи параметров статическим методам используют синтаксис вида

```
[Класс]::Метод(параметр1, параметр2, …)
```

В следующем примере статический метод класса [System.Diagnostics.Process] используется для вывода информации о процессе:

```
[System.Diagnostics.Process]::GetProcessById(0)
```

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
      0       0        0         24     0               0 Idle
```

Типы объектов

Определения форматов и типов объектов, используемые PowerShell по умолчанию, хранятся в следующих файлах, расположенных в каталоге $pshome:

- **Certificate.Format.ps1xml** — объекты сертификатов и сертификатов X.509;
- **Diagnostics.Format.ps1xml** — объекты, созданные во время работы с счетчиками производительности и диагностическими функциями PowerShell;

- **DotNetTypes.Format.ps1xml** — объекты .NET Framework, не описанные в других файлах определений (например, объекты CultureInfo, FileVersionInfo и EventLogEntry);
- **FileSystem.Format.ps1xml** — объекты файловой системы;
- **GetEvent.Types.ps1xml** — объекты журналов событий, записей журналов и счетчиков производителей;
- **Help.Format.ps1xml** — представления, используемые PowerShell для вывода содержимого справки;
- **PowerShellCore.Format.ps1xml** — объекты, генерируемые основными командлетами PowerShell;
- **PowerShellTrace.Format.ps1xml** — объекты PSTraceSource, генерируемые при трассировке в PowerShell;
- **Registry.Format.ps1xml** — объекты системного реестра;
- **Types.ps1xml** — системные объекты;
- **WSMan.Format.ps1xml** — объекты, генерируемые при работе с конфигурациями WS Management в PowerShell.

Автоматическое определение и приведение типов в PowerShell обеспечивается общим объектом, способным динамически предоставлять сведения о своем типе и членах, а также взаимодействовать с другими объектами через унифицированный уровень абстракции. Этот объект называется PSObject. Он способен инкапсулировать любые базовые объекты, включая системные объекты, объекты WMI, COM и ADSI.

Объект PSObject, являясь оболочкой для инкапсулируемого объекта, может обеспечивать для него адаптированное представление либо дополнительные возможности. Адаптированное представление базового объекта поддерживает избранные члены, доступные напрямую, остальные члены доступны через другие представления базового объекта. Вот список доступных представлений базового объекта:

- **PSBase** — открывает доступ к свойствам исходного объекта;
- **PSAdapted** — дает адаптированное представление объекта;
- **PSExtended** — предоставляет дополнительные свойства и методы объекта, объявленные в Types.ps1xml или с помощью Add-Member;
- **PSObject** — открывает доступ к адаптеру, который преобразует базовый объект в PSObject;
- **PSTypeNames** — предоставляет список типов, описывающих объект.

По умолчанию PowerShell возвращает только информацию, полученную от представлений PSObject, PSExtended и PSTypeNames, но можно обращаться и к другим представлениям, используя синтаксис, показанный ниже. Следующая команда получает объект Win32_Process, представляющий процесс winlogon.exe:

```
$pr = Get-WmiObject Win32_Process | where-object { $_.ProcessName -eq
«winlogon.exe» }
```

Далее можно получить из переменной **$pr** любые сведения о процессе, доступные через представления PSObject, PSExtended и PSTypeNames, а следующая команда служит для просмотра членов объекта, доступных через представление PSBase:

```
$pr.PSBase.Properties
```

Иногда требуется самостоятельно дополнить объект нужными свойствами. Это делается с помощью настройки файла определений Types.ps1xml либо командлета Add-Member. Чаще всего используются следующие типы расширений:

- **ScriptProperty** — добавляет к типам свойства, в том числе вычисляемые;
- **AliasProperty** — определяет псевдонимы для свойств;
- **ScriptMethod** — определяет дополнительные свойства.

В пользовательском файле Types.ps1xml можно объявлять вышеперечисленные расширения, используя XML-элементы, ориентируясь на многочисленные примеры, доступные в каталоге $pshome. Разрешается добавлять эти расширения динамически, через командную строку либо сценарии. Так, ScriptProperty — это один из типов членов объекта. Он содержит блок кода, обрабатывающий и извлекающий информацию об объектах. Вот его общий синтаксис:

```
$Объект | add-member -membertype scriptproperty -name Имя -value {Код}
```

Поскольку параметры –Name и–Value интерпретируются в зависимости от позиции в списке параметров, их имена можно не указывать. Рассмотрим следующий пример:

```
$proc = get-process powershell;

$proc | add-member -Type scriptproperty «UpTime» {return ((date) -
($this.starttime))};

$proc | select Name, @{name='Uptime'; Expression={«{0:n0}» -f
$_.UpTime.TotalMinutes}};
```

```
Name                                                          Uptime
----                                                          ------
powershell                                                      242
```

Эти команды получают объекты, представляющие процессы Powershell.exe на компьютере. Далее с помощью командлета Add-Member к стандартному объекту процесса добавляется свойство ScriptProperty. Код, добавленный вместе со ScriptProperty, вычисляет время работы процесса. Следующая команда снова получает объект процесса, и на этот раз к ее выводу добавляется имя процесса и новое свойство Uptime. С помощью регулярного выражения значение Uptime переводится в минуты, после чего выводятся результаты.

Первая команда записывает в переменную $proc набор стандартных объектов-процессов. После этого генерируется второй набор объектов-процессов, инкапсулированных в экземпляры PSObject, к которым добавлено новое свойство Uptime с помощью командлета Add-Member.

Если ScriptProperty расширяет функциональность объектов, то AliasProperty упрощает работу с ними. Рассмотрим это на примере командлета Get-PSDrive, который генерирует новый объект PSDriveInfo, предоставляющий информацию о диске C:

```
$myDrive = get-psdrive C
$myDriveInfo = New-Object System.IO.DriveInfo $myDrive
```

Заметьте, что при наличии объекта, представляющего диск C:, для получения информации об этом диске достаточно ввести **$myDriveInfo**. Вывод этой команды содержит сведения о диске, его состоянии, емкости и размере свободного места:

```
Name                : C:\
DriveType           : Fixed
DriveFormat         : NTFS
IsReady             : True
AvailableFreeSpace  : 302748798976
TotalFreeSpace      : 302748798976
TotalSize           : 490580373504
RootDirectory       : C:\
VolumeLabel         :
```

По умолчанию вывод форматируется в виде списка, но при желании можно получить информацию в виде таблицы, что удобно, например, при работе с несколькими дисками. Вывод команды **$myDriveInfo | format-table –property *** не очень удобно читать, если не растягивать окно консоли до предела. Чтобы сделать вывод более компактным, можно создать псевдонимы для свойств AvailableFreeSpace, TotalFreeSpace и RootDirectory. Это также делается с помощью командлета Add-Member и команд вида:

```
$Объект | add-member -membertype aliasproperty -name Псевдоним -value
Свойство
```

где *Объект* — имя объекта, с которым вы работаете; *Псевдоним* — новый псевдоним для свойства; *Свойство* — исходное имя свойства, например:

```
$myDriveInfo | add-member -membertype aliasproperty -name Free -value
AvailableFreeSpace

$myDriveInfo | add-member -membertype aliasproperty -name Format -value
DriveFormat
```

С псевдонимами работают как с настоящими именами свойств. Например, для просмотра значения свойства AvailableFreeSpace можно ввести как:

```
$myDriveInfo.AvailableFreeSpace
```

или

```
$myDriveInfo.Free
```

Псевдонимы можно использовать и при форматировании вывода команд, например, так:

```
$myDriveInfo | format-table -property Name, Free, Format
```

```
Name                                    Free     Format
----                                    ----     ------
C:\                                302748483584    NTFS
```

Расширения ScriptMethod позволяют добавлять к объектам новые методы с использованием следующего синтаксиса:

```
$Объект | add-member -membertype scriptmethod -name Имя
 -value {Код}
```

Параметры –Name и –Value интерпретируются в зависимости от их положения в список параметров, поэтому указывать их имена не обязательно. Рассмотрим пример:

```
$myDrive = get-psdrive C

$myDrive | add-member -membertype scriptmethod -name Remove
 -value { $force = [bool] $args[0]
 if ($force) {$this | Remove-PSDrive }
 else {$this | Remove-PSDrive -Confirm}
}
```

Здесь к объекту PSDrive добавляется метод Remove. При вызове этого метода без аргументов PowerShell запрашивает подтверждение отключения диска от текущего сеанса. Если же вызвать этот метод и передать ему параметр $true со значением 0 (ноль), PowerShell отключит диск от сеанса, не запрашивая подтверждения.

Близкое знакомство с объектами

Давайте познакомимся с объектами поближе на примере $host. Как сказано в главе 1, окно свойств консоли PowerShell служит для настройки шрифта, цвета и других параметров консоли. То же самое можно сделать с помощью объекта $host, который открывает доступ к интерфейсу консоли либо графической среды PowerShell.

Для просмотра текущей конфигурации объекта $host введите:

```
$host.ui.rawui | format-list -property *
```

Эта команда должна вывести данные примерно такого вида:

```
ForegroundColor       : DarkYellow
BackgroundColor       : DarkMagenta
CursorPosition        : 0,1050
WindowPosition        : 0,1001
CursorSize            : 25
BufferSize            : 120,3000
WindowSize            : 120,50
MaxWindowSize         : 120,95
MaxPhysicalWindowSize : 240,95
eyAvailable           : False
WindowTitle           : Windows PowerShell V2
```

Вывод содержит ряд свойств, включая:

- **ForegroundColor** — задает цвет приглашения и текста;
- **BackgroundColor** — задает цвет фона в окне консоли;
- **WindowTitle** — задает текст заголовка окна PowerShell.

Для настройки окна PowerShell необходимо получить ссылку на объект $host. Проще всего сделать это, записав объект $ в переменную, например, так:

```
$myHostWin = $host.ui.rawui
```

Получив необходимую ссылку, можно обращаться к свойствам и методам объекта. Для назначения цветов фона и текста можно использовать следующие значения:

- Black, DarkBlue, DarkGreen, DarkCyan;
- DarkRed, DarkMagenta, DarkYellow, Gray;
- DarkGray, Blue, Green, Cyan;
- Red, Magenta, Yellow, White.

Для этого необходимо присвоить соответствующему свойству представляющего окно объекта нужное значение, например, так:

```
$myHostWin.ForegroundColor = «White»
```

или так:

```
$myHostWin.BackgroundColor = «DarkGray»
```

Аналогично задают заголовок окна консоли с помощью свойства WindowTitle:

```
$myHostWin.WindowTitle = «PowerShell on $env.computername»
```

Здесь в заголовок окна включается имя компьютера. Соответственно, на компьютере TechPC32 заголовок будет иметь следующий вид:

```
PowerShell on TechPC32
```

Давайте вернемся к перечню свойств объекта $host. У некоторых из них значения представляют списки, разделенные запятыми. Это говорит о том,

что такие свойства содержат массивы «вложенных» свойств. Для просмотра такого массива можно вывести его в виде списка. Например, для просмотра значений свойства CursorPosition введите следующую команду:

```
$host.ui.rawui.CursorPosition | format-list -property *
```

Вывод будет примерно таким:

```
X : 0
Y : 2999
```

Видно, что свойство CursorPosition содержит два вложенных свойства: X и Y. Чтобы обратиться к вложенному свойству, после имени свойства следует поставить точку, затем — имя вложенного свойства:

```
$host.ui.rawui.CursorPosition.X
$host.ui.rawui.CursorPosition.Y
```

Продолжив анализ свойств, несложно обнаружить, что у свойств CursorPosition и WindowPosition также есть вложенные свойства X и Y, а у свойств BufferSize, WindowSize, MaxWindowSize и MaxPhysicalWindowSize — Width и Height.

Получив список вложенных свойств можно узнать их значения. Это делается так же, как определение значений обычных свойств. Напротив, в отличие от обычных свойств, нельзя присваивать вложенным свойствам значение напрямую. Для этого необходимо с помощью командлета New-Object создать экземпляр объекта $host и настроить его свойства.

Это означает, что сначала необходимо получить ссылку на объект $host следующим образом:

```
$myHost = $host.ui.rawui
```

Далее следует создать новый экземпляр объекта и настроить его вложенные свойства, как показано ниже:

```
$myHostWindowSize = New-Object
System.Management.Automation.Host.Size(150,100)
```

Это пример динамической установки размера окна консоли. Первое значение — это ширина, а второе — высота окна.

Работа с объектами COM и .NET Framework

COM (Component Object Model) и .NET Framework — две модели объектов, с которыми чаще всего сталкиваются администраторы, работающие в PowerShell. Многие приложения обеспечивают поддержку сценариев и функций администрирования посредством COM, но использование для этой цели объектов .NET Framework и командлетов PowerShell становится все популярнее.

Создание объектов COM и работа с ними

Экземпляры объектов COM создаются с помощью командлета New-Object, общий синтаксис которого имеет следующий вид:

```
New-Object [-Set Ассоциативный_массив] [-Strict] [-ComObject] Строка
```

Значением параметра –ComObject является программный идентификатор (ProgID) нового объекта. PowerShell поддерживает большинство распространенных объектов COM, включая объекты Windows Script Host (WSH). Вот пример программного создания ярлыка на рабочем столе:

```
$WshShell = New-Object -ComObject WScript.Shell
$scut = $WshShell.CreateShortcut(«$Home\Desktop\PowerShellHome.lnk»)
$scut.TargetPath = $PSHome
$scut.Save()
```

Эти команды создают ярлык *PowerShellHome*, ссылающийся на каталог $PSHome.

Совет Получив ссылку на объект COM, можно просматривать его свойства и методы с помощью клавиши Tab. Например, если объект хранится в переменной $a, введите **$a.** и нажмите Tab либо Shift+Tab для перебора доступных методов и свойств объекта.

Кроме объектов WSH, существует много других полезных объектов COM (см. табл. 6-4).

Табл. 6-4. Объекты COM, поддерживаемые PowerShell

ProgID	Для чего используется
Access.Application	Доступ к Microsoft Office Access
CEnroll.Cenroll	Доступ к службам регистрации заявок на выпуск сертификатов
Excel.Application	Доступ к Microsoft Office Excel
Excel.Sheet	Доступ к листам книг Excel
HNetCfg.FwMgr	Доступ к Брандмауэру Windows
InternetExplorer. Application	Доступ к Internet Explorer
MAPI.Session	Доступ к сеансам MAPI
Messenger. MessengerApp	Доступ к Windows Messenger
Microsoft.Update. AutoUpdate	Доступ к графику автоматического обновления Microsoft Update
Microsoft.Update. Installer	Установка обновлений через Microsoft Update
Microsoft.Update.Searches	Поиск обновлений через Microsoft Update
Microsoft.Update.Session	Доступ к журналу обновлений Microsoft Update

(*см. след. стр.*)

Табл. 6-4. Объекты COM, поддерживаемые PowerShell

ProgID	Для чего используется
Microsoft.Update.SystemInfo	Доступ к системной информации Microsoft Update
Outlook.Application	Доступ к Microsoft Office Outlook
OutlookExpress.MessageList	Поддержка автоматизации электронной почты Microsoft Office Outlook Express
PowerPoint.Application	Доступ к Microsoft Office PowerPoint
Publisher.Application	Доступ к Microsoft Office Publisher
SAPI.SpVoice	Доступ к Microsoft Speech (API)
Scripting.FileSystemObject	Доступ к файловой системе компьютера
SharePoint.OpenDocuments	Доступ к службам Microsoft SharePoint Services
Shell.Application	Доступ к Проводнику (Windows Explorer)
Shell.LocalMachine	Доступ к сведениям об оболочке на локальном компьютере
SQLDMO.SQLServer	Доступ к функциям управления Microsoft SQL Server
WMPlayer.OCX	Доступ к Windows Media Player
Word.Application	Доступ к Microsoft Office Word
Word.Document	Доступ к документам, открытым в Word

Работать с объектами COM просто, я покажу это на простых примерах с Проводником (Explorer), обозревателем Internet Explorer и Excel. Ниже создается экземпляр объекта Проводника и вызывается его метод Windows(), который показывает текущий каталог для каждого экземпляра Проводника и Internet Explorer:

```
$shell = new-object -comobject shell.application
$shell.windows() | select-object locationname
```

```
Data
Computer
Network
Robert Stanek's Bugville Critters
Robert Stanek - Ruin Mist: The Lost Ages
Windows Nation: Home of Technology Author William Stanek
```

Вывод этого метода передается по конвейеру команде Select-Object LocationName, которая показывает значения свойств LocationName каждого из объектов, представляющих оболочку Windows. При этом выводятся названия папок, открытых в разных экземплярах Проводника, а также названия веб-страниц, открытых в Internet Explorer. Для Internet Explorer 7 и более высоких версий

дополнительно выводится список страниц, открытых на вкладках в окне обозревателя. В этом примере мы получаем сведения и о Проводнике, и об Internet Explorer, поскольку обе эти программы используют оболочку Windows.

Для просмотра всех свойств объектов, представляющих оболочку Windows, нужно направить вывод командлету Select-Object:

```
$shell = new-object -comobject shell.application
$shell.windows() | select-object
```

```
Application          : System.__ComObject
Parent               : System.__ComObject
Container            :
Document             : mshtml.HTMLDocumentClass
TopLevelContainer    : True
Type                 : HTML Document
Left                 : 959
Top                  : 1
Width                : 961
Height               : 1169
LocationName         : Windows Nation: Home of Tech Author William
Stanek
LocationURL          : http://www.williamstanek.com/
Busy                 : False
Name                 : Windows Internet Explorer
HWND                 : 854818
FullName             : C:\Program Files\Internet Explorer\iexplore.exe
Path                 : C:\Program Files\Internet Explorer\
Visible              : True
StatusBar            : True
StatusText           : Done
ToolBar              : 1
MenuBar              : True
FullScreen           : False
ReadyState           : 4
Offline              : False
Silent               : False
RegisterAsBrowser    : False
RegisterAsDropTarget : True
TheaterMode          : False
AddressBar           : True
Resizable            : True
```


Совет Для некоторых объектов COM существуют .NET-оболочки, обеспечивающие взаимодействие этих объектов с .NET Framework. Оболочки могут работать иначе, чем «родные» объекты COM, поэтому New-Object поддерживает параметр –Strict, предупреждающий об использовании оболочки. Если указать этот параметр при вызове New-Object, то при обнаружении оболочки PowerShell выводит предупреждение, но все равно создает объект COM.

Следующий пример открывает в Internet Explorer сайт *www.williamstanek.com*:

```
$iexp = new-object -comobject «InternetExplorer.Application»
$iexp.navigate(«www.williamstanek.com»)
$iexp.visible = $true
```

Здесь создается новый объект COM, представляющий Internet Explorer. Этот объект поддерживает те же свойства, что и объект, представляющий окно оболочки (см. выше). Методу Navigate() этого объекта задает адрес веб-страницы, открытой в обозревателе, а свойство Visible управляет отображением окна. Для просмотра полного списка доступных методов и свойств введите показные выше команды, затем **$iexp | get-member**. Любой из перечисленных методов можно вызвать, а свойство — прочитать или установить.

Так, следующая команда озвучивает через Microsoft Speech API строку текста:

```
$v = new-object -comobject «SAPI.SPVoice»
$v.speak(«Well, hello there. How are you?»)
```

В этом примере создается объект COM, представляющий Speech API и вызывается метод Speak(), «проговаривающий» заданную строку. Чтобы получить полный список доступных методов и свойств, введите показанные выше команды, затем **$v | get-member**.

А этот пример работает с приложением Microsoft Excel:

```
$a = New-Object -comobject «Excel.Application»
$a.Visible = $True
$wb = $a.workbooks.add()
$ws = $wb.worksheets.item(1)

$ws.cells.item(1,1) = «Computer Name»
$ws.cells.item(1,2) = «Location»
$ws.cells.item(1,3) = «OS Type»
$ws.cells.item(2,1) = «TechPC84»
$ws.cells.item(2,2) = «5th Floor»
$ws.cells.item(2,3) = «Windows Vista»

$a.activeworkbook.saveas(«c:\data\myws.xls»)
```

В этом примере создается объект COM, представляющий запущенную программу Excel; установкой свойства Visible этого объекта отображается окно Excel. Экземпляр объекта Excel поддерживает методы и свойства для работы с приложением Excel, а также связанные объекты, представляющие книги, листы и ячейки электронной таблицы. Для просмотра значений, назначенных свойствам объекта Excel по умолчанию, введите показанные выше программы, затем **$a**; чтобы получить список доступных свойств и методов объекта Excel, введите **$a | get-member**.

После создания объекта Excel можно создать новую книгу, вызвав метод Add() объекта Workbooks. В результате создается объект Workbook со своими методами и свойствами, а также массивом связанных объектов Worksheets. Для просмотра значений по умолчанию свойств объекта Workbook, введите **$wb**, а для просмотра его методов и свойств — **$wb | get-member**.

Следующая команда выбирает для обработки только что созданный объект книги Excel, вызывая метод Item() массива объектов Worksheets. В результате создается объект листа Excel (Worksheet) с соответствующими методами, свойствами и связанным массивом объектов Cells. Для просмотра значений, назначенных свойствам объекта Worksheet по умолчанию, введите **$ws**; для просмотра свойств и методов объекта Worksheet введите **$ws | get-member**.

Созданный лист можно заполнить данными, вызывая метод Item() массива Cells. При вызове метода Item() задают координаты ячейки в виде номера строки и столбца, а также значение, которое требуется поместить в ячейку. Для просмотра значений, назначенных свойствам объекта Cells по умолчанию, введите **$ws.cells**; для просмотра методов и свойств объекта Cells введите **$ws.cells | get-member**.

Объекты Cell представляют отдельные ячейки электронных таблиц. Например, для просмотра значений по умолчанию свойств объекта Cell, представляющего первую ячейку в первом столбце, введите **$ws.cells.item(1,1)**. Для просмотра методов и свойств этого объекта введите **$ws.cells.item(1,1) | get-member**.

Работа с объектами и классами .NET Framework

Инфраструктура .NET Framework очень тесно интегрирована с PowerShell, поэтому очень сложно рассказывать о PowerShell, не упоминая .NET. Объекты и классы .NET часто используются в примерах, приведенных в этой и других главах.

Один из способов создания экземпляров классов .NET Framework — прямой вызов базового класса. Для этого следует взять имя класса в квадратные скобки, поставить после него два двоеточия, затем указать имя нужного метода или свойства. Аналогичный метод использовался выше для работы с экземплярами классов [System.Datetime] и [System.Math].

Следующий вызов создает экземпляр класса [System.Environment] и получает текущий каталог:

```
[system.environment]::CurrentDirectory
```

```
C:\data\scripts\myscripts
```

Совет Для просмотра статических членов классов .NET Framework также можно использовать клавишу Tab: введите имя класса (в квадратных скобках), затем два двоеточия (**::**) и нажмите Tab или Shift+Tab для просмотра списка доступных методов и свойств класса.

Ссылку на экземпляр класса .NET Framework можно получить и с помощью командлета New-Object. Вот общий синтаксис соответствующей команды:

```
New-Object [-Set Ассоциативный_массив] [-TypePath Строки] [[-ArgumentList]
Объекты] [-TypeName] Строка
```

Следующая команда создает ссылку на объект System.Diagnostic.Eventlog, представляющий журнал событий Application:

```
$log = new-object -type system.diagnostics.eventlog -argumentlist application
```

```
  Max(K) Retain OverflowAction       Entries Name
  ------ ------ --------------       ------- ----
  20,480      0 OverwriteAsNeeded      45,061 application
```

Совет Получив ссылку на экземпляр класса .NET Framework, можно использовать Tab для просмотра членов этого экземпляра. Например, если объект хранится в переменной $log, введите **$log.** затем точку (.) и нажмите Tab или Shift+Tab для просмотра списка его методов и свойств.

В этой главе мы познакомились лишь с некоторыми из многочисленных классов .NET Framework, в табл. 6-5 перечислены другие полезные .NET-классы.

Табл. 6-5. Классы .NET Framework, наиболее востребованные при работе в PowerShell

Класс	Что представляет и(или) обеспечивает
Microsoft.Win32.Registry	Корневые разделы системного реестра
Microsoft.Win32.RegistryKey	Разделы системного реестра
System.AppDomain	Среда исполнения приложения
System.Array	Массивы и средства для работы с ними
System.Console	Потоки консоли для стандартного ввода-вывода и сообщения об ошибках
System.Convert	Предоставляет статические методы и свойства для преобразования типов
System.Datetime	Дату и время
System.Diagnostics.Debug	Методы и свойства для отладки
System.Diagnostics.EventLog	Журналы событий Windows
System.Diagnostics.Process	Процессы Windows и средства для работы с ними
System.Drawing.Bitmap	Растровые изображения
System.Drawing.Image	Изображения и средства для работы с ними
System.Environment	Информация о рабочей среде и платформе
System.Guid	Глобально уникальные идентификаторы (GUID)
System.IO.Stream	Потоки ввода-вывода

Табл. 6-5. Классы .NET Framework, наиболее востребованные при работе в PowerShell

Класс	Что представляет и(или) обеспечивает
System.Management.Automation.PowerShell	Объект PowerShell, к которому можно добавлять свойства и другие члены
System.Math	Статические методы и свойства для математических вычислений
System.Net.Dns	Средства для взаимодействия с DNS
System.Net.NetworkCredential	Удостоверения для проверки подлинности в сети
System.Net.WebClient	Средства для взаимодействия с веб-клиентами
System.Random	Генерация случайных чисел
System.Reflection.Assembly	Сборки .NET и средства для работы с ними
System.Security.Principal.WellKnownSidType	Идентификаторы защиты (SID)
System.Security.Principal.WindowsBuiltInRole	Встроенные роли системы безопасности
System.Security.Principal.WindowsIdentity	Пользователи Windows
System.Security.Principal.WindowsPrincipal	Проверка членства пользователей в группах
System.Security.SecureString	Зашифрованные строки
System.String	Строки и средства для работы с ними
System.Text.RegularExpressions.Regex	Неизменяемые регулярные выражения
System.Threading.Thread	Потоки и средства для работы с ними
System.Type	Объявления типов
System.Uri	Унифицированные идентификаторы ресурса (URI)
System.Windows.Forms.FlowLayoutPanel	Панели форм
System.Windows.Forms.Form	Окна и диалоговые окна приложений

Использование некоторых .NET-объектов требует предварительной загрузки связанных с ними сборок .NET. Сборка — это просто набор файлов, включающий DLL-библиотеки, EXE-файлы и файлы ресурсов, необходимые для корректной работы объектов. Узнать, что объекту требуются сборки, легко: при обращении к такому объекту при отсутствии нужной сборки PowerShell генерирует ошибку, например:

```
Unable to find type [system.drawing.image]: make sure that the assembly
containing this type is loaded.
At line:1 char:23
```

```
+ [system.drawing.image] <<<< |get-member -static
    + CategoryInfo          : InvalidOperation: (system.drawing.
image:String)
[], RuntimeException
    + FullyQualifiedErrorId : TypeNotFound
```

Решение — в загрузке необходимых сборок с помощью класса [Reflection.Assembly], например, вызовом метода LoadWithPartialName(). Его общий синтаксис имеет следующий вид:

```
[Reflection.Assembly]::LoadWithPartialName(«Класс»)
```

где *Класс* — имя .NET-класса, выполняющего нужную операцию. Например, класс System.Drawing.Bitmap позволяет преобразовать изображение из формата GIF в JPEG. Для вызова этого класса должна быть загружена содержащая его сборка, что можно проверить командой следующего вида:

```
[Reflection.Assembly]::LoadWithPartialName(«System.Windows.Forms»)
```

```
GAC    Version        Location
---    -------        --------
True   v2.0.50727     C:\Windows\assembly\GAC_MSIL\System.Windows.Forms
\2.0.0.0__b77a5c561934e089\System.Windows.For...
```

Этот вывод содержит важную информацию. Значение True в поле GAC говорит о том, что необходимая сборка загружена. Значение Version показывает номер версии .NET Framework, необходимой для сборки, а Location — расположение файла сборки.

Я рекомендую форматировать информацию, возвращаемую методами, использующими отражение, следующим образом:

```
[Reflection.Assembly]::LoadWithPartialName(«System.Windows.Forms») |
format-list
```

```
CodeBase : file:///C:/Windows/assembly/GAC_MSIL/System.Windows.Forms/
2.0.0.0__b77a5c561934e089/System.Windows.Forms.dll
EntryPoint :
EscapedCodeBase : file:///C:/Windows/assembly/GAC_MSIL/System.Windows.
Forms/2.0.0.0__b77a5c561934e089/System.Windows.Forms.dll
FullName  : System.Windows.Forms, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
GlobalAssemblyCache    : True
HostContext            : 0
ImageFileMachine       :
ImageRuntimeVersion    : v2.0.50727
Location               : C:\Windows\assembly\GAC_MSIL\System.Windows.
Forms\2.0.0.0__b77a5c561934e089\System.Windows.Forms.dll
ManifestModule         : System.Windows.Forms.dll
MetadataToken          :
```

```
PortableExecutableKind :
ReflectionOnly         : False
```

Это дает дополнительную важную информацию о только что загруженной сборке, включая ее имя, номер версии, идентификатор культуры и открытый ключ (см. пункт FullName). Скопировав содержимое поля FullName начиная с имени сборки, вы получите готовый аргумент для метода Load(). Это удобно, поскольку сборки лучше загружать именно методом Load(), а метод LoadWithPartialName более не рекомендуется к использованию и, возможно, его поддержка вскоре прекратится.

Вернемся к примеру с преобразованием изображения. Загрузив класс [System.Windows.Forms], можно преобразовать GIF-изображение в JPEG следующим образом:

```
$image = New-Object System.Drawing.Bitmap myimage.gif
$image.Save(«mynewimage.jpg»,»JPEG»)
```

Этот код загружает изображение из файла MyImage.gif в текущем каталоге и преобразует его в формат JPEG. Этот прием годится для любых GIF-изображений. Чтобы узнать ширину, высоту и другие свойства изображения в этом примере, введите **$image**; получить список методов для работы с изображением позволит команда **$image | get-member**.

Работа с объектами и запросами WMI

Компьютеры, работающие под управлением Windows XP и более высокой версии, поддерживают инструментарий управления Windows (Windows Management Instrumentation, WMI). WMI — это инфраструктура управления, которая позволяет запрашивать сведения о компьютере и параметры его конфигурации. Например, с помощью WMI-запросов можно узнать версию операционной системы компьютера и размер его оперативной памяти. WMI-запросы — весьма удобная функция, особенно для использования в сценариях.

Запросы WMI позволяют определять практически любые измеримые параметры компьютеров, включая:

- размер установленной оперативной памяти;
- свободное место на жестком диске;
- тип и тактовую частоту процессора;
- тип и скорость сетевой платы;
- версии операционной системы, пакета обновления и исправлений;
- разделы и параметры системного реестра;
- системные службы, работающие на компьютере.

WMI-запросы пишутся на языке запросов WMI Query Language, их общий синтаксис имеет следующий вид:

```
Select * from WMI_класс where Условие
```

где *WMI_класс* — требуемый класс WMI-объектов, а *Условие* — проверяемое условие. Оператор Select возвращает объекты заданного класса. Условие состоит из трех частей:

- имя проверяемого свойства объекта;
- оператор, например =, > или <;
- значение, с которым сравнивают указанное свойство.

В качестве операторов также допускается указывать –Is и –Like. Оператор –Is определяет точное соответствие заданному критерию. При использовании оператора –Like условие считается выполненным, если искомая строка или ключевое слово встречается в составе проверяемого значения. Ниже показан запрос для поиска компьютеров под управлением Windows Vista:

```
Select * from Win32_OperatingSystem where Caption like «%Vista%»
```

КлассWin32_OperatingSystem отслеживает общую конфигурацию операционной системы. Это первый из двух WMI-классов, часто требующихся администраторам, второй класс — Win32_ComputerSystem — отслеживает общую конфигурацию компьютера.

В Windows PowerShell для получения WMI-объектов служит командлет Get-WMIObject. Вот его общий синтаксис:

```
Get-WmiObject -Class WMI_класс -Namespace Пространство_имен -ComputerName Имя_ПК
```

где *WMI_класс* — WMI-класс, *Пространство_имен* — пространство имен WMI, а *Имя_ПК* — имя нужного компьютера.

При работе с WMI следует использовать корневое пространство имен, для этого параметру –Namespace следует присвоить значение root/cimv2. Параметр –Computer позволяет администратору указать нужный ему компьютер. Чтобы выбрать локальный компьютер, следует ввести точку (.) вместо имени компьютера. Направив вывод запроса командлету Format-List *, вы получите полный список свойств объекта с их значениями.

Вышеописанным способом можно изучить объект Win32_OperatingSystem и его свойства. Чтобы получить сводную информацию о конфигурации операционной системы и компьютера, введите следующую команду:

```
Get-WmiObject -Class Win32_OperatingSystem -Namespace root/cimv2
 -ComputerName . | Format-List *
```

Чтобы сохранить вывод команды в файле, просто перенаправьте его в файл, такой как os_save.txt:

```
Get-WmiObject -Class Win32_OperatingSystem -Namespace root/cimv2
-ComputerName . | Format-List * > os_save.txt
```

Эти сведения позволяют судить о многих аспектах операционной системы, работающей на компьютере. То же самое верно и для конфигурации компьютера, которую можно выяснить с помощью следующей команды:

```
Get-WmiObject -Class Win32_ComputerSystem -Namespace root/cimv2
-ComputerName . | Format-List *
```

Отбирать компьютеры для анализа можно не только по конфигурации операционной системы или оборудования, но и по типу файловой системы и размеру свободного места на дисках. Следующая команда отбирает компьютеры, у которых на диске C, D или G свободно более 100 мегабайт:

```
get-wmiobject -query 'Select * from Win32_LogicalDisk where (Name = «C:»
OR Name = «D:»  OR Name = «G:» ) AND DriveType = 3 AND FreeSpace
> 104857600 AND FileSystem = «NTFS»'
```

Параметр *DriveType = 3* в этом примере задает локальный диск, а значение *FreeSpace* указано в байтах (100 Мб = 104 857 600 байтов). Заданные разделы должны располагаться на локальных жестких дисках с файловой системой NTFS. Учтите, что PowerShell «понимает» разные единицы измерения информации, включая мегабайты и килобайты, а вот язык WMI-запросов поддерживает только байты.

Для вывода полного списка свойств объекта Win32_LogicalDisk введите команду следующего вида:

```
Get-WmiObject -Class Win32_LogicalDisk -Namespace root/cimv2
-ComputerName . | Format-List *
```

У этого объекта много полезных свойств, таких как Compressed, которое указывает, используется ли сжатие на диске. Другие важные классы WMI-объектов перечислены в табл. 6-6.

Табл. 6-6. WMI-классы, часто применяемые при работе в PowerShell

WMI-класс	Что представляет
Win32_BaseBoard	Материнскую плату компьютера
Win32_BIOS	Атрибуты микропрограммы BIOS
Win32_BootConfiguration	Параметры загрузки компьютера
Win32_CacheMemory	Кэш-память
Win32_CDROMDrive	Оптический привод
Win32_ComputerSystem	Windows-компьютер
Win32_Desktop	Общие параметры Рабочего стола
Win32_DesktopMonitor	Тип монитора или другого видеоустройства вывода
Win32_DiskDrive	Физические диски
Win32_DiskPartition	Раздел физического диска
Win32_DiskQuota	Монитор использования места NTFS-томов
Win32_Environment	Параметры окружения
Win32_LogicalDisk	Логический диск
Win32_LogonSession	Текущий сеанс вошедшего в систему пользователя
Win32_NetworkAdapter	Сетевая плата

(*см. след. стр.*)

Табл. 6-6. WMI-классы, часто применяемые при работе в PowerShell

WMI-класс	Что представляет
Win32_NetworkAdapter-Configuration	Конфигурация сетевой платы
Win32_ NetworkConnection	Активное сетевое подключение
Win32_OperatingSystem	Рабочую среду операционной системы
Win32_ OSRecoveryConfiguration	Файлы восстановления и дампа памяти
Win32_PageFileUsage	Страничный файл виртуальной памяти
Win32_PhysicalMemory	Модуль оперативной памяти
Win32_ PhysicalMemoryArray	Все запоминающие устройства компьютера с учетом их емкости и числа
Win32_Printer	Печатающее устройство
Win32_ PrinterConfiguration	Конфигурация печатающего устройства
Win32_PrintJob	Активные задания печати, сгенерированные приложениями
Win32_Processor	Физический или логический процессор (ядро)
Win32_ QuickFixEngineeering	Установленные обновления
Win32_Registry	Системный реестр Windows
Win32_SCSIController	SCSI-контроллер
Win32_Service	Службу, настроенную на компьютере
Win32_Share	Сетевой диск
Win32_SoundDevice	Дискретную или встроенную звуковую плату

Показанные выше методы позволяют анализировать конфигурацию любого из объектов, перечисленных в табл. 6-6, средствами PowerShell. Так, легко убедиться, что у объекта Win32_PhysicalMemoryArray есть свойство MaxCapacity, отслеживающее суммарный размер физической памяти (в Кб). На его основе можно без труда написать WMI-запрос для поиска компьютеров с 256 и более Мб ОЗУ:

```
if (get-wmiobject -query «Select * from Win32_PhysicalMemoryArray where
MaxCapacity > 262000») {write-host $env:computername}
```

```
CORPC87
```

Я указал размер памяти как 262000, поскольку 256 Мб = 262 144 Кб, а нам требуется найти компьютеры, размер памяти которых больше заданного. Комбинируя методы удаленной работы (см. главу 4) и приемы, показанные

в этой главе, вы сможете отыскать в корпоративной сети любой компьютер с заданными параметрами.

Чтобы получить полный список WMI-объектов, введите:

```
Get-WmiObject -list -Namespace root/cimv2 -ComputerName . | Format-List name
```

Это очень длинный список, поэтому лучше перенаправить вывод команды в файл, например в FullWMIObjectList.txt:

```
Get-WmiObject -list -Namespace root/cimv2 -ComputerName . |
Format-List name > FullWMIObjectList.txt
```

При необходимости можно выбрать из WMI-классов только Win32-классы:

```
Get-WmiObject -list | where {$_.name -like «*Win32_*»}
```

Глава 7

Управление компьютерами с помощью команд и сценариев

Сценарии PowerShell представляют собой текстовые файлы с командами. Это те же самые команды, которые обычно вводят в командной строке PowerShell. Если вы сохраните нужные команды в виде сценария, вам не потребуется каждый раз вводить их с клавиатуры — достаточно запустить сценарий, что намного проще.

Сценарии, несомненно, удобны, но администраторам все-таки чаще приходится работать с командной строкой. В PowerShell можно написать команды, состоящие из единственной строки, практически «на все случае жизни». Если же одной строки не хватит, это тоже не беда: консоль PowerShell поддерживает ввод многострочных команд так же, как и в сценариях. Кроме того, консоль PowerShell позволяет копировать и вставлять многострочные команды, эта процедура ничем не отличается от копирования-вставки однострочных команд, но PowerShell исполняет вставленные команды строго поочередно.

Эффективная работа с профилями и сценариями

Поскольку сценарии — это обычные текстовые файлы, их можно редактировать в любом стандартном текстовом редакторе, включая встроенный редактор среды PowerShell ISE. Отдельные команды и группы команд, которые должны исполняться вместе, разделяются при вводе точкой с запятой или просто вводятся с новой строки. Готовый сценарий необходимо сохранить в файл с расширением .ps1.

Сохраненные сценарии можно вызвать как обычные командлеты или внешние программы: просто введите имя сценария и путь к нему, затем нажмите Enter. Далее PowerShell прочитает сценарий и поочередно выполнит все содержащиеся в нем команды. Вывод этих команд отображается на консоли PowerShell, если в сценарии не указано обратное.

Помните, что профили PowerShell — это тоже сценарии, причем одни из самых мощных по функциональности. В профили можно включать объявления псевдонимов, функций и переменных, постоянно требующихся при работе. Конфигурацию командной оболочки для локального пользователя определяют профили $pshome и $home на локальном компьютере, а при удаленной работе действуют одноименные профили, определенные на удаленном компьютере.

Ниже показан пример профиля (листинг 7-1) с определениями функций и псевдонимов. Функция Prompt настраивает приглашение, добавляя в него путь к текущему каталогу и имя компьютера. Функция GetWinRm выводит состояние службы Windows Remote Management, функция GetS выводит список настроенных служб с указанием их состояния (Running или Stopped). Функция Inventory выводит свойства объекта Win32_OperatingSystem. Псевдонимы обеспечивают поддержку коротких имен для следующих функций: gr для GetWinRm, gs для GetS и inv для Inventory.

Листинг 7-1. Пример профиля

```
function prompt {«PS $(get-location) [$env:computername]> »}

new-alias gr getwinrm
new-alias gs gets
new-alias inv inventory
function getwinrm {
   get-service -name winrm | format-list
}

function gets {param ($status)
   get-service | where { $_.status -eq $status}
}

function inventory {param ($name = «.»)
   get-wmiobject -class win32_operatingsystem -namespace root/cimv2 `
    -computername $name | format-list *
}
```

Примечание Третья снизу строка заканчивается обратным апострофом (`). Как сказано в предыдущих главах, этот знак является символом продолжения строки в PowerShell. Если вам по каким-то причинам неудобно пользоваться этим знаком, просто вводите разделенные им команды как единую строку.

Переменные, псевдонимы и функции, объявленные в профиле, всегда доступны, поскольку загружены в глобальную рабочую среду. Конечно, администратору удобно, когда привычные средства работы (а их со временем становится все больше) оказываются «под рукой» сразу после входа в систему. В некоторых случаях этого удается добиться копированием своих профилей на компьютер, с которым вы работаете, например на вверенные вам серверы,

но вряд ли удобно копировать свои профили на все компьютеры организации. Тем не менее, можно воспользоваться профилем и без этого, посвятив немного времени подготовительным операциям.

Вот эти операции: отредактируйте копию профиля, объявив его элементы как глобальные (листинг 7-2), затем сохраните измененный профиль в сетевую папку, доступную другим компьютерам. После этого для загрузки элементов из профиля в глобальное окружение достаточно запустить профиль как сценарий (не забудьте указать полный путь к сценарию, например \\FileServer84\DataShare\wrstanek\profile.ps1).

Листинг 7-2. Профиль, настроенный для использования на удаленных компьютерах

```
function global:prompt {«PS $(get-location) [$env:computername]> »}

new-alias gr getwinrm -scope global
new-alias gs gets -scope global
new-alias inv inventory -scope global

function global:getwinrm {
   get-service -name winrm | format-list
}

function global:gets {param ($status)
   get-service | where { $_.status -eq $status}
}

function global:inventory {param ($name = «.»)
   get-wmiobject -class win32_operatingsystem -namespace root/cimv2 -
computername $name | format-list *
}
```

Учтите, что для успешного применения этого приема политика исполнения должна разрешать исполнение сценариев, загруженных из удаленных источников. Если политика исполнения требует, чтобы такие сценарии имели цифровую подпись, сценарий потребуется еще и подписать перед запуском. Кроме того, перед запуском сценария PowerShell может показать такое предупреждение:

```
 Security Warning
Run only scripts that you trust. While scripts from the Internet can be
useful, this script can potentially harm your computer. Do you want to
run \\192.168.1.252\wrs\profileold.ps1?
[D] Do not run  [R] Run once  [S] Suspend  [?] Help (default is «D»):
```

Чтобы запустить сценарий, нажмите **R** и Enter; подробнее о политиках исполнения см. в главе 1.

Журналы исполнения

Консоль PowerShell поддерживает регистрацию действий при исполнении сценариев в *журналах исполнения* (transcripts); графическая среда PowerShell ISE еще не поддерживала журналы исполнения к моменту выхода этой книги. Для работы с журналами используются следующие командлеты:

- **Start-Transcript —** открывает файл журнала и регистрирует в нем все действия, совершаемые в сеансе PowerShell.

```
Start-Transcript [[-path] Путь] [-force] [-noClobber] [-append]
```

- **Stop-Transcript —** останавливает ведение журнала и закрывает его файл.

```
Stop-Transcript
```

Командлет Start-Transcript включает ведение журнала PowerShell. При этом создается текстовый файл, в который записываются все команды, которые вводятся в командной строке, вместе с их выводом, отображаемым на консоли. Общий синтаксис Start-Transcript выглядит так:

```
Start-Transcript [[-path] Путь]
```

где *Путь* — путь к файлу журнала, в котором нельзя использовать подстановочные знаки, но можно использовать переменные. Если хоть один из каталогов в заданном пути не существует, исполнение команды завершается неудачей.

Примечание Если путь не задан, Start-Transcript открывает файл журнала в каталоге, заданном глобальной переменной $Transcript, а если эта переменная не объявлена — в каталоге $Home\[My]Documents. Файлы журнала получают имена вида PowerShell_transcript.*Дата_Время*.txt, где *Дата_Время* — дата и время создания журнала.

Параметр –Force позволяет обойти ограничения, препятствующие удачному завершению команды. Например, –Force переопределяет файловый атрибут Read-Only, но не может изменить параметры защиты и разрешения на доступ к файлу.

По умолчанию Start-Transcript перезаписывает существующие журналы без предупреждения. Параметр –noClobber запрещает перезапись, а параметр –Append позволяет дописать новые данные к концу существующего файла.

Чтобы остановить ведение журнала, закройте консоль или введите **Stop-Transcript**. Командлет Stop-Transcript вызывают без параметров.

Совет PowerShell инициализирует новые журналы, вставляя в них заголовок следующего вида:

```
**********************
Windows PowerShell Transcript Start
Start time: 20100211134826
Username  : CPANDL\Bubba
```

```
Machine  : TECHPC85 (Microsoft Windows NT 6.0.6001
Service Pack 1)
**********************
Transcript started, output file is C:\Users\Bubba\Documents\
PowerShell_transcript.20100211134826.txt
```

В заголовке отмечается время начала ведения журнала; пользователь, открывший журнал, и полный путь к файлу журнала. Заметьте, что имя пользователя указывается в формате ДОМЕН\Имя_пользователя или Имя_компьютера\Имя_пользователя, а сведения о компьютере включают версию Windows и пакета обновления. Эта информация полезна при устранении неполадок, поскольку позволяет быстро выявлять случаи запуска сценария в неверном контексте пользователя и в несовместимых версиях Windows. При остановке ведения журнала PowerShell вставляет строки следующего вида с указанием времени остановки:

```
**********************
Windows PowerShell Transcript End
End time: 20090211134958
**********************
```

Разность между временем начала и окончания ведения журнала представляет собой время исполнения сценария либо время отладки в командной строке.

Транзакции в PowerShell

Транзакцией (transaction) называют группу команд, исполняемых как единый блок. Только при успешном завершении всех команд блока транзакция считается успешной, при сбое любой из команд выполняется отмена или откат (rollback) всех остальных команд блока. Транзакции применяют, когда требуется гарантированное исполнение всех команд блока во избежание повреждения и данных и нестабильного состояния компьютера.

Введение в транзакции

Транзакции — одно из мощнейших средств для работы с реляционными базами данных, вычислительными средами и PowerShell. Почему? А потому, что для транзакции возможны лишь два исхода: успешное внесение заданных изменений либо, в случае сбоев, полный откат с восстановлением исходного состояния рабочей среды.

Для восстановления исходного состояния необходимо отслеживать изменения, которые вносят команды в составе транзакции, а также сохранения исходных значений и данных. Таким образом, для поддержки транзакций необходима особая функциональность, которая имеется не у всех команд. Поддержка транзакций в PowerShell может быть реализована на двух уровнях:

- на уровне **поставщика** командлетов;
- на уровне отдельных **командлетов**.

Из основных компонентов PowerShell в Windows Vista и выше транзакции поддерживает лишь поставщик Registry, а именно его командлеты New-

Item, Set-Item, Clear-Item, Copy-Item, Move-Item и Remove-Item. Кроме того, класс System.Management.Automation.TransactedString позволяет выполнять группы команд как транзакции в любых версиях Windows, поддерживающих PowerShell. Добавить поддержку транзакций к другим поставщикам можно путем их обновления. Для поиска поставщиков, совместимых с транзакциями, используйте следующую команду:

```
get-psprovider | where {$_.Capabilities -like «*transactions*»}
```

Очень упрощенно работу транзакций можно описать так:

1. Вы запускаете транзакцию.
2. PowerShell выполняет команды, составляющие транзакцию.
3. Транзакция подтверждается или откатывается.

В составе транзакции могут быть «вложенные» транзакции, которые обрабатываются как отдельная команда транзакции и называются *подписчиками* (subscribers). Например, для работы с базой данных в PowerShell используется транзакция, которая может запускать отдельные транзакции для обработки наборов данных. Успешное или неудачное завершение «вложенных» транзакций определяет, соответственно, успех или неудачу «родительской» транзакции, из которой они были запущены. Следовательно, до подтверждения «родительской» транзакции необходимо подтвердить «вложенные» транзакции.

Для работы с транзакциями используют следующие командлеты:

- **Get-Transaction** — получает объект, представляющий транзакцию, исполняемую в данный момент (активную транзакцию) или последнюю выполненную транзакцию в текущем сеансе. Позволяет определить настройки отката, число подписчиков и состояние транзакции.

  ```
  Get-Transaction
  ```

- **Complete-Transaction** — подтверждает активную транзакцию, применяя и сохраняя внесенные ей изменения. Для полного подтверждения транзакции с подписчиками этот командлет необходимо вызвать для каждого подписчика.

  ```
  Complete-Transaction
  ```

- **Start-Transaction** — запускает новую транзакцию (независимую либо «вложенную» транзакцию-подписчик). Параметр отката по умолчанию — Error (откат при ошибке). Время ожидания для транзакций, запущенных из командной строки, по умолчанию не задано; для транзакций, запущенных из сценария, период ожидания по умолчанию составляет 30 мин.

  ```
  Start-Transaction [-Independent] [-RollbackPreference {Error |
  TerminatingError | Never}] [-Timeout Минуты]
  ```

- **Undo-Transaction** — откатывает активную транзакцию, отменяя изменения, внесенные во время ее исполнения, и восстанавливая исходное со-

стояние рабочей среды. Если у транзакции, для которой вызван Undo-Transaction, есть транзакции-подписчики, они также откатываются.

```
Undo-Transaction
```

- **Use-Transaction** — добавляет к активной транзакции блок сценария, обеспечивая поддержку транзакций для сценариев с использованием объектов .NET Framework, таких как System.Management.Automation.TransactedString. В блоке сценария, исполняемом в составе транзакции, разрешается использовать только совместимые объекты. Для входа в активную транзакцию необходимо использовать параметр –UseTransaction, иначе команда не даст результата.

```
Use-Transaction [-UseTransaction] [-TransactedScript] Блок_сценария
```

Совет У командлетов, поддерживающих транзакции, имеется параметр –UseTransaction. Поскольку перечень таких параметров меняется от версии к версии PowerShell, лучше вывести их список с помощью команды **get-help * -parameter UseTransaction**. Только учтите, что некоторые командлеты, поддерживающие транзакции (например, командлеты поставщика Registry для работы с элементами реестра) могут и не фигурировать в справке как командлеты с параметром –UseTransaction.

При использовании транзакций для изменения конфигурации компьютеров следует иметь в виду, что изменения в конфигурацию вносятся только после подтверждения транзакции. Большинство систем, поддерживающих транзакции, обеспечивают блокировку данных во время их изменения, но в PowerShell такой функции нет. Поэтому во время исполнения транзакции те же самые параметры конфигурации могут изменять и другие команды, что может отражаться на состоянии рабочей среды.

В сеансе PowerShell допустимо существование только одной активной транзакции в каждый момент времени. При запуске новой независимой транзакции запущенная транзакция становится активной, и внесение изменений в прежней транзакции становится возможным только после подтверждения или отката новой активной транзакции.

Если транзакция завершается успешно, то после подтверждения все изменения, внесенные командами этой транзакции, применяются к рабочей среде. Если же транзакция завершается неудачей, все внесенные ее командами изменения отменяются, и восстанавливается исходное состояние рабочей среды. По умолчанию, транзакция автоматически откатывается, если хоть одна из ее команд вызовет ошибку.

Работа с транзакциями

Чтобы запустить новую независимую или «вложенную» транзакцию, введите **start-transaction** в командной строке или сценарии PowerShell. Если командлет Start-Transaction запущен во время исполнения транзакции, для новой транзакции используется существующий объект транзакции, при этом число его подписчиков увеличивается на один. Другими словами, новая транзакция становится «вложенной» или, как говорят, подписчиком

(subscriber) прежней транзакции. Чтобы завершить транзакцию, у которой имеется несколько подписчиков, необходимо выполнить команду Complete-Transaction для каждого из подписчиков.

Примечание Поддержка вложенных транзакций или подписчиков в PowerShell предусмотрена для ситуаций, когда транзакция, запущенная из сценария, запускает транзакцию из другого сценария. Поскольку эти транзакции связаны, они должны подтверждаться либо откатываться как единое целое.

Командлет Start-Transaction поддерживает три параметра: –Independent, –RollbackPreference и –Timeout. Параметр –Independent действует, только если в сеансе уже запущена транзакция. В этом случае новая транзакция, запущенная с использованием этого параметра, будет независимой, то есть, ее можно будет завершить или откатить независимо от исходной транзакции. Но, поскольку допустимо существование только одной активной транзакции, предыдущая транзакция может быть продолжена только после подтверждения или отката новой транзакции.

В следующем примере запускается транзакция, для которой отключен автоматический откат и задан период ожидания, равный 30 мин.:

```
start-transaction -rollbackpreference «never» -timeout 30
```

```
Suggestion [1,Transactions]: Once a transaction is started, only
commands that get called with the -UseTransaction flag become part of
that transaction.
```

Параметр –RollbackPreference управляет автоматическим откатом транзакций. Его допустимые значения:

- **Error —** транзакция автоматически откатывается при возникновении любых ошибок (это значение по умолчанию);
- **TerminatingError —** транзакция автоматически откатывается только при возникновении ошибки, прерывающей исполнение (terminating error);
- **Never —** автоматический откат транзакции запрещен.

Параметр –Timeout задает максимальный период (в минутах) активности транзакции, по истечении которого транзакция автоматически откатывается. Для транзакций, запущенных из командной строки, период ожидания по умолчанию не задан, а для транзакций, запущенных из сценария, он составляет 30 минут.

После запуска транзакции можно исполнять команды в ее составе двумя способами:

- добавляя к активной транзакции блоки сценария с помощью Use-Transaction;
- добавляя к активной транзакции отдельные команды, вызывая их с параметром –UseTransaction.

PowerShell исполняет блоки сценария и команды по мере их добавления к транзакции, а также выполняет заданные действия при возникновении

ошибок и истечении периода ожидания. Для получения сведений о транзакциях служит командлет Get-Transaction:

```
get-transaction
```

```
RollbackPreference   SubscriberCount   Status
------------------   ---------------   ------
Never                3                 Active
```

Видно, что автоматический откат этой транзакции запрещен (Rollback Preference = Never), у нее 3 подписчика, и она является активной (Status = Active).

Если PowerShell не может автоматически откатить транзакцию из-за ошибки или по тайм-ауту, транзакцию можно подтвердить или откатить вручную. Чтобы подтвердить транзакцию, введите **complete-transaction** для каждого подписчика, а чтобы полностью откатить ее, достаточно одной команды **undo-transaction**.

Рассмотрим следующий пример:

```
start-transaction

cd hkcu:\Software
new-item MyKey -UseTransaction
new-itemproperty -path MyKey -Name Current -value «Windows PowerShell» `
-UseTransaction

complete-transaction
```

В этом примере запускается транзакция, обеспечивающая безопасность работы с реестром. Далее открывается куст HKCU\Software, в нем создается новый раздел MyKey, а в нем — новый параметр. Если при исполнении этих действий не возникнет ошибок, эта транзакция остается активной, и после подтверждения транзакции внесенные изменения применяются к реестру.

Общие элементы сценариев

Итак, мы познакомились с основными приемами работы со сценариями, а теперь перейдем к общим элементам сценариев, таким как:

- комментарии;
- инициализирующие команды;
- условные конструкции;
- циклы.

Инициализирующие команды и комментарии

Большинство сценариев начинается с комментариев, в которых сообщается, для чего предназначен этот сценарий и как им пользоваться. PowerShell поддерживает два типа комментариев:

- Однострочные — они начинаются знаком #. Все, что находится между этим знаком и концом строки, PowerShell интерпретирует как комментарий. Вот пример:

```
$myVar = «$env:computername» #Получаем имя компьютера
```

Примечание В строках, заключенных в двойные или одинарные кавычки, знак не интерпретируется как литерал, а не как начало комментария.

- Многострочные, которые открываются строкой <# и закрываются строкой #>. Открытый комментарий обязательно должен быть закрыт. Все, что находится между открывающей и закрывающей строками, PowerShell интерпретирует как комментарий. Вот пример многострочного комментария:

```
<# -------------------------

   Имя сценария: EvaluateComp.ps1
   Описание: этот сценарий проверяет рабочую среду
компьютера на предмет недостатка места на дисках,
сбоев сетевых подключений и других неполадок.

------------------------- #>
```

В каждом сценарии должны быть комментарии, содержащие:

- дату создания сценария и внесения последних изменений;
- имя автора;
- описание назначения сценария;
- способ связи с автором;
- описание способа вывода данных сценарием.

Эта информация (см. листинг 7-3) не только облегчит другим администраторам работу с вашими сценариями, но и позволит без труда вспомнить назначение этого сценария даже спустя долгое время.

Листинг 7-3. Пример заголовка сценария

```
 <# -------------------------
   Имя сценария: CheckDNS.ps1
   Дата создания: 2/28/2010
   Дата последнего изменения: 3/15/2010
   Автор: William R. Stanek
   E-mail: williamstanek@aol.com
   ************************
   Описание: проверка конфигурации DNS и IP.
   ************************
   Выходной файл: c:\data\checkdns.txt.
------------------------- #>
```

Совет Не забывайте, что комментарии также позволяют:

1. Вставлять в текст сценария пояснения, например, с описанием функций.
2. Полностью блокировать исполнение команды. Например, чтобы «закомментировать» однострочную команду, поставьте перед ней знак #.
3. Блокировать исполнение части команды. Чтобы заставить PowerShell игнорировать часть командной строки, поставьте знак # перед этой частью.

После заголовка имеет смысл разместить код, инициализирующий консоль, чтобы при каждом запуске сценария получать рабочую среду с идентичной конфигурацией. Например, можно использовать команду Cls (или Clear-Host) для очистки окна консоли и экранного буфера, а также команду Start-Transcript для регистрации вывода сценария в журнале (только не забудьте закрыть журнал командой Stop-Transcript).

Во время инициализации также можно настраивать размер, цвета текста и фона, а также заголовок окна PowerShell, например, так:

```
if ($host.name -eq «ConsoleHost») {
$size=New-Object System.Management.Automation.Host.Size(120,80);
$host.ui.rawui.WindowSize=$size }

$myHostWin = $host.ui.rawui
$myHostWin.ForegroundColor = «Blue»
$myHostWin.BackgroundColor = «Yellow»
$myHostWin.WindowTitle = «Working Script»
```

Этот код получает экземпляр объекта System.Management.Automation.Host, который затем используется для настройки ширины и высоты окна (120 строк по 80 знаков) консоли PowerShell. Далее с помощью свойств объекта $host.ui.rawui устанавливается синий текст на желтом фоне и текст заголовка окна (Working Script). Подробнее о работе с объектами — в главе 6.

Примечание При динамической настройке размера окна консоли PowerShell необходимо учитывать текущее разрешение монитора и действующие размеры окна. Не задавайте для окна размеры, превышающие размеры экрана. При попытке назначения консоли ширины, превышающей размеры буфера, вы получите ошибку. Графическая среда PowerShell ISE не поддерживает такой способ настройки размеров окна. Поэтому перед попыткой установки размеров рекомендуется проверять свойство $Host.Name: оно содержит строку «Windows PowerShell ISE Host», если вы работаете с графической средой, или «ConsoleHost», если вы работаете с консолью PowerShell.

При инициализации сценария также имеет смысл проверить, что:

- сценарий запущен на компьютере с совместимой версией PowerShell;
- хост-приложением действительно является Windows PowerShell;
- доступны необходимые оснастки, поставщик и модули PowerShell.

Для этих целей в PowerShell предусмотрена директива #Requires, которая позволяет проверить:

- версию PowerShell командой вида:

```
#requires -Version N[.n]
```

где *N* — номер версии, *n* — номер сборки (необязательный параметр). Для проверки наличия PowerShell версии 2.0 введите такую команду:

```
#requires -version 2
```

или такую:

```
#requires -version 2.0
```

- идентификатор хост-приложения командой вида:

```
#requires -ShellId ShellId
```

где *ShellID* — идентификатор хост-приложения. Идентификатор *Microsoft.PowerShell* соответствует консоли или графической среде PowerShell: #requires –ShellId «Microsoft.PowerShell»

- доступность в сеансе нужной оснастки и даже определенной ее версии командой:

```
#requires -PsSnapIn PsSnapIn [-Version N[.n]]
```

где *PSSnapIn* — идентификатор нужной оснастки, а *N* и *.n* значения, составляющие номер версии, например:

```
#requires -PsSnapIn ADRMS.PS.Admin -Version 2
```

При использовании директивой #Requires сценарий будет запущен только на компьютере, соответствующем требованиям, заданным директивной #Requires (подробнее см. в главе 3).

Подобно большинству командлетов и внешних утилит, сценарии могут принимать параметры. Параметры используются для управления работой сценариев, а также для передачи им необходимой информации. При запуске сценария список аргументов, разделенных пробелами (взятый, при необходимости, в двойные кавычки), следует после имени сценария. В следующем примере сценарию Check-Computer, запускаемому из текущего рабочего каталога, передаются аргументы FileServer26 и Extended:

```
.\check-computer fileserver26 extended
```

Переданные сценарию аргументы содержатся в массиве $args. Ссылка на первый элемент этого массива имеет вид $args[0], ссылка на второй аргумент — $args[1], и т.д. Имя сценария хранится как значение свойства $MyInvocation.MyCommand.Name, а полный путь к нему — в свойстве $MyInvocation.MyCommand.Path property.

Примечание Поскольку PowerShell хранит аргументы в виде массива, число аргументов сценария не ограничено. Кроме того, независимо от числа аргументов, ссылка на последний аргумент имеет вид $args[$arg.length - 1], а общее число аргументов представлено значением $args.count; если аргументы сценарию не передавались, оно равно нулю.

Работа с условными конструкциями

Условные конструкции применяются для управления ходом выполнения сценариев в зависимости от тех или иных обстоятельств, известных только во время выполнения. Поддерживаются следующие условные конструкции:

- **If…Else** — исполняет один блок команд, если выполняется определенное условие, и другой блок, если это условие не выполняется;
- **If…ElseIf… Else** — исполняет один блок команд, если выполняется определенное условие, другой блок, если выполняется второе условие, либо третий блок, если не выполняется ни первое, ни второе условие;
- **If Not** — исполняет заданный блок команд, если проверка условия дает False; если это условие не выполняется, этот блок игнорируется;
- **Switch** — выполняет альтернативные операции в зависимости от проверки трех и более условий.

Конструкции If, If…Else и If…ElseIf…Else

Условные конструкции уже использовались в некоторых из приведенных выше примеров, а теперь мы разберем поподробнее их синтаксис. Гибкость и мощность условных конструкций впечатляют, особенно тех, кто не занимался программированием раньше.

Ключевое слово If используется для условного ветвления и направляет исполнение сценария по двум альтернативным маршрутам. Вот его общий синтаксис:

```
if (условие) {блок_кода1} [else {блок_кода2}]
```

Каждый блок кода здесь содержит одну или несколько команд. В качестве условия может выступать любое выражение, которое дает True или False. Часть Else — необязательная, ее синтаксис имеет следующий вид:

```
if (условие1) {блок_кода1}
```

Конструкция If работает так: если проверка условия дает True (условие выполняется), выполняются команды *блок_кода1*, а когда условие не выполняется, т.е. при его проверке получается False — *блок_кода2* (в том случае, если определена конструкция Else). Код из обеих ветвей (If и Else) сразу не исполняется ни при каких обстоятельствах. Часто в условных конструкциях используется проверка равенства. Простейший пример — сравнение двух строк с помощью оператора =, например:

```
if (Строка_A = Строка_B) {блок_кода}
```

В этом примере строки сравниваются как литералы; если они идентичны, исполняется заданный блок кода. Такое сравнение работает в случае литералов, но идеально для использования в сценариях. Значения параметров, свойств и аргументов нередко содержат пробелы, а в некоторых случаях — неопределенные переменные. В подобных обстоятельствах сравнение лите-

ралов дает ошибку. Поэтому рекомендуется использовать интеллектуальные методы сравнения, например, с помощью операторов –eq, –like, –match и –contains.

Рассмотрим эти методы на примере:

```
if ($args.count -eq 0) {throw «No arguments passed to script»}
else {write-host «You passed args to the script»}
```

```
   No arguments passed to script
   At C:\Users\Bubba\dat.ps1:8 char:24
   + if ($args.count) {throw <<<<  «No arguments passed to script»}
       + CategoryInfo          : OperationStopped: (No arguments passed to
   script:String) [], RuntimeException
       + FullyQualifiedErrorId : No arguments passed to script
```

При вызове этого сценария без аргументов возникает ошибка. Поскольку PowerShell по умолчанию прерывает исполнение при возникновении ошибок, это удобный прием для прекращения работы сценария при несоответствии реальных условий ожидаемым. Если сценарию переданы аргументы, исполнение идет по ветви Else и сценарий выводит «You passed args to the script».

Альтернативный способ — использовать условную конструкцию для ввода недостающих аргументов с помощью командлета Read-Host, как в следующем примере:

```
if ($args.count -eq 0) {
   $compName = read-host «Enter the name of the computer to check»
} else {
   $compName = $args[0]
}
```

```
   Enter the name of the computer to check: FileServer84.cpandl.com
```

Если этот сценарий вызван без аргумента, командлет Read-Host запрашивает аргумент у пользователя и после ввода сохраняет его в переменной $compName. В противном случае переменной $compName присваивается значение первого аргумента, переданного сценарию.

Использование логических операторов (табл. 5-7) позволяет проверять два условия. Так, в следующем примере условие If выполняется, только если оба выражения, связанных через AND, дают True:

```
if (($args.count -ge 1) -and ($args[0] -eq «Check»)) {
   write-host «Performing system checks...»
 } else {
   write-host «Script will not perform system checks...»
}
```

```
    .\check-sys.ps1 Check
   Performing system checks...
```

Кроме того, PowerShell поддерживает конструкцию If...ElseIf...Else, которая исполняет один блок кода, если выполняется первое условие, второй блок — если выполняется второе условие, либо третий блок, если не выполняется ни одно из этих условий. Вот общий синтаксис этой конструкции:

```
if (условие_1) {блок_кода1} elseif (условие_2) {блок_кода2} else {блок_кода3}
```

В следующем примере конструкция If...ElseIf...Else позволяет выполнить разные действия в зависимости от значения первого аргумента сценария:

```
if (($args.count -ge 1) -and ($args[0] -eq «Check»)) {
   write-host «Performing system checks...»
 } elseif (($args.count -ge 1) -and ($args[0] -eq «Test»)) {
   write-host «Performing connectivity tests...»
} else {
   write-host «Script will not perform system checks or tests.»
}
```

```
   .\check-sys.ps1 Test
   Performing connectivity tests...
```

Исполнить некоторую команду в ситуации, когда определенное условие дает False, позволяет конструкция If Not, которую можно записать так:

```
if (!условие) {блок_кода1} [else {блок_кода2}]
```

или так:

```
if (-not (условие)) {блок_кода1} [else {блок_кода2}]
```

Если в этом примере проверка условия даст False, PowerShell исполняет первый блок кода, в противном случае исполняется второй блок, если он определен. Конструкция Else — необязательная, то есть, допустима и следующая запись:

```
if (!условие) {блок_кода1}
```

Рассмотрим следующий пример:

```
if (!$args.count -ge 1) {
   read-host «Enter the name of the computer to check»
}
```

Этот сценарий исполняет заданный код, если ему передано ни одного аргумента.

Совет Конструкция If, размещенная вну три другой к онструкции If, называется вложенной условной конструкцией. Вложенные условные конструкции часто применяются в программировании, включая написание сценариев PowerShell. Вложенные конструкции If…Else (или If…ElseIf…Else) вместе с соответствующим кодом помещают вместо блоков кода «родительской» условной конструкции.

Конструкция Switch

Проверка множества условий с помощью конструкции If...ElseIf...Else трудоемка и громоздка, проще сделать это с использованием конструкции Switch. Конструкция Switch позволяет проверить сразу несколько условий, при этом ее код понятен и прост. К ней также можно добавить блок Default, содержащий код, который будет исполнен, если не выполняется ни одно из условий.

Общий синтаксис конструкции Switch имеет следующий вид:

```
switch (выражение) {
   значение1 { блок_кода1 }
   значение2 { блок_кода2 }
   значение3 { блок_кода3 }
. . .
   значениеN { блок_кодаN }
   default { код_по_умолчанию }
```

В сущности, конструкция Switch — это расширенный набор конструкций If, принимающий проверяемое значение. Если переданное значение соответствует одному из вариантов, определенных в конструкции Switch, исполняется соответствующий блок кода, после чего PowerShell покидает конструкцию Switch. Если переданы дополнительные значения, PowerShell проверяет их поочередно, выполняя те же действия.

Примечание В конструкции Switch разрешается определять не бо лее одного блока кода по умолчанию, иначе возникает ошибка.

Конструкцию Switch можно использовать с любыми допустимыми типами данных. Если проверяемое значение является массивом чисел, строк или объектов, все элементы массива проверяются поочередно, начиная с нулевого элемента. В этом случае в массиве должен быть хотя бы один элемент, соответствующий одному из условий, иначе PowerShell генерирует ошибку. В следующем примере объявляется массив $myValue с четырьмя значениями, который обрабатывается с помощью конструкции Switch:

```
$myValue = 4, 5, 6, 0

switch ($myValue) {
   0 { write-host «The value is zero.»}
   1 { write-host «The value is one.»}
   2 { write-host «The value is two.»}
   3 { write-host «The value is three.»}
   default { write-host «The value doesn't match expected parameters.»}
}

The value doesn't match expected parameters.
The value doesn't match expected parameters.
```

```
The value doesn't match expected parameters.
The value is zero.
```

Обнаружив одно выполняющееся условие и исполнив соответствующий код, PowerShell продолжает проверять все остальные условия. Чтобы выйти из конструкции Switch сразу после исполнения кода, связанного с выполняющимся условием, следует добавить к концу соответствующего блока ключевое слово Break:

```
switch (выражение) {

   значение1 { блок_кода1 break}
   значение2 { блок_кода2 break}
   значение3 { блок_кода3 break}
. . .
   значениеN { блок_кодаN break}
   default { код_по_умолчанию }
```

Вот пример конструкции Switch с оператором Break:

```
$myError = «Green», «Red», «Yellow», «Green»

switch ($myError) {
   «Red» { write-host «A critical error occurred. Break.»; break}
   «Yellow» { write-host «A warning occurred. Break.»; break}
   «Green» { write-host «No error occurred yet. No break.» }
   default { write-host «The values don't match expected parameters.»}
}
```

```
   No error occurred yet. No break.
   A critical error occurred. Break.
```

По умолчанию, при проверке условий в конструкции Switch регистр символов не учитывается. Для управления проверкой используются дополнительные флаги:

- **–regex —** проверка соответствия строки результату регулярного выражения. Если проверяемое значение не является строкой, данный параметр игнорируется; не используется с флагами –wildcard и –exact;
- **–wildcard —** включает использование подстановочных знаков. Если проверяемое значение не является строкой, данный параметр игнорируется; не используется с флагами – regex и –exact;
- **–exact —** требует точного совпадения проверяемых строк. Если проверяемое значение не является строкой, данный параметр игнорируется; не используется с флагом – regex;
- **–casesensitive —** включает проверку с учетом регистра. Если проверяемое значение не является строкой, данный параметр игнорируется;
- **–file —**заставляет Switch загружать аргументы из файла, а не принимать их от команды. При этом каждая строка, прочитанная из файла, проверя-

ется конструкцией Switch отдельно. В файле должна быть хотя бы одна строка, соответствующая хотя бы одному условию, иначе PowerShell генерирует ошибку.

Ниже показан пример сценария, проверяющего состояние службы, имя которой задано первым аргументом сценария либо введено пользователем:

```
if (!$args.count -ge 1) {
   $rh = read-host «Enter the name of the service to check»
   $myValue = get-service $rh
} else {
   $myValue = get-service $args[0]
}

$serName = $myValue.Name

switch -wildcard ($myValue.Status) {
   «S*» { write-host «The $serName service is stopped.»}
   «R*» { write-host «The $serName service is running.»}
   «P*» { write-host «The $serName service is paused.»}
   default { write-host «Check the service.»}
}
```

```
Enter the name of the service to check: w32time
The W32Time service is running.
```

Циклы

Циклы используются для многократного исполнения одной и той же команды или набора команд. *Цикл* (loop) также является средством управления потоком исполнения команд. В PowerShell поддерживаются следующие циклы:

- For;
- ForEach;
- While;
- Do While;
- Do Until.

Простейшим циклом является цикл For, который позволяет исполнить блок кода заданное число раз. Вот общая структура цикла For:

```
for (начальное_значение; условие; шаг) { тело_цикла }
```

где *начальное_значение* — оператор, задающий начальное значение счетчика цикла, *условие* — условие, определяющее выход из цикла по достижении конечного значения счетчика, *шаг* — оператор, задающий шаг цикла, а *тело* — повторно исполняемый код. Заметьте, что элементы в фигурных скобках разделяются точкой с запятой, а следующее счетчика вычисляется после каждого оборота цикла, т.е. перед следующим оборотом.

В следующем примере начальное значение счетчика равно 1, оно увеличивается на 1 при каждом обороте цикла, который «крутится», пока значение счетчика не достигнет 10:

```
for ($c=1; $c -le 10; $c++){write-host $c}
```

```
1
2
3
4
5
6
7
8
9
10
```

В следующем примере счетчик цикла уменьшается с 10 до 0 с шагом 1:

```
for ($c = 10; $c -ge 0; $c--) {Write-Host $c}
```

```
10
9
8
7
6
5
4
3
2
1
0
```

Так же просто создавать циклы с шагом 2, 3 и т.п.:

```
for ($c=1; $c -le 100; $c += 2){write-host $c}
for ($c=1; $c -le 100; $c += 3){write-host $c}

for ($c = 20; $c -ge 0; $c -= 2) {Write-Host $c}
for ($c = 20; $c -ge 0; $c -= 3) {Write-Host $c}
```

Другой тип циклов, ForEach, используется для перебора групп элементов, чаще всего — массивов. Цикл ForEach похож на стандартный цикл For, отличаясь ли тем, что число оборотов цикла ForEach определяется числом обрабатываемых элементов. Вот его общий синтаксис:

```
ForEach ( Элемент in Набор) { тело_цикла }
```

где *Элемент* — переменная, которую PowerShell создает автоматически при исполнении цикла ForEach, *Набор* — группа элементов для перебора, кото-

рая может поступать прямо из конвейера. В следующем примере цикл используется для обработки набора объектов-процессов:

```
foreach ($p in get-process) {
   if ($p.handlecount -gt 500) {
      Write-Host $p.Name, $p.pm }
}
```

```
   aolsoftware 14766080
   csrss 1855488
   csrss 22507520
   explorer 34463744
```

Для каждого процесса данный код проверяет число открытых описателей файлов. Если оно превышает 500, выводится имя процесса и размер его закрытой памяти. Этот прием удобен для выявления процессов, использующих много системных ресурсов.

Следующий пример демонстрирует перебор файлов в цикле:

```
if (!$args.count -ge 1) {
   $path = read-host «Enter the name of the base directory to check»
} else {
   $path = $args[0]
}

foreach ($file in Get-ChildItem -path $path -recurse) {
    if ($file.length -gt 1mb) {
      $size = [Math]::Round($file.length/1MB.ToString(«F0»))
      Write-Host $file, $size, $file.lastaccesstime }
}
```

```
   A Catalog Section 1.pdf 8095845 11/23/2009 8:10:16 PM
   A Catalog Section 2.pdf 12021788 11/23/2009 8:10:16 PM
```

Данный код ищет в заданном каталоге и вложенных в него папках все файлы размером больше мегабайта и выводит для них имя, размер в Мб и время последнего обращения. Этот метод позволяет выявлять большие файлы, которые давно не используются.

Совет Иногда требуется выйти из цикла For или ForEach до его завершения. Для этого можно воспользоваться оператором Break, который рекомендуется помещать в блоки кода условных конструкций If, If…Else или If…ElseIf…Else.

Иногда требуется повторять исполнение некоторого кода до тех пор, пока выполняется некоторое условие. Это делается с помощью цикла While:

```
while (условие) {тело_цикла}
```

Этот цикл будет крутиться, пока выполняется заданное условие. Для выхода из него необходимо изменить значение, проверяемое условием, в теле цикла, например:

```
$x = 0
$continuetoggle = $true
while ($continuetoggle) {
   $x = $x + 1
   if ($x -lt 5) {write-host «x is less than 5.»}
   elseif ($x -eq 5) {write-host «x equals 5.»}
   else { write-host «exiting the loop.»
     $continuetoggle = $false }
}
```

```
   X is less than 5.
   X is less than 5.
   X is less than 5.
   X is less than 5.
   X equals 5.
   Exiting the loop.
```

Поместив проверку условия в начало тела цикла, вы гарантируете исполнение кода в теле цикла только при выполнении заданного условия. В предыдущем примере это означает, что код в теле цикла не будет выполнен, если переменная continueToggle заранее будет установлена в False.

В некоторых случаях требуется, чтобы цикл совершил хотя бы один оборот до проверки условия. Для таких ситуаций предусмотрен цикл Do While, где проверка условия выполняется до исполнения кода в теле цикла:

```
do {тело_цикла} while (условие)
```

Код в теле следующего цикла будет исполнен как минимум один раз:

```
$x = 0
$continuetoggle = $true
do { $x = $x + 1
   if ($x -lt 5) {write-host «x is less than 5.»}
   elseif ($x -eq 5) {write-host «x equals 5.»}
   else { write-host «exiting the loop.»
     $continuetoggle = $false }
}
while ($continuetoggle)
```

```
   X is less than 5.
   X is less than 5.
   X is less than 5.
   X is less than 5.
   X equals 5.
   Exiting the loop.
```

Последний из типов циклов PowerShell — Do Until. В отличие от Do While, он «крутится» *до тех пор*, пока не будет выполнено заданное условие,

при этом (как и в случае Do While) проверка условия выполняется после исполнения кода в теле цикла. Вот общий синтаксис цикл Do Until:

```
do {тело_цикла} until (условие)
```

В этом примере цикл «крутится», пока не будет выполнено заданное условие:

```
do {
   $cont = read-host «Do you want to continue? [Y/N]»
} until ($cont -eq «N»)
```

Глава 8

Управление ролями, службами ролей и компонентами

Тем, кто работает с Windows Server 2008 и выше, доступно куда больше возможностей настройки, чем в Windows Vista или Windows 7. После установки сервер с Windows Server 2008 настраивают путем установки и конфигурирования следующих компонентов:

- **Серверные роли** Серверные роли и соответствующие им наборы программного обеспечения позволяют серверам предоставлять определенные сервисы пользователям и компьютерам в сети. Серверу может быть назначена единственная роль, например поддержка файловых служб (File Services), либо несколько ролей.
- **Службы ролей** Службы ролей — это программные компоненты, поддерживающие функции серверных ролей. Некоторые роли включают единственную функцию, обеспечивающий ее компонент устанавливается при установке этой роли. Большинство же серверных ролей предполагает использование нескольких взаимосвязанных служб, которые можно выбрать при установке роли.
- **Дополнительные компоненты** Эти компоненты предоставляют дополнительные возможности и устанавливаются независимо от ролей и служб ролей. В зависимости от конфигурации, на компьютере может быть установлено различное число дополнительных компонентов.

Обычно для управления ролями, службами ролей и компонентов используется административная утилита командной строке ServerManagerCmd либо GUI-утилита Server Manager, для тех же целей можно использовать командную оболочку PowerShell 2.0 и выше, подключив к ней модуль ServerManager.

Введение в Server Manager

Для управления конфигурацией серверов с помощью PowerShell необходимо подключить к консоли модуль ServerManager. Этот модуль позволяет не только добавлять и удалять роли, службы ролей и компонентов; его команд-

леты также позволяют получать сведения об их конфигурации и состоянии этих программных элементов. Чтобы подключать модуль ServerManager при каждом запуске, добавьте в свой профиль следующую команду:

```
import-module servermanager
```

Модуль ServerManager поддерживается в Windows Server 2008. Этот модуль обладает рядом преимуществ по сравнению с GUI-утилитой Server Manager версий 6.0 и 6.1, доступных в Windows Server 2008 и Windows Server 2008 Release 2, соответственно. Например, возможен запуск сразу нескольких экземпляров модуля ServerManager для одновременного добавления и удаления компонентов; в случае Server Manager 6.1 это ограничение удается обойти только путем обновления. Возможность одновременного запуска нескольких экземпляров Server Manager действительно полезна: так, в одном сеансе Server Manager можно использовать для добавления ролей, а в другом — для удаления компонентов.

Совет Дополнительные компоненты для Windows Server 2008 и выше, такие как Windows Media Server 2008 и Windows SharePoint Server 2008, можно загружать с веб-сайта Майкрософт. Для установки и настройки этих компонентов средствами Server Manager необходимо загрузить их в виде пакетов обновлений Microsoft Update Standalone Package (.msu-файлов). Зарегистрировать загруженный пакет и подготовить его к использованию можно двойным щелчком.

Команды Server Manager

Для работы с ServerManager необходимо запускать консоль PowerShell с администраторскими полномочиями. После этого можно использовать для управления ролями, службами ролей и компонентами следующие командлеты:

- **Get-WindowsFeature** — выводит текущее состояние ролей, служб ролей и компонентов на сервере.

  ```
  Get-WindowsFeature [[-Name] Компонент] [-LogPath Журнал.txt]
  ```

- **Add-WindowsFeature** — устанавливает заданную роль, службу роли или компонент. Параметр –IncludeAllSubFeature позволяет установить заданный элемент со всеми подчиненными службами и компонентами.

  ```
  Add-WindowsFeature [-Name] Компонент [-IncludeAllSubFeature]
  [-LogPath Журнал.txt] [-Restart] [-Concurrent]
  ```

- **Remove-WindowsFeature** — удаляет заданную роль, службу роли или компонент.

  ```
  Remove-WindowsFeature [-Name] Компонент [-LogPath Журнал.txt] [-Restart]
  [-Concurrent]
  ```

  При необходимости используют следующие параметры:

- **–LogPath** — чтобы регистрировать ошибки в альтернативном файле журнала;
- **–Restart** — чтобы автоматически перезагрузить компьютер, если перезагрузка необходима для завершения установки;
- **–Concurrent** — чтобы разрешить одновременную установку и удаление компонентов разным экземплярам модуля;
- **–WhatIf** — чтобы сымитировать заданную команду.

Вышеперечисленные команды принимают следующие параметры:

- **компонент** — задает реальное имя (не отображаемое имя) роли, службы роли или компонента. Параметр –Name работает с «настоящими», а не отображаемыми именами компонентов. Командлет Get-WindowsFeature поддерживает подстановочные знаки, а Add-WindowsFeature и Remove-WindowsFeature принимают имена компонентов через конвейер от других командлетов, таких как Get-WindowsFeature;
- **LogFile.txt** — задет имя и путь к файлу журнала ошибок.

У большинства ролей, служб ролей и компонентов имеется имя, по которому на них можно ссылаться в командах PowerShell, это верно и для компонентов, загружаемых с веб-сайта Майкрософт.

Доступные роли и службы ролей

В табл. 8-1 перечислены имена ролей, а также имена связанных с ними служб и компонентов. Для установки ролей и служб вместе с подчиненными компонентами используется параметр –IncludeAllSubFeature.

Примечание В табл. 8-1 ниже надстрочным индексом[1] обозначены переработанные роли, доступные в Windows Server 2008 R2 и выше, индексом[2] — устаревшие и более не доступные роли.

Табл. 8-1. Имена ключевых ролей и служб ролей

Имя Роль	Служба Компонент
AD-Certificate	Active Directory Certificate Services (Службы сертификации Active Directory)
ADCS-Cert-Authority	Certification Authority (Центр Сертификации)
ADCS-Web-Enrollment	Certification Authority Web Enrollment (Служба подачи заявок в центр сертификации через Интернет)
ADCS-Online-Cert	Online Responder (Сетевой ответчик)
ADCS-Device-Enrollment	Network Device Enrollment Service (Служба подачи заявок на сетевые устройства)
ADCS-Enroll-Web-Svc	Certificate Enrollment Web Service
ADCS-Enroll-Web-Pol	Certificate Enrollment Policy Web Service

Табл. 8-1. Имена ключевых ролей и служб ролей

Имя Роль	Служба	Компонент
AD-Domain-Services	Active Directory Domain Services, AD DS (Доменные службы Active Directory)	
ADDS-Domain-Controller	Active Directory Domain Controller (Контроллер домена Active Directory)	
ADDS-Identity-Mgmt	Identity Management for UNIX (Диспетчер удостоверений для UNIX)	
ADDS-NIS		Server for Network Information Services (NIS) [Средства сервера для NIS]
ADDS-Password-Sync		Password Synchronization (Синхронизация паролей)
ADDS-IDMU-Tools		Administration Tools (Средства администрирования)
AD-Federation-Services	Active Directory Federation Services, AD FS (Службы федерации Active Directory)	
ADFS-Federation	Federation Service (Служба федерации)	
ADFS-Proxy	Federation Service Proxy (Прокси-агент службы федерации)	
ADFS-Web-Agents	AD FS Web Agents (Веб-агенты AD FS)	
ADFS-Claims		Claims-Aware Agent (Агент, поддерживающий утверждения)
ADFS-Windows-Token		Windows Token-Based Agent (Агент Windows на основе маркеров)
ADLDS	Active Directory Lightweight Directory Services, AD LDS (Службы облегченного доступа к каталогам Active Directory)	
ADRMS	Active Directory Rights Management Services (Службы управления правами Active Directory)	
ADRMS-Server	Active Directory Rights Management Services Server (Сервер службы управления правами Active Directory)	
ADRMS-Identity	Identity Federation Support (Поддержка федерации удостоверений)	
DHCP	Dynamic Host Configuration Protocol (DHCP) Server (DHCP-сервер)	
DNS	Domain Name System (DNS) Server (DNS-сервер)	
Fax	Fax Server (Факс-сервер)	
File-Services	File Services (Файловые службы)	

(*см. след. стр.*)

Табл. 8-1. Имена ключевых ролей и служб ролей

Имя Роль	Служба	Компонент
FS-FileServer	File Server (Файловый сервер)	
FS-DFS	Distributed File System (Распределенная файловая система)	
FS-DFS-Namespace		DFS Namespace (Пространство имен DFS)
FS-DFS-Replication		DFS Replication (Репликация DFS)
FS-Resource-Manager	File Server Resource Manager (Диспетчер ресурсов файлового сервера)	
FS-NFS-Services	Services for Network File System (NFS) [Службы для NFS]	
FS-Search-Service	Windows Search Service (Служба поиска Windows)	
FS-Win2003-Services	Windows Server 2003 File Services (Файловые службы Windows Server 2003)	
FS-Replication		File Replication Service[2] (Служба репликации файлов)
FS-Indexing-Service		Indexing Service (Служба индексирования)
FS-BranchCache	BranchCache for Remote Files ()	
Hyper-V	Hyper-V	
NPAS	Network Policy and Access Services (Службы сетевой политики и доступа)	
NPAS-Policy-Server	Network Policy Server (Сервер сетевой политики)	
NPAS-RRAS-Services	Routing and Remote Access Services (Службы маршрутизации и сетевого доступа)	
NPAS-RRAS		Remote Access Service (Служба удаленного доступа)
NPAS-Routing		Routing (Маршрутизация)
NPAS-Health	Health Registration Authority (Центр регистрации работоспособности)	
NPAS-Host-Cred	Host Credential Authorization Protocol (Протокол авторизации учетных данных узла)	
Print-Services	Print and Document Services	
Print-Server	Print Server (Сервер печати)	
Print-LPD-Service	LPD Service (Служба LPD)	
Print-Internet	Internet Printing (Печать через Интернет)	

Табл. 8-1. Имена ключевых ролей и служб ролей

Имя Роль	Служба Компонент
Print-Scan-Server	Distributed Scan Management Server
Remote-Desktop-Services	Remote Desktop Services
RDS-RD-Server	Remote Desktop Server (Сервер удаленного рабочего стола)
RDS-Licensing	Remote Desktop (RD) Services Licensing
RDS-Connection-Broker	RD Connection Broker
RDS-Gateway	RD Gateway
RDS-Web-Access	RD Web Access
RDS-Virtualization	RD Virtualization
WDS	Windows Deployment Services (Службы развертывания Windows)
WDS-Deployment	Deployment Server (Сервер развертывания)
WDS-Transport	Transport Server (Транспортный сервер)
OOB-WSUS	Windows Server Update Services (Службы обновления Windows Server)

Доступные компоненты

В табл. 8-2 перечислены имена компонентов, включая починенные компоненты. Для установки компонента вместе с подчиненными компонентами служит параметр –IncludeAllSubFeature.

Примечание Звездочкой (*) помечены компоненты, которые обычно устанавливают с подчиненными компонентами при помощи параметра –IncludeAllSubFeature. В табл. 8-2 надстрочным индексом[1] обозначены переработанные роли, доступные в Windows Server 2008 R2 и выше; индексом[2] — устаревшие и более не доступные роли, а индексом[3] — компоненты, встроенные в операционную систему.

Табл. 8-2. Имена основных и подчиненных компонентов

Имя	Компонент	Подчиненный компонент	Подчиненный компонент третьего уровня
NET-Framework*	.NET Framework 3.5.1 Features		
BitLocker*	BitLocker Drive Encryption (Шифрование диска BitLocker)		
BITS	Background Intelligent Transport Services (BITS) Server Extensions (Серверные расширения BITS)		
BranchCache	BranchCache		
CMAK	Connection Manager Administration Kit (Пакет администрирования диспетчера подключений)		

(*см. след. стр.*)

Табл. 8-2. Имена основных и подчиненных компонентов

Имя	Компонент	Подчиненный компонент	Подчиненный компонент третьего уровня
Desktop-Experience	Desktop Experience (Возможности рабочего стола)		
DAMC	Direct Access Management Console		
Failover-Clustering	Failover Clustering (Средство отказоустойчивости кластеров)		
GPMC	Group Policy Management Console (Консоль управления групповыми политиками)		
Ink-Handwriting*	Ink and Handwriting Services		
Internet-Print-Client	Internet Printing Client (Клиент Интернет-печати)		
ISNS	Internet Storage Name Server (Служба имен хранилищ интернета)		
LPR-Port-Monitor	LPR Port Monitor (Монитор порта LPR)		
MSMQ*	Message Queuing (Очередь сообщений)		
Multipath-IO	Multipath I/O (Многопутевой ввод-вывод)		
NLB	Network Load Balancing (Балансировка сетевой нагрузки)		
PNRP	Peer Name Resolution Protocol (Протокол PNRP)		
qWave	Quality Windows Audio Video Experience (qWave)		
Remote-Assistance	Remote Assistance (Удаленный помощник)		
RDC	Remote Differential Compression (Удаленное разностное сжатие)		
RSAT	Remote Server Administration Tools (Средства удаленного администрирования сервера)		
RSAT-Role-Tools	Role Administration Tools (Средства администрирования ролей)		
RSAT-ADCS*		Active Directory Certificate Services Tools (Средства служб сертификации Active Directory)	
RSAT-AD-Tools		AD DS and AD LDS Tools	
RSAT-ADDS*			Active Directory Domain Services Tools (Средства доменных служб Active Directory)
RSAT-ADLDS			Active Directory Lightweight Directory Services Tools (Средства облегченного доступа к каталогам Active Directory)

Табл. 8-2. Имена основных и подчиненных компонентов

Имя	Компонент	Подчиненный компонент	Подчиненный компонент третьего уровня
RSAT-AD-PowerShell			Оснастка Active Directory PowerShell
RSAT-RMS		Active Directory Rights Management Services Tools (Средства служб управления правами Active Directory)	
RSAT-DHCP		DHCP Server Tools (Средства DHCP-сервера)	
RSAT-DNS-Server		DNS Server Tools (Средства DNS-сервера)	
RSAT-Fax		Fax Server Tools (Средства факс-сервера)	
RSAT-File-Services*		File Services Tools (Средства файловых служб)	
RSAT-NPAS*		Network Policy and Access Services Tools (Средства служб сетевой политики и доступа)	
RSAT-Print-Services		Print Services Tools (Средства служб печати)	
RSAT-RDS*		Remote Desktop Services Tools	
RSAT-UDDI		Universal Description, Discovery, and Integration (UDDI) Services Tools[2] (Средства служб UDDI)	
RSAT-Web-Server		Web Server (Internet Information Services, IIS) Tools (Средства служб IIS)	
RSAT-WDS		Windows Deployment Services Tools (Средства служб развертывания Windows)	
RSAT-Hyper-V		Hyper-V Tools (Средства Hyper-V)	
RSAT-Feature-Tools		Feature Administration Tools (Средства администрирования возможностей)	
RSAT-BitLocker*			BitLocker Drive Encryption Tools (Средства шифрования дисков BitLocker)
RSAT-BITS-Server			BITS Server Extensions Tools (Средства серверных расширений BITS)
RSAT-Clustering			Failover Clustering Tools (Средства отказоустойчивости кластеров)
RSAT-NLB			Network Load Balancing Tools (Средства балансировки сетевой нагрузки)

(*см. след. стр.*)

Табл. 8-2. Имена основных и подчиненных компонентов

Имя	Компонент	Подчиненный компонент	Подчиненный компонент третьего уровня
RSAT-SMTP			Simple Mail Transfer Protocol (SMTP) Server Tools (Средства SMTP-сервера)
RSAT-WINS			Windows Internet Naming Service (WINS) Server Tools (Средства WINS-сервера)
Removable-Storage	Removable Storage Manager[2] (Диспетчер съемных носителей)		
RPC-over-HTTP-Proxy	RPC over HTTP Proxy (Прокси RPC через HTTP)		
Simple-TCPIP	Simple TCP/IP Services (Простые службы TCP/IP)		
SMTP-Server	Simple Mail Transfer Protocol (SMTP) Server (SMTP-сервер)		
SNMP-Services*	Simple Network Management Protocol Services (Службы SNMP)		
Storage-Mgr-SANS	Storage Manager for SANs (Диспетчер хранилища для сетей SAN)		
Subsystem-UNIX-Apps	Subsystem for UNIX-Based Applications (Подсистема для UNIX-приложений, SUA)		
Telnet-Client	Telnet Client (Клиент Telnet)		
Telnet-Server	Telnet Server (Сервер Telnet)		
TFTP-Client	Trivial File Transfer Protocol (TFTP) Client (TFTP-клиент)		
Biometric-Framework	Windows Biometric Framework		
Windows-InternalDB	Windows Internal Database (Внутренняя база данных Windows)		
PowerShell	Windows PowerShell[3]		
Backup-Features	Windows Server Backup Features (Возможности системы архивации данных Windows)		
Backup		Windows Server Backup (Архивации Windows Server)	
Backup-Tools		Command-Line Tools (Средства командной строки)	
Migration	Windows Server Migration Tools		
WSRM	Windows System Resource Manager (Диспетчер системных ресурсов)		
WinRM-IIS-Ext	WinRM IIS Extension (Расширение WinRM IIS)		

Табл. 8-2. Имена основных и подчиненных компонентов

Имя	Компонент	Подчиненный компонент	Подчиненный компонент третьего уровня
WINS-Server	WINS Server (WINS-сервер)		
Wireless-Networking	Wireless Local Area Network (LAN) Service (Служба беспроводной сети)		
XPS-Viewer	XML Paper Specification (XPS) Viewer (Средство просмотра XPS)		

Проверка установленных ролей, служб ролей и компонентов

Перед настройкой сервера следует изучить его текущую конфигурацию и тщательно спланировать свои действия с учетом влияния установки и удаления ролей, служб ролей и компонентов на работу сервера. Администраторы нередко стараются комбинировать несколько ролей на одном сервере, но это повышает нагрузку на сервер и потому требует установки соответствующего оборудования.

Чтобы выяснить, какие роли, службы ролей и компоненты установлены на сервере, введите **Get-WindowsFeature** в командной строке консоли PowerShell, запущенной с обычными или администраторскими полномочиями. Вы получите список доступных ролей, служб ролей и компонентов с указанием их конфигурации, а также установленных компонентов (см. листинг ниже), первыми в этом списке идут роли и службы ролей, а после них — компоненты:

```
get-windowsfeature
```

```
Display Name                                            Name
------------                                            ----
[ ] Active Directory Certificate Services               AD-Certificate
    [ ] Certification Authority                         ADCS-Cert-Authority
    [ ] Certification Authority Web Enrollment          ADCS-Web-Enrollment
    [ ] Online Responder                                ADCS-Online-Cert
    [ ] Network Device Enrollment Service               ADCS-Device-
Enrollment
    [ ] Certificate Enrollment Web Service              ADCS-Enroll-Web-Svc
    [ ] Certificate Enrollment Policy Web Service       ADCS-Enroll-Web-Pol
[X] Active Directory Domain Services                    AD-Domain-Services
    [X] Active Directory Domain Controller              ADDS-Domain-
Controller
    [X] Identity Management for UNIX                    ADDS-Identity-Mgmt
        [X] Server for Network Information Services     ADDS-NIS
        [X] Password Synchronization                    ADDS-Password-Sync
```

```
        [X] Administration Tools                    ADDS-IDMU-Tools
[X] Active Directory Federation Services            AD-Federation-
Services
    [X] Federation Service                          ADFS-Federation
    [X] Federation Service Proxy                    ADFS-Proxy
    [X] AD FS Web Agents                            ADFS-Web-Agents
        [X] Claims-aware Agent                      ADFS-Claims
        [X] Windows Token-based Agent               ADFS-Windows-Token
[ ] Active Directory Lightweight Directory Services  ADLDS
[X] Active Directory Rights Management Services     ADRMS
    [X] Active Directory Rights Management Server   ADRMS-Server
    [X] Identity Federation Support                 ADRMS-Identity

. . .

[X] .NET Framework 3.5.1 Features                   NET-Framework
    [X] .NET Framework 3.5.1                        NET-Framework-Core
    [X] WCF Activation                              NET-Win-CFAC
        [X] HTTP Activation                         NET-HTTP-Activation
        [X] Non-HTTP Activation                     NET-Non-HTTP-Activ
[ ] Background Intelligent Transfer Service (BITS)  BITS
    [ ] Compact Server                              BITS-LWDLServer
    [ ] IIS Server Extension                        BITS-IIS-Ext
[ ] BitLocker Drive Encryption                      BitLocker
```

Для проверки состояния отдельных групп компонентов разрешается использовать в именах подстановочные знаки. Например, для проверки компонентов, связанных с Active Directory, введите **Get-WindowsFeature -name ad*** или **Get-WindowsFeature ad***.

Чтобы сохранить сведения о конфигурации серверов, направьте вывод Get-WindowsFeature в файл, например так:

```
Get-WindowsFeature > ServerConfig03-21-2010.txt
```

Установка ролей, служб ролей и компонентов

Для установки ролей, служб ролей и компонентов средствами PowerShell используется модуль ServerManager. При установке ролей, служб и компонентов, требующих для своей работы дополнительные компоненты, Server Manager запрашивает разрешение на добавление этих компонентов.

Процедура установки

Для установки ролей, служб ролей и компонентов следует открыть консоль с администраторскими полномочиями и ввести в командной строке **add-windowsfeature *Компонент***, где *Компонент* — имя устанавливаемого ком-

понента (см. табл. 8-1 и 8-2). Для установки ролей и служб вместе с подчиненными компонентами используйте параметр –IncludeAllSubFeature как в следующем примере:

```
add-windowsfeature fs-dfs -IncludeAllSubFeature
```

```
Success   Restart Needed   Exit Code   Feature Result
-------   --------------   ---------   -------------
True      No               Success     {DFS Replication, DFS Namespaces}
```

В этом примере устанавливается служба роли Distributed File System вместе с подчиненными службами DFS Namespaces и DFS Replication. Во время выполнения команды PowerShell отображает индикатор хода операции, а после ее завершения выводит результаты (см. предыдущий пример). Из результатов видно, чем закончилась установка (успехом или неудачей), требуется ли перезагрузка компьютера, также приводится код завершения и полный список внесенных изменений. Коды успешного завершения могут различаться. Например, при попытке установки уже установленных компонентов возвращается код NoChangeNeeded, например:

```
add-windowsfeature –name net-framework -includeallsubfeature
```

```
Success   Restart Needed   Exit Code       Feature Result
-------   --------------   ---------       -------------
True      No               NoChangeNeeded  {}
```

Из вывода этой команды видно, что она завершилась успешно, но никаких изменений при этом внесено не было (см. также поле Feature Result).

Командлет Add-WindowsFeature может принимать информацию от других команд через конвейер, например:

```
get-windowsfeature bits* | add-windowsfeature
```

```
WARNING: [Installation] Succeeded: [Background Intelligent Transfer
Service (BITS)] Compact Server. You must restart this server to finish
the installation process.
Success   Restart Needed   Exit Code               Feature Result
-------   --------------   ---------               -------------
True      Yes              SuccessRestartRequired  { BITS-LWDLServer,
BITS-IIS-Ext }
```

Здесь Add-WindowsFeature выводит список подлежащих установке компонентов, полученный от командлета Get-WindowsFeature (службу BITS и подчиненные службы Compact Server и IIS Server Extension). Поскольку для завершения установки нужна перезагрузка, команда выводить соответствующее предупреждение.

Чтобы выполнить перезагрузку автоматически, вызывайте Add-Windows Feature с параметром –Restart, но учтите, что автоматическая переза-

грузка может прервать установку или удаление других компонентов. Соответствующее сообщение об ошибке выводится вместе со стандартным выводом:

```
Add-WindowsFeature: Please restart the computer before trying to install
more roles/features.

Success   Restart Needed   Exit Code              Feature Result
-------   --------------   ---------              -------------
False     Yes              FailedRestartRequired  { }
```

Обработка ошибок при установке

Некоторые компоненты невозможно установить из командной строки. Попытка сделать это оканчивается выводом предупреждения:

```
get-windowsfeature ad-fe* | add-windowsfeature
```

```
WARNING: Installation of 'Active Directory Federation Services' is not
supported on the command line. Skipping . . .
Success   Restart Needed   Exit Code        Feature Result
-------   --------------   ---------        -------------
True      No               NoChangeNeeded   { }
```

Для проверки результатов установки перед ее выполнением используется параметр –WhatIf. При попытке установить уже установленные компоненты выводится сообщение о том, что изменения не были внесены:

```
get-windowsfeature ad-d* | add-windowsfeature -whatif
```

```
What if: Checking if running in 'WhatIf' Mode.
Success   Restart Needed   Exit Code       Feature Result
-------   --------------   ---------       -------------
True      No               NoChangeNeeded  {}
```

Если команда завершается неудачей, Add-WindowsFeature выводит сообщение об ошибке (как правило, оно красного цвета) с ее флагом и описанием:

```
The term 'add-windowsfeature' is not recognized as a cmdlet, function,
operable program, or script file. Verify the term and try again.
At line:1 char:19
+ add-windowsfeature <<<<  fs-dfs
    + CategoryInfo          : ObjectNotFound: (add-windowsfeature:String)
[], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
```

В этом примере PowerShell не «узнает» командлет Add-WindowsFeature, поскольку консоли не был подключен модуль ServerManager командой import-module servermanager.

Также распространены ошибки из-за недостаточного уровня привилегий у консоли PowerShell:

```
Add-WindowsFeature: Because of security restrictions imposed by User
Account Control you must run Add-WindowsFeature in a Windows PowerShell
session opened with elevated rights.
```

Чтобы устранить такую ошибку, щелкните правой кнопкой ярлык PowerShell и выберите Run As Administrator (Запуск от имени Администратора).

Add-WindowsFeature записывает подробную информацию о каждой операции при установке компонентов в журнал %SystemRoot%\logs\servermanager.log. Чтобы указать другой файл журнала, используйте параметр –LogPath. Так, в следующем примере для журнала задан файл C:\logs\install.log:

```
add-windowsfeature BITS -IncludeAllSubFeature -LogPath
c:\logs\install.log
```

Поскольку PowerShell возвращает вывод в качестве объектов, при необходимости можно передавать его через конвейер другим командлетам для альтернативного форматирования, например:

```
get-windowsfeature net-* | add-windowsfeature | format-list *
```

```
Success       : True
RestartNeeded : No
FeatureResult : { }
ExitCode      : NoChangeNeeded
```

Удаление ролей, служб ролей и компонентов

Для удаления ролей, служб ролей и компонентов также используется модуль ServerManager. Удалять компоненты, необходимые для работы других ролей, служб и компонентов, возможно только вместе с зависимыми компонентами. При попытке удаления компонента, от которого зависят другие службы и роли, Server Manager предупредит, что удалить этот компонент можно только вместе с зависимыми ролями, службами и компонентами.

Процедура удаления

Чтобы удалить роль, службу роли или компонент, откройте консоль с административными полномочиями и введите в командной строке **remove-windowsfeature *Компонент***, где *Компонент* — имя удаляемого компонента (см. табл. 8-1 и 8-2). При удалении компонента автоматически удаляются все его подчиненные компоненты. Рассмотрим следующий пример:

```
remove-windowsfeature net-framework
```

```
WARNING: [Removal] Succeeded: [Background Intelligent Transfer Service
(BITS)] Compact Server. You must restart this server to finish the
removal process.
Success   Restart Needed   Exit Code              Feature Result
-------   --------------   ---------              -------------
True      Yes              SuccessRestartRequired {Compact Server, IIS
Server Extension}
```

Эта команда удаляет службу BITS вместе с подчиненными службами Compact Server и IIS Server Extension. Во время исполнения команды PowerShell выводит индикатор хода операции, а после ее завершения — результаты. В данном примере показаны результаты успешного завершения операции. Если для завершения удаления необходима перезагрузка, выводится соответствующее предупреждение.

Чтобы выполнить перезагрузку автоматически, вызывайте Remove-WindowsFeature с параметром –Restart. Для проверки результатов удаления, как и при установке, используется параметр –WhatIf. При попытке удалить отсутствующий компонент выводится сообщение о том, что изменения не были внесены.

```
remove-windowsfeature net-framework -whatif
```

```
What if: Checking if running in 'WhatIf' Mode.
Success   Restart Needed   Exit Code       Feature Result
-------   --------------   ---------       -------------
True      No               NoChangeNeeded  {}
```

Remove-WindowsFeature может принимать ввод от других командлетов через конвейер:

```
get-windowsfeature fs-* | remove-windowsfeature
```

```
Success   Restart Needed   Exit Code    Feature Result
-------   --------------   ---------    -------------
True      No               Success      {DFS Replication, DFS Namespaces}
```

В этом примере Remove-WindowsFeature получает список подлежащих удалению компонентов от командлетов Get-WindowsFeature, удаляя службу Distributed File System с подчиненными службами.

Обработка ошибок при удалении

Если вызов Remove-WindowsFeature завершается неудачей, выводится сообщение об ошибке, похожее на сообщения об ошибках при установке. В некоторых случаях удаление компонентов оказывается невозможным, обычно по причине зависимости от этого компонента других ролей, служб и компонентов. Рассмотрим следующий пример:

```
get-windowsfeature fs-* | remove-windowsfeature
```

```
Remove-WindowsFeature : DFS Replication and DFS Namesspace cannot be
removed from a domain controller.

Success   Restart Needed   Exit Code       Feature Result
-------   --------------   ---------       -------------
False     No               Failed          {}
```

Эта попытка удаления службы Distributed File System с подчиненными службами оканчивается неудачей, поскольку эта служба необходима на контроллере домена.

Remove-WindowsFeature записывает подробную информацию о каждой операции при установке компонентов в журнал %SystemRoot%\logs\servermanager.log. Чтобы указать другой файл журнала, используйте параметр –LogPath. Так, в следующем примере для журнала задан файл C:\logs\uninstall.log:

```
remove-windowsfeature net-framework -logpath c:\logs\uninstall.log
```

Глава 9

Инвентаризация и диагностика Windows-компьютеров

Администраторам нередко требуется анализировать конфигурацию удаленных рабочих станций и серверов. Например, иногда необходимо узнать, кто вошел в систему, определить текущее системное время и список локальных учетных записей пользователей, какой процессор и сколько оперативной памяти установлено на компьютере. Получение этой и другой информации о компьютерах называется инвентаризацией. Эта процедура позволяет не только определять конфигурацию оборудования, но и выявлять неполадки, требующие вмешательства администратора. Так, вовремя узнав о нехватке оперативной памяти или места на жестком диске, администратор может принять соответствующие меры.

Получение основных сведений о системе

Иногда администратор работает с компьютером, не имея о нем даже основной информации, такой как имя компьютера, домена входа и записи текущего пользователя. Это возможно, когда его вызывают для устранения неполадок на компьютере другого отдела, а также при удаленной работе. В подобных случаях можно быстро собрать необходимую информацию, используя следующие команды:

- $env:computername — выводит имя компьютера:

```
$env:computername
```

- $env:username — выводит имя пользователя, вошедшего в систему:

```
$env:username
```

- $env:userdomain — выводит имя домена входа пользователя:

```
$env:userdomain
```

- Get-Date — выводит системную дату и время:

```
Get-Date [-Date] Дата_и_время
```

```
Get-Date -DisplayHint [Date | Time]
```

- Set-Date — задает системное время и дату:

```
Set-Date [-Date] Дата
Set-Date [-Adjust] Временной_сдвиг
```

- Get-Credential — получает у пользователя удостоверения, необходимые для аутентификации.

```
Get-Credential [-Credential] Удостоверения
```

Определение имени текущего пользователя, компьютера и домена

Часто удается получить необходимую информацию о рабочей среде путем опроса переменных окружения. Чаще всего это имена компьютера, текущего пользователя и домена входа, которые содержатся в переменных $env:username, $env:userdomain и $env:computername, соответственно. Вот как вывести эту информацию на консоль:

```
write-host «Domain: $env:userdomain 'nUser: $env:username 'nComputer:
$env:computername»
```

```
Domain: CPANDL
User: wrstanek
Computer: TECHPC76
```

В этом примере wrstanek — имя пользователя, TechPC76 — имя компьютера, а CPANDL — домен входа пользователя.

Альтернативный способ хорош тем, что позволяет быстрее получить более подробную информацию о рабочей среде компьютера, включая все переменные окружения с их значениями:

```
get-childitem env:
```

```
Name                           Value
----                           -----
ALLUSERSPROFILE                C:\ProgramData
APPDATA                        C:\Users\WilliamS\AppData\Roaming
CommonProgramFiles             C:\Program Files\Common Files
COMPUTERNAME                   TECHPC125
ComSpec                        C:\Windows\system32\cmd.exe
FP_NO_HOST_CHECK               NO
HOMEDRIVE                      C:
HOMEPATH                       \Users\WilliamS
LOCALAPPDATA                   C:\Users\WilliamS\AppData\Local
LOGONSERVER                    \\ENGPC42
```

```
NUMBER_OF_PROCESSORS          4
OS                            Windows_NT
Path                          C:\Windows\system32;C:\Windows;
PATHEXT                       .COM;.EXE;.BAT;.CMD;.VBS; JS;.PSC1
PROCESSOR_ARCHITECTURE        x86
PROCESSOR_IDENTIFIER          x86 Family GenuineIntel
PROCESSOR_LEVEL               6
PROCESSOR_REVISION            0f07
ProgramData                   C:\ProgramData
ProgramFiles                  C:\Program Files
PSMODULEPATH                  C:\Windows\System32\WindowsPowerShell;
PUBLIC                        C:\Users\Public
SystemDrive                   C:
SystemRoot                    C:\Windows
TEMP                          C:\Users\WilliamS\AppData\Local\Temp
USERDOMAIN                    ENGPC42
USERNAME                      WilliamS
USERPROFILE                   C:\Users\WilliamS
```

Определение и установка даты и времени

Для получения текущей даты и времени служит командлет Get-Date — просто введите его имя в командной строке и нажмите Enter:

```
get-date
```

```
Tuesday, March 16, 2010 10:40:51 AM
```

Если нужно узнать только время, либо только дату, используйте параметр –DisplayHint. Команда Get-Date –DisplayHint Date выводит дату (например, Tuesday, March 16, 2010), а Get-Date –DisplayHint Time — время (10:40:51 AM). Вот пример:

```
get-date -displayhint time
```

```
10:40:51 AM
```

Чтобы установить дату и (или) время, необходимо открыть консоль PowerShell с администраторскими полномочиями и вызвать Set-Date с нужной датой и (или) временем, взятым в двойные кавычки. Дата вводится в формате ММ-ЧЧ-ГГ, где *ММ* — месяц, *ЧЧ* — число, а *ГГ* — год. Например, аргумент **03-20-10** в следующем примере задает дату 20 марта 2010 г.:

```
set-date «03-20-10»
```

Время вводится в формате ЧЧ:ММ или ЧЧ:ММ:СС, где *ЧЧ* —часы, *ММ* — минуты, а *СС* — секунды. Если при вводе не указана половина суток (AM или PM), время считается заданным в 24-ч формате. Например, так можно задать время 3:30 дня:

```
set-date «3:30 PM»
```

Дату и время можно устанавливать одновременно, например:

```
set-date «03-20-10 03:30 PM»
set-date «03-20-10 03:30:00 PM»
set-date «03-20-10 15:30:00»
```

«Перевести» системные часы вперед или назад позволяет параметр –Adjust. Введите **Set-Date -Adjust**, затем период, на который нужно перевести часы (в формате ЧЧ:ММ:СС). Например, так можно перевести часы на 30 минут вперед:

```
set-date -adjust 00:30:00
```

Чтобы перевести часы назад, введите время со знаком «–». Так, следующая команда переводит часы на час назад:

```
set-date -adjust -01:00:00
```

Совет Командлет Get-Date возвращает объект DateTime. Прежде, чем экспериментировать с системными часами, рекомендуется на всякий случай сохранить текущую дату и время в переменной командой **$date = get-date**, чтобы при необходимости восстановить его командой **set-date $date**.

Установка учетных данных для проверки подлинности

Чтобы получить доступ к некоторым командлетам и объектам в PowerShell порой требуется предоставить учетные данные для проверки подлинности. И в командной строке, и в сценарии проще всего сделать это, получив с помощью Get-Credential объект Credential и сохранив его в переменной, как показано ниже:

```
$cred = get-credential
```

Прочитав эту команду, PowerShell запрашивает у вас имя и пароль учетной пользователя, после чего сохраняет введенные данные в переменной $cred. Важно отметить, что запрос на ввод учетных данных появляется после ввода **Get-Credential**.

Можно также указать имя пользователя и домен, для которых требуется ввести пароль. В следующем примере запрашивается пароль для учетной записи пользователя TestUser в домене DevD:

```
$cred = get-credential -credential devd\testuser
```

Объект Credential поддерживает свойства UserName и Password. Имя пользователя хранится «открытым текстом», а пароль — в зашифрованном виде, получить их, например, из переменной $cred, можно с помощью следующих команд:

```
$user = $cred.username
$password = $cred.password
```

Проверка конфигурации системы и рабочей среды

При работе с компьютерами администраторам часто требуется подробная информация об операционных системах. Для важных серверов эти сведения имеет смысл сохранить и даже распечатать, чтобы держать их под рукой. Для получения подробных сведений о системе используются следующие команды:

- **Get-HotFix** — получает сведения о пакетах обновлений и исправлениях, установленных на заданных компьютерах. Параметр –Id задает идентификатор нужных исправлений, а –Description — их тип.

```
Get-HotFix [[-Id | -Description] Исправления] {AddtlParams}

AddtlParams=
[-Credential Удостоверения] [-ComputerName Компьютер1, Компьютер2, ...]
```

- **Win32_ComputerSystem** — выводит подробные сведения о компьютере.

```
Get-Wmiobject -Class Win32_ComputerSystem [-ComputerName
Компьютер1, Компьютер2, ...] [-Credential Удостоверения] | format-list *
```

- **Win32_OperatingSystem** — выводит подробную информацию об операционной системе локального или заданного компьютера.

```
Get-Wmiobject -Class Win32_OperatingSystem [-ComputerName
Компьютер1, Компьютер2, ...] [-Credential Удостоверения] | format-list *
```

- **Win32_UserAccount** — выводит доступные пользовательские учетные записи, как локальные, так и доменные.

```
Get-Wmiobject -Class Win32_UserAccount [-ComputerName
Компьютер1, Компьютер2, ...] [-Credential Удостоверение] | format-list
Caption, Name, Domain, FullName, SID
```

- **Win32_Group** — выводит список доступных групп, включающих доменные и локальные учетные записи пользователей.

```
Get-Wmiobject -Class Win32_Group [-ComputerName Компьютер1,
Компьютер2, ...] [-Credential Удостоверения] | format-list Caption,
Name, Domain, SID
```

Инвентаризация установленных обновлений

Командлет Get-HotFix поддерживает параметр –ComputerName, с помощью которого можно задать нужные компьютеры в виде списка, разделенного запятыми:

```
get-hotfix -ComputerName fileserver84, dcserver32, dbserver11
```

Для доступа к удаленным компьютерам обычно требуется ввести учетные данные, их можно предоставить с помощью параметра –Credential. Однако ввести учетные данные на локальном компьютере таким способом невозможно (именно поэтому необходимо заранее запускать консоль с администраторскими полномочиями). Ниже показана команда, запрашивающая учетные данные у пользователя:

```
get-hotfix -Credential (get-credential) -ComputerName fileserver84,
dcserver32, dbserver11
```

А эта команда использует учетные данные, ранее сохраненные в переменной:

```
$cred = get-credential
get-hotfix -Credential $cred -ComputerName fileserver84, dcserver32,
dbserver11
```

Если вы часто работаете с одними и теми же удаленными компьютерами, имеет смысл загружать их имена из файла. Для этого следует подготовить файл, где имена нужных компьютеров занимают отдельные строки, и сохранить его в доступном каталоге, например на сетевом диске. Загрузить содержимое файла можно так:

```
get-hotfix -Credential (get-credential) -ComputerName (get-content c:\data\
servers.txt)
```

или так:

```
$comp = get-content c:\data\servers.txt
$cred = get-credential
get-hotfix -Credential $cred -ComputerName $comp
```

```
Source       Description    HotFixID       InstalledBy          InstalledOn
------     -----------   --------      -----------          -----------
TECHPC75                  {026C2636-...                        12/19/2009
12:00:00 AM
TECHPC75                  {AFB4DC8C-...                        10/22/2010
12:00:00 AM
TECHPC75     Update        KB937286      TECHPC75\wrstanek    10/15/2010
3:50:09 AM
TECHPC75     Software Update  928439     TECHPC75\wrstanek    4/18/2010
4:10:35 PM
TECHPC75     Security Update  KB925902   NT AUTHORITY\SYSTEM  4/17/2010
4:57:48 PM
TECHPC75     Update           KB929399   NT AUTHORITY\SYSTEM  4/17/2010
4:57:48 PM
TECHPC75     Update           KB929777   NT AUTHORITY\SYSTEM  4/17/2010
4:57:48 PM
```

В этом примере выводится список исправлений, установленных на заданных удаленных компьютерах, для доступа к которым запрашиваются учетные данные. Для каждого исправления и обновления указаны следующие данные:

- **source** — имя исходного компьютера;
- **description** — тип исправления: обновление для приложения, исправление системы безопасности или пакет обновления. Для исправления обычно приводится значение *update* или *hotfix*;
- **HotFixID** — идентификатор исправления (GUID, номер обновления или его идентификатор в Базе знаний Майкрософт);
- **InstalledBy** — имя пользователя, установившего обновление, либо пользователя, от имени которого оно было установлено. Для обновлений, автоматически установленных службой Windows Update, выводится NT AUTHORITY\SYSTEM;
- **InstalledOn** — дата и время установки.

Параметры –Id и –Description позволяют задавать исправления по идентификатору и типу. Следующая команда ищет пакеты обновлений, установленные на локальном компьютере:

```
get-hotfix -description «Service Pack»
```

```
Source      Description    HotFixID    InstalledBy          InstalledOn
------      -----------    --------    -----------          -----------
TECHPC16    Service Pack   KB936330    TECHPC16\CharleneD   10/15/2008
2:18:22 AM
```

В расширенном варианте этот прием можно использовать для ведения журнала обновлений и поиска компьютеров без заданных обновлений. Например, если требуется установить на заданных компьютерах обновление KB936330, можно сохранить имена этих компьютеров в текстовом файле computers.txt, каждое на отдельной строке. Далее можно проверить эти компьютеры командой Get-HotFix и сохранить результаты в файл, например, так:

```
$comp = get-content c:\data\computers.txt
$comp | foreach { if (!(get-hotfix -id KB936330 -computername $_)) { add-
content $_ -path log.txt }}
```

Эти команды загружают имена компьютеров в массив $comp, после чего элементы этого массива обрабатываются в цикле ForEach. Для каждого элемента конструкция If Not вызывает командлет Get-HotFix на каждом из заданных компьютеров и записывает в файл log.txt имена компьютеров, на которых не установлено нужное обновление. В этом примере переменная $_ содержит ссылку на текущий элемент заданного массива, в котором находится имя проверяемого компьютера.

Получение подробной информации о системе

Важные инструменты для инвентаризации компьютеров — объекты Win32_OperatingSystem и Win32_ComputerSystem. Объект Win32_OperatingSystem и его свойства содержат сведения о конфигурации операционной системы, которые можно получить следующим образом:

```
Get-WmiObject -Class Win32_OperatingSystem | Format-List *
```

```
Status                                  : OK
Name                          : Microsoft® Windows Server® 2008 Enterprise
                                |C:\Windows|\Device\Harddisk1\Partition1
FreePhysicalMemory                      : 679172
FreeSpaceInPagingFiles                  : 3749368
FreeVirtualMemory                       : 2748020
BootDevice                              : \Device\HarddiskVolume2
BuildNumber                             : 7000
BuildType                               : Multiprocessor Free
Caption                                 : Microsoftr Windows Server 2008
CodeSet                                 : 1252
CountryCode                             : 1
```

Важнейшие сведения, которые предоставляет объект Win32_OperatingSystem, включают:

- размер свободной физической и виртуальной памяти (свойства TotalVisibleMemorySize и TotalVirtualMemorySize, соответственно);
- загрузочное устройство, системный каталог, номер и тип сборки, тип операционной системы (свойства BootDevice, SystemDirectory, BuildNumber, BuildType и Caption);
- уровень шифрования, архитектура операционной системы (свойства EncryptionLevel и OSArchitecture;
- время последнего запуска (свойство LastBootUpTime).

Значения свойств TotalVisibleMemorySize и TotalVirtualMemorySize выражаются в килобайтах, чтобы быстро перевести их в мегабайты, скопируйте нужное значение в командную строку PowerShell и введите **/1kb**. Так, если TotalVisibleMemorySize = 3403604, то команда **3403604/1kb** даст результат 3323,832 Мб.

Примечание Внимательным читателям бу дет интересно, по чему я не испо льзовал в предыдущем примере **/1mb**. Дело в том, что к онстанта kb равна 1024, а к онстанта mb — 1048576. Если бы значение TotalVisibleMemorySize выражалось в байтах, то для перевода его в мегабайты дейс твительно пришлось вводить **/1mb**. Но оно выраж ено не в байтах, а в килобайтах, поэтому его требуется поделить на 1024.

С учетом сказанного выше, оценить размер памяти в мегабайтах помогут следующие команды:

```
$os = get-wmiobject -class win32_operatingsystem
```

```
$AvailMemInMB = $os.totalvisiblememorysize/1kb
$VirtualMemInMB = $os.totalvirtualmemorysize/1kb
```

Переменные $AvailMemInMB и $VirtualMemInMB доступны как в командной строке, так и в сценариях.

Загрузочное устройство, системный диск, системный каталог и номер сборки — важные компоненты конфигурации операционной системы. С помощью этой информации можно определить физический диск, на котором установлена Windows, реальный каталог, в котором находится операционная система, тип ее сборки (для одно- или многопроцессорных систем), а также точную версию операционной системы с помощью команд следующего вида:

```
$os = get-wmiobject -class win32_operatingsystem
$BootDevice = $os.bootdevice
$SystemDevice = $os.systemdevice
$SystemDirectory = $os.systemdirectory
$BuildType = $os.buildtype
$OSType = $os.caption
```

Эти переменные можно использовать в своих целях. Например, чтобы выполнить некоторые действия только над компьютерами с Windows Vista, используйте следующий прием:

```
$os = get-wmiobject -class win32_operatingsystem
$OSType = $os.caption

if ($OSType -match «Vista») {
   #На компьютере есть Vista, выполняем следующие команды
} else {
   #Vista отсутствует, выполняем следующие команды
}
```

С помощью свойства LastBootUpTime объекта Win32_OperatingSystem можно узнать время работы компьютера после включения, сравнив текущее время и дату с значением LastBootUpTime. Однако значением этого свойства является строка, а не объект DateTime, поэтому ее сначала необходимо преобразовать в DateTime методом ConvertToDateTime(). Вот пример:

```
$date = get-date
$os = get-wmiobject -class win32_operatingsystem
$uptime = $os.ConvertToDateTime($os.lastbootuptime)
write-host ($date - $uptime)
```

```
09:20:57.2639083
```

Здесь текущая дата и время записывается в переменную $date; затем вызовом Get-WmiObject извлекается объект Win32_OperatingSystem. Далее метод ConvertToDateTime() используется для преобразования значения LastBootUpTime в объект DateTime. В завершение выполняется сравнение по-

лученного объекта (даты и времени последней загрузки) с текущей датой и временем. Таким образом, в данном примере компьютер проработал со времени последней загрузки 9 ч 20 мин.

Объект Win32_ComputerSystem и его свойства содержат сводную информацию о конфигурации компьютера:

```
Get-WmiObject -Class Win32_ComputerSystem | Format-List *
```

```
AdminPasswordStatus         : 1
BootupState                 : Normal boot
ChassisBootupState          : 3
KeyboardPasswordStatus      : 2
PowerOnPasswordStatus       : 1
PowerSupplyState            : 3
PowerState                  : 0
FrontPanelResetStatus       : 2
ThermalState                : 3
Status                      : OK
Name                        : CORPSERVER84
```

Объект Win32_ComputerSystem содержит много ценной информации о компьютере и его конфигурации, включая:

- состояние загрузки и общее состояние компьютера (свойства BootUpState и Status, соответственно);
- имя, DNS-имя, домен и роль в домене (свойства Name, DNSHostName, Domain и DomainRole);
- тип системы и размер физической памяти в байтах (SystemType и TotalPhysicalMemory).

Состояние загрузки и состояние компьютера позволят узнать, закончилась ли загрузка и можно ли приступать к изменению конфигурации компьютера. Эта возможность особенно полезна при удаленной работе. В следующем примере исполняются различные команды в зависимости от состояния компьютера:

```
$cs = get-wmiobject -class win32_computersystem
$BootUpState = $cs.bootupstate

if ($BootUpState -match «Normal») {
   «Computer is in a normal state, so run these commands.»
} else {
   «Computer not in a normal state, so run these commands instead.»
}
```

Зная имя и роль компьютера в домене, администратор может отобрать для настройки только нужные компьютеры. Например, ему может потребоваться перенастроить настольные компьютеры и лэптопы, не трогая серверы и контроллеры домена. В этом случае перед настройкой компьютера можно выполнить следующую проверку:

```
$cs = get-wmiobject -class win32_computersystem
$DomainRole = $cs.domainrole

switch -regex ($DomainRole) {
  [0-1] { «This computer is a workstation.» }
  [2-3] { «This computer is a server but not a domain controller.»}
  [4-5] { «This computer is a domain controller.»}
  default { «Unknown value.»}
}
```

При переходе организации на 64-разрядные вычислительные платформы администратору требуется вести учет компьютеров, поддерживающих 64-разрядные операционные системы, а также компьютеры, на которых такие ОС уже установлены. Чтобы узнать, установлена ли на компьютере 64-разрядная ОС, проверьте свойство OSArchitecture объекта Win32_Operating-System. Чтобы определить, поддерживает ли компьютер 64-разрядные ОС, проверьте свойства Name и Description объекта Win32_Processor.

Имена подлежащих проверке компьютеров можно сохранить в текстовом файле computers.txt (каждое имя на отдельной строке), затем проверить эти компьютеры с помощью объекта Get-WmiObject и записать результаты в файл журнала:

```
$comp = get-content computers.txt

#Получаем список компьютеров без 64-разрядной ОС
$comp | foreach {
 $os = get-wmiobject -class win32_operatingsystem -computername $_
 $OSArch = $os.osarchitecture
 if (!($OSArch -match «32-bit»)) { add-content $_ -path next.txt }
}

#Ищем среди них компьютеры, поддерживающие 64-разрядные ОС
$comp2 = get-content next.txt
$comp2 | foreach {
 $ps = get-wmiobject -class win32_processor -computername $_
 $SystemType = $ps.description
 if ($SystemType -like «*x64*») { add-content $_ -path final.txt }
}
```

В этом примере список имен компьютеров загружается из файла computers.txt, расположенного в текущем рабочем каталоге, в массив $comp. Далее элементы этого массива, содержащие имена компьютеров, обрабатываются в цикле ForEach. Если на компьютере установлена 32-разрядная ОС, его имя записывается в файл next.txt, также в текущем рабочем каталоге. В переменной $_ в этом примере находится ссылка на текущий элемент заданного массива, который в данном примере содержит имя компьютера.

Команды второго блока загружают список компьютеров, на которых не установлены 64-разрядные ОС, в массив $comp2. Далее элементы этого мас-

сива обрабатываются в цикле ForEach. При этом имена компьютеров, поддерживающих 64-разрядные ОС, записываются в файл final.txt в текущем рабочем каталоге. Такая проверка «на скорую руку» обеспечивает быструю инвентаризацию компьютеров. В итоге получается файл final.txt, содержащий имена компьютеров, поддерживающих 64-разрядные ОС, но все еще работающих под управлением 32-разрядных ОС.

Инвентаризация учетных записей пользователей и групп

В ходе инвентаризации компьютеров организации часто требуется составить списки учетных записей пользователей и групп. Один из способов сделать это заключается в использовании объектов Win32_UserAccount и Win32_Group как показано в следующем примере:

```
Get-Wmiobject -Class Win32_UserAccount | format-list Caption,Name,Domain

   Caption : TECHPC76\Administrator
   Name : Administrator
   Domain : TECHPC76

   Caption : TECHPC76\Barney
   Name : Barney
   Domain : TECHPC76
```

Поскольку этот код работает с локальным компьютером, в имена учетных записей вместо домена подставляется имя локального компьютера; для удаленных компьютеров в качестве домена будет указано имя домена Active Directory.

Параметр –ComputerName задает удаленные компьютеры для проверки, а параметр –Credential — учетные данные для аутентификации. Рассмотрим их применение на следующем примере:

```
$cred = get-credential
$comp = get-content c:\data\computers.txt
$comp | foreach { Get-Wmiobject -Class Win32_UserAccount `
-ComputerName $_ -Credential $cred }
```

Этот код запрашивает у пользователя учетные данные и сохраняет их в переменной $cred. Далее список имен компьютеров загружается в массив $comp, затем элементы этого массива (имена компьютеров) обрабатываются в цикле ForEach, и для каждого из них составляется список учетных записей, доступных на соответствующем компьютере.

Совет Показанные выше примеры работают как в командной строке, так и в сценариях. В командной строке команды следует вводить поочередно, а в сценарии размещать каждую команду на отдельной строке. Длинные команды можно разбивать на фрагменты, не забывая заканчивать каждый фрагмент знаком продолжения строки (`).

Для выполнения некоторых заданий и архивации по расписанию на компьютере должны быть определенные локальные записи пользователей и групп, доступность которых необходимо проверять заблаговременно. Вот один из способов сделать это:

```
$comp = get-content computers.txt

#Получаем список компьютеров, на которых нет учетной записи BackUpUser
$comp | foreach {

 $Global:currentc = $_
 $ua = get-wmiobject -class win32_useraccount -computername $_

 $ua | foreach {
  $user = $_.name
  if ($user -eq «sqldb») {add-content $currentc -path valid.txt}
  }
}
```

Этот код загружает список имен компьютеров из файла computers.txt в текущем рабочем каталоге в массив $comp. Далее элементы этого массива (имена компьютеров) обрабатываются в цикле ForEach. При этом текущее имя компьютера записывается в глобальную переменную, затем извлекаются все учетные записи на этом компьютере и сохраняются в массиве $ua. После этого второй цикл ForEach используется для обработки элементов массива $ua (учетных записей). Если на компьютере есть учетная запись группы SqlDb, его имя записывается в файл valid.txt в текущем рабочем каталоге.

Поскольку здесь используется прямое сравнение, для поиска учетных записей можно сразу вызывать Get-WmiObject с параметром –Filter. Параметр –Filter работает подобно конструкции Where в WMI-запросах. С учетом сказанного код предыдущего примера можно переписать следующим образом:

```
$comp = get-content computers.txt

# Получаем список компьютеров, на которых нет учетной записи BackUpUser
$comp | foreach {
if (get-wmiobject -class win32_useraccount -computername $_ -filter `
«Name='sqldb'») { add-content $_ -path valid.txt }
}
```

Обратите внимание на синтаксис аргумента параметра –Filter: вся строка взята в двойные кавычки, поэтому значения отдельных свойств заключаются в одинарные кавычки.

С объектом Win32_Group работают практически так же. Хотя показанный выше прием позволяет легко и быстро получать списки пользователей и групп, для управления пользователями и группами рекомендуется использовать специализированные командлеты Active Directory. Для целей инвен-

таризации также полезны следующие объекты Win32: Win32_BIOS, Win32_NetworkAdapterConfiguration, Win32_PhysicalMemory, Win32_Processor и Win32_LogicalDisk.

Инвентаризация оборудования компьютеров

Администратору, обслуживающему компьютеры в организации, часто требуется подробная информация о конфигурации их оборудования. Эта информация нужна для выявления и устранения неполадок, включая сложные случаи, вызванные отказами и неверной настройкой оборудования.

Проверка версии и состояния микропрограммы BIOS

Неполадки компьютеров часто возникают из-за неправильной настройки или устаревшей версии микропрограммы BIOS. Для проверки состояния, версии, языка и производителя BIOS служит объект Win32_BIOS. Определить актуальность текущей версии BIOS компьютера позволяет свойство SMBIOSBIOSVersion:

```
get-wmiobject win32_bios | format-list * |
Out-File -append -filepath save.txt
```

```
__PATH                : \\ENGPC42\root\cimv2:Win32_BIOS.Name=«Default
                        System BIOS»,SoftwareElementID=«Default
                        System BIOS»,SoftwareElementState=3,
                        TargetOperatingSystem=0,Version=«GATEWA - 11d»
Status                : OK
Name                  : Default System BIOS
SMBIOSPresent         : True
BIOSVersion           : {GATEWA - 11d}
CurrentLanguage       : enUS
Manufacturer          : Intel Corp.
SMBIOSBIOSVersion     : LA97510J.15A.0285.2007.0906.0226
```

Рекомендуется отслеживать версию BIOS компьютеров с помощью свойства SMBIOSBIOSVersion, чтобы своевременно обновлять микропрограмму. Ниже показаны команды, получающие номер версии BIOS для группы компьютеров и записывающие его вместе с именами соответствующих компьютеров в файл bioscheck.txt:

```
$comp = get-content computers.txt

#Сохраняем версию BIOS для каждого компьютера
$comp | foreach {
 $bios = get-wmiobject -class win32_bios -computername $_
 $BiosVersion = $bios.SMBIOSBIOSVersion
```

```
 add-content («$_ $BiosVersion») -path bioscheck.txt
}
```

Если в организации используются компьютеры стандартной конфигурации, имеет смысл проверять версию BIOS у группы стандартных компьютеров и регистрировать в файле журнала компьютеры, у которых версия BIOS отличается от стандартной:

```
$comp = get-content computers.txt

$comp | foreach {
$bios = get-wmiobject -class win32_bios -computername $_
$BiosVersion = $bios.SMBIOSBIOSVersion
if (!($BiosVersion -match «LA97510J.15A.0285.2007.0906.0226»)) {
   add-content («$_ $BiosVersion») -path checkfailed.txt }
}
```

Определение размера ОЗУ и типа процессора

Оперативная память и процессор — ключевые компоненты, определяющие производительность компьютера, поэтому для эффективной работы на компьютере должно быть установлено достаточное количество памяти и подходящий по мощности процессор.

Объект Win32_PhysicalMemory предоставляет подробную информацию о каждом модуле оперативной памяти и ряд других свойств, позволяющих диагностировать различные сбои. Модулем называется группа микросхем памяти, смонтированных на одной плате, с которыми компьютер работает как с единым массивом памяти.

У большинства компьютеров четное число банков памяти, обычно два или четыре, в которые устанавливаются модули памяти с одинаковой разрядностью и скоростью. Вот пример использования объекта Win32_Physical Memory:

```
get-wmiobject Win32_PhysicalMemory | format-list *  |
Out-File -append -filepath save.txt
```

```
__PATH : \\ENGPC42\root\cimv2:Win32_PhysicalMemory.Tag=«Physical Memory
0»
BankLabel            : CHAN A DIMM 0
Capacity             : 1073741824
Caption              : Physical Memory
DataWidth            : 64
Description          : Physical Memory
DeviceLocator        : J6H1
FormFactor           : 8
HotSwappable         :
InstallDate          :
```

```
InterleaveDataDepth  : 1
InterleavePosition   : 1
Manufacturer         : 0xAD00000000000000
MemoryType           : 21
Model                :
Name                 : Physical Memory
Speed                : 667
Status               :
Tag                  : Physical Memory 0
TotalWidth           : 64
TypeDetail           : 128
Version              :
```

Свойство BankLabel содержит номер канала и модуля, например CHAN A DIMM 0, емкость (Capacity) модуля отображается в байтах. Чтобы быстро перевести это значение в мегабайты, скопируйте его в командную строку PowerShell и введите **/1mb**, например так: **1073741824/1mb**.

Свойство Status содержит текущий код ошибки, по которому можно выявлять сбойные и несовместимые модули.

Совет Windows Vista и более поздние версии Windows обладают встроенными средствами для выявления и диагностики проблем с оперативной памятью. При подозрении на возникновение проблем, которые система не может диагностировать автоматически, запустите утилиту Windows Memory Diagnostics следующим образом:

1. Щелкните **Start (Пуск)**, введите **mdsched.exe** в поле **Search (Начать поиск)** и нажмите Enter.
2. Выберите удобный вариант: перезагрузить компьютер и запустить утилиту сразу либо запустить утилиту при следующей перезагрузки компьютера.
3. После перезагрузки компьютера автоматически запустится утилита Windows Memory Diagnostics, которая приступит к стандартной проверке памяти. Чтобы изменить набор тестов, нажмите F1, выберите клавишами-стрелками вариант Test Mix: Basic, Standard или Extended, затем нажмите F10, чтобы возобновить проверку.
4. По завершении проверки компьютер автоматически перезагрузится, после входа в систему вы увидите результаты проверки памяти.

Если компьютер «зависает» из-за сбоев оперативной памяти и Windows Memory Diagnostics обнаружит это, она попросит запланировать проверку после следующей перезагрузки.

Общий размер оперативной памяти вычисляется как сумма емкостей всех модулей, установленных в банках памяти компьютера. Как сказано в главе 6, объект Win32_PhysicalMemoryArray поддерживает свойства MaxCapacity, отслеживающее общий размер оперативной памяти (в килобайтах), и MemoryDevices, содержащее число банков памяти.

Объект Win32_Processor предоставляет подробные сведения о процессорах компьютера, включая тактовую частоту, разрядность, deviceID, описание кэша, число ядер и логический номер. В однопроцессорной системе может быть установлен многоядерный процессор, каждое из ядер которого являет-

ся логическим процессором. Размер кэша процессора обычно отображается в килобайтах. Процессоры для настольных компьютеров обычно оснащаются кэш-памятью до второго уровня, а процессоры для серверов — кэшем третьего уровня и выше. Вот пример содержимого объекта Win32_Processor:

```
get-wmiobject Win32_Processor | format-list *  |
Out-File -append -filepath save.txt
```

```
__PATH : \\ENGPC42\root\cimv2:Win32_Processor.DeviceID=»CPU0»
CpuStatus                  : 1
CreationClassName          : Win32_Processor
CurrentClockSpeed          : 2660
CurrentVoltage             : 16
DataWidth                  : 64
Description                : x64 Family 6 Model 15 Stepping 7
DeviceID                   : CPU0
ErrorCleared               :
ErrorDescription           :
ExtClock                   : 266
Family                     : 190
InstallDate                :
L2CacheSize                : 4096
L2CacheSpeed               :
L3CacheSize                : 0
L3CacheSpeed               : 0
LastErrorCode              :
Level                      : 6
LoadPercentage             : 2
Manufacturer               : GenuineIntel
MaxClockSpeed              : 2660
Name                       : Intel(R) Core(TM)2 Quad CPU    @ 2.66GHz
NumberOfCores              : 4
NumberOfLogicalProcessors  : 4
PowerManagementCapabilities :
PowerManagementSupported   : False
ProcessorId                : BFEBFBFF000006F7
ProcessorType              : 3
Revision                   : 3847
Role                       : CPU
SocketDesignation          : LGA 775
Status                     : OK
```

Важные свойства — ErrorCleared, ErrorDescription и Status, они позволяют выявлять сбои процессора. Обратите внимание на описание и условия возникновения ошибки и примите меры к ее устранению. Так, если ошибка сохраняется и после перезагрузки, не исключено, что ее причина — в сбоях материнской платы, процессора либо обоих устройств. Иногда эти ошибки удается исправить путем обновления BIOS.

Проверка жесткого диска и разделов

Жесткие диски используются для хранения данных. Емкости диска должно быть достаточно для размещения файлов операционной системы, рабочей среды и пользовательских данных. Для обеспечения оптимальной производительности на жестком диске также должно оставаться определенное свободное место, необходимое для автоматической очистки диска и других сервисных задач.

WMI поддерживает несколько Win32-классов для работы с дисками. Класс Win32_DiskDrive позволяет работать с физическими дисками, включая жесткие диски и запоминающие устройства, подключаемые через USB. Чтобы скрыть все диски кроме жестких, можно использовать фильтрацию по типу носителя, например так:

```
get-wmiobject -class win32_diskdrive -filter '
«MediaType='Fixed hard disk media'»
```

```
Partitions : 2
DeviceID   : \\.\PHYSICALDRIVE0
Model      : ST3500630AS
Size       : 500105249280
Caption    : ST3500630AS

Partitions : 1
DeviceID   : \\.\PHYSICALDRIVE1
Model      : ST3500630AS
Size       : 500105249280
Caption    : ST3500630AS
```

Этот компьютер оснащен парой жестких дисков, диск PhysicalDrive0 разбит на два раздела, а PhysicalDrive1 содержит единственный раздел.

Фильтруя вывод этой команды по идентификатору устройства (DeviceID) или модели (Caption), можно получать сведения об отдельных жестких дисках, как в следующем примере:

```
get-wmiobject -class win32_diskdrive -filter «Caption='ST3500630AS'» |
format-list *
```

```
ConfigManagerErrorCode      : 0
LastErrorCode               :
NeedsCleaning               :
Status                      : OK
DeviceID                    : \\.\PHYSICALDRIVE0
StatusInfo                  :
Partitions                  : 2
BytesPerSector              : 512
ConfigManagerUserConfig     : False
```

```
DefaultBlockSize          :
Index                     : 0
InstallDate               :
InterfaceType             : SCSI
SectorsPerTrack           : 63
Size                      : 500105249280
TotalCylinders            : 60801
TotalHeads                : 255
TotalSectors              : 976768065
TotalTracks               : 15504255
TracksPerCylinder         : 255
Caption                   : ST3500630AS
CompressionMethod         :
ErrorCleared              :
ErrorDescription          :
ErrorMethodology          :
FirmwareRevision          : 3.AA
Manufacturer              : (Standard disk drives)
MediaLoaded               : True
MediaType                 : Fixed hard disk media
Model                     : ST3500630AS
Name                      : \\.\PHYSICALDRIVE0
SCSIBus                   : 0
SCSILogicalUnit           : 0
SCSIPort                  : 0
SCSITargetId              : 0
```

Вывод этой команды содержит подробные сведения о конфигурации физического диска, включая:

- число байтов в секторе, секторов в дорожке и дорожек в цилиндре;
- тип интерфейса (SCSI или IDE);
- емкость (в байтах, чтобы перевести ее в Гб, необходимо поделить это число на константу 1gb);
- общее число цилиндров, головок, секторов и дорожек;
- тип шины, логический номер, порт и идентификатор TargetID.

Также стóит обратить внимание на свойства ErrorCleared, ErrorDescription, ErrorMethodology и Status — они позволяют выявлять сбои дисков. Изучите описание и условия возникновения ошибки, чтобы принять меры для ее устранения. Если ошибка не исчезает после перезагрузки, возможно, причина в сбое контроллера, самого диска или обоих устройств. Иногда ошибки удается устранить путем обновления микропрограммы контроллера.

Объект Win32_DiskPartition предоставляет сведения о разбиении дисков на разделы. Этот объект представляет разделы, на которые вы разбиваете диски с помощью оснастки Disk Management (Управление дисками), например:

```
get-wmiobject -class win32_diskpartition
```

```
 NumberOfBlocks   : 18603207
BootPartition    : False
Name             : Disk #0, Partition #0
PrimaryPartition : True
Size             : 9524841984
Index            : 0

NumberOfBlocks   : 958164795
BootPartition    : True
Name             : Disk #0, Partition #1
PrimaryPartition : True
Size             : 490580375040
Index            : 1

NumberOfBlocks   : 976768002
BootPartition    : False
Name             : Disk #1, Partition #0
PrimaryPartition : True
Size             : 500105217024
Index            : 0
```

Вся ключевая информация содержится в стандартном выводе этой команды, поэтому вам, скорее всего, не потребуется обращаться к дополнительным свойствам. В этом примере на компьютере установлено два диска: Disk 0 и Disk 1. Диск Disk 0 разбит на два раздела: Partition 0 и Partition 1, а на Disk 1 имеется единственный раздел — Partition 0. Размеры разделов отображаются в байтах. Перевести это значение в гигабайты можно путем деления на константу 1gb.

В Windows форматированные разделы представлены логическими дисками, для работы с которыми используется WMI-объект Win32_LogicalDisk. Этот объект предоставляет подробные сведения о каждом из логических дисков компьютера. Учтите, однако, что сетевые и оптические диски, а также сменные носители, включая флэш-диски и оптические диски, также получают представление в виде логических дисков. Различать их можно по значению свойства Description, которое может быть следующим:

- CD-ROM Disc для оптических дисков;
- Removable Disk для сменных дисков;
- Local Fixed Disk для фиксированных жестких дисков.

При работе с логическим дисками, представляющими разделы жестких дисков, важно знать идентификатор (DeviceID), состояние сжатия, тип файловой системы, размер свободного места, емкость и поддерживаемые возможности. Значение DeviceID — это буква диска, например C:. Вот пример получения содержимого объекта Win32_LogicalDisk:

```
get-wmiobject -class win32_logicaldisk -filter «name='c:'» |
format-list * | Out-File -append -filepath save.txt
```

```
Status                        :
Availability                  :
DeviceID                      : C:
StatusInfo                    :
Access                        : 0
BlockSize                     :
Caption                       : C:
Compressed                    : False
ConfigManagerErrorCode        :
ConfigManagerUserConfig       :
CreationClassName             : Win32_LogicalDisk
Description                   : Local Fixed Disk
DriveType                     : 3
FileSystem                    : NTFS
FreeSpace                     : 298870042624
InstallDate                   :
LastErrorCode                 :
MaximumComponentLength        : 255
MediaType                     : 12
Name                          : C:
NumberOfBlocks                :
QuotasDisabled                : True
QuotasIncomplete              : False
QuotasRebuilding              : False
Size                          : 490580373504
SupportsDiskQuotas            : True
SupportsFileBasedCompression  : True
SystemCreationClassName       : Win32_ComputerSystem
SystemName                    : ENGPC42
VolumeDirty                   : False
VolumeName                    :
VolumeSerialNumber            : 008EA097
```

Значения свойств FreeSpace и Size выражаются в байтах. Чтобы быстро перевести их в гигабайты, скопируйте значение в командную строку PowerShell и введите **/1gb**, например **302779912192/1gb**. Вот пример:

```
$dr = get-wmiobject -class win32_logicaldisk -filter «name='c:'»
$free = [Math]::Round($dr.freespace/1gb)
$capacity = [Math]::Round($dr.size/1gb)

write-host $dr.name «on» $dr.systemname
write-host «Disk Capacity: $capacity»
write-host «Free Space: $free»
```

```
C: on ENGPC42
Disk Capacity: 457
Free Space: 278
```

Проверка драйверов устройств и управление ими

К компьютерам подключают самые разные устройства, и каждому из них для работы требуется драйвер. Администраторам весьма полезно знать:

- активно ли устройство или отключено;
- работает ли устройство или оно остановлено;
- настроен ли драйвер для автоматического запуска.

Для работы с драйверами используется Win32-класс Win32_SystemDriver. Он предоставляет подробные сведения о конфигурации и состоянии драйверов устройств, установленных на компьютере, например отображаемое имя устройства (свойством DisplayName), состояние (свойство State) или тип запуска (свойство StartMode). Отображаемые имена устройства и его драйвера совпадают, то же самое имя отображается Диспетчером устройств (Device Manager).

В следующем примере свойство DisplayName используется для проверки RAID-контроллера компьютера:

```
get-wmiobject -class win32_systemdriver | where-object '
{$_.displayname -like «*raid c*»} | format-list *
```

```
Status                  : OK
Name                    : iaStor
State                   : Running
ExitCode                : 0
Started                 : True
ServiceSpecificExitCode : 0
AcceptPause             : False
AcceptStop              : True
Caption                 : Intel RAID Controller
CreationClassName       : Win32_SystemDriver
Description             : Intel RAID Controller
DesktopInteract         : False
DisplayName             : Intel RAID Controller
ErrorControl            : Normal
InstallDate             :
PathName                : C:\Windows\system32\drivers\iastor.sys
ServiceType             : Kernel Driver
StartMode               : Boot
StartName               :
SystemCreationClassName : Win32_ComputerSystem
SystemName              : ENGPC42
```

```
TagId                   : 25
Site                    :
Container               :
```

Свойство State принимает значение Running или Stopped, что можно использовать при поиске работающих либо остановленных драйверов:

```
get-wmiobject -class win32_systemdriver -filter «state='Running'»
```

```
DisplayName : Microsoft ACPI Driver
Name        : ACPI
State       : Running
Status      : OK
Started     : True

DisplayName : Ancillary Function Driver for Winsock
Name        : AFD
State       : Running
Status      : OK
Started     : True
```

Поддерживаются следующие типы запуска драйверов устройств:

- Boot — при загрузке;
- Manual — вручную;
- Auto — автоматически;
- System — режим для системных драйверов.

Используя свойство StartMode, можно отобрать драйверы, которые запускаются во время загрузки системы, как показано ниже:

```
get-wmiobject -class win32_systemdriver -filter «startmode='Boot'»
```

Класс Win32_SystemDriver поддерживает ряд методов для управления системными драйверами:

- **Change()** — изменяет конфигурацию драйвера устройства; принимает следующие параметры (по порядку): DisplayName, PathName, ServiceTypeByte, ErrorControlByte, StartMode, DesktopInteractBoolean, StartName, StartPassword, LoadOrderGroup, LoadOrderGroupDependenciesArray, and ServiceDependenciesArray.

Внимание! Настройка устройств через командную строку PowerShell требует большой осторожности и тщательного планирования, иначе компьютер может перестать загружаться. Перед внесением любых изменений рекомендуется создавать точки восстановления (см. гл. 12) или даже выполнять полное резервное копирование.

- **ChangeStartMode()** — изменяет тип запуска драйвера устройства; принимает единственный параметр (имя типа запуска), допустимые значения: boot, manual, auto или system.

Внимание! Перед изменением типа запуска драйвера убедитесь, что этот драйвер поддерживает тип запуска, который вы хотите ему назначить, и новый тип запуска не помешает загрузке компьютера.

- **Delete()** — удаляет драйвер устройства (если состояние драйвера допускает это). Удаление драйвера не препятствует использованию устройства; чтобы сделать это, необходимо отключить драйвер. Если удалить драйвер, не отключая его, Windows обычно обнаруживает устройство и переустанавливает его при следующей загрузке. Иногда драйвер преднамеренно удаляют при устранении неполадок, чтобы вынудить Windows переустановить устройство;

Внимание! Удалять драйверы устройств средствами PowerShell следует очень осторожно, так как PowerShell не предупреждает о действиях, способных нарушить работу компьютера.

- **InterrogateService()** — подключается к устройству, используя его драйвер. Если метод возвращает ноль, WMI удалось опросить устройство; если же возвращается другое значение, при попытке опроса возникли сбои. Если работа устройства остановлена либо приостановлена, этот метод всегда возвращает ошибку;
- **PauseService()** — приостанавливает работу устройства в целях диагностики и устранения неполадок. Приостановить устройство возможно, если свойство AcceptPause его драйвера установлено в True, а само устройство находится в состоянии, допускающем приостановку;
- **ResumeService()** — возобновляет работу приостановленного устройства;
- **StopService()** — останавливает работу устройства в целях диагностики и устранения неполадок. Остановить устройство возможно, если свойство AcceptStop его драйвера установлено в True, а само устройство находится в состоянии, допускающем остановку;
- **StartService()** — запускает остановленные устройства, включая устройства, настроенные для ручного запуска.

Для изменения типа запуска драйвера устройства вызовите метод ChangeStartMode(), указав нужный режим. Вот общий синтаксис вызова этого метода:

```
$Объект_драйвера.ChangeStartMode(Режим)
```

где *$Объект_драйвера* — ссылка на объект Win32_SystemDriver, а *StartMode* — новый тип запуска:

```
$d = get-wmiobject -class win32_systemdriver | where-object '
{$_.displayname -like «creative audio*»}
$d.changestartmode(«auto»)
```

```
__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
```

```
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0
```

Код в этом примере задает для устройства Creative Audio тип запуска Auto. В выводе следует обратить внимание на значение ReturnValue. Если оно равно 0, операция завершилась успешно, любое другое значение говорит об ошибке. Обычно ошибки возникают из-за недостаточных полномочий (если консоль PowerShell открыта с пользовательскими, а не администраторскими привилегиями), обращения к неверному драйверу, либо к драйверу, состояние которого не допускает настройки. Учтите, что после изменения конфигурации драйверов компьютер может перестать загружаться, поэтому все действия по настройке драйверов должны быть тщательно продуманы заранее.

Углубленный анализ системы

Хотите получить более полную картину конфигурации компьютера? Это возможно. Так, следующая команда выводит полный список доступных .NET-типов:

```
[System.AppDomain]::CurrentDomain.GetAssemblies() |
Foreach-Object { $_.GetTypes() }
```

Можно написать эту команду как функцию, чтобы вызвать ее с разными фильтрами для поиска определенных .NET-типов:

```
function ListType() {
  [System.AppDomain]::CurrentDomain.GetAssemblies() |
  Foreach-Object { $_.GetTypes() }
}
```

Для получения полного списка .NET-типов, достаточно вызвать функцию ListType без фильтров, например, так:

```
ListType
```

Чтобы проверить определенные .NET-типы, следует ввести их имена. Например, чтобы найти все .NET-типы, у которых в имени есть слово «parser», введите:

```
ListType | ? { $_.Name -like «*parser*» }
```

Продолжим «анатомировать» .NET-типы. Следующая команда выводит конструкторы для всех .NET-типов, у которых в имени есть слово «parser»:

```
ListType | ? { $_.Name -like «*parser*» } |
% { $_.GetConstructors() }
```

Здорово, правда? Однако не все нужные .NET-типы загружаются по умолчанию, поэтому перед использованием некоторые .NET-типы необходимо загрузить.

Другой замечательный прием — проверка объектов COM, доступных на компьютере. Поскольку объекты COM регистрируются в системном реестре, их поиск ведется путем просмотра соответствующих разделов реестра с помощью функций, подобных следующей:

```
function ListProgID {
 param()
 $paths = @(«REGISTRY::HKEY_CLASSES_ROOT\CLSID»)
 if ($env:Processor_Architecture -eq «amd64») {
   $paths+=«REGISTRY::HKEY_CLASSES_ROOT\Wow6432Node\CLSID» }
 Get-ChildItem $paths -include VersionIndependentPROGID -recurse |
 Select-Object @{
   Name='ProgID'
   Expression={$_.GetValue(«»)}
 }, @{
   Name='Type'
   Expression={
     if ($env:Processor_Architecture -eq «amd64») { «Wow6432» }
     else { «32-bit» }
   }
 }
}
```

Этот код проверяет архитектуру процессора, установленного на компьютере. На компьютерах с 32-разрядными ОС поиск 32-разрядных объектов COM ведется в разделе HKEY_CLASSES_ROOT\CLSID, а на компьютерах с 64-разрядными ОС — в HKEY_CLASSES_ROOT\CLSID; дополнительные объекты регистрируются в разделе HKEY_CLASSES_ROOT\Wow6432Node\CLSID. Найденные объекты выводятся с указанием их идентификатора (progID) и типа.

Чтобы получить полный список объектов COM, введите

```
ListProgID
```

Для поиска определенных объектов COM следует указать хотя бы часть их имени. Например, чтобы найти все объекты COM, в имени которых есть слово «Microsoft», введите

```
ListProgID | Where-Object { $_.ProgID -like «*Microsoft*» }
```

В общем, изучать объекты очень весело, поскольку о них можно узнать много интересного; подробнее о работе с объектами см. в главе 6.

Глава 10

Управление, защита и аудит файловой системы

В этой главе вы познакомитесь с методами управления файловой системой и ее защиты. Поверьте, здесь кроется куда больше возможностей, чем многие могут себе представить. Вы можете создавать, копировать и перемещать файлы и каталоги; читать, записывать и дописывать содержимое файлов; вступать во владение файлами и каталогами. Но это еще не все: PowerShell позволяет обрабатывать параметры защиты не только для отдельных файлов и каталогов, но и для групп файлов и папок, соответствующих заданным параметрам.

Управление дисками, каталогами и файлами в PowerShell

В PowerShell есть средства для управления дисками, каталогами и файлами (см. также главу 3), в их число входит поставщик FileSystem, а также командлеты для работы с хранилищами данных (табл. 3-4) и дисками поставщиков (табл. 3-5).

Подключение и отключение дисков PowerShell

Командлет Get-PSDrive позволяет просматривать доступные в данный момент диски PowerShell. Как видно из следующего примера, их список включает настоящие диски и другие ресурсы, с которыми в PowerShell работают как с дисками:

```
get-psdrive
```

```
Name       Provider      Root
----       --------      ----
Alias      Alias
C          FileSystem    C:\
cert       Certificate   \
D          FileSystem    D:\
```

```
E           FileSystem    E:\
Env         Environment
F           FileSystem    F:\
Function    Function
G           FileSystem    G:\
HKCU        Registry      HKEY_CURRENT_USER
HKLM        Registry      HKEY_LOCAL_MACHINE
I           FileSystem    I:\
J           FileSystem    J:\
K           FileSystem    K:\
L           FileSystem    L:\
M           FileSystem    M:\
N           FileSystem    N:\
O           FileSystem    O:\
P           FileSystem    P:\
Q           FileSystem    Q:\
Variable    Variable
W           FileSystem    W:\
WSMan       WSMan
X           FileSystem    X:\
Y           FileSystem    Y:\
Z           FileSystem    Z:\
```

Примечание Для опроса нескольких компьютеров пользуйтесь командлетом Invoke-Command (см. главу 4 и следующий пример):

```
invoke-command -computername Server43, Server27, Server82
-scriptblock { get-psdrive }
```

Для перехода между дисками служит командлет Set-Location: просто укажите после него нужный диск или относительный путь, например, так:

```
set-location c:
```

или так:

```
set-location c:\logs
```

При переходе между дисками PowerShell запоминает рабочий каталог и переводит пользователя в прежний каталог, если он возвращается на соответствующий диск (подробнее об этом — в главе 3).

Командлет New-PSDrive создает диски PowerShell, сопоставленные каталогам хранилищ данных, которыми могут быть локальные или сетевые папки и разделы реестра. Такой диск является своего рода ярлыком и доступен только в текущем сеансе консоли PowerShell. Например, если вы часто работаете с каталогом C:\Data\Current\History\Files, для оперативного доступа к этому каталогу можно создать диск PowerShell. При создании диска его псевдоним задают с помощью параметра –Name, тип поставщика — параметром –PSProvider, а корневой каталог — параметром –Root:

```
new-psdrive -name hfiles -psprovider filesystem -root c:\data\current\
history\files
```

В этом примере создается диск *hfiles* на основе файловой системы (тип — FileSystem), который открывает доступ к папке C:\Data\Current\History\Files. Чтобы перейти на этот диск, введите **set-location hfiles:**. Ошибки при создании дисков возникают из-за недостаточных полномочий и при попытках создать диск с именем, уже занятым другим диском.

Созданные таким образом диски существуют только в текущем сеансе PowerShell, поэтому они исчезают после завершения сеанса. Удалить диск PowerShell также можно командой Remove-PSDrive. Пользователю разрешено удалять диски, которые он подключил к консоли, но удалять в PowerShell диски Windows и сетевые диски, подключенные другими методами, он не сможет.

Примечание Возможно также создание дисков, сопоставленных разделам реестра. Для этого необходимо присвоить параметру PSProvider значение registry. Чтобы получить список допустимых значений PSProvider, введите **get-psprovider**.

Создание файлов и каталогов и управление ими

Приемы работы в PowerShell с файлами и каталогами практически не отличаются. Для просмотра каталогов и файлов служит командлет Get-ChildItem, демонстрировавшийся во множестве приведенных выше примеров. Для создания файлов и каталогов применяют командлет New-Item. Вот его общий синтаксис:

```
new-item -type [Directory | File] -path Путь
```

При вызове этого командлета необходимо указать, что именно создается — каталог или файл, указав значение *Directory* или *File*, соответственно; также необходимо задать путь к создаваемому файлу или каталогу (параметр -path). Чтобы проверить созданный файл или каталог, используйте командлет New-Item. В следующем примере создается каталог C:\Logs\Backup, после чего выполняется его проверка:

```
new-item -type directory -path c:\logs\backup
```

```
    Directory: C:\logs
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----         2/18/2009   4:54 PM            backup
```

Примечание Для создания каталогов и файлов на удаленных компьютерах применяют командлет Invoke-Command (подробнее о нем — в главе 4). Вот пример:

```
invoke-command -computername Server43, Server27, Server82
-scriptblock { new-item -type directory -path c:\logs\backup }
```

Если у вас есть необходимые разрешения, проблем с созданием файлов и папок быть не должно. Командлет New-Item «умеет» автоматически создавать вложенные каталоги. Так, в показанном выше примере каталог C:\Logs не существовал, поэтому PowerShell создала сначала его, а в нем — каталог Backup. Файлы PowerShell всегда создает пустыми.

Копирование файлов и каталогов

Для копирования файлов и каталогов используется командлет Copy-Item. Общий синтаксис команд для копирования каталогов имеет следующий вид:

```
copy-item Исходный_каталог Целевой_каталог -recurse
```

где *Исходный_каталог* — путь к копируемому каталогу, а *Целевой_каталог* — путь, к каталогу, в который его нужно скопировать. В следующем примере каталог C:\Logs со всем содержимым копируется в каталог C:\Logs_Old:

```
copy-item c:\logs c:\logs_old -recurse
```

Если каталог Logs_Old не существует, команда создаст его. А так выглядит общий синтаксис для копирования файлов:

```
copy-item Исходный_файл Целевой_каталог
```

где *Исходный_файл* — имя копируемого файла и путь к нему, а *Целевой_каталог* — путь к каталогу, в который требуется скопировать файл. В следующем примере из каталога C:\Logs копируются все файлы с расширением .txt в каталог C:\Logs_Old:

```
copy-item c:\logs\*.txt c:\logs_old
```

При наличии необходимых разрешений проблем с копированием быть не должно. Копирование файлов и папок с диска на диск выполняется почти так же:

```
copy-item c:\logs d:\logs_old -recurse
```

Перемещение файлов и каталогов

Для перемещения файлов и каталогов используется командлет Move-Item. Вот его общий синтаксис:

```
move-item Исходный_путь Целевой_путь
```

где *Исходный_путь* — путь к перемещаемому каталогу или файлу, а *Целевой_путь* — путь к каталогу, в который их требуется переместить. Если при перемещении возникает ошибка, выводится соответствующее сообщение, а при успешном завершении операции никаких сообщений не выводится. В следующем примере каталог C:\Logs со всем содержимым перемещается в каталог C:\Backup\Logs:

```
move-item c:\logs c:\backup\logs
```

А эта команда перемещает все .txt-файлы из каталога C:\Logs в C:\Backup\Logs:

```
move-item c:\logs\*.txt c:\backup\logs
```

Как и в предыдущих случаях, при наличии необходимых разрешений операция выполняется без проблем, но кое-какие «подводные камни» тут все же есть. Так, Move-Item не перемещает файлы и каталоги между разными дисками, поэтому корневые каталоги в исходном и целевом путях должны быть одинаковыми. Если каталог или какой-либо из перемещаемых файлов занят другой программой, переместить их не удастся.

Переименование файлов и каталогов

Для переименования файлов и каталогов служит командлет Rename-Item, общий синтаксис его таков:

```
rename-item ИсходноеИмя_и_путь НовоеИмя
```

где *ИсходноеИмя_и_путь* — полный путь к каталогу или файлу, который требуется переименовать, а *НовоеИмя* — новое имя каталога или файла. В следующем примере файл Log1.txt в каталоге C:\Logs переименовывается в Log1_hist.txt:

```
rename-item c:\logs\log1.txt log1_hist.txt
```

И здесь при наличии необходимых разрешений у вас все должно быть в порядке, если только нужные файлы или каталоги не заняты другой программой.

Удаление файлов и каталогов

Для удаления файлов и каталогов используется командлет Remove-Item, общий синтаксис которого имеет следующий вид:

```
remove-item Имя_и_путь [-force]
```

где *Имя_и_путь* — полный путь к каталогу или файлу, который требуется удалить; *–Force* — необязательный параметр, с которым удаляются даже файлы и каталоги, доступные только для чтения. В следующем примере удаляется каталог D:\Logs_Old со всем содержимым:

```
remove-item d:\logs_old
```

Для исполнения этой операции нужны соответствующие разрешения, а удаляемые файлы и каталоги не должны быть заняты другими программами. Каталоги и файлы с атрибутами Read-Only, Hidden и System удаляются только с параметром –Force.

Работа с содержимым файлов

При работе с компьютерами часто требуется вести журналы и инвентарные записи, для чего в PowerShell используются простые команды, обеспечивающие чтение и запись данных в файлы.

Команды для работы с содержимым файлов

Команды для работы с файловыми ресурсами включают:

- **Get-Content** — выводит содержимое заданного файла. Параметр –Force открывает доступ к файлам с атрибутами Hidden, System и Read-Only; –TotalCount задает число выводимых строк; –Include явно задает файлы для обработки; –Exclude заставляет игнорировать заданные файлы. Параметры –Include и –Exclude принимают подстановочные знаки.

```
Get-Content [-LiteralPath | -Path] Файл {AddtlParams}

AddtlParams=
[-Credential Удостоверения] [-Delimiter Строка] [-Encoding Кодировка]
[-Exclude Файлы] [-Force] [-Include Файлы]
[-TotalCount Число]
```

Совет Многие командлеты PowerShell принимают параметры –Path и –LiteralPath, задающие путь к файлу, но значение –LiteralPath, в отличие от –Path, используется «как есть», то есть без замены подстановочных знаков. Спецсимволы в составе пути необходимо брать в одинарные кавычки, чтобы PowerShell воспринимала их как литералы.

- **Set-Content** — перезаписывает содержимое заданного файла. Параметр –Force открывает доступ к файлам с атрибутами Hidden, System и Read-Only; новое содержимое файла задается параметром – Value или принимается через конвейер; –Include явно задает файлы для обработки; –Exclude заставляет игнорировать заданные файлы. Параметры –Include и –Exclude принимают подстановочные знаки.

```
Set-Content [-LiteralPath | -Path] Файл [-Value Содержимое]
{AddtlParams}

AddtlParams=
[-Credential Удостоверения] [-Encoding Кодировка] [-Exclude
Файлы] [-Force] [-Include Файлы]
```

- **Add-Content** — дописывает содержимое к заданному файлу. Параметр –Force открывает доступ к файлам с атрибутами Hidden, System и Read-Only; новое содержимое файла задается параметром – Value или принимается через конвейер.

```
Add-Content [-LiteralPath | -Path] Файл [-Value Новое_содержимое]
{AddtlParams}
```

```
AddtlParams=
[-Credential Удостоверения] [-Encoding Кодировка] [-Exclude
Файлы] [-Force] [-Include Файлы]
```

- **Clear-Content** — удаляет содержимое заданного файла. Параметр –Force открывает доступ к файлам с атрибутами Hidden, System и Read-Only; –Include явно задает файлы для обработки; –Exclude заставляет игнорировать заданные файлы.

```
Add-Content [-LiteralPath | -Path] Файл {AddtlParams}

AddtlParams=
[-Credential Удостоверения] [-Exclude Файлы] [-Force]
[-Include Файлы]
```

Чтение и запись файлов

По умолчанию Get-Content ищет в текущем каталоге файлы или файлы, заданные с использованием подстановочных знаков, и выводит их содержимое в виде текстов. Это означает, что можно быстро вывести содержимое любого текстового файла в текущем каталоге на консоли, если ввести **Get-Content** и имя искомого файла. Следующая команда выводит содержимое файла log1.txt из текущего каталога:

```
get-content log1.txt
```

Чтобы вывести содержимое файла, расположенного в другом каталоге, необходимо указать полный путь к файлу. Следующая команда выводит содержимое файла log1.txt из каталога C:\Logs:

```
get-content c:\logs\log1.txt
```

С помощью подстановочных знаков можно вывести содержимое заданной группы файлов. Следующая команда выводит содержимое всех файлов в каталоге C:\Logs, имя которых начинается словом «log»:

```
get-content c:\logs\log*
```

Чтобы сузить область поиска файлов, используйте параметр –Include, а чтобы исключить определенные файлы — параметр –Exclude. Следующая команда выводит только файлы с расширениями .txt и .log:

```
get-content -include *.txt, *.log -path c:\logs\log*
```

Чтобы исключить из вывода .xml-файлы и показать все остальные файлы, имя которых начитается с «log», введите

```
get-content -exclude *.xml -path c:\logs\log*
```

Увидеть первые строки содержимого файла позволяет параметр –TotalCount, который задает число выводимых строк. Следующий пример выводит первые 10 строк каждого из заданных файлов:

```
get-content -totalcount 10 -path c:\logs\log*
```

Для работы с содержимым файлов также используются командлеты Set-Content, Add-Content и Clear-Content. Set-Content перезаписывает заданные файлы новым содержимым. Add-Content дописывает новые данные к концу заданных файлов. Clear-Content очищает заданные файлы, но не удаляет их, в результате получаются пустые файлы.

Работа с дескрипторами защиты

Важнейшие задачи администратора включают настройку и обслуживание защиты файловой системы. Для решения этих задач PowerShell предоставляет ряд команд, обеспечивающих просмотр и настройку дескрипторов защиты. Записав необходимые команды в журнал (transcript) или сценарий, можно без труда воспроизвести их на других компьютерах.

Команды для работы с дескрипторами защиты

Команды, управляющие доступом к файловым ресурсам, включают:

- **Get-Acl** — получает объект, представляющий дескриптор защиты файла, раздела реестра или другого ресурса от поставщика, поддерживающего дескрипторы защиты. Параметр –Audit получает данные аудита для заданного дескриптора защиты из списка управления доступом.

  ```
  Get-Acl [-Path] Файл {AddtlParams}

  AddtlParams=
  [-Audit] [-Exclude Файлы] [-Include Файлы]
  ```

- **Set-Acl** — изменяет дескриптор защиты файла, раздела реестра или другого ресурса от поставщика, поддерживающего дескрипторы защиты. Параметр –AclObject задает настройки защиты.

  ```
  Set-Acl [-Path] Файл [-Aclobject] Параметры_защиты {AddtlParams}

  AddtlParams=
  [-Exclude Файлы] [-Include Файлы]
  ```

Примечание Доступ к файлам и каталогам на NTFS-томах контролируется разрешениями, без которых работать с этими ресурсами невозможно.

Получение и установка дескрипторов защиты

Администратору при работе с такими ресурсами, как каталоги, файлы и разделы реестра требуется просматривать и настраивать их дескрипторы защиты. Для этого служит командлет Get-Acl с параметром –Path, задающим путь к нужному ресурсу. Как и Get-Content, этот командлет поддерживает пути с подстановочными знаками, а также параметры –Include и –Exclude.

Для каждого из заданных файлов Get-Acl возвращает объект, содержащий данные системы безопасности. По умолчанию Get-Acl отображает путь к ресурсу, его владельца и записи списка управления доступом. Списком управления доступом (access control list, ACL) управляет владелец ресурса. Для получения дополнительной информации, включая группу системы безопасности владельца, параметров аудита и полного дескриптора защиты на языке SDDL (Security Descriptor Definition Language), следует отформатировать вывод Get-Acl так:

```
get-acl c:\windows\system32\windowspowershell | format-list
```

```
   Path: Microsoft.PowerShell.Core\FileSystem::
  C:\windows\system32\windowspowershell
  Owner  : NT AUTHORITY\SYSTEM
  Group  : NT AUTHORITY\SYSTEM
  Access : NT SERVICE\TrustedInstaller Allow  FullControl
           NT SERVICE\TrustedInstaller Allow  268435456
           NT AUTHORITY\SYSTEM Allow  FullControl
           NT AUTHORITY\SYSTEM Allow  268435456
           BUILTIN\Administrators Allow  FullControl
           BUILTIN\Administrators Allow  268435456
           BUILTIN\Users Allow  ReadAndExecute, Synchronize
           BUILTIN\Users Allow  -1610612736
           CREATOR OWNER Allow  268435456
  Audit  :
  Sddl   : O:SYG:SYD:AI(A;ID;FA;;;S-1-5-80-956008885-3418522649-
  1831038044-
  1853292631-2271478464)(A;CIIOID;GA;;;S-1-5-80-956008885-3418522649-
  1831038044-1853292631-2271478464)(A;ID;FA;;;SY) (A;OICIIOID;GA;;;SY)
  (A;ID;FA;;;BA)(A;OICIIOID;GA;;;BA)(A;ID;0x1200a9;;;BU)
  (A;OICIIOID;GXGR;;;
  BU)(A;OICIIOID;GA;;;CO)
```

В этом примере Get-Acl возвращает объект DirectorySecurity, представляющий дескриптор защиты каталога C:\Windows\System32\WindowsPowerShell, и передает его командлету Format-List.

С файлами работают аналогичным образом. Вот пример:

```
get-acl -include *.txt, *.log -path c:\logs\log* | format-list
```

Для каждого из заданных файлов Get-Acl возвращает объект FileSecurity, представляющий дескриптор защиты, который затем выводится через Format-List. доступны следующие свойства этого объекта:

- **Owner** — имя владельца ресурса;
- **Group** — основная группа, членом которой является владелец ресурса;
- **Access** — правила управления доступом к ресурсу;

- **Audit** — правила аудита ресурса;
- **Sddl** — полный дескриптор защиты в виде SDDL-строки.

Примечание Объекты FileSecurity и DirectorySecurity поддерживают свойства, отсутствующие в стандартном выводе. Чтобы увидеть их, направьте вывод Format-List *, в результате к выводу будут добавлены свойства *PSPath* (путь PowerShell к ресурсу), PSParentPath (путь PowerShell к родительскому ресурсу), *PSChildName* (имя ресурса), *PSDrive* (диск PowerShell, на котором находится ресурс), *AccessToString* (альтернативное представление правил доступа к ресурсу) и *AuditToString* (альтернативное представление правил аудита ресурса).

С помощью объектов, возвращаемых Get-Acl, можно устанавливать дескрипторы защиты и для других ресурсов, таких как каталоги, файлы и разделы реестра. Для этого необходимо:

1. Открыть консоль PowerShell с администраторскими полномочиями.
2. Получить объект дескриптора защиты для нужного ресурса.
3. Используя этот объект, настроить параметры защиты ресурса как требуется.

При настройке дескрипторов защиты в PowerShell рекомендуется явно указывать объекты, которые требуется настроить, либо объекты, которые настраивать не требуется (при необходимости — и то и другое), либо конкретный ресурс. В приведенных выше примерах использовался файл log1.txt, расположенный в каталоге C:\Logs. Так, чтобы скопировать его дескриптор защиты на файл log2.txt, введите

```
set-acl -path c:\logs\log2.txt -aclobject (get-acl c:\logs\log1.txt)
```

Этот пример легко расширить. Например, можно скопировать дескриптор защиты log1.txt на все .txt- и .log-файлы в каталоге C:\Logs:

```
$secd = get-acl c:\logs\log1.txt
set-acl -include *.txt, *.log -path c:\logs\* -aclobject $secd
```

Чтобы обработать все вложенные каталоги, получите объекты всех расположенных в них файлов с помощью Get-ChildItem. Вот пример:

```
$s = get-acl c:\logs\log1.txt
gci c:\logs -recurse -include *.txt, *.log -force | set-acl -aclobject $s
```

Здесь *gci* — это псевдоним командлета Get-ChildItem. Первая команда получает дескриптор защиты файла log1.txt. Следующая команда получает ссылки на все .txt- и .log-файлы в каталоге C:\Logs и вложенных в него каталогах. В завершение дескриптор защиты log1.txt назначается всем этим файлам.

Если требуется обработать только каталоги, не трогая файлы, необходимо ограничить результаты, возвращаемые Get-ChildItem. Объекты файлов и каталогов, которые возвращает Get-ChildItem, поддерживают свойство Mode:

```
get-childitem c:\
```

```
    Directory: C:\
Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----         6/20/2008   1:20 PM            Backup
d----         2/19/2008  10:57 AM            cabs
d----        12/18/2008   9:16 AM            Documents
-a---         9/18/2006   2:43 PM         24 autoexec.bat
-ar-s         2/29/2008   9:53 AM       8192 BOOTSECT.BAK
-a---         9/18/2006   2:43 PM         10 config.sys
```

Это свойство принимает следующие значения:

- d (каталог);
- a (архив);
- r (только для чтения);
- h (скрытый);
- s (системный).

Таким образом, чтобы обработать только каталоги, следует отобрать ресурсы, у которых значение свойства Mode начинается с *d* (то есть искать строку *d**):

```
where-object {$_.mode -like «d*»}
```

Чтобы обработать только файлы, нужно искать ресурсы по противоположному критерию:

```
where-object {$_.mode -notlike «d*»}
```

Так, чтобы скопировать дескриптор защиты C:\Data на C:\Logs и все вложенные в него каталоги, используйте следующую команду:

```
gci c:\logs -recurse -force | where-object {$_.mode -like «d*»} |
set-acl -aclobject (get-acl c:\data)
```

Чтобы скопировать дескриптор защиты C:\Data\key.txt на все файлы в C:\Logs и вложенных в него каталогах, используйте команду:

```
gci c:\logs -recurse -force | where-object {$_.mode -notlike «d*»} |
set-acl -aclobject (get-acl c:\data\key.txt)
```

мечание Для работы показанных выше примеров должны существовать соответствующие файлы и каталоги. Чтобы воспроизвес ти их на своем к омпьютере, создайте каталоги C:\Data и C:\Logs и добавьте в них несколько .txt-файлов, включая key.txt.

Работа с правилами доступа

Как видите, процедура копирования дескрипторов защиты проста и понятна, но важнее то, что она одинакова для всех типов ресурсов, поддерживающих дескрипторы защиты — файлов, каталогов, разделов реестра и пр. Чтобы научиться создавать собственные дескрипторы защиты, вам придется получ-

ше познакомиться с моделью объектов системы безопасности. В эту модель входят объекты, представляющие правила управления доступом, включая дескрипторы защиты. Правила, регулирующих доступ к ресурсам, хранятся в свойстве Access этих объектов. Правила доступа к файлам и каталогам представляют объекты типа:

```
System.Security.AccessControl.FileSystemAccessRule
```

Есть несколько способов просмотра объектов, управляющих доступом к ресурсу. Первый состоит в получении объекта-дескриптора и выводе значения его свойства Access, как показано в следующем примере:

```
$s = get-acl c:\logs
$s.access
```

```
  FileSystemRights  : FullControl
  AccessControlType : Allow
  IdentityReference : BUILTIN\Administrators
  IsInherited       : True
  InheritanceFlags  : None
  PropagationFlags  : None

  FileSystemRights  : ReadAndExecute, Synchronize
  AccessControlType : Allow
  IdentityReference : BUILTIN\Users
  IsInherited       : True
  InheritanceFlags  : ContainerInherit, ObjectInherit
  PropagationFlags  : None

  FileSystemRights  : Modify, Synchronize
  AccessControlType : Allow
  IdentityReference : NT AUTHORITY\Authenticated Users
  IsInherited       : True
  InheritanceFlags  : None
  PropagationFlags  : None
```

Этот код получает объект DirectorySecurity для каталога C:\Logs и отображает значение его свойства Access. Это свойство содержит все определения правил доступа, но работать с ними по отдельности не позволяет. В показанном выше выводе следует обратить внимание на следующие значения:

- **FileSystemRights** — действующие права файловой системы;
- **AccessControlType** — тип управления доступа (Allow или Deny);
- **IdentityResource** — пользователь или группа, на которую действует правило;
- **IsInherited** — наследуется ли это правило доступа;
- **InheritanceFlags** — порядок наследования;
- **PropagationFlags** — будет ли это правило доступа наследоваться и дальше.

Второй способ позволяет работать с отдельными правилами доступа при помощи цикла ForEach, как показано в следующем примере:

```
$s = get-acl c:\logs

foreach($a in $s.access) {

 #обрабатываем объекты управления доступом

 if ($a.identityreference -like «*administrator*») {$a | format-list *}
}
```

```
   FileSystemRights  : FullControl
   AccessControlType : Allow
   IdentityReference : BUILTIN\Administrators
   IsInherited       : True
   InheritanceFlags  : None
   PropagationFlags  : None
```

Этот код анализирует объекты правил доступа по отдельности, позволяя манипулировать ими. Данный пример ищет правила доступа, действующие для администраторов.

Администраторам часто требуется отыскать файлы и каталоги, параметры доступа к которым препятствуют проведению автоматической архивации и другим административным манипуляциям. Выше показано, как с помощью Get-ChildItem обрабатывать файлы и каталоги. Объекты файлов и каталогов, возвращаемые Get-ChildItem, поддерживают свойство Mode, которое позволяет выборочно обрабатывать каталоги либо файлы. Следующий пример создает список всех каталогов и файлов на диске C, к которым у администраторов нет полного доступа:

```
$resc = gci c:\ -recurse -force | where-object {$_.mode -notlike «*hs*»}

foreach($r in $resc) {
  $s = get-acl $r.FullName
  $found = $false

  foreach($a in $s.access) {
   if (($a.identityreference -like «*administrator*») -and `
       ($a.filesystemrights -eq «fullcontrol»)) {
           if ($a.accesscontroltype -eq «allow») { $found = $true }
   }
  }

  if (-not $found) { write-host $r.FullName}
}
```

```
C:\logs\backup
C:\logs\backup2
C:\logs\logs
C:\logs\data.ps1
C:\logs\log1.txt
C:\logs\log2.txt
C:\logs\log3.txt
C:\logs\log4.txt
C:\logs\backup\backup
C:\logs\backup\backup\b2
```

Код этого примера необходимо запускать в консоли PowerShell, открытой с администраторскими правами. В переменную $resc записывается набор объектов, представляющих все файлы и каталоги на диске C:\ за исключением скрытых и системных. Далее эти объекты обрабатываются в цикле ForEach. Сначала извлекается ACL объекта с использованием его полного имени. Далее переменная $found инициализируется значением False — этот нужно для мониторинга файлов с заданными правами доступа.

Вложенный цикл ForEach обрабатывает все объекты управления доступом, связанные с текущим файлом или каталогом. Если администраторы имеют полный доступ к текущему ресурсу, переменная $found устанавливается в True. Всего требуется проверить три свойства; сначала проверяются первые два, связанные логическим оператором AND. Если они равны True, то проверяется и третье свойство (также на предмет равенства True). В итоге если переменная $found не равна True (то есть, равна False), полное имя файла или папки выводится на консоли. В результате получается полный список файлов и папок, к которым у администраторов нет полного доступа.

Настройка разрешений на доступ к файлам и каталогам

Файлы и папки на NTFS-томах поддерживают два типа разрешений: базовые (Basic) и особые (Special), регламентирующих доступ для пользователей и групп.

Настройка базовых разрешений

Базовые разрешения на доступ к файлам и каталогам перечислены в табл. 10-1 и 10-2. Эти разрешения представляют собой комбинации специальных разрешений. Обратите внимания, что у каждого разрешения имеется флаг правила (rule flag), который необходимо указывать при создании правил доступа.

Табл. 10-1. Базовые разрешения на доступ к каталогам

Имя	Что разрешает	Флаг правила
Full Control (Полный доступ)	Чтение, запись, изменение и удаление. Обладая полным доступом к папке, пользователь может удалять из нее файлы, независимо от имеющихся у него разрешений на доступ к ним	FullControl
Modify (Изменение)	Чтение и запись файлов и вложенных папок, а также удаление папки	Modify
List Folder Contents (Содержание папки)	Просмотр списков файлов и вложенных папок, а также выполнение файлов; наследуется только папками	Synchronize
Read & Execute (Чтение и выполнение)	Просмотр файлов и вложенных папок, а также выполнение файлов; наследуется файлами и папками	ReadAndExecute
Write (Запись)	Создание файлов и папок	Write
Read (Чтение)	Просмотр файлов и вложенных каталогов	Read

Табл. 10-2. Базовые разрешения на доступ к файлам

Имя	Что разрешает	Флаг правила
Full Control (Полный доступ)	Чтение, запись, изменение и удаление	FullControl
Modify (Изменение)	Чтение, запись и удаление	Modify
Read & Execute (Чтение и выполнение)	Просмотр содержимого и запуск (для файлов программ)	ReadAndExecute
Write (Запись)	Запись и удаление содержимого файла (удаление самого файла не разрешается)	Write
Read (Чтение)	Просмотр. Этого разрешения достаточно для запуска сценария, оно необходимо для доступа к ярлыку и ресурсу, с которым он связан	Read

Чтобы запретить или разрешить доступ, при настройке базовых разрешений следует указать тип управления Allowed либо Denied, соответственно. Базовые разрешения настраивают с использованием правил доступа, которые сложены из массивов, определяющих:

- пользователя или группу, на которую действует правило;
- действующее разрешение;
- состояние (Allow или Deny).

Таким образом, общий синтаксис для работы с правилами доступа имеет следующий вид:

«*Пользователь*», «*Разрешение*», «*Тип*»

где *Пользователь* — имя пользователя или группы, на которую действует правило, *Разрешение* — имя базового разрешения, а *Тип* — его состояние (Allow или Deny). Имена пользователей и групп указывают в формате *ИМЯ_КОМПЬЮТЕРА\Имя_пользователя* или *ИМЯ_ДОМЕНА\Имя_пользователя*. В следующем примере пользователю BackupOpUser предоставляется полный доступ:

```
«BackupOpUser», «FullControl», «Allow»
```

Тот же синтаксис можно использовать и при настройке разрешений для каталогов, но в этом случае допустим и расширенный синтаксис, который позволяет настраивать разрешения для каталога и его содержимого:

«*Пользователь*», «*Разрешение*», «*Флаг1*», «*Флаг2*»,»*Тип*"

где *Пользователь* — имя пользователя или группы, на которую действует правило; *Разрешение* — имя разрешения; *Флаг1* — флаг, управляющий наследованием; *Флаг2* — флаг, управляющей распространением унаследованных правил; *ControlType* — состояние разрешения (Allow или Deny). В следующем примере группа DeploymentTesters получает полный доступ к некоторому каталогу и вложенным в него каталогам:

```
«BackupOpUser», «FullControl», «ContainerInherit», «None», «Allow»
```

Флаг, управляющий наследованием, может принимать следующие значения:

- **None** — правило доступа не наследуется;
- **ContainerInherit** — наследуется вложенными каталогами (дочерними объектами-контейнерами);
- **ObjectInherit** — наследуется файлами (дочерними объектами);
- **ContainerInherit**, **ObjectInherit** — наследуется файлами и вложенными каталогами (дочерними объектами и объектами-контейнерами).

Флаг, управляющий дальнейшим распространением унаследованных правил, принимает следующие значения:

- **None** — правило доступа наследуется без изменений;
- **InheritOnly** — правило доступа наследуется «прямыми потомками» объекта (дочерними объектами и объектами-контейнерами);
- **NoPropagateInherit** — правило доступа наследуется дочерними объектами и объектами-контейнерами, но не наследуется «внучатыми» объектами (т.е. объектами, вложенными в дочерние объекты-контейнеры);
- **NoPropagateInherit**, **InheritOnly** — правило доступа действует только на дочерние объекты-контейнеры.

Для добавления правил доступа к ресурсам используются методы SetAccessRule() и AddAccessRule() объекта ACL, а для удаления правил доступа — метод RemoveAccessRule() того же объекта. Как сказано выше, правила доступа представлены объектами System.Security.AccessControl.FileSystemAccessRule.

Проще всего добавлять и удалять правила доступа следующим образом:

1. Получите объект ACL для ресурса, доступ к которому требуется настроить, либо объект ACL с набором разрешений, максимально близким к нужному.
2. Создайте нужное число объектов System.Security.AccessControl.FileSystemAccessRule и запишите в них требуемые разрешения;
3. Вызывая AddAccessRule() или RemoveAccessRule() объекта ACL (см. шаг 1), добавьте или удалите соответствующие правила доступа.
4. Чтобы внесенные изменения вступили в силу, примените измененный объект ACL к нужному ресурсу.

Рассмотрим следующий пример (чтобы он сработал, должна существовать группа Dev в домене Cpandl):

```
$acl = get-acl c:\logs
$perm = «cpandl\dev»,«fullcontrol»,«allow»
$r = new-object system.security.accesscontrol.filesystemaccessrule $perm
$acl.addaccessrule($r)
$acl | set-acl c:\logs
```

Этот код получает объект ACL для каталога C:\Logs, записывает правила доступа в переменную $perm и создает экземпляр класса FileSystemAccessRule для хранения этого правила. Для добавления нового разрешения к ранее полученному объекту ACL вызывается его метод AddAccessRule(). В завершение измененный объект ACL применяется к каталогу C:\Logs с помощью командлета Set-Acl.

Достаточного небольшой доработки, чтобы этот код смог настраивать разрешения не только для каталога C:\Logs, но и всех содержащихся в нем ресурсов (для работы этого примера нужна группа Test на локальном компьютере Room5):

```
$acl = get-acl c:\logs
$perm = «room5\test»,«fullcontrol»,«allow»
$r = new-object system.security.accesscontrol.filesystemaccessrule $perm
$acl.addaccessrule($r)

$resc = gci c:\logs -recurse -force
foreach($f in $resc) {

  write-host $f.fullname
  $acl | set-acl $f.FullName
}
```

```
C:\logs\backup
C:\logs\backup2
C:\logs\logs
C:\logs\data.ps1
C:\logs\log1.txt
C:\logs\log2.txt
C:\logs\log3.txt
C:\logs\log4.txt
C:\logs\backup\backup
C:\logs\backup\backup\b2
```

Для простоты отслеживания изменений в этот пример добавлена команда, которая выводит имена файлов и каталогов с измененными разрешениями.

Настройка особых разрешений

Особые разрешения на доступ к файлам и каталогам перечислены ниже в табл. 10-3. Поскольку базовые разрешения представляют собой различные комбинации особых разрешений, последние также называют *атомарными* (atomic permissions). Как и базовым, особым разрешениям сопоставлены флаги, которые следует указывать при создании правил доступа. Если разрешению сопоставлено два флага, достаточно указать один из них.

Табл. 10-3. Особые разрешения

Имя	Что разрешает	Флаг правила
Traverse Folder/ Execute File (Обзор папок/Выполнение файлов)	Traverse Folder — доступ к папке даже без разрешения на чтение ее содержимого; Execute File — запуск программ	Traverse, ExecuteFile
List Folder/Read Data (Содержание папки/Чтение данных)	List Folder — просмотр имен файлов и папок; Read Data — чтение содержимого файлов	ListDirectory, ReadData
Read Attributes (Чтение атрибутов)	Чтение атрибутов Read-Only, Hidden, System и Archive у файлов и папок	ReadAttributes
Read Extended Attributes (Чтение дополнительных атрибутов)	Чтение файловых атрибутов (именованных потоков данных), включая поля Summary (Title, Subject, Author) и других данных	ReadExtendedAttributes
Create Files/Write Data (Создание файлов/Запись данных)	Create Files — создание файлов в папке; Write Data — перезапись существующих файлов (но не дозапись данных к ним: это прерогатива разрешения Append Data; см. ниже)	CreateFiles, WriteData

(*см. след. стр.*)

Табл. 10-3. Особые разрешения

Имя	Что разрешает	Флаг правила
Create Folders/ Append Data (Создание папок/ Дозапись данных)	Create Folders — создание вложенных папок; Append Data — дозапись (но не перезапись: это прерогатива разрешения Write Data; см. выше) данных к концу существующих файлов	CreateFolders, AppendData
Write Attributes (Запись атрибутов)	Изменение атрибутов Read-Only, Hidden, System и Archive файлов и папок	WriteAttributes
Write Extended Attributes (Запись дополнительных атрибутов)	Изменение файловых атрибутов (именованных потоков данных), включая поля Summary (Title, Subject, Author) и других данных	WriteExtended-Attributes
Delete Subfolders and Files (Удаление подпапок и файлов)	Удаление содержимого папки даже без разрешения Delete для отдельных вложенных файлов и папок	DeleteSubdirectories-AndFiles
Delete (Удаление)	Удаление файлов и папок. Если удаляемая папка не пуста, и у вас нет разрешения Delete для *всех* содержащихся в ней файлов и папок, удалить эту папку удастся только при наличии разрешения Delete Subfolders and Files	Delete
Read Permissions (Чтение разрешений)	Чтение любых разрешений на доступ к файлу или папке	ReadPermissions
Change –Permi-ssions (Смена разрешений)	Изменение любых разрешений на доступ к файлу или папке	ChangePermissions
Take Ownership (Смена владельца)	Вступление во владение файлом или папкой. По умолчанию администраторы имеют это разрешение и могут наделять им других пользователей	TakeOwnership

В табл. 10-4 и 10-5 показаны комбинации особых разрешений, составляющие базовые разрешения на доступ к файлам и папкам.

Табл. 10-4. Особые разрешения на доступ к папкам

Особые разрешения	Full control	Modify	Read & execute	List folder contents	Read	Write
Traverse Folder/ Execute File	X	X	X	X		
List Folder/Read Data	X	X	X	X	X	
Read Attributes	X	X	X	X	X	
Read Extended –Attributes	X	X	X	X	X	
Create Files/Write Data	X	X				X
Create Folders/ Append Data	X	X				X
Write Attributes	X	X				X
Write Extended –Attributes	X	X				X
Delete Subfolders and Files	X					
Delete	X	X				
Read Permissions	X	X	X	X	X	X
Change Permissions	X					
Take Ownership	X					

Табл. 10-5. Особые разрешения на доступ к файлам

Особые разрешения	Full control	Modify	Read & execute	Read	Write	
Traverse Folder/Execute File	X	X	X			
List Folder/Read Data	X	X	X	X		
Read Attributes	X	X	X	X		
Read Extended Attributes	X	X	X	X		
Create Files/Write Data	X	X			X	
Create Folders/Append Data	X	X			X	
Write Attributes	X	X			X	
Write Extended Attributes	X	X			X	

(*см. след. стр.*)

Табл. 10-5. Особые разрешения на доступ к файлам

Особые разрешения	Full control	Modify	Read & execute	Read	Write
Delete Subfolders and Files	X				
Delete	X	X			
Read Permissions	X	X	X	X	X
Change Permissions	X				
Take Ownership	X				

Особые разрешения настраивают так же, как и базовые, назначая ресурсам правила доступа с помощью методов SetAccessRule() и AddAccessRule() объекта ACL. Для удаления правил доступа применяют метод RemoveAccessRule() того же объекта.

Рассмотрим следующий пример (для его работы должна существовать группа Dev в домене Cpandl):

```
$acl = get-acl c:\logs
$p1 = «cpandl\dev», «executefile», «allow»
$r1 = new-object system.security.accesscontrol.filesystemaccessrule $p1
$acl.addaccessrule($r1)

$p2 = «cpandl\dev», «listdirectory», «allow»
$r2 = new-object system.security.accesscontrol.filesystemaccessrule $p2
$acl.addaccessrule($r2)

$acl | set-acl c:\logs
```

Этот код получает объект ACL для каталога C:\Logs, определяет правило доступа и записывает его в переменную $p1 и создает соответствующий объект FileSystemAccessRule, после чего добавляет новое разрешение к объекту ACL вызовом метода AddAccessRule(). Далее определяется второе правило доступа и записывается в переменную $p2, затем создается еще один объект FileSystemAccessRule и новое разрешение добавляется к тому же объекту ACL вызовом метода AddAccessRule(). В завершение измененный объект ACL применяется к каталогу C:\Logs с помощью командлета Set-Acl.

Смена владельца

Создатель файла или каталога в Windows может и не быть его владельцем, вместо него право владения имеет тот, кто обладает непосредственным контролем над файлом или каталогом. Владельцы файлов и каталогов могут раздавать пользователям различные разрешения в отношении этих ресурсов, включая разрешение на смену их владельца.

Кто станет владельцем, зависит от того, где был создан файл или каталог. По умолчанию владельцем ресурса становится его создатель, но право

владения может переходить и делегироваться различными способами. Во владение ресурсом может вступить любой администратор, а также пользователь или группа, обладающая разрешением Take Ownership. Владельцем ресурса также может стать любой пользователь — обладатель права Restore Files And Directories (Восстановление файлов и каталогов), например член группы Backup Operators (Операторы резервного копирования). Наконец, текущий владелец ресурса может передать владение любому другому пользователю.

Для вступления во владение файлом или каталогом вызывают метод SetOwner() объекта ACL. Проще всего это сделать следующим образом:

1. Получите объект ACL для нужного ресурса.
2. Получите IdentityReference для пользователя или группы, которой перейдет владение. У этого пользователя или группы должно быть разрешение на доступ к ресурсу (см. выше).
3. Вызовите метод SetOwner, чтобы предать владение нужному пользователю или группе.
4. Назначьте измененный объект ACL ресурсу.

   Рассмотрим следующий пример:

```
$acl = get-acl c:\logs
$found = $false
foreach($rule in $acl.access) {
 if ($rule.identityreference -like «*administrators*») {
   $global:ref = $rule.identityreference; $found = $true; break}
}

if ($found) {
 $acl.setowner($ref)
 $acl | set-acl c:\logs
}
```

Этот код получает объект ACL для каталога C:\Logs. Далее проверяются все правила доступа в составе этого объекта в поисках правил, действующих на нужную нам группу. При обнаружении такого правила в переменную $ref записывается объект IdentityReference для заданной группы, переменная $found устанавливается в $true, после чего цикл ForEach прерывается. После выхода из цикла проверяется значение $found. Если оно равно True, в ранее полученном объекте ACL устанавливается разрешение на смену владельца, после чего этот объект назначается каталогу C:\Logs с помощью командлета Set-Acl.

Настройка аудита файлов и каталогов

Администраторы используют аудит для наблюдения за событиями на вверенных им компьютерах. В ходе аудита производится сбор информации об

использовании ресурсов, таких как файлы и каталоги. Каждый раз, когда происходит событие, для которого настроен аудит, сведения об этом событии записываются в системный журнал Security. Для просмотра журнала используется консоль Event Viewer (Просмотр событий). В большинстве случаев для выполнения аудита необходимо членство в группе Administrators (Администраторы) либо разрешение Manage Auditing And Security Log (Управление аудитом и журналом безопасности), назначенное через групповую политику.

Политики аудита необходимы для обеспечения безопасности и целостности вычислительных систем. Практически каждый компьютер в сети должен быть охвачен тем или иным видом аудита. Политики аудита файлов и каталогов задают с использованием правил аудита (auditing rules), сложенных из массивов, которые определяют:

- пользователя или группу, на которую действует правило;
- разрешение, использование которого отслеживается;
- тип аудита.

Таким образом, общий синтаксис правил аудита выглядит так:

```
«Пользователь», «Разрешение», «Тип»
```

где *Пользователь* — имя пользователя или группы, на которую действует аудит, *Разрешение* — разрешение, использование которого отслеживается, а *Тип* — тип аудита. Тип аудита Success отслеживает успешное применение заданного разрешения, тип Failure — неудачного использование того же разрешения, тип None отключает аудит, а тип Both — включает аудит успешного и неудачного использования этого разрешения.

Как и в случае разрешений системы безопасности, при настройке аудита имя пользователя задается в формате ИМЯ_КОМПЬЮТЕРА\Имя_пользователя либо ИМЯ_ДОМЕНА\Имя_пользователя. Следующее правило аудита отслеживает попытки обращения к ресурсу пользователей из домена, оканчивающиеся неудачей из-за отсутствия нужных разрешений:

```
«CPANDL\USERS», «ReadData», «Failure»
```

При настройке аудита каталогов можно использовать общий синтаксис правил, но в этом случае доступен и расширенный синтаксис правил аудита:

```
«Пользователь», «Разрешение», «Флаг1», «Флаг2», «Тип»
```

где *Пользователь* — имя пользователя или группы, на которую действует правило, *Разрешение* — разрешение, использование которого отслеживается, *Флаг1* — флаг, управляющий наследованием, *Флаг2* — флаг, управляющий дальнейшим распространение унаследованных правил, а *Тип* — тип аудита. В следующем примере правило аудита назначается ресурсу и вложенным в него файлам и папкам:

```
«CPANDL\USERS», «ReadData», «ContainerInherit», «None», «Failure»
```

Флаг, управляющий наследованием, может принимать следующие значения:

- **None** — правило доступа не наследуется;
- **ContainerInherit** — наследуется вложенными каталогами (дочерними объектами-контейнерами);
- **ObjectInherit** — наследуется файлами (дочерними объектами);
- **ContainerInherit**, **ObjectInherit** — наследуется файлами и вложенными каталогами (дочерними объектами и объектами-контейнерами).

Флаг, управляющий дальнейшим распространением унаследованных правил, принимает следующие значения:

- **None** — правило доступа наследуется без изменений;
- **InheritOnly** — правило доступа наследуется «прямыми потомками» объекта (дочерними объектами и объектами-контейнерами);
- **NoPropagateInherit** — правило доступа наследуется дочерними объектами и объектами-контейнерами, но не наследуется «внучатыми» объектами (т.е. объектами, вложенными в дочерние объекты-контейнеры);
- **NoPropagateInherit**, **InheritOnly** — правило доступа действует только на дочерние объекты-контейнеры.

Для добавления правил аудита к ресурсам используются методы SetAuditRule() и AddAuditRule() объекта ACL, а для удаления правил доступа — метод RemoveAuditRule() того же объекта. Как сказано выше, правила доступа представлены объектами System.Security.AccessControl.FileSystemAuditRule.

Проще всего добавлять и удалять правила доступа следующим образом:

1. Получите объект ACL для ресурса, доступ к которому требуется настроить, либо объект ACL с набором правил аудита, максимально близким к нужному.
2. Создайте нужное число объектов System.Security.AuditControl.FileSystemAccessRule и запишите в них требуемые правила аудита;
3. Вызывая AddAuditRule () или RemoveAuditRule () объекта ACL (см. шаг 1), добавьте или удалите соответствующие правила аудита.
4. Чтобы внесенные изменения вступили в силу, примените измененный объект ACL к нужному ресурсу.

Рассмотрим следующий пример (для работы этого и следующего примеров должна существовать группа Users в домене Cpandl):

```
$acl = get-acl d:\data
$audit = «cpandl\users», «readdata», «failure»
$r = new-object system.security.accesscontrol.filesystemauditrule $audit
$acl.addauditrule($r)
$acl | set-acl d:\data
```

Этот код получает объект ACL для каталога D:\Data, записывает правила аудита в переменную $audit и создает экземпляр класса FileSystemAuditRule для хранения этого правила. Для добавления нового правила аудита к ранее

полученному объекту ACL вызывается его метод AddAuditRule (). В завершение измененный объект ACL применяется к каталогу D:\Data с помощью командлета Set-Acl.

Достаточного небольшой доработки, чтобы этот код смог настраивать правила аудита не только для каталога D:\Data, но и всех содержащихся в нем ресурсов:

```
$acl = get-acl d:\data
$audit = «cpandl\users»,«readdata»,«failure»
$r = new-object system.security.accesscontrol.filesystemauditrule $audit
$acl.addauditrule($r)

$resc = gci d:\data -recurse -force
foreach($f in $resc) {

  write-host $f.fullname
  $acl | set-acl $f.FullName
}
```

```
  D:\data\backup
  D:\data\backup\historydat.txt
  D:\data\logs\datlog.log
  D:\data\data.ps1
  D:\data\transcript1.txt
  D:\data\transcript2.txt
  D:\data\backup\backup
```

Для простоты отслеживания изменений в этот пример добавлена команда, которая выводит имена файлов и каталогов с измененными правилами аудита.

Глава 11

Управление TCP/IP-сетями, сетевыми дисками и принтерами

В задачи администраторов входит обеспечение взаимодействия сетевых компьютеров и общего доступа к ресурсам, таким как сетевые диски и сетевые принтеры, с использованием встроенных сетевых функций Windows. Взаимодействие в сети осуществляется, как правило, с использованием TCP/IP.

Управление сетевыми дисками и каталогами

Открыв общий доступ через сеть к диску или каталогу, вы сделаете его содержимое доступным определенной группе пользователей. Сетевой диск можно создать только на основе диска с файловой системой NTFS либо каталога, расположенного на NTFS-диске. Доступ к сетевым дискам регламентируется двумя наборами разрешений: NTFS-разрешениями и разрешениями на доступ к сетевому диску. Используя их, можно задавать пользователей, которым разрешен доступ, и настраивать уровень доступа. Чтобы открыть доступ к файлам через сеть, перемещать сами файлы не требуется.

Разрешения на доступ к сетевому диску применяются только при обращении к ресурсу с других компьютеров сети; разрешения файловой системы действуют при обращении как с локального, так и с сетевых компьютеров. При обращении к ресурсу с удаленного компьютера сначала применяются разрешения на доступ к сетевому ресурсу, а затем разрешения файловой системы. При обращении с локального компьютера действуют только разрешения файловой системы.

Совет Возможность создания сетевых дисков определяется настройками, сделанными в Network And Sharing Center (Центре управления сетями и общим доступом). Чтобы открыть его, щелкните Start | Network (Пуск | Сеть) либо, на панели инструментов Проводника щелкните Network And Sharing Center (Центр управления сетями и общим доступом).

Получение сведений о сетевых дисках

Сетевыми дисками управляют иначе, чем локальными. Для работы с сетевыми дисками в PowerShell служит класс Win32_Share. Чтоб получить сведения о доступных сетевых ресурсах, введите **Get-Wmiobject -Class Win32_Share**:

```
get-wmiobject -class win32_share
```

```
Name            Path                                    Description
----            ----                                    -----------
ADMIN$          C:\Windows                              Remote Admin
C$              C:\                                     Default share
D$              D:\                                     Default share
E$              E:\                                     Default share
HP3505-PCL6     HP CLJ CP3505 PCL6,LocalsplOnly         HP CLJ CP3505
IPC$            Remote IPC
print$          C:\Windows\system32\spool\drivers       Printer
Driver
W$              W:\                                     Default share
```

Сетевые ресурсы поддерживают ряд свойств. Для просмотра свойств определенного ресурса используют WMI-запросы и выражения на основе Where-Object. В следующем примере выводятся свойства сетевого диска C$:

```
get-wmiobject -class win32_share | where-object {$_.Name -eq «C$»} |
format-list *
```

```
Status           : OK
Type             : 2147483648
Name             : C$
__GENUS          : 2
__CLASS          : Win32_Share
__SUPERCLASS     : CIM_LogicalElement
__DYNASTY        : CIM_ManagedSystemElement
__RELPATH        : Win32_Share.Name="C$"
__PROPERTY_COUNT : 10
__DERIVATION     : {CIM_LogicalElement, CIM_ManagedSystemElement}
__SERVER         : TECHPC89
__NAMESPACE      : root\cimv2
__PATH           : \\TECHPC89\root\cimv2:Win32_Share.Name="C$"
AccessMask       :
AllowMaximum     : True
Caption          : Default share
Description      : Default share
InstallDate      :
MaximumAllowed   :
```

```
Path             : C:\
Scope            : System.Management.ManagementScope
Options          : System.Management.ObjectGetOptions
ClassPath        : \\TECHPC89\root\cimv2:Win32_Share
Properties       : {AccessMask, AllowMaximum, Caption, Description...}
SystemProperties : {__GENUS, __CLASS, __SUPERCLASS, __DYNASTY...}
Qualifiers       : {dynamic, Locale, provider, UUID}
Site             :
Container        :
```

Примечание Чтобы опросить несколько компьютеров, используйте параметр –ComputerName командлета Get-Wmiobject, например так:

```
get-wmiobject -class win32_share -computername Server43, Server27,
Server82
```

Свойство Type указывает тип сетевого ресурса. Допустимы следующие типы:

- **0** — сетевой диск;
- **1** — сетевой принтер;
- **2** — общее сетевое устройство;
- **3** — сетевой IPC-ресурс;
- **2147483648** — административный сетевой диск;
- **2147483649**— административный сетевой принтер;
- **2147483650**— административное сетевое устройство;
- **2147483651**— административный IPC-ресурс.

Настройка сетевых ресурсов

Используя консоль PowerShell, открытую с администраторскими полномочиями, можно настраивать некоторые параметры сетевых ресурсов, включая максимальное число пользователей и описание ресурса. Свойство The MaxAllow определяет максимальное число одновременных подключений к ресурсу. Свойство содержит текст с описанием ресурса.

Свойства MaxAllow и Description устанавливают с помощью методов SetShareInfo() объекта Win32_Share. Вот общий синтаксис этого метода:

```
$ОбъектРеурса.setShareInfo(МаксЧислоПодключений,Описание)
```

где *$ОбъектРесурса* — ссылка на объект Win32_Share object, *МаксЧисло Подключений* — желаемое максимальное число одновременных подключений, а *Описание* — текст описания. Допустимое число пользователей — от 1 до 2 147 483 647, значение 0 соответствует неограниченному числу пользователей. Среди полезных свойств объекта Win32_Share отметим также Name, Path, Caption, Description и MaximumAllowed.

Чтобы получить ссылку на объект Win32_Share, сохраните этот объект в переменной. Задать WMI-запрос проще всего с помощью параметра –Filter. Этот параметр принимает Where-конструкцию, которая работает как фильтр объектов. Следующая команда получает ссылку на объект, представляющий сетевой ресурс Logs, и сохраняет ее в переменной $share:

```
$share = get-wmiobject -class win32_share -filter «name='logs'»
```

Обратите внимание на синтаксис конструкции Where, переданной как строковый аргумент параметра –Filter. Заданный сетевой ресурс должен существовать, в противном случае значение установлено не будет. Поскольку аргумент Filter заключен в двойные кавычки, искомое значение свойства необходимо взять в одинарные кавычки. Получив ссылку на объект Win32_Share, вызовите метод SetShareInfo(), чтобы установить желаемые значения, например, так:

```
$share = get-wmiobject -class win32_share -filter «name='logs'»
$share.setShareInfo(255,«Logging Share»)
```

```
__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 2
```

В этом выводе важно возвращаемое значение (ReturnValue). Если оно равно 0, операция выполнена успешно, любое другое значение говорит об ошибке. В данном случае ReturnValue = 2 свидетельствует об ошибке из-за отказа в доступе. Такие ошибки возникают, если вы забыли запустить консоль с администраторскими полномочиями, либо у вас нет достаточных разрешений для управления этим сетевым ресурсом. Возвращаемое значение можно сохранить в переменой для дальней обработки, например:

```
$share = get-wmiobject -class win32_share -filter «name='logs'»
$results = $share.setShareInfo(255,«Logging Share»)
write-host $results.returnvalue
```

```
0
```

В этом примере мы получаем ссылку на объект ресурса Logs, устанавливаем для него 255 как максимальное число пользователей и в качестве описания — строку «Logging Share». Чтобы убедиться в том, что изменения

были внесены, возвращаемое значение записывается в переменную и выводится на консоль.

Создание сетевых дисков и папок

Для создания сетевых ресурсов в командной строке PowerShell, открытой с администраторскими полномочиями, используется статический метод Create() класса Win32_Share. Его общий синтаксис выглядит так:

```
$Объект_ресурса.Create(Путь, Имя, Тип, МаксЧисло_пользователей, Описание)
```

где *$Объект_ресурса* — ссылка на объект Win32_Share; *Путь* — путь к каталогу, который требуется сделать доступным через сеть; *Имя* — имя сетевого ресурса; *Тип* — допустимый идентификатор типа; *МаксЧисло_пользователей* — желаемое максимальное число пользователей; *Описание* — описание ресурса. При наличии необходимых разрешений PowerShell сможет создать ресурс, если только указанное имя уже не «занято» другим сетевым ресурсом.

Проще всего создавать экземпляры класса Win32_Share с помощью его псевдонима [wmiclass], как показано в следующем примере:

```
$share = [wmiclass]«Win32_Share»
```

Получив экземпляр класса Win32_Share, можно создать сетевой ресурс.

```
$share.Create(«c:\data», «Data», 0, 255,«Data Share»)
```

Эта команда возвращает те же результаты, что и вызов SetShareInfo(). Возвращаемое значение 0 говорит об успешном завершении команды, любое другое значение свидетельствует об ошибке. Вот полный код этого примера:

```
$share = [wmiclass]«Win32_Share»
$results = $share.Create(«c:\data», «Data», 0, 0,«Data Share»)
write-host $results.returnvalue
```

```
0
```

Здесь мы получаем ссылку на класс Win32_Share и создаем ресурс Data, сопоставленный каталогу C:\Data. Для нового ресурса задан тип «сетевой диск» неограниченное число пользователей. Чтобы убедиться, что изменения были внесены, возвращаемое значение записывается в переменную и выводится на консоли.

Примечание Созданному сетевому ресурсу можно назначить разрешения на доступ. По умолчанию неявно определенной группе Everyone разрешен доступ для чтения (Read), остальным пользователям и группам доступ запрещен.

Удаление сетевых ресурсов

Для удаления сетевых ресурсов в командной строке PowerShell, запущенной с администраторскими полномочиями, используется метод Delete() объекта Win32_Share:

```
$Объект_ресурса.delete()
```

где *$Объект_ресурса* — ссылка на объект Win32_Share ресурса, который требуется удалить. Как и другие методы, Delete() возвращает значение, по которому можно узнать об успешном или неудачном завершении вызова.

Чтобы удалить ранее созданный ресурс Data, введите

```
$share = get-wmiobject -class win32_share -filter «name='data'»
$results = $share.delete()

write-host $results.returnvalue
```

```
0
```

Управление принтерами

Для обеспечения возможности печати с Windows-компьютеров через сеть администраторам приходится решать две задачи: настраивать серверы печати, а также рабочие станции для печати через сеть. Сервер печати — центральный сервер, к которому подключены общие сетевые принтеры. Чтобы предоставить пользователям домена доступ к общим принтерам, следует настроить в домене сервер печати.

В сетях используется два типа печатающих устройств: *локальные* и *сетевые*. Локальным печатающим устройством называется принтер, подключенный к тому же компьютеру, на который вошел работающий с ним пользователь. Сетевым печатающим устройством называется принтер, настроенный для удаленного доступа через сеть. Такой принтер может быть подключен к серверу печати либо непосредственно к сети через встроенную сетевую плату.

Функции сервера печати может выполнять рабочая станция или сервер, к которому подключены общие принтеры, либо принтер с самостоятельным подключением к сети. Недостатком использования рабочей станции в качестве сервера печати является малое число одновременных подключений; у серверов это ограничение менее жесткое. Использование Windows Server 2008 и выше позволяет не беспокоиться о нехватке сетевых подключений.

Задача сервера печати — обеспечение общего доступа к печатающему устройству через сеть и управление очередями печати. Основное преимущество использования серверов печати в том, что они предоставляют централизованно управляемую очередь печати, и освобождают от необходимости установки драйверов принтера на все рабочие станции.

Впрочем, можно обойтись и без сервера печати, если принтер поддерживает прямое подключение к сети. В этом случае с сетевым принтером рабо-

тают почти так же, как с локальным, с тем лишь отличием, что к сетевому принтеру подключается множество пользователей, и у каждого из них своя очередь печати. Управление очередями печати при этом осуществляется независимо, что затрудняет администрирование и устранение неполадок.

Получение сведений о принтерах

Для работы с принтерами в PowerShell используется класс Win32_Printer. Для просмотра доступных принтеров введите **Get-Wmiobject -Class Win32_Printer**:

```
get-wmiobject -class win32_printer
```

```
Location      : 18th Floor
Name          : HP CLJ CP3505 PCL6
PrinterState  : 131072
PrinterStatus : 1
ShareName     : HP3505-PCL6
SystemName    : PrinterServer08

Location      : 17th Floor
Name          : magicolor 2300 DL
PrinterState  : 0
PrinterStatus : 3
ShareName     : Color2300
SystemName    : PrintServer21
```

Примечание Чтобы опросить несколько компьютеров, вызывайте командлет Get-WmiObject с параметром –ComputerName, например:

```
get-wmiobject -class win32_printer -computername EngPC45, TechPC15,
EngPC82
```

У объекта принтера имеются разнообразные свойства, доступные администратору. Для просмотра свойств конкретного принтера выводимые результаты можно фильтровать с использованием WMI-запросов и выражений на основе Where-Object. В следующем примере выводятся свойства общего принтера HP3505-PCL6:

```
get-wmiobject -class win32_printer -filter «ShareName='HP3505-PCL6'» |
format-list *
```

```
Status                    : OK
Name                      : HP CLJ CP3505 PCL6
__GENUS                   : 2
__CLASS                   : Win32_Printer
__SUPERCLASS              : CIM_Printer
__DYNASTY                 : CIM_ManagedSystemElement
__RELPATH                 : Win32_Printer.DeviceID="HP CLJ CP3505
```

```
PCL6"
__PROPERTY_COUNT            : 86
__DERIVATION                : {CIM_Printer, CIM_LogicalDevice, CIM_
LogicalElement, CIM_ManagedSystemElement}
__SERVER                    : TECHPC87
__NAMESPACE                 : root\cimv2
__PATH                      : \\TECHPC87\root\cimv2:Win32_Printer.
DeviceID="HP CLJ CP3505 PCL6"
Attributes                  : 588
Availability                :
AvailableJobSheets          :
AveragePagesPerMinute       : 0
Capabilities                : {4, 2, 3, 5}
CapabilityDescriptions      : {Copies, Color, Duplex, Collate}
Caption                     : HP CLJ CP3505 PCL6
Description                 :
DetectedErrorState          : 5
DeviceID                    : HP CLJ CP3505 PCL6
Direct                      : False
DoCompleteFirst             : True
DriverName                  : HP Color LaserJet CP3505 PCL 6
```

Для проверки параметров TCP/IP сетевого принтера используется класс Win32_TcpIpPrinterPort. Команда **Get-Wmiobject -Class Win32_TcpIp-PrinterPort** выводит информацию о настроенных TCP/IP-портах принтеров:

```
get-wmiobject -class win32_tcpipprinterport
```

```
__GENUS                 : 2
__CLASS                 : Win32_TCPIPPrinterPort
__SUPERCLASS            : CIM_ServiceAccessPoint
__DYNASTY               : CIM_ManagedSystemElement
__RELPATH               : Win32_TCPIPPrinterPort.Name="192.168.0.90"
__PROPERTY_COUNT        : 17
__DERIVATION            : {CIM_ServiceAccessPoint, CIM_LogicalElement,
CIM_ManagedSystemElement}
__SERVER                : TECHPC87
__NAMESPACE             : root\cimv2
__PATH                  : \\TECHPC87\root\cimv2:Win32_TCPIPPrinterPort.
Name="192.168.0.90"
ByteCount               :
Caption                 :
CreationClassName       : Win32_TCPIPPrinterPort
Description             :
HostAddress             : 192.168.0.90
InstallDate             :
Name                    : 192.168.0.90
```

```
PortNumber              : 9100
Protocol                : 1
Queue                   :
SNMPCommunity           :
SNMPDevIndex            :
SNMPEnabled             : False
Status                  :
SystemCreationClassName : Win32_ComputerSystem
SystemName              :
Type                    :
```

Примечание Чтобы опросить неск олько к омпьютеров, вызывайте к омандлет Get-WmiObject с параметром –ComputerName, как в следующем примере:

```
get-wmiobject -class win32_tcpipprinterport -computername EngPC45,
TechPC15, EngPC82
```

Проверка драйверов принтеров

Драйверы для большинства стандартных принтеров автоматически устанавливаются на компьютерах, работающих под управлением операционных систем Windows. В Windows драйверы хранятся в каталоге %SystemRoot%\Inf. Имена файлов драйверов принтеров начинаются на «Prn»; файлы определений драйверов обладают расширениями .pnf, а файлы, собственно, драйверов — .inf. Зная это, можно написать примерно такую команду для вывода списка доступных драйверов принтеров:

```
get-childitem ((get-item env:systemroot).value + «\inf») -exclude *.pnf |
where-object {$_.name -match «prn»} | format-list Fullname
```

```
D FullName : C:\Windows\inf\prnao001.inf
FullName : C:\Windows\inf\prnbr001.inf
FullName : C:\Windows\inf\prnca001.inf
```

В этом примере командлет Get-ChildItem ищет в каталоге %System Root%\Inf файлы с именами, начинающимися на «Prn», с расширением .pnf. Ко-манда Get-Item Env:Systemroot получает значение переменной окружения %SystemRoot%, сцепляет ее со строкой «\Inf», получая полный путь к папке с драйверами, например C:\Windows\Inf.

Управление подключениями к принтерам

Во время работы над этой книгой в PowerShell 2.0 не было удобных средств для создания подключений к принтерам и управлениям ими, несмотря на поддержку WMI. Ниже я поделюсь своим любимым приемом, позволяющим обойти это ограничение. Этот прием основан на использовании сервера сценариев Windows (Windows Script Host, WSH) и COM. Прежде всего, не-

обходимо создать экземпляр объекта WScript.Network посредством COM, после чего можно будет использовать его методы для работы с принтерами. Заметьте, что для настройки принтера для некоторого пользователя необходимо войти в систему под его учетной записью либо воспроизвести назначенные этому пользователю разрешения.

Принтер по умолчанию является основным для пользователя. Именно этот принтер используется при отправке документа на печать без предварительного выбора принтера. Для установки принтера по умолчанию вызовите метод SetDefaultPrinter() объекта WScript.Network — он автоматически запишет в профиль пользователя новый принтер по умолчанию.

При установке принтера по умолчанию необходимо указать имя сетевого принтера либо путь к нему, например **«HP CLJ CP3505 PCL6»** или **«\\PrintServer72\ColorPrinter03»**. Вот пример:

```
$wn = New-Object -ComObject WScript.Network
$wn.SetDefaultPrinter(«\\PrintServer72\ColorPrinter03»)
```

Управление подключениями к сетевым принтерам во многом напоминает управление подключениями к сетевым дискам. Для подключения принтеров используется метод AddWindowsPrinterConnection() объекта WScript.Network, а для отключения — метод RemovePrinterConnection() того же объекта.

Метод AddWindowsPrinterConnection() принимает путь к сетевому принтеру:

```
$wn = New-Object -ComObject WScript.Network
$wn.AddWindowsPrinterConnection(«\\PrintServer72\ColorPrinter03»)
```

После вызова AddWindowsPrinterConnection() WSH пытается подключиться к серверу печати для проверки доступности принтера. Если проверка прошла удачно, Windows создает подключение к принтеру и автоматически передает с сервера печати на локальный компьютер подходящие драйверы.

Закончив работу с сетевым принтером, можно отключить его, удалив подключение вызовом метода RemovePrinterConnection(). Этот метод принимает «локальное» имя отключаемого принтера, например:

```
$wn = New-Object -ComObject WScript.Network
$wn.RemovePrinterConnection(«HP CLJ CP3505 PCL6»)
```

Управление TCP/IP-сетями

PowerShell предоставляет динамическую среду для работы с TCP/IP-сетями. Чтобы установить на компьютере поддержку сети, следует настроить сетевой протокол TCP/IP и сетевую плату. В Windows TCP/IP используется как протокол по умолчанию для работы с глобальными сетями (WAN).

Как правило, поддержка сети устанавливается во время установки Windows. Протокол TCP/IP также можно установить через окно свойств под-

ключения по локальной сети. Windows-компьютеры поддерживают IP-адресацию версий 4 (IPv4) и 6 (IPv6).

Получение сведений о сетевых платах

Подключение по локальной сети создается автоматически, если компьютер оснащен сетевой платой и подключен к сети. Если на компьютере установлено несколько сетевых адаптеров, подключенных к сети, для каждого из них создается подключение по локальной сети. Если подключение не было создано автоматически, следует подключить компьютер к сети, либо создать сетевое подключение другого типа.

Компьютеры используют IP-адреса для взаимодействия по протоколу TCP/IP. Windows поддерживает несколько вариантов IP-адресации:

- **ручная** — IP-адреса назначаются вручную (такие IP-адреса называются *статическими*). Статические IP-адреса фиксированы, их можно изменить только вручную. Как правило, статические IP-адреса назначают серверам. При этом для работы в сети серверы требуют ряда дополнительных настроек;
- **динамическая** (задана по умолчанию) — при запуске компьютера IP-адреса автоматически назначаются DHCP-сервером (если таковой есть в сети); динамические IP-адреса со временем могут изменяться;
- **альтернативная** (только для IPv4) — если компьютер настроен для использования DHCPv4, но в сети нет DHCPv4-сервера, Windows Server 2008 назначает т.н. автоматические частные IP-адреса из диапазона 169.254.0.1–169.254.255.254 с маской подсети 255.255.0.0. Кроме того, можно вручную настроить автоматический адрес IPv4, что особенно удобно владельцам лэптопов.

Для анализа конфигурации сетевых адаптеров используются объекты Win32_NetworkAdapter и Win32_NetworkAdapterConfiguration. Первый хранит базовые, а второй — подробные сведения о сетевых адаптерах. С учетом этого администраторы предпочитают использовать только Win32_NetworkAdapterConfiguration.

Однако на большинстве компьютеров настроено немалое число виртуальных сетевых карт. Следовательно, если вы не знаете, какая именно сетевая карта вам нужна, готовьтесь разбирать кучу посторонней информации. Лучше действовать так:

1. С помощью объекта Win32_NetworkAdapter получите номер нужно сетевого адаптера. На большинстве компьютеров основной сетевой плате присвоен идентификатор «Local Area Connection» («Подключение по локальной сети»).
2. С помощью объекта Win32_NetworkAdapterConfiguration и фильтра либо конструкции Where получите информацию о нужной сетевой плате. Так можно получить полные сведения о конфигурации TCP/IP, а также

MAC-адрес сетевой платы, необходимый для резервирования IP-адресов на DHCP-сервере.

Вот как использовать этот прием на практике:

```
$na = get-wmiobject Win32_NetworkAdapter -filter `
«NetConnectionID='Local Area Connection'»

$index = $na.index
get-wmiobject Win32_NetworkAdapterConfiguration -filter «Index=$index»
```

```
__PATH : \\ROOM5\root\cimv2:Win32_NetworkAdapterConfiguration.Index=4
DHCPLeaseExpires             : 20090129095440.000000-480
Index                        : 4
Description                  : Intel(R) PRO/1000 PM Network Connection
DHCPEnabled                  : True
DHCPLeaseObtained            : 20090128095440.000000-480
DHCPServer                   : 192.168.1.1
DNSDomain                    :
DNSDomainSuffixSearchOrder   :
DNSEnabledForWINSResolution  : False
DNSHostName                  : TechPC242
DNSServerSearchOrder         : {68.87.78.177, 68.77.75.78, 68.77.79.176}
DomainDNSRegistrationEnabled : False
FullDNSRegistrationEnabled   : True
IPAddress                    : {192.168.1.104, fe80::7330:2226:ee62:2312}
IPConnectionMetric           : 1
IPEnabled                    : True
IPFilterSecurityEnabled      : False
WINSEnableLMHostsLookup      : True
WINSHostLookupFile           :
WINSPrimaryServer            :
WINSScopeID                  :
WINSSecondaryServer          :
DatabasePath                 : %SystemRoot%\System32\drivers\etc
DeadGWDetectEnabled          :
DefaultIPGateway             : {192.168.1.1}
DefaultTOS                   :
DefaultTTL                   :
ForwardBufferMemory          :
GatewayCostMetric            : {0}
IGMPLevel                    :
InterfaceIndex               : 7
IPPortSecurityEnabled        :
IPSecPermitIPProtocols       : {}
IPSecPermitTCPPorts          : {}
IPSecPermitUDPPorts          : {}
```

```
IPSubnet                    : {255.255.255.0, 64}
MACAddress                  : 89:76:FF:D4:D6:35
MTU                         :
NumForwardPackets           :
TcpipNetbiosOptions         : 0
TcpMaxConnectRetransmissions :
TcpMaxDataRetransmissions   : 5
TcpNumConnections           :
TcpWindowSize               :
Scope                       : System.Management.ManagementScope
```

Из вывода этой команды можно получить следующую информацию:

- IPv4- и IPv6-адреса;
- маску подсети для IPv4;
- основной шлюз IP;
- MAC-адрес;
- домен DNS;
- порядок просмотра доменных суффиксов DNS;
- порядок просмотра DNS-серверов;
- состояние регистрации в DNS;
- состояние DHCP;
- конфигурация DHCP-аренды;
- параметры DHCP-сервера.

Чтобы проверить с помощью объекта Win32_NetworkAdapter состояние только активных сетевых плат, следует отбирать объекты, у которых свойство NetConnectionStatus равно 2. Вот пример:

```
$na = get-wmiobject Win32_NetworkAdapter -filter «NetConnectionStatus=2»

$na | foreach {
   $index = $_.index
   get-wmiobject Win32_NetworkAdapterConfiguration -filter «Index=$index»
}
```

```
DHCPEnabled      : True
IPAddress        : {192.168.10.152}
DefaultIPGateway : {192.168.10.1}
DNSDomain        :
ServiceName      : e1express
Description      : Intel(R) PRO/1000 PM Network Connection
Index            : 4
```

Полный список значений NetConnectionStatus выглядит так: 0 (отключен), 1 (подключается), 2 (подключен), 3 (отключается), 4 (нет сетевой платы), 5 (сетевая плата отключена), 6 (сбой сетевой платы), 7 (отключен носи-

тель), 8 (аутентификация), 9 (аутентификация прошла успешно) и 10 (сбой аутентификации).

Следующий код выводит состояние подключения по локальной сети:

```
$na = get-wmiobject Win32_NetworkAdapter -filter «NetConnectionID='Local
Area Connection'»
$status = $na.NetConnectionStatus
switch -regex ($status) {
 [0] { «Disconnected.» }
 [1] { «Connecting.» }
 [2] { «Connected. The connection is active.» }
 [3] { «Disconnecting.» }
 [4-6] { «Hardware is disabled, malfunctioning or not present. Check and
enable hardware.»}
 [7] { «Media is disconnected; connect the network cable.»}
 [8-9] { «Authenticating.»}
 [10] { «Authentication failed.»}
}
```

```
   Connected. The connection is active.
```

Настройка статического IP-адреса

Назначая статический IP-адрес, вы должны указать вместе с адресом маску подсети и, при необходимости, адрес основного шлюза для взаимодействия с другими сетями. IP-адрес — это числовой идентификатор компьютера. Схема IP-адресации зависит от конфигурации сети, но адрес компьютера, как правило, зависит от адреса подсети.

Адресация в IPv6 и IPv4 сильно различается. Первые 64 бита адреса IPv6 представляют идентификатор (ил адрес) сети, а остальные 64 бита — адрес сетевого интерфейса. Число битов адреса IPv4, отведенное для идентификатора сети, варьируется; от него зависит число битов, представляющих адрес хоста. Например, если компьютер, использующий IPv4, подключен к сегменту сети с адресом 192.168.1.0 и маской подсети 255.255.255.0, первые 24 бита представляют адрес сети, для компьютеров в этой сети доступен диапазон адресов 192.168.1.1–192.168.1.254 (адрес 192.168.1.255 зарезервирован для широковещательной передачи).

В частной сети, напрямую подключенной к Интернету, следует использовать частные адреса IPv4 (см. табл. 11-1).

Табл. 11-1. Частные адреса IPv4

Адрес сети	Маска подсети	Диапазон адресов хостов
10.0.0.0	255.0.0.0	10.0.0.0–10.255.255.255
172.16.0.0	255.240.0.0	172.16.0.0–172.31.255.255
192.168.0.0	255.255.0.0	192.168.0.0–192.168.255.255

Остальные адреса IPv4 являются публичными, их сдают в аренду (обычно платную) провайдеры. Если диапазон адресов IPv4 предоставил Вам провайдер Интернет-услуг, эти адреса можно использовать в сети, подключенной к Интернету напрямую.

В консоли PowerShell, запущенной с администраторскими полномочиями, можно вызвать методы объекта Win32_NetworkAdapterConfiguration для настройки параметров TCP/IP на компьютере, включая статические IP-адреса. Вот некоторые свойства этого объекта, хранящие полезную информацию:

- **DNSDomain** — хранит DNS-суффикс для подключения. Переопределяет ранее настроенный суффикс, обычно не используется;
- **DNSDomainSuffixSearchOrder** — хранит DNS-суффикс подключения и порядок просмотра DNS-суффиксов при разрешении имен компьютеров. Обычно первым в списке суффиксов идет родительский домен, то есть при разрешении имен к имени компьютера добавляется имя родительского домена;
- **DNSEnabledForWINSResolution** — указывает, разрешено ли использовать DNS при разрешении WINS-имен; по умолчанию установлено в False;
- **DNSServerSearchOrder** — хранит IP-адреса DNS-серверов (в порядке их просмотра);
- **DomainDNSRegistrationEnabled** — указывает, разрешена ли регистрация в DNS IP-адреса для данного подключения и связывание их с DNS-суффиксом, заданным для подключения; по умолчанию установлено в False;
- **FullDNSRegistrationEnabled** — указывает, разрешена ли регистрация в DNS IP-адреса для данного подключения и связывание их с полным доменным именем компьютера; по умолчанию установлено в True.

У этого объекта также есть полезные администратору методы:

- **SetDNSDomain()** — DNS-суффикс для подключения, переопределяет ранее настроенный суффикс. Принимает суффикс в виде строки;
- **SetDNSServerSearchOrder()** — задает IP-адреса DNS-серверов (в порядке их просмотра). Принимает IP-адреса в виде строки или массива строк;
- **SetDynamicDNSRegistration()** — включает автоматическую регистрацию IP-адреса для данного подключения в DNS с привязкой к полному доменному имени компьютера. Принимает булево значение $True или $False;
- **SetGateways()** — задает IP-адрес и метрику шлюза. Первый параметр метода — строка или массив строк с IP-адресами, а второй аргумент — целое число или массив целочисленных значений метрики;
- **SetWINSServer()** — задает IP-адреса WINS-серверов в порядке просмотра. Принимает строку или массив строк с IP-адресами WINS-серверов.

Чтобы изменить конфигурацию TCP/IP на компьютере с динамическим или статическим IP-адресом, вызовите метод EnableStatic(), который вклю-

чает статическую IP-адресацию и задает IP-адрес с маской подсети. Вот его общий синтаксис:

```
$Подключение.EnableStatic(IP_адрес,Маска)
```

где *$Подключение* — это ссылка на объект Win32_NetworkAdapterConfiguration, *IP_адрес* — новый IP-адрес (в виде строки), а *Маска* — новая маска подсети (также в виде строки). Вот пример:

```
$na = get-wmiobject Win32_NetworkAdapter -filter `
«NetConnectionID='Local Area Connection'»
$index = $na.index
$nac = get-wmiobject Win32_NetworkAdapterConfiguration -filter `
«Index=$index»

$nac.EnableStatic(«192.168.1.100»,»255.255.255.0»)
```

```
__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0
```

В этом примере устанавливается IP-адрес 192.168.1.100 и маска подсети 255.255.255.0. В выводе команды следует обратить внимание на возвращаемое значение (ReturnValue). Если оно равно 0, операция завершилась успешно, любое другое значение говорит об ошибке. Обычная причина ошибок — отсутствие администраторских полномочий у консоли и неверный выбор сетевой платы;

Если компьютер не получает адрес основного шлюза через DHCP, можно задать шлюз методом SetGateways(), принимающим IP-адрес и метрику шлюза. При назначении нескольких шлюзов Windows использует шлюз с наименьшей метрикой. Вот общий синтаксис этого метода:

```
$Подключение.SetGateways(Адрес,Метрика)
```

где *$Подключение* — это ссылка на объект Win32_NetworkAdapterConfiguration, *Адрес* — IP-адрес шлюза (в виде строки или массива строк), а *Метрика* — метрика шлюза (в виде целого числа или массива целых чисел). Следующая команда назначает основной шлюз с адресом 192.168.1.1 и метрикой 1:

```
$nac.SetGateways(«192.168.1.1»,1)
```

Для этого и других методов, описанных в данном разделе, возврат значения 0 говорит об успехе, а любые другие результаты — об ошибке. В следующем примере назначаются три шлюза с разной метрикой:

```
$g = «192.168.1.1», «192.168.2.1», «192.168.3.1»
$m = 1,2,3
$nac.setgateways($g,$m)
```

Если компьютер не получает IP-адреса DNS-серверов через DHCP, можно задать их методом SetDNSServerSearchOrder(). Вот его общий синтаксис:

```
$Подключение.SetDNSServerSearchOrder(Адреса)
```

где *$Подключение* — это ссылка на объект Win32_NetworkAdapterConfiguration, а *Адреса* — IP-адреса DNS-сервера в виде строки или массива строк. В следующем примере компьютеру назначаются IP-адреса основного и дополнительного DNS-серверов:

```
$dns = «10.10.10.52», «10.10.10.68»
$nac.SetDNSServerSearchOrder($dns)
```

Настройка динамической IP-адресации

IP-адреса могут назначаться компьютерам динамически с помощью DHCP-серверов. При назначении IP-адреса клиентскому компьютеру DHCP-сервером говорят, что клиент получил IP-адрес в *аренду* (lease), поскольку динамический IP-адрес предоставляется на ограниченный срок. DHCP-сервер определяет длительность аренды во время ее предоставления.

Объект Win32_NetworkAdapterConfiguration поддерживает методы и свойства для работы с динамической IP-адресацией. Полезные свойства этого объекта включают:

- **DHCPEnabled** — указывает, используется ли DHCP данной сетевой платой. Если это свойство равно True, то свойство DHCPLeaseObtained содержит сведения об аренде IP-адреса у DHCP-сервера;
- **DHCPServer** — хранит IP-адреса DHCP-серверов;
- **DHCPLeaseObtained** — хранит время получения DHCP-аренды как объект DateTime в строковой форме;
- **DHCPLeaseExpires** — хранит время истечения аренды IP-адреса как объект DateTime в строковой форме.

Этот объекта также поддерживает методы, полезные администратору (эти методы вызываются без параметров):

- **EnableDHCP()** — включает DHCP для заданной сетевой платы;
- **ReleaseDHCPLease()** — освобождает DHCP-аренду и связанные данные;
- **RenewDHCPLease()** — освобождает DHCP-аренду и обновляет ее.

Имея ссылку на объект Win32_NetworkAdapterConfiguration для сетевого подключения, использующего DHCP, можно воспользоваться свойствами этого объекта для просмотра сведений о работе DHCP, например, так:

```
$na = get-wmiobject Win32_NetworkAdapter -filter `
«NetConnectionID='Local Area Connection'»
$index = $na.index
$nac = get-wmiobject Win32_NetworkAdapterConfiguration -filter `
«Index=$index»

$nac | format-list DHCPLeaseExpires, DHCPEnabled, DHCPLeaseObtained, `
DHCPServer
```

```
DHCPLeaseExpires             : 20090220080155.000000-480
DHCPEnabled                  : True
DHCPLeaseObtained            : 20090219080155.000000-480
DHCPServer                   : 192.168.1.1
```

Чтобы перевести строки DateTime в более понятный вид, вызовите ConvertToDateTime() как показано ниже:

```
$na = get-wmiobject Win32_NetworkAdapter -filter `
«NetConnectionID='Local Area Connection'»
$index = $na.index
$nac = get-wmiobject Win32_NetworkAdapterConfiguration -filter `
«Index=$index»

$gotlease = $nac.ConvertToDateTime($nac.dhcpleaseobtained)
$explease = $nac.ConvertToDateTime($nac.dhcpleaseexpires)

write-host («DHCP Lease Obtained: $gotlease»)
write-host («DHCP Lease Expires: $explease»)
```

```
DHCP Lease Obtained: 02/19/2009 14:42:56
DHCP Lease Expires: 02/20/2009 14:42:56
```

мечание С сетевой платой, использующей статический IP-адрес, этот пример не сработает, поскольку DHCP-параметры в этом случае содержат Null и не мог ут быть переведены в объекты DateTime.

Для управления динамической IP-адресацией необходима консоль PowerShell с администраторскими полномочиями. Получив ссылку на объект Win32_NetworkAdapterConfiguration, можно вызвать его метод EnableDHCP(), чтобы включить DHCP:

```
$na = get-wmiobject Win32_NetworkAdapter -filter `
«NetConnectionID='Local Area Connection'»
$index = $na.index
$nac = get-wmiobject Win32_NetworkAdapterConfiguration -filter `
«Index=$index»
```

```
$nac.EnableDHCP()
```

```
__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0
```

В этом примере включается поддержка DHCP для подключения по локальной сети. Метод EnableDHCP() возвращает 0 в случае успеха и другие значения в случае неудачи. Типичные причины ошибок — использование консоли PowerShell без администраторских полномочий и выбор неверной сетевой платы.

Объект Win32_NetworkAdapterConfiguration позволяет обновить DHCP-аренду вызовом метода ReleaseDHCPLease() или RenewDHCPLease(). Вот как сделать это для подключения по локальной сети:

```
$na = get-wmiobject Win32_NetworkAdapter -filter `
«NetConnectionID='Local Area Connection'»
$index = $na.index
$nac = get-wmiobject Win32_NetworkAdapterConfiguration -filter `
«Index=$index»

$nac.RenewDHCPLease()
```

Настройка Брандмауэра Windows

Брандмауэр Windows (Windows Firewall) поддерживается Windows XP с Service Pack 2 и более высокими версиями. При настройке Windows-компьютеров нередко требуется определить, активен ли брандмауэр Windows, какие порты открыты или закрыты, а затем открыть либо закрыть отдельные порты. Эти задачи легко решаются средствами PowerShell.

Просмотр и настройка параметров брандмауэра Windows

Брандмауэр Windows поддерживает три типа профилей:

- **Domain** — для компьютеров, подключенных к домену;
- **Private** — для компьютеров в рабочей группе или домашней сети;
- **Public** — для компьютеров, подключенных к публичным сетям.

Для управления брандмауэром Windows служит COM-объект HNetCfg.FwMgr. получить ссылку на него можно так:

```
$firewall = new-object -com HNetCfg.FwMgr
```

Для просмотра параметров брандмауэра Windows через консоль PowerShell достаточно стандартных полномочий, но для изменения их потребуется консоль, открытая с администраторскими полномочиями. Следующий пример иллюстрирует работу с методами объекта брандмауэра:

```
$firewall = new-object -com HNetCfg.FwMgr
$firewall | get-member
```

```
   TypeName: System.__ComObject#{f7898af5-cac4-4632-a2ec-da06e5111af2}

Name               MemberType Definition
----               ---------- ----------
IsIcmpTypeAllowed  Method     void IsIcmpTypeAllowed (NET_FW_IP_VERSION
IsPortAllowed      Method     void IsPortAllowed (string, NET_FW_IP_
VERSION
RestoreDefaults    Method     void RestoreDefaults ()
CurrentProfileType Property   NET_FW_PROFILE_TYPE_ CurrentProfileType ()
LocalPolicy        Property   INetFwPolicy LocalPolicy () {get}
```

Вызов метода RestoreDefaults() объекта брандмауэра восстанавливает его параметры по умолчанию:

```
$firewall = new-object -com HNetCfg.FwMgr
$firewall.restoredefaults()
```

Свойство CurrentProfileType содержит код типа текущего профиля: 0 (Private), 1 (Public), or 2 (Domain), прочитать его можно так:

```
$firewall = new-object -com HNetCfg.FwMgr
$firewall.currentprofiletype
```

```
1
```

Чаще всего требуется свойство LocalPolicy, которое возвращает объект, представляющий локальную политику брандмауэра. Этот объект используется для представления как текущего, так и любых других профилей, заданных по имени.

Текущий профиль можно получить из свойства CurrentProfile объекта локальной политики брандмауэра :

```
$firewall = new-object -com HNetCfg.FwMgr
$current = $firewall.localpolicy.currentprofile
$current | format-list *
```

```
Type                                    : 1
FirewallEnabled                         : False
```

```
ExceptionsNotAllowed                          : False
NotificationsDisabled                         : False
UnicastResponsesToMulticastBroadcastDisabled : False
RemoteAdminSettings                           : System.__ComObject
IcmpSettings                                  : System.__ComObject
GloballyOpenPorts                             : {Intel(R) Viiv(TM) Media}
Services                                      : {File and Printer
Sharing}
AuthorizedApplications                       : {Roxio Upnp Service, SPCM}
```

Метод GetProfileByType() объекта локальной политики брандмауэра позволяет получить сведения о профиле:

```
$firewall = new-object -com HNetCfg.FwMgr
$private = $firewall.localpolicy.getprofilebytype(0)
$private | format-list *
```

```
Type                                          : 0
FirewallEnabled                               : True
ExceptionsNotAllowed                          : False
NotificationsDisabled                         : True
UnicastResponsesToMulticastBroadcastDisabled : True
RemoteAdminSettings                           : System.__ComObject
IcmpSettings                                  : System.__ComObject
GloballyOpenPorts                             : {Intel(R) Viiv(TM) Media}
Services                                      : {File and Printer
Sharing}
AuthorizedApplications                       : {Roxio Upnp Service, SPCM}
```

Независимо от типа профиля, с которым вы работаете, вам доступны следующие свойства:

- **AuthorizedApplications** — возвращает набор объектов, представляющих приложения, которым разрешено передавать данные через брандмауэр (авторизованными приложениями). Эти объекты служат для управления авторизованными приложениями. Например, так можно вывести список всех объектов, представляющих авторизованные приложения:

```
$firewall = new-object -com HNetCfg.FwMgr
$apps = $firewall.localpolicy.currentprofile.authorizedapplications
foreach ($a in $apps) {$a}
```

```
Name                : Roxio Upnp Service
ProcessImageFileName : C:\Program Files\Roxio\Easy Media Creator 8\
Digital Home\RoxUpnpServer.exe
IpVersion           : 2
Scope               : 0
RemoteAddresses     : *
Enabled             : True
```

- **ExceptionsNotAllowed** — разрешает или запрещает исключения брандмауэра и выводит их состояние. Это свойство принимает значение типа Boolean.

```
$firewall = new-object -com HNetCfg.FwMgr
$cp = $firewall.localpolicy.currentprofile
$cp.exceptionsnotallowed = $True
write-host $cp.exceptionsnotallowed
```

```
True
```

- **FirewallEnabled** — включает либо выключает брандмауэр и выводит его состояние. Это свойство принимает значение типа Boolean.

```
$firewall = new-object -com HNetCfg.FwMgr
$cp = $firewall.localpolicy.currentprofile
$cp.firewallenabled = $True
write-host $cp.firewallenabled
```

```
True
```

- **GloballyOpenPorts** — возвращает набор объектов, представляющих открытые порты и позволяющих управлять портами. Так, например, можно вывести свойства этих объектов:

```
$firewall = new-object -com HNetCfg.FwMgr
$oports = $firewall.localpolicy.currentprofile.globallyopenports
foreach ($o in $oports) {$o}
```

```
Name            : Adobe Version Cue CS3 Server
IpVersion       : 2
Protocol        : 6
Port            : 50901
Scope           : 0
RemoteAddresses : *
Enabled         : True
BuiltIn         : False
```

- **IcmpSettings** — возвращает набор объектов, представляющих параметры протокола ICMP и позволяющих управлять ими (см. пример ниже).

```
$firewall = new-object -com HNetCfg.FwMgr
$icmpsettings = $firewall.localpolicy.currentprofile.icmpsettings
foreach ($i in $icmpsettings) {$i}
```

```
AllowOutboundDestinationUnreachable : False
AllowRedirect                       : False
AllowInboundEchoRequest             : False
AllowOutboundTimeExceeded           : False
AllowOutboundParameterProblem       : False
AllowOutboundSourceQuench           : False
AllowInboundRouterRequest           : False
```

```
AllowInboundTimestampRequest        : False
AllowInboundMaskRequest             : False
AllowOutboundPacketTooBig           : True
```

- **NotificationsDisabled** — управляет выводом оповещений и отображает текущие настройки их вывода. Если вывод оповещений включен, пользователь уведомляется о блокировании входящих соединений для какой-либо программы. Это свойство принимает значение типа Boolean.

```
$firewall = new-object -com HNetCfg.FwMgr
$cp = $firewall.localpolicy.currentprofile
$cp.notificationsdisabled = $True
write-host $cp.notificationsdisabled
```

```
True
```

- **RemoteAdminSettings** — возвращает набор объектов, представляющих параметры удаленного администрирования и позволяющих управлять этими параметрами.

```
$firewall = new-object -com HNetCfg.FwMgr
$ras = $firewall.localpolicy.currentprofile.remoteadminsettings
foreach ($r in $ras) {$r | format-list *}
```

```
IpVersion       : 2
Scope           : 1
RemoteAddresses : LocalSubnet
Enabled         : False
```

- **UnicastResponsesToMulticastBroadcastDisabled** — разрешает либо запрещает одноадресные отклики и выводит состояние этого параметра. Если одноадресные отклики разрешены, брандмауэр Windows передает одноадресные отклики на многоадресные и широковещательные сообщения. Это свойство принимает значение типа Boolean.

```
$firewall = new-object -com HNetCfg.FwMgr
$cp = $firewall.localpolicy.currentprofile
$cp.unicastresponsestomulticastbroadcastdisabled = $True
write-host $cp.unicastresponsestomulticastbroadcastdisabled
```

```
True
```

- **Type** — выводит тип профиля, с которым вы работаете: 0 (Private), 1 (Public), or 2 (Domain).

```
$firewall = new-object -com HNetCfg.FwMgr
$cp = $firewall.localpolicy.currentprofile
write-host $cp.type
```

```
1
```

Управление портами брандмауэра

Консоль PowerShell, открытая с администраторскими полномочиями, позволяет открывать и закрывать порты брандмауэра с помощью методов Add() и Remove() COM-объекта HNetCfg.FWOpenPort. Чтобы открыть порт, следует получить ссылку на этот объект и задать параметры порта следующим образом:

```
$PROTOCOL_TCP = 6
$firewall = new-object -com HNetCfg.FwMgr
$port = new-object –com HNetCfg.FWOpenPort

$port.name = «Web Services»
$port.port = 8080
$port.protocol = $PROTOCOL_TCP

$firewall.localpolicy.currentprofile.globallyopenports.Add($port)
```

Свойство name содержит понятное имя порта, по которому можно догадаться о его назначении. Номер порта (свойство port) представляет открытый TCP- или UDP-порт брандмауэра. Свойство protocol определяет используемый протокол (TCP или UDP).

Проверить настройки созданного порта позволяет свойство GloballyOpenPorts соответствующего профиля следующим образом:

```
$firewall = new-object -com HNetCfg.FwMgr
$oports = $firewall.localpolicy.currentprofile.globallyopenports
$oports | where-object {$_.name -eq «Web Services»}
```

```
Name            : Web Services
IpVersion       : 2
Protocol        : 6
Port            : 8080
Scope           : 0
RemoteAddresses : *
Enabled         : True
BuiltIn         : False
```

Чтобы закрыть порт, удалите его объект. Для этого необходимо получить ссылку на COM-объект HNetCfg.FWOpenPort и указать подлежащий удалению порт, а затем передать адрес порта и протокол, как показано ниже:

```
$PROTOCOL_TCP = 6
$firewall = new-object -com HNetCfg.FwMgr
$cp = $firewall.localpolicy.currentprofile
$cp.globallyopenports.Remove(8080, $PROTOCOL_TCP)
```

Далее можно проверить, закрыт ли порт. Эта проверка проводится так же, как проверка открытого порта: если найти порт по имени его объекта не удается, порт был успешно закрыт.

Глава 12

Управление реестром и его защита

В системном реестре Windows хранятся различные параметры конфигурации. Для работы с реестром используется встроенный поставщик PowerShell под названием Registry. Этот поставщик позволяет просматривать, создавать, удалять, сравнивать и копировать разделы реестра. Системный реестр жизненно важен для нормальной работы Windows, поэтому изменять его можно, лишь полностью представляя последствия вносимых изменений. Любые манипуляции над реестром следует выполнять как транзакции (о транзакциях см. в главе 7). Для этого:

1. Запустите транзакцию с помощью Start-Transaction.
2. Внесите в реестр необходимые изменения и проверьте их.
3. Подтвердите транзакцию, вызвав Stop-Transaction, либо отмените ее с помощью Undo-Transaction.

После подтверждения транзакции командлетом Stop-Transaction отмена внесенных изменений становится невозможной. Кроме того, использование транзакций не позволит выявить изменения, способные привести к сбоям в работе компьютера и установленных на нем программа. Поэтому перед внесением в реестр любых изменений необходимо создать точку восстановления, чтобы застраховаться от возможных ошибок.

Внимание! Ошибки при изменении реестра Windows чреваты серьезными неполадками. Не исключено, что при повреждении реестра вам придется переустановить систему. Поэтому перед запуском тщательно проверьте каждую команду и убедитесь, что она делает именно то, что надо.

Введение в разделы и параметры реестра

В реестре Windows хранятся сведения о конфигурации операционной системы, приложений, пользователей и оборудования. Эта информация хранится в виде разделов (registry keys) и параметров (registry values), содержащихся в корневых разделах (root keys); корневые разделы контролируют использование расположенных в них разделов и параметров.

В табл. 12-1 приводится список корневых разделов реестра, с которыми можно работать в PowerShell, с кратким описанием и указанием имен, по

которым на корневые разделы можно ссылаться при работе с поставщиком Registry. В корневых разделах содержатся вложенные разделы, управляющие системой, пользователями, приложениями и оборудованием. Разделы реестра образуют иерархическую структуру, напоминающую дерево каталогов файловой системы, в котором разделы занимают место папок.

Табл. 12-1. Корневые разделы системного реестра Windows

Корневой раздел	Имя для ссылки	Какую информацию хранит
HKEY_CURRENT_USER	HKCU	Параметры текущего пользователя
HKEY_LOCAL_MACHINE	HKLM	Общесистемные параметры конфигурации
HKEY_CLASSES_ROOT	HKCR	Параметры конфигурации файлов и приложений. Обеспечивает вызов приложений, соответствующих типу открываемого файла
HKEY_USERS	HKU	Параметры профиля пользователя по умолчанию и других профилей
HKEY_CURRENT_CONFIG	HKCC	Используемый профиль оборудования

Нужный раздел реестра задают как путь, подобный пути к каталогу файловой системы. Например, в разделе HKLM\SYSTEM\CurrentControlSet\Services содержатся сведения о службах, а параметры в разделе DNS управляют службой DNS. Для хранения разделов и параметров используются особые типы данных (табл. 12-2).

Табл. 12-2. Типы данных системного реестра

Тип данных	Описание	Имя для ссылки	Пример
REG_BINARY	Двоичные значения, которые хранятся в двоичном (в виде 0 и 1), но отображаются в шестнадцатеричном формате	Binary	01 00 14 80 90 00 00 9C 00
REG_DWORD	Двоичные значения, представляющие четырехбайтовые (32-разрядные) целые числа, отображаемые в шестнадцатеричном формате	Dword	0x00000002
REG_EXPAND_SZ	Строки с подставляемыми значениями, обычно используются для хранения путей	Expandstring	%SystemRoot%\dns.exe
REG_MULTI_SZ	Многострочное значение	Multistring	Tcpip Afd RpcSc

Табл. 12-2. Типы данных системного реестра

Тип данных	Описание	Имя для ссылки	Пример
REG_NONE	Данные неопределенного типа, которые хранятся в двоичном и отображаются в шестнадцатеричном формате	None	23 45 67 80
REG_ QWORD	Двоичные значения, представляющие восьмибайтовые (64-разрядные) целые числа, отображаемые в шестнадцатеричном формате	Qword	0x0000EA3FC
REG_SZ	Строковые значения	String	DNS Server

Просмотр реестра

Для работы с реестром применяются, главным образом, поставщик Registry (подробнее о нем — в главе 3) и командлеты, управляющие хранилищами данных (табл. 3-4 и 3-5). Определив путь к нужному разделу и типы содержащихся в нем данных, можно выполнять над этим разделом самые разные манипуляции с помощью поставщика Registry.

По умолчанию, из корневых разделов в PowerShell доступны только HKLM и HKCU. Чтобы открыть остальные корневые разделы, следует зарегистрировать их как диски PowerShell. Например, следующие команды регистрируют разделы HKCR, HKU и HKCC:

```
new-psdrive -name hkcr -psprovider registry -root hkey_classes_root
new-psdrive -name hku -psprovider registry -root hkey_users
new-psdrive -name hkcc -psprovider registry -root hkey_current_config
```

После их исполнения можно обращаться к зарегистрированным корневым разделам реестра напрямую. Так, например, можно открыть раздел HKCC:

```
set-location hkcc:
```

Командлет Set-Location позволяет обращаться к любым разделам реестра через поставщик Registry, например:

```
set-location hklm:
```

Выполнив эту команду, можно работать с разделами и параметрами, находящимися в HKLM. Содержимое HKLM (и других корневых разделов) просматривают так же, как содержимое дисков файловой системы — с помощью путей. Так, если уже открыли корневой раздел HKLM, то можете с помощью командлета Set-Location (или его псевдонима CD) перейти в один из его подразделов:

```
set-location system\currentcontrolset\services
```

Если же HKLM еще не открыт, включите его в путь к нужному разделу:

```
set-location hklm:\system\currentcontrolset\services
```

Примечание Если заданный путь, раздел или параметр не существует, выводится сообщение об ошибке примерно такого вида: «Cannot find ____ because it does not exist».

Чтобы получить список параметров в текущем разделе, введите **get-childitem** (или **dir**):

```
set-location hklm:\system\currentcontrolset\services
get-childitem
```

```
    Hive: HKEY_LOCAL_MACHINE\system\currentcontrolset\services

SKC  VC Name                           Property
---  -- ----                           --------
  2   0 .NET CLR Data                  {}
  2   0 .NET CLR Networking            {}
  2   0 .NET Data Provider for Oracle  {}
  2   0 .NET Data Provider for SqlS... {}
  1   0 .NETFramework                  {}
  2   7 ACPI                           {Tag, DisplayName, Group...}
  1   7 Adobe LM Service               {Description, DisplayName...}
  1   6 Adobe Version Cue CS2          {DisplayName, Type, Start...}
  1   7 Adobe Version Cue CS3          {Type, Start, ErrorControl...}
```

Вывод этой команды содержит следующие поля:

- SKC — число подразделов в данном разделе;
- VC — число параметров в данном разделе;
- Name — имя подраздела;
- Property — список свойств соответствующего раздела.

Давайте более подробно рассмотрим команды для просмотра реестра на примере раздела APCI. Чтобы получить список свойств из этого раздела, введите **get-itemproperty**:

```
set-location hklm:\system\currentcontrolset\services\acpi
get-itemproperty .
```

```
PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\
system\currentcontrolset\services\acpi
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\
system\currentcontrolset\services
PSChildName  : acpi
PSDrive      : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry
```

```
Tag          : 1
DisplayName  : Microsoft ACPI Driver
Group        : Boot Bus Extender
ImagePath    : system32\drivers\acpi.sys
ErrorControl : 3
Start        : 0
Type         : 1
```

Свойства можно просматривать, не открывая соответствующий раздел. Для этого нужно указать путь при вызове Get-ItemProperty. В предыдущем примере точкой (.) обозначен текущий раздел. Если вам требуется просмотреть свойства другого подраздела, в том числе из другого корневого раздела, укажите полный путь к нему:

```
get-itemproperty hklm:\system\currentcontrolset\services\acpi
```

Чтобы вывести подразделы раздела ACPI, введите **get-childitem** (или **dir**):

```
set-location hklm:\system\currentcontrolset\services\acpi
get-childitem
```

```
    Hive: HKEY_LOCAL_MACHINE\system\currentcontrolset\services\acpi

SKC  VC Name                           Property
---  -- ----                           --------
  1   2 Parameters                     {AMLIMaxCTObjs,
WHEAOSCImplemented}
  0   3 Enum                           {0, Count, NextInstance}
```

Как и Get-ItemProperty, командлет Get-ChildItem принимает путь в качестве параметра. То есть, показанный выше результат можно получить и от такой команды:

```
get-childitem hklm:\system\currentcontrolset\services\acpi
```

В подразделе enum, вложенном в раздел ACPI, содержится ряд свойств, включая Count и NextInstance. Полный список свойств можно получить с помощью Get-ChildItem, но чаще приходится работать с отдельными свойствами. Для этого следует сначала получить все свойства и сохранить их в переменной для дальнейшей работы:

```
$p = get-itemproperty hklm:\system\currentcontrolset\services\acpi\enum
$p.count
```

```
1
```

Альтернативный вариант — прочитать нужный параметр, указав его имя и полный путь к нему. Вот общий синтаксис соответствующей команды:

```
get-itemproperty [-path] Путь [-name] Имя
```

где *Путь* — путь к нужному разделу, а *Имя* — необязательный параметр, который задает имя нужного параметра (свойства). Вот пример:

```
get-itemproperty hklm:\system\currentcontrolset\services\acpi\enum count
```

```
PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\
system\currentcontrolset\services\acpi\enum
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\
system\currentcontrolset\services\acpi
PSChildName  : enum
PSDrive      : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry
Count        : 1
```

Как видите, вывод этой команды включает путь, имя корневого раздела («диска») и поставщика, а также искомое значение. С помощью Format-List можно отфильтровать вывод этой команды, оставив в нем лишь значение свойства:

```
get-itemproperty hklm:\system\currentcontrolset\services\acpi\enum `
count | format-list -property count
```

```
Count        : 1
```

Примечание Для работы с реестром на удаленных компьютерах используется командлет Invoke-Command (см. главу 4), как показано ниже:

```
invoke-command -computername Server43, Server27, Server82 `
-scriptblock { get-itemproperty `
hklm:\system\currentcontrolset\services\acpi\enum }
```

Другой способ предполагает запуск удаленного сеанса с помощью командлета New-PSSession с автоматическим вызовом необходимых команд, как в следующем примере:

```
$s = new-PSSession -computername Server43, Server27, Server82
invoke-command -session $s -scriptblock {get-itemproperty `
hklm:\system\currentcontrolset\services\acpi\enum}
```

Управление разделами и параметрами реестра

Поставщик Registry поддерживает множество командлетов для управления разделами и параметрами реестра. Эти командлеты позволяют создавать, копировать, перемещать, переименовывать и удалять элементы реестра.

Создание разделов и параметров

Используя PowerShell, можно без труда создавать подразделы и параметры в реестре Windows. Для создания разделов используется командлет New-

Item, а для создания параметров — New-ItemProperty. Вот общий синтаксис создания разделов:

```
new-item [-type registrykey] [-path] Путь
```

Registrykey — необязательный параметр, который задает тип создаваемого элемента, а *Путь* — путь, по которому должен быть создан раздел. После создания раздела New-Item выводит подтверждение. В следующем примере создается раздел HKCU:\Software\Test и выводится подтверждение успешного завершения операции:

```
new-item -type registrykey -path hkcu:\software\test
```

```
    Hive: HKEY_CURRENT_USER\software
SKC  VC Name                           Property
---  -- ----                           --------
  0   0 test                           {}
```

Примечание Для создания разделов и параметров в реес тре удаленных компьютеров используйте командлет Invoke-Command (см. главу 4) следующим образом:

```
invoke-command -computername Server43, Server27, Server82 `
-scriptblock { new-item hkcu:\software\test }
```

При наличии разрешений на создание подразделов в заданном разделе реестра операция должна завершиться успешно. Если заданный подраздел уже существует, выводится сообщение об ошибке следующего вида: «A key at this path already exists».

Общий синтаксис создания параметров выглядит так:

```
new-itemproperty [-path] Путь [-name] Имя [-type Тип] [-value Значение]
```

где *Путь* — путь к существующему разделу реестра, *Имя* — имя параметра, *Тип* — тип данных параметра, а *Значение* — значение нового параметра. Допустимые типы данных параметров — Binary, Dword, Expandstring, Multistring, None и String (табл. 12-2).

После создание параметра New-ItemProperty выводит подтверждение. В следующем примере создается строковый параметр Data в разделе HKCU:\Software\Test и выводится подтверждение успешного завершения операции:

```
new-itemproperty -path hkcu:\software\test Data -type «string» `
-value «Current»
```

```
PSPath       : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\
software\test
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\
software
PSChildName  : test
PSDrive      : HKCU
PSProvider   : Microsoft.PowerShell.Core\Registry
Data         : Current
```

При наличии разрешения на создание параметров в заданном разделе эта операция завершается успешно, но если заданный параметр уже существует, выводится сообщение об ошибке «The property already exists».

Копирование разделов и параметров реестра

Для копирования разделов реестра служит командлет Copy-Item. Вот его общий синтаксис:

```
copy-item Исходный_путь Целевой_путь
```

где *Исходный_путь* — путь к копируемому разделу реестра, а *Целевой_путь* — путь к новой копии раздела. В следующем примере раздел HKCU:\Software\Test key (со всем содержимым) копируется в HKLM:\Software\Dev:

```
copy-item hkcu:\software\test hklm:\software\dev
```

При наличии необходимых разрешений копирование должно завершиться успехом. Учтите, что Copy-ItemProperty выводит сообщения о возникающих при копировании ошибках, но никак не подтверждает успешно завершенное копирование.

Для копирования параметров служит командлет Copy-ItemProperty с общим синтаксисом следующего вида:

```
copy-itemproperty [-path] Исходный_путь [-destination] Целевой_путь [-name]
Параметр
```

где *Исходный_путь* — путь к копируемому параметру, *Целевой_путь* — раздел, в который следует его скопировать, а *Параметр* — имя копируемого параметра. В следующем примере копируется параметр Data из раздела HKCU:\Software\Test в HKLM:\Software\Dev:

```
copy-itemproperty -path hkcu:\software\test -destination `
hklm:\software\dev -name data
```

Если копируемый параметр существует, то при наличии необходимых разрешений копирование должно завершиться успехом. Командлет Copy-ItemProperty позволяет копировать и группы параметров, которые можно задавать в виде списков, разделенных запятыми, либо с помощью подстановочных знаков.

Перемещение разделов и параметров реестра

Для перемещения разделов и параметров служит командлет Move-Item с общим синтаксисом следующего вида:

```
move-item Исходный_путь Целевой_путь
```

где *Исходный_путь* — путь к перемещаемому разделу, а *Целевой_путь* — путь, по которому этот раздел нужно переместить. При перемещении раздела командлет Move-Item выводит сообщения об ошибках, но никак не

подтверждает успешное завершение операции. В следующем примере раздел HKCU:\Software\Test key со всем содержимым перемещается в раздел HKCU:\Software\Test2:

```
move-item hkcu:\software\test hkcu:\software\test2
```

При наличии необходимых разрешений перемещение должно выполняться успешно. Move-Item также позволяет перемещать подразделы одного корневого раздела в другой, например HKCU:\Software\Test в HKLM:\Software\Test.

Для перемещения параметров используется командлет Move-ItemProperty. Вот его общий синтаксис:

```
move-itemproperty [-path] Исходный_путь [-destination] Целевой_путь [-name] Имя
```

где *Исходный_путь* — текущий путь к перемещаемому параметру, *Целевой_путь* — путь после перемещения, а *Имя* — имя перемещаемого параметра. Move-ItemProperty выводит сообщения об ошибках, но никак не подтверждает успешное перемещение. В следующем примере параметр Data перемещается из раздела HKCU:\Software\Test в раздел HKLM:\Software\Test:

```
move-itemproperty -path hkcu:\software\test -destination `
hklm:\software\test2 -name data
```

Если у вас есть необходимые разрешения, а исходный и целевой разделы существуют, операция должна завершиться успешно. Командлет Move-ItemProperty также позволяет перемещать параметры группами, которые задают как списки, разделенные запятыми.

Переименование разделов и параметров реестра

Для переименования разделов используется командлет Rename-Item, общий синтаксис которого имеет следующий вид:

```
rename-item Исходное_имя Новое_имя
```

где *Исходное имя* — полный путь и прежнее имя раздела, а *Новое_имя* — его новое имя. Например, следующая команда переименовывает подраздел Test из раздела HKCU:\Software в Test2:

```
rename-item hkcu:\software\test test2
```

Для переименования параметров служит командлет Rename-ItemProperty с синтаксисом следующего вида:

```
rename-item [-path] Исходный_путь [-name] Имя [-newname] Новое_имя
```

где *Исходный_путь* — полный путь к параметру, который требуется переименовать, а *Новое_имя* — новое имя этого параметра. В следующем примере параметр Data из раздела HKCU:\Software\Test переименовывается в EntryType:

```
rename-itemproperty -path hkcu:\software\test2 -name data `
-newname entrytype
```

Удаление разделов и параметров реестра

Для удаления разделов реестра применяют командлет Remove-Item. Вот его общий синтаксис:

```
remove-item Имя [-Force]
```

где *Имя* — полный путь к удаляемому разделу, а –Force необязательный параметр для принудительного удаления разделов. В следующем примере удаляется раздел HKCU:\Software\Test2 со всем содержимым:

```
remove-item hkcu:\software\test2
```

Обладая необходимыми разрешениями, вы без труда удалите разделы реестра.

Для удаления параметров используется командлет Remove-ItemProperty со следующим синтаксисом:

```
remove-itemproperty [-Path] Путь [-Name] Имя [-Force]
```

где *Путь* — полный путь к разделу, в котором находится удаляемый параметр, *Имя* — имя удаляемого раздела, а –Force — необязательный аргумент для принудительного удаления параметров реестра. Так, в следующем примере из раздела HKCU:\Software\Test удаляется параметр Data:

```
remove-itemproperty -path hkcu:\software\test -name data
```

Сравнение элементов реестра

Возможность сравнения содержимого реестра на разных компьютерах либо разных разделов реестра одного и того же компьютера удобна:

- **при устранении неполадок служб и сбоев в конфигурации приложений.** В этом случае сравнение конфигурации реестра на правильно настроенном компьютерах и компьютере, на котором возникают сбои, позволяет выявить причину сбоев;
- **когда требуется одинаково настроить приложение и службу на нескольких компьютерах**, например, чтобы проверить одну и ту же конфигурацию на разных компьютерах.

Рассмотрим сравнение элементов реестра на разных компьютерах на следующем примере:

```
$c1 = «techpc18»
$c2 = «engpc25»

$p = invoke-command -computername $c1 -scriptblock { get-itemproperty `
hklm:\system\currentcontrolset\services\acpi\enum }
```

```
$h = invoke-command -computername $c2 -scriptblock { get-itemproperty `
hklm:\system\currentcontrolset\services\acpi\enum }

if ($p = $h) {write-host $True} else {
write-host «Computer: $c1»
write-host $p

write-host «Computer: $c2»
write-host $h}
```

```
True
```

Чтобы выполнить эти команды, потребуется консоль PowerShell, открытая с администраторскими полномочиями, и два компьютера, настроенные для удаленной работы с PowerShell. Если содержимое заданных разделов одинаково на обоих компьютерах, PowerShell выводит True, а в противном случае — значения различающихся параметров.

Этот пример легко расширить и получить код для сравнения конфигурации нескольких компьютеров с «эталонным» компьютером с заведомо верными настройками. Вот пример:

```
$clist = «techpc18», «techpc25», «techpc36»
$src = «engpc25»

$ps = invoke-command -computername $clist -scriptblock { `
get-itemproperty hklm:\system\currentcontrolset\services\acpi\enum }

$h = invoke-command -computername $src -scriptblock { get-itemproperty `
hklm:\system\currentcontrolset\services\acpi\enum }

$index = 0

foreach ($p in $ps) {

if ($p = $h) {write-host $clist[$index] «same as $src» } else {
write-host «Computer:» $clist[$index]
write-host $p

write-host «Computer: $src»
write-host $h

$index++}
}
```

```
techpc18 same as engpc25
techpc25 same as engpc25
techpc36 same as engpc25
```

Здесь конфигурация реестра нескольких компьютеров сравнивается с конфигурацией «эталонного» компьютера. При обнаружении различий выводится список отличающихся параметров с указанием компьютера, на котором обнаружено различие.

Просмотр и настройка параметров защиты реестра

Администраторам нередко требуется просматривать и настраивать параметры защиты реестра. Для этой цели в PowerShell применяются командлеты Get-Acl и Set-Acl:

- **Get-Acl** — получает объекты, представляющие дескрипторы защиты разделов реестра. Параметр –Audit to дополнительно получает данные аудита дескриптора защиты из ACL.

```
Get-Acl [-Path] Путь_к_разделу {AddtlParams}

AddtlParams=
[-Audit] [-Exclude Разделы] [-Include Разделы]
```

- **Set-Acl** — устанавливает для разделов реестра новый дескриптор защиты, заданный параметром –Aclobject.

```
Set-Acl [-Path] Путь_к_разделу [-Aclobject] Объект_ACL {AddtlParams}

AddtlParams=
[-Exclude Разделы] [-Include Разделы]
```

При работе с разделами реестра часто приходится просматривать и изменять их дескрипторы защиты. Используйте для просмотра командлет Get-Acl, указав нужный ресурс с помощью параметра –Path (разрешены подстановочные знаки); параметры –Include и –Exclude, соответственно, включают либо исключают разделы из поиска.

Работа с дескрипторами защиты элементов реестра

Get-Acl возвращает отдельный объект с информацией системы безопасности для каждого из заданных разделов. По умолчанию Get-Acl выводит путь, владельца и список записей ACL для заданного ресурса. Списком ACL управляет владелец ресурса. Для получения дополнительной информации, включая имя группы владельца, список записей аудита и полный дескриптор защиты (в виде SDDL-строки) следует отформатировать вывод команды, как показано ниже:

```
get-acl hklm:\software\test | format-list
```

```
Path:Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\software\test
Owner  : BUILTIN\Administrators
```

```
Group  : ENGPC18\None
Access : BUILTIN\Users Allow  ReadKey
         BUILTIN\Users Allow  -2147483648
         BUILTIN\Administrators Allow  FullControl
         BUILTIN\Administrators Allow  268435456
         NT AUTHORITY\SYSTEM Allow  FullControl
         NT AUTHORITY\SYSTEM Allow  268435456
         CREATOR OWNER Allow  268435456
Audit  :
Sddl   : O:BAG:S-1-5-21-3603280705-3559929044-3306537903-
513D:AI(A;ID;KR;;;BU)(A;CIIOID;GR;;;BU)(A;ID;KA;;;BA)(A;CIIOID;GA;;;BA)
(A;ID;KA;;;SY)(A;CIIOID;GA;;;SY)(A;CIIOID;GA;;;CO)
```

Примечание Для просмотра и настройки параметров защиты реестра у удаленных компьютеров используют методы, описанные в главе 4, например:

```
invoke-command -computername Server16, Server12, Server18 `
-scriptblock { get-acl hklm:\software\test | format-list }
```

В этом примере Get-Acl возвращает объект RegistrySecurity, представляющий дескриптор защиты раздела HKLM:\Software\Test, который передается командлету Format-List. Вам доступны следующие свойства объекта RegistrySecurity, которые хранят:

- **Owner** — имя владельца ресурса;
- **Group** — имя основной группы владельца;
- **Access** — правила управления доступом к ресурсу;
- **Audit** — правила аудита для ресурса;
- **Sddl** — полный дескриптор защиты в виде SDDL-строки.

Примечание У объектов RegistrySecurity имеются дополнительные свойства, не отображаемые в стандартном выводе. Для их просмотра передайте объект командлету Format-List *. Так удается вывести свойства PSPath (путь PowerShell к ресурсу), PSParentPath (путь PowerShell к родительскому ресурсу), PSChildName (имя ресурса), PSDrive (PowerShell-диск, на котором находится ресурс), AccessToString (альтернативное представление правил доступа к ресурсу) и AuditToString (альтернативное представление правил аудита ресурса).

Объекты, возвращаемые Get-Acl, можно использовать для установки дескрипторов защиты разделов реестра. Для этого откройте консоль PowerShell с администраторскими правами, получить объект дескриптора защиты раздела реестра с нужными параметрами и назначьте его соответствующему разделу реестра. Вот пример:

```
set-acl -path hkcu:\software\dev -aclobject (get-acl hklm:\software\test)
```

Здесь дескриптор защиты раздела HKLM:\Software\Test назначается разделу HKCU:\Software\Dev.

Этот метод легко расширить. Так, ниже дескриптор защиты HKLM:\Software\Test назначается всем подразделам раздела HKCU:\Software\Dev:

```
$secd = get-acl hklm:\software\test
set-acl -path hkcu:\software\dev\* -aclobject $secd
```

Для получения доступа к подразделам используется командлет Get-ChildItem:

```
$s = get-acl hklm:\software\test
gci hkcu:\software\dev -recurse -force | set-acl -aclobject $s
```

gci в этой команде — псевдоним Get-ChildItem. Первая команда получает дескриптор защиты HKLM:\Software\Test. Вторая команда получает ссылки на объекты, представляющие подразделы HKCU:\Software\Dev и назначает им дескриптор защиты, полученный от раздела HKLM:\Software.

Работа с правилами доступа к реестру

Для создания собственных дескрипторов защиты необходимо настраивать правила доступа к реестру. Эти правила хранятся как значение свойства Access объектов системы безопасности. Правила доступа к разделам реестра представлены объектами следующего типа:

```
System.Security.AccessControl.RegistryAccessRule
```

Вот один из способов просмотра объектов правил доступа к реестру:

```
$s = get-acl hklm:\software\test
$s.access
```

```
  RegistryRights    : FullControl
  AccessControlType : Allow
  IdentityReference : ENGPC72\Bubba
  IsInherited       : True
  InheritanceFlags  : ContainerInherit, ObjectInherit
  PropagationFlags  : None

  RegistryRights    : FullControl
  AccessControlType : Allow
  IdentityReference : NT AUTHORITY\SYSTEM
  IsInherited       : True
  InheritanceFlags  : ContainerInherit, ObjectInherit
  PropagationFlags  : None
  RegistryRights    : FullControl

  RegistryRights    : FullControl
  AccessControlType : Allow
  IdentityReference : BUILTIN\Administrators
  IsInherited       : True
  InheritanceFlags  : ContainerInherit, ObjectInherit
  PropagationFlags  : None
```

Здесь мы получаем объект RegistrySecurity для раздела HKLM:\Software\Test и выводим содержимое его свойства Access. Этот прием отображает отдельные правила доступа, но не позволяет работать с ними. Среди отображаемых свойств объектов правил обратите внимание на следующие:

- **RegistryRights** — указывает действующее право на доступ к реестру;
- **AccessControlType** — указывает тип управления доступом (Allow или Deny);
- **IdentityReference** — имя группы или пользователя, для которых действует правило;
- **IsInherited** — указывает, наследуется ли правило;
- **InheritanceFlags** — указывает способ наследования правил;
- **PropagationFlags** — указывает дальнейшее наследование правил доступа.

Другой прием позволяет работать с объектами правил доступа по отдельности. В этом примере для обработки объектов правил используется цикл ForEach:

```
$s = get-acl hklm:\software\test

foreach($a in $s.access) {

#обрабатываем объекты правил управления доступом

 if ($a.identityreference -like «*administrator*») {$a | format-list *}
}
```

```
RegistryRights    : FullControl
AccessControlType : Allow
IdentityReference : BUILTIN\Administrators
IsInherited       : True
InheritanceFlags  : ContainerInherit, ObjectInherit
PropagationFlags  : None
```

Этот код проверяет каждый объект правил доступа, выполняя над ним определенное действие (здесь отбираются правила, действующие для администраторов).

Настройка разрешений на доступ к реестру

Доступом пользователей и групп к элементам реестра управляют с помощью разрешений двух типов: базовых и особых.

Базовые разрешения на доступ к разделам реестра, перечисленные в табл. 12-3, представляют собой комбинации особых разрешений. В таблице обратите внимание на флаги разрешений — эти флаги используются при создании правил доступа.

Табл. 12-3. Базовые разрешения на доступ к реестру

Имя	Что разрешает	Флаг правила
Full Control (Полный доступ)	Чтение, запись, изменение и удаление разделов и параметров реестра	FullControl
Read (Чтение)	Чтение разделов и параметров	ReadKey, ExecuteKey
Write (Запись)	Чтение и запись разделов и параметров	WriteKey

При настройке базовых разрешений на доступ к реестру для пользователей и групп указывают тип управления доступом Allowed или Denied, чтобы разрешить или запретить доступ, соответственно.

Базовые разрешения настраивают с использованием правил доступа. Правило доступа содержит набор массив, определяющих:

- пользователей и группы, на которое оно действует;
- действующие разрешения;
- состояние (Allow или Deny).

Таким образом, общий синтаксис правил доступа выглядит так:

```
«Имя_пользователя», «Разрешение», «Тип»
```

где *Имя_пользователя* — имя пользователя или группы, на которых действует правило, *Разрешение* — применяемое базовое разрешение, а *Тип* — состояние (Allow или Deny). Имена пользователей и групп указывают в формате ИМЯ_КОМПЬЮТЕРА\Имя_пользователя или ИМЯ_ДОМЕНА\Имя_пользователя. В следующем примере группа DeploymentTesters получает полный доступ:

```
«DeploymentTesters», «FullControl», «Allow»
```

Расширенный синтаксис правил доступа выглядит так:

```
«Имя_пользователя», «Разрешение», «Флаг1», «Флаг2», «Тип»
```

где *Имя_пользователя* — имя пользователя или группы, на которые действует правило, *Разрешение* — применяемое базовое разрешение, *Флаг1* — флаг, управляющий наследованием, *Флаг2* — флаг, управляющий распространением унаследованных правил, а *Тип* — тип управления доступом (Allow или Deny). В следующем примере группа DeploymentTesters получает полный доступ к разделу и его содержимому:

```
«DeploymentTesters», «FullControl», «ContainerInherit», «None», «Allow»
```

Флаг, управляющий наследованием, может принимать следующие значения:

- **None** — правило доступа не наследуется;
- **ContainerInherit** — наследуется вложенными разделами;
- **ObjectInherit** — наследуется только параметрами.

Все разделы реестра являются контейнерами, поэтому из флагов, управляющих наследованием, для разделов имеет смысл только ContainerInherit. Если этот флаг не указан, другие флаги игнорируются, и правило применяется только к явно заданному разделу. Если же флаг ContainerInherit указан, правило наследуется вложенными элементами согласно одному из перечисленных ниже флагов.

Флаг, управляющий дальнейшим распространением унаследованных правил, принимает следующие значения:

- **None** — правило доступа наследуется без ограничений;
- **InheritOnly** — правило доступа наследуется только «прямыми потомками» объекта (вложенными разделами и параметрами);
- **NoPropagateInherit** — правило доступа наследуется дочерними объектами и объектами-контейнерами, но не наследуется «внучатыми» объектами (т.е. объектами, вложенными в дочерние объекты-контейнеры);
- **NoPropagateInherit**, **InheritOnly** — правило доступа наследуется только дочерними разделами (контейнерами) заданного раздела, но не наследуется их подразделами.

Для назначения правил доступа ресурса используются методы SetAccessRule() и AddAccessRule() объекта ACL, а для удаления правил доступа — метод RemoveAccessRule() того же объекта. Как сказано выше, тип объектов правил доступа —System.Security.AccessControl.RegistryAccessRule.

Проще всего добавлять и удалять правила доступа следующим образом:

1. Получите объект ACL (например, скопировав его у раздела с параметрами доступа, наиболее близкими к желаемым).
2. Создайте нужное число экземпляров объекта System.Security.AccessControl.RegistryAccessRule и запишите в них соответствующие разрешения;
3. Вызовите метод AddAccessRule() или RemoveAccessRule() ранее полученного объекта ACL, чтобы добавить либо удалить правила доступа как требуется.
4. Чтобы изменения вступили в силу, следует назначить настроенный объект ACL соответствующему разделу реестра.

   Рассмотрим следующий пример:

```
$acl = get-acl hklm:\software\test
$perm = «cpandl\deploymenttesters»,«fullcontrol»,«allow»
$r = new-object system.security.accesscontrol.registryaccessrule $perm
$acl.addaccessrule($r)
$acl | set-acl hkcu:\software\dev
```

Этот код получает объект ACL для раздела HKLM:\Software\Test и записывает его в переменную $perm, далее создается экземпляр объекта RegistryAccessRule для нового правила доступа. Для сохранения правила доступа затем вызывается метод AddAccessRule() ранее полученного объекта ACL. Последняя команда назначает настроенный объект ACL заданному разделу с помощью командлета Set-Acl.

Данный пример легко расширить, чтобы получить код, обрабатывающий группы разделов:

```
$acl = get-acl hklm:\software\test
$perm = «cpandl\devtesters», «fullcontrol», «allow»
$r = new-object system.security.accesscontrol.registryaccessrule $perm
$acl.addaccessrule($r)

$resc = gci hkcu:\software\test -force
foreach($f in $resc) {

  write-host $f.pspath
  $acl | set-acl $f.pspath
}
```

```
  Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\software\test\Dt\IO
  Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\software\test\Queue
  Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\software\test\Stats
```

В этом примере ACL с измененным набором разрешений применяется к разделу HKCU:\Software\Test и всем его подразделам, имена измененных разделов выводятся на консоли.

Особые разрешения на доступ к реестру перечислены в табл. 12-4. Как и для базовых разрешений, для них указан флаг, применяемый при создании правил доступа.

Табл. 12-4. Особые разрешения

Имя	Что разрешает	Флаг правила
Query Values (Запрос значения)	Чтение параметров из заданного раздела	QueryValues
Set Value (Задание значения)	Установку значений параметров раздела	SetValue
Create Subkey (Создание подраздела)	Создание подразделов в заданном разделе	CreateSubKey
Enumerate Subkeys (Перечисление подразделов)	Вывод списка подразделов заданного раздела	EnumerateSubKeys
Notify (Уведомление)	Получение уведомлений об изменении разделов	Notify
Create Link (Создание связи)	Создание связей между разделами	CreateLink

Табл. 12-4. Особые разрешения

Имя	Что разрешает	Флаг правила
Delete (Удаление)	Удаление разделов, подразделов и параметров. Для удаления раздела у пользователя или группы должно быть разрешение Delete не только в отношении этого раздела, но и в отношении его подраздела	Delete
Write DAC (Запись DAC)	Изменение разрешений системы безопасности для разделов	ChangePermissions
Take –Ownership (Смена владельца)	Смену владельца раздела	TakeOwnership
Read Control (Чтение разрешений)	Чтение разрешений, назначенных разделу	ReadPermissions

В табл. 12-5 приводятся комбинации особых разрешений, составляющие базовые разрешения на доступ к разделам реестра.

Табл. 12-5. Комбинации особых разрешений

Разрешение	Full control	Read	Write
Query Values	X	X	
Set Value	X		X
Create Subkey	X		X
Enumerate Subkeys	X	X	
Notify	X	X	
Create Link	X		
Delete	X		
Write DAC	X		
Write Owner	X		
Read Control	X	X	X

Особые разрешения на доступ к реестру настраивают так же, как и базовые разрешения. Для назначения правил доступа к ресурсам вызывают метод SetAccessRule() или AddAccessRule() объекта ACL, а для удаления правил — метод RemoveAccessRule() того же объекта.

Рассмотрим следующий пример:

```
$acl = get-acl hklm:\software\test
$p1 = «cpandl\dev», «queryvalues», «allow»
$r1 = new-object system.security.accesscontrol.registryaccessrule $p1
$acl.addaccessrule($r1)
```

```
$p2 = «cpandl\dev»,«enumeratesubkeys»,«allow»
$r2 = new-object system.security.accesscontrol.registryaccessrule $p2
$acl.addaccessrule($r2)

$acl | set-acl hklm:\software\test
```

В этом примере мы получаем объект ACL для раздела HKLM:\Software\Test (он раздел должен существовать, иначе код примера не сработает). Созданное правило доступа записывается в переменную $p1, далее создается экземпляр объекта RegistryAccessRule и разрешение добавляется к объекту ACL вызовом метода AddAccessRule(). Далее создается второе правило доступа, записывается в переменную $p2. Создается еще один объект RegistryAccessRule и второе правило добавляется к ACL вызовом AddAccessRule(). В завершение настроенный объект ACL назначается заданному разделу реестра.

Смена владельца разделов реестра

Для смены владельца раздела реестра служит метод SetOwner() объекта ACL. Проще всего сменить владельца следующим образом:

1. Получите объект ACL для нужного раздела реестра.
2. Получите идентификатор пользователя или группы, которую планируется сделать владельцем; у этого пользователя или группы уже должно быть соответствующе разрешение для данного раздела реестра.
3. Вызовите метод SetOwner с указанием IdentityReference нового владельца.
4. Примените изменения к нужному разделу реестра.
   Рассмотрим следующий пример:

```
$acl = get-acl hklm:\software\test
$found = $false
foreach($rule in $acl.access) {
 if ($rule.identityreference -like «*administrators*») {
   $global:ref = $rule.identityreference; $found = $true; break}
}

if ($found) {
 $acl.setowner($ref)
 $acl | set-acl hklm:\software\test
}
```

Этот код получает объект ACL для раздела HKLM:\Software\Test и проверяет каждое правило доступа из этого объекта, чтобы отобрать правила, действующие на заданную группу. Если такое правило обнаружено, в переменную $ref записывается идентификатор этой группы, а $found устанавливается в $true, и цикл ForEach прерывается. После выхода из цикла проверяется равенство $found = True. Если значение этой переменной равно

True, в ранее полученный объект ACL записывается разрешение на смену владельца, после чего это объект назначается с помощью Set-Acl разделу HKLM:\Software\Test.

Аудит системного реестра

Администраторам доступна возможность аудита обращений к системному реестру. Аудит позволяет регистрировать пользователей, обращающихся к реестру, и их действия. Поддерживается аудит всех разрешений, перечисленных в табл. 12-1 и 12-2. Впрочем, желательно ограничиваться аудитом необходимого минимума ресурсов, чтобы уменьшить объем данных, записываемых в журналы системы безопасности и сэкономить системные ресурсы.

Прежде, чем включать аудит реестра, необходимо активировать поддержку аудита на компьютере, с которым вы работаете. Это делается через локальную политику сервера либо соответствующую групповую политику. Политика, управляющая аудитом, называется Computer Configuration\Windows Settings\Security Settings\Local Policies\Audit Policy (Конфигурация компьютера\Конфигурация Windows\Параметры безопасности\Локальные политики\Политика аудита).

Включив аудит на компьютере, можно настраивать параметры аудита отдельных разделов реестра. Благодаря механизму наследования, вам не придется вручную настраивать аудит каждого раздела: достаточно включить аудит для нужного корневого раздела или подраздела, и все его содержимое будет автоматически охвачено аудитом.

Политики аудита системного реестра задают с помощью правил аудита. Правило аудита содержит набор массивов, определяющих:

- пользователей или группы, на которые действует правило;
- разрешения, использование которых отслеживается;
- флаг, включающий наследование подразделами параметров аудита, назначенных их родительскому разделу;
- флаг, управляющий дальнейшим наследованием;
- тип аудита.

Таким образом, общий синтаксис правила аудита выглядит так:

«*Имя_пользователя*», «*Разрешение*», «*Флаг1*», «*Флаг2*», «*Тип*»

где *Имя_пользователя* — имя пользователя или группы, на которых действует правило, *Разрешение* — отслеживаемое разрешение, *Флаг1* — флаг, управляющий наследованием правил, *Флаг2* — флаг, управляющий распространением унаследованных правил, а *Тип* — тип аудита.

Флаг, управляющий наследованием, может принимать следующие значения:

- **None** — правило аудита не наследуется;

- **ContainerInherit** — наследуется вложенными разделами;
- **ObjectInherit** — наследуется только параметрами.

Все разделы реестра являются контейнерами, поэтому из флагов, управляющих наследованием, для разделов имеет смысл только ContainerInherit. Если этот флаг не указан, другие флаги игнорируются, и правило применяется только к явно заданному разделу. Если же флаг ContainerInherit указан, правило наследуется вложенными элементами согласно одному из перечисленных ниже флагов.

Флаг, управляющий дальнейшим распространением унаследованных правил, принимает следующие значения:

- **None** — правило аудита наследуется без ограничений;
- **InheritOnly** — правило аудита наследуется только «прямыми потомками» объекта (вложенными разделами и параметрами);
- **NoPropagateInherit** — правило аудита наследуется дочерними объектами и объектами-контейнерами, но не наследуется «внучатыми» объектами (т.е. объектами, вложенными в дочерние объекты-контейнеры);
- **NoPropagateInherit**, **InheritOnly** — правило аудита наследуется только подразделами (контейнерами) заданного раздела, но не наследуется их подразделами и параметрами.

Типа аудита Success (Успехи) позволяет отслеживать успешное применение заданного разрешения, а тип Failure (Отказы) — неудачные попытки использования данного разрешения; тип Both отслеживает и то и другое.

Как и в случае разрешений, имена пользователей и групп при настройке правил аудита указывают в формате ИМЯ_КОМПЬЮТЕРА\Имя_пользователя или ИМЯ_ДОМЕНА\Имя_пользователя. В следующем примере отслеживаются неудачные попытки запроса пользователями из домена CPANDL некоторых параметров реестра:

```
«CPANDL\USERS», «QueryValues», «ContainerInherit», «None», «Failure»
```

Для назначения правил аудита ресурса используются методы SetAuditRule() и AddAuditRule() объекта ACL, а для удаления правил аудита — метод RemoveAuditRule() того же объекта. Тип объектов правил аудита — System.Security.AccessControl.RegistryAuditRule.

Проще всего добавлять и удалять правила доступа следующим образом:

1. Получите объект ACL (например, скопировав его у раздела с параметрами аудита, наиболее близкими к желаемым).
2. Создайте нужное число экземпляров объекта System.Security.AuditControl.RegistryAuditRule и запишите в них соответствующие правила;
3. Вызовите метод AddAuditRule() или RemoveAuditRule() ранее полученного объекта ACL, чтобы добавить либо удалить правила аудита как требуется.
4. Чтобы изменения вступили в силу, следует назначить настроенный объект ACL соответствующему разделу реестра.

Рассмотрим следующий пример:

```
$acl = get-acl hklm:\software\test
$audit = «cpandl\users», «queryvalues», «containerinherit», «none», «failure»
$r = new-object system.security.accesscontrol.registryauditrule $audit
$acl.addauditrule($r)
$acl | set-acl hklm:\software\test
```

Этот код получает объект ACL для раздела HKLM:\Software\Test и записывает его в переменную $audit, далее создается экземпляр объекта RegistryAuditRule для нового правила аудита. Для сохранения правила аудита затем вызывается метод AddAuditRule() ранее полученного объекта ACL. Последняя команда назначает настроенный объект ACL заданному разделу с помощью командлета Set-Acl.

Данный пример легко расширить, чтобы получить код, обрабатывающий группы разделов:

```
$acl = get-acl hklm:\software\test
$audit = «cpandl\users», «queryvalues», «containerinherit», «none», «failure»
$r = new-object system.security.accesscontrol.registryauditrule $audit
$acl.addauditrule($r)

$resc = gci hkcu:\software\test -recurse -force
foreach($f in $resc) {

  write-host $f.pspath
  $acl | set-acl $f.pspath
}
```

```
  Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\software\test\Dt\IO
  Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\software\test\Queue
  Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\software\test\Stats
```

Этот код применяет заданное правило аудита ко всем подразделам раздела HKCU:\Software\Test и выводит на консоли список измененных разделов.

Глава 13

Мониторинг и оптимизация работы Windows-компьютеров

Возможности PowerShell позволяют администраторами диагностировать системные сбои, наблюдать за работой приложений и служб, а также обеспечивать информационную безопасность. Если система стала медленно работать и в ней часто возникают ошибки, имеет смысл поискать причину проблемы в журналах событий. Выявив причины сбоев, можно принять меры к ее устранению и исключению возможности ее повторного возникновения. Мониторинг производительности позволяет вовремя обнаруживать неблагоприятные ситуации при работе систем и делать все необходимое, чтобы такие ситуации больше не повторялись.

Управление журналами событий Windows

Событием в Windows считается любое стечение обстоятельств при работе операционной системы, о котором требуется уведомить администратора или пользователей. События регистрируются в журналах событий Windows, в которых накапливается важная хронологическая информация, необходимая для наблюдения за работой вверенных администратору систем, обеспечения их безопасности, диагностики и устранения сбоев. Регулярно анализировать информацию журналов не просто важно, а жизненно необходимо. Администратор должен внимательно просматривать все содержимое журналов на всех серверах, а на рабочих станциях — хотя бы важнейшие системные события. Он должен обеспечить надежную защиту серверов, бесбойную работу серверных приложений и служб, а также устранять ошибки, снижающие производительность системы. На рабочих станциях следует настроить регистрацию в журналах событий, необходимых для обслуживания систем и устранения неполадок, и обеспечить доступность журналов в любое время.

Работа с журналами событий

Служба Windows, управляющая журналами событий, называется Windows Event Log (Журнал событий Windows), она регистрирует в журналах важную информацию. Набор доступных журналов зависит от роли компьютера и установленных на нем служб. Существует два основных типа файлов журналов:

- **журналы Windows** — в них регистрируются общие события операционной системы, связанные с приложениями, системой безопасности, установкой программ и работой системных компонентов;
- **журналы приложений и служб** — они содержат только события, генерируемые определенными приложениями и службами.

Вот наиболее распространенные журналы событий с описанием регистрируемых в них событий:

- **Application (Приложение)** — важные события, возникающие во время работы приложений. Например, в журналах Microsoft Exchange Server регистрируются события, связанные с обменом почтовыми сообщениями, работой хранилищ данных, почтовых ящиков и состоянием связанных служб. По умолчанию хранится в %SystemRoot%\System32\Winevt\Logs\Application.evtx.
- **DFS Replication (Репликация DFS)** — на контроллерах домена, участвующих в репликации DFS, в этом журнале регистрируются события, связанные с репликацией файлов, изменением состояния соответствующих служб и управлением ими, а также сканированием системных томов и управлением наборами репликации. По умолчанию хранится в %SystemRoot%\System32\Winevt\Logs\DFS Replication.Evtx.
- **Directory Service (Служба каталогов)** — на контроллерах домена в этом журнале регистрируются события, связанные с работой доменных служб Active Directory (Active Directory Domain Services, AD DS), запуском служб каталогов и проверкой целостности каталога. По умолчанию хранится в %SystemRoot%\System32\Winevt\Logs\Directory Service.Evtx.
- **DNS Server (DNS-сервер)** — на DNS-серверах в этом журнале регистрируются запросы, отклики и другие действия, связанные с работой DNS. По умолчанию находится в %SystemRoot%\System32\Winevt\Logs\DNS Server.Evtx.
- **File Replication Service (Служба репликации файлов)** — в этом журнале регистрируются действия, связанные с репликацией файлов. По умолчанию хранится в %SystemRoot%\System32\Winevt\Logs\File Replication Service.Evtx.
- **Forwarded Events (Пересылаемые события)** — в этом журнале регистрируются события, которые переслали другие серверы (если настроена пересылка событий). По умолчанию хранится в %SystemRoot%\System32\Winevt\Logs\FordwardedEvents.Evtx.

- **Hardware Events (События оборудования)** — если в системе включен прием событий от аппаратных подсистем, то события, сгенерированные оборудованием, регистрируются в этом журнале. По умолчанию хранится в %SystemRoot%\System32\Winevt\Logs\HardwareEvent.Evtx.
- **Microsoft\Windows** — группа журналов, регистрирующих события, связанные с работой определенных служб и компонентов Windows. События в этих журналах упорядочены по категории события и имени компонента.
- **Security (Безопасность)** — в этом журнале регистрируются события системы безопасности, например вход и выход пользователей, использование привилегий и защищенных ресурсов. По умолчанию хранится в %SystemRoot%\System32\Winevt\Logs\Security.Evtx.

Совет Для доступа к журналам системы безопасности необходимо право Manage Auditing And Security Log (Управление аудитом и журналом безопасности). По умолчанию это разрешение есть у администраторов. Подробнее о назначении прав см. в главе 10 Справочника администратора по Windows Server 2008 (БХВ-Петербург, Русская Редакция, 2009).

- **Setup (Установка)** — в этом журнале log records events logged by the operating system or its components during setup and installation. The default location is %SystemRoot%\System32\Winevt\Logs\Setup.Evtx.
- **System (Система)** — в этом журнале регистрируются события, связанные с работой операционной системы и ее компонентов: запуск или сбои служб, инициализация драйверов, общесистемные сообщения. По умолчанию хранится в %SystemRoot%\System32\Winevt\Logs\System.Evtx.
- **Windows PowerShell** — в этом журнале регистрируются события, связанные с PowerShell. По умолчанию он хранится в файле %SystemRoot%\System32\Winevt\Logs\Windows PowerShell.Evtx.

По уровню серьезности события делятся на несколько категорий, от информационных сообщений и предупреждений до сообщений о критических сбоях:

- **Information (информационное событие)** — уведомление, как правило, об успешно завершенной операции;
- **Warning (событие предупреждения)** — общее предупреждение; общие предупреждения позволяют предотвратить возникновение проблем в будущем;
- **Error (событие ошибки)** — критическая ошибка, например, неполадки при настройке адреса DHCPv6;
- **Critical (Критическое событие)** — критическое событие, например перезагрузка компьютер из-за отказа электропитания или краха системы;
- **Audit Success (Успех проверки)** — успешное исполнение операции, включенной в аудит (например, использование привилегии);
- **Audit Failure (Ошибка аудита)** — неудача операции, включенной в аудит (например, неудачный вход в систему).

Совет Типов событий много, но особого внимания требуют предупреждения и ошибки. Если причина их возникновения неясна, необх одимо тщательно изучить соответствующие записи журнала, чтобы понять, требуется ли дальнейшее вмешательство.

Наряду с уровнем, события обладают рядом общих свойств:

- **Computer** — имя компьютера, вызвавшего генерацию события;
- **Data** — данные или код ошибки, сгенерированный вместе с событием;
- **Date and Time** — дата и время события;
- **Description** — подробное описание события; может включать ссылки на дополнительные источники сведений о неполадках и способах их устранения, открывается по двойному щелчку записи события в консоли Event Viewer;
- **Event ID** — числовой уникальный идентификатор события, сгенерированный его источником;
- **Log Name** — имя журнала, в котором зарегистрировано событие;
- **Source** — источник события (приложение, служба или системный компонент), это свойство облегчает поиск причин возникновения события;
- **Task Category** — категория события, иногда используется для дополнительного описания связанного действия. У каждого источника событий могут быть свои категории. Так, система безопасности генерирует следующие категории событий: logon/logoff, privilege use, policy change и account management;
- **User** — имя учетной записи пользователя (или системной учетной записи, такой как Local Service, Network Service или Anonymous Logon), вызвавшего событие. Если событие вызвано не пользователем, то вместо имени пользователя это свойство содержит строку «N/A».

Для управления событиями используется средство с графическим интерфейсом — консоль Event Viewer (Просмотр событий). Чтобы открыть эту консоль с журналами локального компьютера, введите в командной строке PowerShell **eventvwr**; а чтобы открыть Event Viewer с данными удаленного компьютера, введите **eventvwr *Имя_компьютера***, где *Имя_компьютера* — имя удаленного компьютера, журналы которого вам требуются. Как и все GUI-инструменты, консоль Event Viewer проста в работе, поэтому вы можете продолжать использовать ее для некоторых задач по обслуживанию компьютеров, но для работы с журналами событий PowerShell поддерживает специальные команды:

- **Get-WinEvent** — получает сведения о событиях или трассировке из журналов заданного компьютера (только в Windows Vista, Server 2008 и выше).

```
Get-WinEvent [-ListLog] Журнал {BasicParams}
Get-WinEvent [-ListProvider] Поставщик {BasicParams}

Get-WinEvent [-Path] Файл_журнала {BasicParams} {AddtlParams}
Get-WinEvent [-LogName] Журнал {BasicParams} {AddtlParams}
```

```
Get-WinEvent [-ProviderName] Имя {BasicParams} {AddtlParams}
Get-WinEvent -FilterHashTable Значения {BasicParams} {AddtlParams}

{BasicParams}
[-ComputerName Имя_компьютера] [-Credential Объект_удостоверения]

{AddtlParams}
[-FilterXPath XPathQuery] [-Oldest] [-MaxEvents Число_событий]
```

Совет С помощью XPath-запросов можно создавать «отфильтрованные» представления журналов событий, позволяющие быстро отбирать события, соответствующие заданным критериям. При этом для воссоздания такого представления на любом совместимом компьютере достаточно исполнить соответствующий XPath-запрос.

- **Clear-EventLog** — удаляет все записи из заданного журнала событий указанного компьютера.

```
Clear-EventLog [[-ComputerName] Имя_компьютера] [-LogName] Журнал
```

- **Get-EventLog** — получает список журналов или событий в заданном журнале на указанном компьютере.

```
Get-EventLog [-List] [-ComputerName Имя_компьютера] [-AsString]

Get-EventLog [-ComputerName Имя_компьютера] [-LogName] Журнал
[-AsBaseObject] [-After Дата_время] [-Before Дата_время]
[-EntryType Тип_записей] [-Index Индекс] [[-InstanceID] ID]
[-Message Сообщение] [-Newest Число_событий] [-Source Источник]
[-UserName Пользователь]
```

- **Limit-EventLog** — устанавливает предельный размер и сроки (в сутках) хранения событий в заданных журналах на указанных компьютерах.

```
Limit-EventLog [-ComputerName Имя_компьютера] [-MaximumKiloBytes
Макс_размер] [-OverFlowAction {DoNotOverwrite | OverwriteAsNeeded |
OverwriteOlder] [-Retention Мин_срок] [-LogName] Журнал
```

- **Show-EventLog** — выводит содержимое заданного журнала событий на указанном компьютере с помощью Event Viewer.

```
Show-EventLog [[-ComputerName] Имя_компьютера]
```

Примечание Для работы с журналом событий PowerShell используются командлеты Get-Event, Wait-Event и Remove-Event, а для создания собственных журналов — New-EventLog, Register-ObjectEvent, Register-EngineEvent, Unregister-Event, Get-EventSubscriber и Register-WmiEvent.

Мониторинг системных событий должен проводиться регулярно, а не от случая к случаю. Так, на сервере журналы событий необходимо анализировать минимум раз в сутки, тогда как на рабочих станциях это можно делать по необходимости, например, если пользователь пожаловался на некоторую проблему.

Просмотр и фильтрация журналов событий

Для извлечения подробной информации из журналов событий используются командлеты Get-EventLog и Get-WinEvent. Достоинствами Get-EventLog являются гибкость и простота в использовании. С Get-WinEvent удобно применять сложные фильтры, основанные на XPath-запросах и хэш-значениях. Если сложные фильтры не нужны, то без Get-WinEvent можно обойтись.

Вот общий синтаксис Get-EventLog:

```
get-eventlog «Журнал» [-computername Имя_компьютера]
```

где *Журнал* — это имя журнала, например «Application», «System» или «Directory Service», а *Имя_компьютера* — имя удаленного компьютера или компьютеров. Вот пример проверки журнала Application:

```
get-eventlog «Application» -computername fileserver87, dbserver23
```

Примечание Технически, брать имя журнала в кавычки обязательно, только если в нем есть пробел, как в случае журналов DNS Server, Directory Service или File Replication Service.

Вывод этой команды выглядит примерно так:

```
Index  Time          EntryType   Source        InstanceID Message
-----  ----          ---------   ------        ---------- -------
22278  Feb 27 10:54 Information DQLWinService          0 The description
for Event ID '0' in Source 'DQL
22277  Feb 27 10:49  Information DQLWinService         0 The description
for Event ID '0' in Source 'DQL
```

Как видите, команда выводит свойства событий: Index, Time, EntryType, Source, InstanceID и Message. Index — это номер записи событий в журнале; Time — время регистрации события; EntryType — категория события; Source — источник события; InstanceID — числовой идентификатор события (это значение идентично EventID в журнале), а Message — описание события.

Судя по значению свойства Index, вывод содержит описание событий под номерами 22 277 и 22 278. По умолчанию Get-EventLog возвращает все события из заданного журнала, начиная с самого нового. В большинстве случаев такой большой объем информации просто не требуется, поэтому полезные сведения приходится отбирать из общей массы данных с помощью фильтров. Так, можно извлечь сведения лишь о самых последних событиях.

Параметр –Newest ограничивает число возвращаемых событий. Например, чтобы получить 50 самых новых событий из журнала Security, введите:

```
get-eventlog «security» -newest 50
```

К достоинствам командлета Get-EventLog относится поддержка группирования и фильтрации результирующих наборов событий. Группируя события по типу, проще разделить уведомления, предупреждения и сообщения об ошибках. Сортировка событий по источнику упрощает наблюдение за со-

бытиями из определенных источников, а сортировка по EventID позволяет выявлять повторно возникающие события.

Сгруппировать события по значениям свойств Source, EventId, EntryType или TimeGenerated можно следующим образом:

1. Получите нужные события и запишите их в переменную, например так:

```
$e = get-eventlog -newest 100 -logname «application»
```

2. Вызовите Group-Object, чтобы отсортировать объекты событий. Следующая команда группирует события по значению свойства EventType:

```
$e | group-object -property eventtype
```

События также могут быть отсортированы по значению заданного свойства, например Source, EventId, EntryType или TimeGenerated. Для этого:

1. Получите нужные события и запишите их в переменную, например так:

```
$e = get-eventlog -newest 100 -logname «application»
```

2. Вызовите командлет Sort-Object, чтобы отсортировать объекты событий по по значению заданного свойства (в данном примере — EntryType):

```
$e | sort-object -property entrytype
```

Как правило, просматривать все события, сгенерированные системой, не требуется. Чаще нужно проверять лишь предупреждения и сообщения о критических ошибках, их-то и требуется отфильтровать. Фильтры позволяют извлечь из журнала события, соответствующие любым заданным вами критериям. В данном случае необходимо отобрать события, у которых в значении свойства EntryType содержится строка *error*. Вот как это делается:

1. Получите нужные события и запишите их в переменную, например так:

```
$e = get-eventlog -newest 500 -logname «application»
```

2. Воспользуйтесь командлетом Where-Object для поиска в переменной $e объектов событий, у которых в значении заданного свойства содержится определенная строка (в этом примере — Error):

```
$e | where-object {$_.EntryType -match «error»}
```

При использовании параметра –Match командлет Where-Object ведет поиск без учета регистра символов, при котором критерии поиска **Error**, **error** и **ERROR** равноценны. Аналогичным образом можно отобрать события-предупреждения, критические события и события-уведомления. Поскольку Where-Object считает поиск успешным даже при частичном совпадении с искомой строкой, не обязательно даже вводить тип события полностью, например **warn**, **crit** или **info**:

```
$e = get-eventlog -newest 100 -logname «application»
$e | where-object {$_.EntryType -match «warn»}
```

Where-Object работает и с другими свойствами объектов-событий. В следующем примере ведется поиск событий, свойство Source которых содержит текст *User Profile Service*:

```
$e = get-eventlog -newest 500 -logname «application»
$e | where-object {$_.Source -match «User Profile Service»}
```

А эти команды отбирают события с EventID *1530*:

```
$e = get-eventlog -newest 500 -logname «application»
$e | where-object {$_.EventID -match «1530»}
```

Иногда администратору требуется найти события, сгенерированные до или после заданной даты и времени. Для этой цели служат параметры –Before и –After, соответственно.

В следующем примере извлекаются все события-ошибки, записанные в журнал System в июле 2010 года:

```
$Jun30 = get-date 6/30/10
$Aug1 = get-date 8/01/10
get-eventlog -log «system» -entrytype Error -after $jun30 -before $aug1
```

А здесь мы получаем все события-ошибки, записанные в журнал System за последнюю неделю:

```
$startdate = (get-date).adddays(-7)
get-eventlog -log «system» -entrytype Error -after $startdate
```

Чтобы автоматизировать поиск нужных событий, напишите соответствующий сценарий, записывающий результаты поиска в файл. Рассмотрим следующий пример:

```
$e = get-eventlog -newest 100 -logname «system» $e | where-object
{$_.EntryType -match «error»} > \\FileServer18\www\currentlog.txt

$e = get-eventlog -newest 100 -logname «application» $e | where-object
{$_.EntryType -match «error»} >> \\FileServer18\www\currentlog.txt

$e = get-eventlog -newest 100 -logname «security» $e | where-object
{$_.EntryType -match «error»} >> \\FileServer18\www\currentlog.txt
```

Этот код пытается найти среди 100 последних событий в журналах System, Application и Security события-ошибки; найденные события записываются на сетевой диск сервера FileServer18 в файл CurrentLog.txt. В первой команде оператор > перезаписывает существующий выходной файл, а остальные команды содержат оператор >>, который дописывает данные в конец CurrentLog.txt. Таким образом, файл Currentlog.txt перезаписывается при каждом запуске сценария и содержит только актуальные события. Кроме того, можно создать задание планировщика, чтобы автоматически запускать этот сценарий один или несколько раз в сутки с заданным интервалом.

Настройка параметров журнала

С помощью параметров журнала настраивают предельные размеры и способ записи событий. По умолчанию для журнала задан размер, максимально допустимый для файла. По достижении этого предела события новые события начинают записываться поверх самых старых.

Для настройки параметров журнала используется командлет Limit-EventLog. Вот его общий синтаксис:

```
Limit-EventLog [-ComputerName Компьютеры] [-LogName] Журналы Параметры
```

где *Компьютеры* — имена компьютеров, на которых требуется настроить журнал, *Журналы* — имена настраиваемых журналов, а возможные *Параметры* перечислены ниже:

- **–MaximumSize** — задает максимальный размер журнала (в байтах), допустимый размер — 64 Кб–4 Гб (с шагом в 64 Кб). Настраивая максимальный размер журнала, учитывайте размер свободного места на диске. Большинство файлов журналов по умолчанию хранится в каталоге %SystemRoot%\System32\Winevt\Logs.
- **–OverFlowAction** — задает режим перезаписи событий. Допустимые значения: DoNotOverwrite, OverwriteAsNeeded и OverwriteOlder. Если для этого параметра установлено значение DoNotOverwrite, то при достижении максимального размера журнала компьютер генерирует сообщение об ошибке. Если выбрано значение OverwriteAsNeeded, то при заполнении журнала новые события записываются поверх самых старых. Значение OverwriteOlder разрешает при заполнении журнала перезаписывать только события, зарегистрированные раньше даты, заданной свойством Retention; если же в журнале не окажется таких событий, компьютер генерирует сообщения об ошибке, свидетельствующие о невозможности перезаписи событий.
- **–Retention** — задает минимальный срок (в сутках) хранения записей в журнале.

В следующем примере для журнала System задается максимальный размер 4096 кб и параметр OverwriteOlder, разрешающий перезаписывать события после семи дней хранения:

```
limit-eventlog -maximumsize 4096kb -overflowaction overwriteolder
-retention 7 -logname system
```

Если требуется настроить журналы на нескольких компьютерах, задайте их имена с помощью свойства –ComputerName либо загрузите список компьютеров из текстового файла, как показано ниже:

```
limit-eventlog -computername (get-content c:\data\clist.txt)
-maximumkilobytes 4096 -overflowaction overwriteolder -retention 7
-logname system
```

В этом примере список удаленных компьютеров, подлежащих настройке, загружается из файла CList.txt, расположенного в каталоге C:\Data.

Архивация и очистка журналов

На важных серверах сети, таких как контроллеры домена и серверы приложений, рекомендуется хранить журналы за последние несколько месяцев. Однако не очень удобно накапливать эти данные, устанавливая максимальный размер для журналов. Вместо этого лучше периодически архивировать нужные журналы вручную либо автоматически, средствами Windows. Для архивации журналов используются следующие форматы:

- **«родной» формат журналов** (.evtx-файлы, которые просматривают через консоль Event Viewer);
- **текстовые файлы с разделителями-табуляторами** (.txt-файлы). Для просмотра и редактирования таких файлов используются текстовые редакторы, электронные таблицы и СУБД;
- **текстовые файлы с разделителями-запятыми** (.csv-файлы) — формат, предназначенный для импорта в электронные таблицы и СУБД;
- **XML** (.xml-файлы) — файлы с данными на языке Extensible Markup Language (XML).

Для архивации оптимален формат .evtx, используйте его, если планируете просматривать старые архивы с помощью консоли Event Viewer (Просмотр событий). Если же для анализа журналов планируется использовать другие программы лучше сохранить журналы как текстовые файлы с разделителями-запятыми или табуляторами. Иногда для правильной интерпретации в электронных таблицах текстовые файлы журналов требуется «подчищать» в текстовом редакторе. Впрочем, если у вас есть файл журнала в формате .evtx, всегда можно получить его текстовую версию, выбрав команду Save As (Сохранить как) в меню консоли Event Viewer.

Windows автоматически архивирует журнал, для которого в консоли Event Viewer включен режим Archive The Log When Full, Do Not Overwrite Events (Архивировать журнал при заполнении; не перезаписывать события).

Чтобы заархивировать журнал вручную, выполните следующие действия:

1. Откройте консоль Event Viewer, щелкните правой кнопкой журнал, который требуется заархивировать, и выберите из контекстного меню команду **Save Events As (Сохранить события как)**.
2. В диалоге сохранения файла укажите каталог и тип файла журнала.
3. По умолчанию для сохранения журналов задан тип «Файлы событий» (Event Files; *.evtx). Выберите нужный формат и щелкните **Save (Сохранить)**.
4. Если откроется окно **Display Information (Отображение сведений)**, выберите нужные сведения и щелкните **OK**.

Для регулярной архивации журналов желательно создать специальный каталог — так будет проще находить нужные архивы. Кроме того, желательно назначить файлам журналов имена, по которым было бы легко определить формат журнала и период времени, которому он соответствует. Например, файл архива журнала System за январь 2010 года можно назвать «System Log January 2010».

Для очистки заполненных журналов событий используется командлет Clear-EventLog. Вот его общий синтаксис:

```
Clear-EventLog [-ComputerName Компьютеры] [-LogName] Журналы
```

где *Компьютеры* — имена компьютеров, на которых требуется очистить журналы, а *Журналы* — имена очищаемых журналов, например:

```
clear-eventlog system
```

Запись пользовательских событий в журналы

Иногда требуется, чтобы сценарий, запланированное задание или другая программа регистрировала нестандартные события в журнале. Например, после успешного завершения сценария можно записать в журнал Application уведомление, а при неудачном завершении — предупреждение или сообщение об ошибке. Так будет проще узнать, чем завершился запуск сценария и принять соответствующие меры.

PowerShell поддерживает регистрацию событий в журнале PowerShell. Для просмотра содержимого этого журнала используется командлет Get-EventLog. Для управления регистрацией событий используются следующие переменные окружения, которые необходимо установить в соответствующем профиле. Эти переменные управляют регистрацией следующих событий:

- **$LogCommandHealthEvent** — ошибок и исключений при инициализации и обработке команд;
- **$LogCommandLifecycleEvent** — запуска и остановки команд, а также исключений системы безопасности, возникающих при обнаружении команд;
- **$LogEngineHealthEvent** — ошибок и сбоев сеансов;
- **$LogEngineLifecycleEvent** — запуска и завершения сеансов;
- **$LogProviderHealthEvent** — ошибок поставщиков (ошибок чтения-записи, поиска и вызова);
- **$LogProviderLifecycleEvent** — подключения и отключения поставщиков PowerShell.

Если некоторая переменная установлена в $True, соответствующие события записываются в журнал PowerShell, а если в $False, то эти события не записываются.

Для создания нестандартных событий используется утилита Eventcreate. Такие события могут записываться в любой журнал, кроме Security. Опре-

деление нестандартного события включает его источник, идентификатор и описание. Вот синтаксис вызова Eventcreate:

```
eventcreate /l Журнал /so Источник /t Тип /id ID
/d Описание
```

где

- *Журнал* — имя журнала, в который должно записываться событие. Если имя журнала содержит пробелы, его необходимо взять в кавычки, например «DNS Server».

Совет Нестандартные события можно записывать в любые журналы, кроме Security. Первое нестандартное событие всегда записывается в журнал Application, а все последующие — в заданный вами журнал.

- *Источник* — задает источник событий. Значением этого параметра может быть любая строка; если эта строка содержит пробелы, ее нужно взять в кавычки, например: «Event Tracker». Обычно этот параметр идентифицирует приложение, задачу или сценарий, генерирующий событие.

Совет Имена источников нестандартных событий необходимо тщательно продумывать. Имя нового источника не должно совпадать с именами источников, связанных с существующими службами и приложениями, ролями, службами ролей и компонентами. Так, имена DNS, W32Time и Ntfrs в Windows Server 2008 уже «заняты».

Кроме того, после записи события его источник регистрируется, и попытка записи события из того же источника в другой журнал на том же компьютере приводит к ошибке. Например, если событие с источником «EventChecker» записано в журнал Application сервера FILESERVER82, при попытке записи события с тем же источником в журнал System на том же сервере вы получите сообщение «ERROR: Source already exists in 'Application' log. Source cannot be duplicated».

- *Тип* — задет тип событий (Information, Warning или Error). Типы Audit Success и Audit Failure не поддерживаются, поскольку они используются с журналами системы безопасности, в которые запрещено записывать нестандартные события.
- *ID* — задает численный идентификатор события, допустимы идентификаторы из диапазона 1–1 000. Назначение идентификаторов желательно спланировать. Для этого составьте список событий и разбейте его на категории. Например, идентификаторы 100–199 можно назначать общим событиям, 201–299 — событиям, связанным с изменением состояния, 500–599 — предупреждениям, а 900–999 — ошибкам.
- *Описание* — задает описание события, им может быть произвольная строка (не забывайте брать ее в кавычки).

На локальном компьютере Eventcreate по умолчанию запускается с полномочиями текущего зарегистрированного в системе пользователя. При необходимости можно указать удаленный компьютер, а также другую учетную запись с помощью команды **/S *Компьютер* /u [*Домен*\]*Пользователь* [/P *Пароль*]**, где *Компьютер* — имя или IP-адрес компьютера, *Домен* — имя до-

мена, в котором расположена учетная запись пользователя (необязательный параметр), *Пользователь* — имя учетной записи, под которой требуется запустить Eventcreate, а *Пароль* — пароль этой учетной записи (необязательный параметр).

Рассмотрим использование Eventcreate на примерах:

Создание события-уведомления с источником «event tracker» и идентификатором 209 для записи в журнал Application:

```
eventcreate /l «application» /t information /so «Event Tracker»
/id 209 /d «evs.bat script ran without errors.»
```

Создание события-предупреждения с источником «custapp» и идентификатором 511 для записи в журнал System:

```
 eventcreate /l «system» /t warning /so «CustApp» /id 511
/d «sysck.exe didn't complete successfully.»
```

Создание события-ошибки с источником «sysmon» и идентификатором 918 для записи в журнал System на сервере fileserver18:

```
eventcreate /s fileserver18 /l «system» /t error /so «sysmon»
/id 918 /d «sysmon.exe was unable to verify write operation.»
```

Создание и использование сохраненных запросов

В Vista, Server 2008 и более высоких версиях Windows функции фильтрации и запросов в Event Viewer были существенно улучшены. Благодаря этим усовершенствованиям консоль Event Viewer получила поддержку создания настраиваемых представлений и фильтрации журналов с помощью XPath-запросов. XPath — это отличный от XML язык, применяемый для описания отдельных частей XML-документов. Event Viewer использует XPath-выражения для выборки информации из журналов событий и создания настраиваемых и отфильтрованных представлений журнала.

Чтобы воссоздать ранее определенное представление в Event Viewer, можно скопировать соответствующий XPath-запрос и сохранить его в файле настраиваемого представления (Event Viewer Custom View). Чтобы получить такое же представление на любом компьютер с Windows Vista или Windows Server 2008, просто запустите этот запрос еще раз. Например, чтобы сохранить фильтрованное представление журнала Application, созданное для поиска неполадок SQL Server, скопируйте соответствующий XPath-запрос и запищите его в файл представления — этот файл позволит воссоздать это представление на любом компьютере вашей организации.

Некоторые фильтрованные представления Event Viewer создает автоматически, эти представления содержит узел Custom Views (Настраиваемые представления). Щелкнув узел Administrative Events (События управления), вы получите полный список ошибок и предупреждений из всех жур-

налов. Узлы, вложенные в узел Server Roles (Серверные роли), отображают списки событий, специфичных для конкретных ролей.

Чтобы создать и сохранить настраиваемое представление, сделайте следующее:

1. Откройте консоль Event Viewer (Просмотр событий), щелкнув меню **Administrative Tools | Event Viewer (Администрирование | Просмотр событий)**.
2. Щелкните узел **Custom Views (Настраиваемые представления)** и выберите в меню **Action (Действие)** команду **Create Custom View (Создать настраиваемое представление)** либо щелкните соответствующую кнопку на панели действий.
3. В окне **Create Custom View (Создание настраиваемого представления)** выберите в списке **Logged (Запись событий)** период мониторинга событий. Доступные варианты — Anytime (Любое время), Last Hour (Последний час), Last 12 Hours (Последние 12 часов), Last 24 Hours (Последние 24 часа), Last 7 Days (Последние 7 дней) или Last 30 Days (Последние 30 дней). Также можно указать собственный период времени.
4. Флажками **Event Level (Уровень события)** выберите типы события для записи в журнал, флажок **Verbose (Подробно)** позволяет регистрировать дополнительную информацию.
5. Далее можно создать настраиваемое представление для заданных журналов или событий:
   - выберите в списке **Event Logs (Журналы событий)** нужные журналы, пометив соответствующие флажки;
   - выберите в списке **Event Sources (Источники событий)** нужные источники событий, пометив соответствующие флажки.
6. При желании укажите пользователя и компьютеры в полях **User (Пользователь)** и **Computer(s) [Компьтер(ы)]**, в противном случае будут обрабатываться события, сгенерированные для всех пользователей на всех компьютерах.
7. Перейдите на вкладку **XML** — откроется соответствующий XPath-запрос.
8. Щелкните **OK**, чтобы закрыть окно Create Custom View. В окне **Save Filter To Custom View (Сохранение фильтра в настраиваемое представление)** укажите имя и описание настраиваемого представления.
9. Укажите место сохранения настраиваемого представления. По умолчанию такие представления сохраняются в узле **Custom View (Настраиваемые представления)**. Можно также создать новый узел, для этого щелкните **New Folder (Создать папку)**, введите ее имя и щелкните **OK**.
10. Щелкните **OK**, чтобы закрыть окно **Save Filter To Custom View** — список событий будет отфильтрован, как задано.

11. Щелкните правой кнопкой настраиваемое представление и выберите **Export Custom View (Экспортировать настраиваемое представление)**. В диалоге сохранения файла укажите место сохранения и имя файла представления.

Полученный файл представления содержит XPath-запрос, который отображался на вкладке **XML** (см. шаг 7). Члены группы Event Log Readers (Читатели журнала событий), администраторы и другие лица с соответствующими разрешениями смогут запустить сохраненный запрос для просмотра событий, в том числе на удаленных компьютерах, используя следующий синтаксис:

```
eventvwr Компьютер /v: Файл_запроса
```

где *Компьютер* — имя компьютера, события которого требуется изучить, а *Файл_запроса* — имя и путь к файлу представления с XPath-запросом, например:

```
eventvwr fileserver18 /v: importantevents.xml
```

После запуска Event Viewer вы найдете соответствующее настраиваемое представление в узле Custom Views (Пользовательские представления).

Управление системными службами

Службы выполняют ключевые функции, необходимые для работы серверов и рабочих станций. Для управления системными службами на локальном и удаленных компьютерах используются следующие команды, в которых службы задают по именам, отображаемым именам либо с помощью ссылок на объекты служб.:

- **Get-Service** — получает информацию о работающих и остановленных службах на локальном компьютере.

```
Get-Service [[-Name] Службы] [AddlParams]
Get-Service -DisplayName Службы [AddlParams]
Get-Service [-InputObject Объекты_служб] [AddlParams]

{AddlParams}
[-ComputerName Компьютеры] [-DependentServices] [-Exclude
Службы] [-Include Службы] [-ServicesDependedOn]
```

- **Stop-Service** — останавливает службы. У служб, поддерживающих остановку, свойство CanStop установлено в True. Учтите, что останавливать можно только службы в состоянии, допускающем остановку.

```
Stop-Service [-Name] Службы [AddlParams]
Stop-Service -DisplayName Службы [AddlParams]
Stop-Service [-InputObject Объекты служб] [AddlParams]
```

```
{AddlParams}
[-Include Службы] [-Exclude Службы] [-Force]
[-PassThru]
```

- **Start-Service** — запускает остановленные службы.

```
Start-Service [-Name] Службы [AddlParams]
Start-Service -DisplayName Службы [AddlParams]
Start-Service [-InputObject Объекты_служб] [AddlParams]

{AddlParams}
[-Include Службы] [-Exclude Службы] [-Force]
[-PassThru]
```

- **Suspend-Service** — приостанавливает службы. У служб, допускающих приостановку, свойство CanPauseAndContinue установлено в True; приостановка службы возможна, только если служба находится в состоянии, допускающем приостановку.

```
Suspend-Service [-Name] Службы [AddlParams]
Suspend-Service -DisplayName Службы [AddlParams]
Suspend-Service [-InputObject Объекты_служб] [AddlParams]

{AddlParams}
[-Exclude Службы] [-Include Службы] [-PassThru]
```

- **Resume-Service** — возобновляет приостановленные службы, для служб в другом состоянии эта команда игнорируется.

```
Resume-Service [-Name] Службы [AddlParams]
Resume-Service -DisplayName Службы [AddlParams]
Resume-Service [-InputObject Объекты_служб] [AddlParams]

{AddlParams}
[-Exclude Службы] [-Include Службы] [-PassThru]
```

- **Restart-Service** — перезапускает работающие и запускает остановленные службы.

```
Restart-Service [-Name] Службы [AddlParams]
Restart-Service -DisplayName Службы [AddlParams]
Restart-Service [-InputObject Объекты_служб] [AddlParams]

{AddlParams}
[-Include Службы] [-Exclude Службы] [-Force]
[-PassThru]
```

- **Set-Service** — изменяет свойства и(или) состояние служб.

```
Set-Service [-Name] Служба [ | -InputObject Объекты_служб]
[-DisplayName Отображаемое_имя] [-Description Описание]
```

```
[-StartupType {Automatic | Manual | Disabled}]
[-Status {Running | Stopped | Paused}] [-PassThru] [-ComputerName
Компьютеры]
```

- **New-Service** — регистрирует новую службу Windows в системном реестре и базе данных служб, при необходимости принимает учетные данные.

```
New-Service [-Credential Объект_учетных_данных] [-DependsOn
Службы] [-Description Описание] [-DisplayName Отображаемое_имя]
[-StartupType {Automatic | Manual | Disabled}]
[-Name] Name [-BinaryPathName] Путь_к_EXE-файлу
```

Некоторые из перечисленных выше команд позволяют управлять службами удаленных компьютеров. Для этого укажите NetBIOS-имя, IP-адрес или полное доменное имя (FQDN) нужных компьютеров как значение параметра –ComputerName. Чтобы указать локальный компьютер, достаточно ввести точку (.) или «localhost».

Проверка настроенных служб

Чтобы получить полный список служб, настроенных в системе, введите в командной строке **get-service**, указав нужный компьютер с помощью параметра –ComputerName:

```
get-service -computername fileserver86
```

```
Status   Name               DisplayName
------   ----               -----------
Stopped  AppMgmt            Application Management
Running  AudioEndpointBu... Windows Audio Endpoint Builder
Running  Audiosrv           Windows Audio
Running  BFE                Base Filtering Engine
Running  BITS               Background Intelligent Transfer Ser...
Running  Browser            Computer Browser
Stopped  CertPropSvc        Certificate Propagation
```

Параметр –ComputerName принимает и списки имен, так удается проверить состояние служб на нескольких компьютерах, например:

```
get-service -computername fileserver86, dcserver22, printserver31
```

Чтобы не вводить имена компьютеров вручную, сохраните их в текстовом файле (каждое имя — на отдельной строке), который можно использовать так:

```
get-service -computername (get-content c:\data\clist.txt)
```

В этом примере список удаленных компьютеров загружается из файла CList.txt, расположенного в каталоге C:\Data.

Чтобы проверить конкретную службу, укажите ее имя либо отображаемое имя (разрешается использование подстановочных знаков):

```
get-service -displayname *browser* -computername fileserver86
```

```
Status   Name               DisplayName
------   ----               -----------
Running  Browser            Computer Browser
```

Эта команда выводит все службы, у которых отображаемое имя содержит слово *browser*.

Командлет Get-Service возвращает все объекты, представляющие службы, соответствующе заданным критериям. Ее стандартный вывод содержит значения свойств Status, Name и DisplayName этих объектов (см. пример выше). Чтобы вывести все свойства, отформатируйте вывод команды как показано ниже:

```
get-service -displayname *browser* -computername server12 | format-list *
```

```
Name                : Browser
CanPauseAndContinue : False
CanShutdown         : False
CanStop             : True
DisplayName         : Computer Browser
DependentServices   : {}
MachineName         : Server12
ServiceName         : Browser
ServicesDependedOn  : {LanmanServer, LanmanWorkstation}
ServiceHandle       :
Status              : Running
ServiceType         : Win32ShareProcess
Site                :
Container           :
```

Вывод этой команды содержит полные сведения о конфигурации службы. Администраторам чаще всего требуются следующие свойства:

- **Name** и **ServiceName** — краткое имя службы; если в этом столбце отсутствует имя какой-либо службы, это свойство выводится только для установленных служб; если имени нужной службы нет в этом поле, эту службу необходимо установить;
- **DisplayName** — понятное имя службы;
- **Status** — состояние службы (Running, Paused или Stopped);
- **DependentServices** — службы, для работы которых требуется данная служба;
- **ServicesDependedOn** — службы, необходимые для работы других служб;
- **Type** — тип службы, разделяет ли она общий процесс с другими службами;
- **MachineName** — имя компьютера, на котором настроена данная служба (выводится только при использовании параметра –ComputerName).

Совет При настройке служб важно знать, работает ли служба в отдельном либо в общем с другими службами контексте. Первые помечены идентификатором WIN32OWNPROCESS, а вторые — WIN32SHAREPROCESS.

По умолчанию Get-Service просматривает все службы, независимо от их состояния. Свойство Status позволяет отобрать службы в определенном состоянии, например Stopped или Paused. Рассмотрим следующие примеры:

```
get-service | where-object {$_.status -eq «Running»}
get-service | where-object {$_.status -eq «Stopped»}
```

Первая команда выводит все работающие службы, а вторая — все остановленные.

Запуск, остановка и приостановка служб

Администраторам часто приходится запускать, останавливать и приостанавливать службы Windows. В консоли PowerShell, открытой с административными правами, это делается с использованием соответствующих командлетов или методов класса Win32_Service. Вот примеры:

Запуск службы:

```
start-service Служба
start-service -displayname Отображаемое_имя
get-service Служба | start-service
```

Приостановка службы:

```
suspend-service Служба
suspend-service -displayname Отображаемое_имя
get-service Служба | suspend-service
```

Возобновление приостановленной службы:

```
resume-service Служба
resume-service -displayname Отображаемое_имя
get-service Служба | resume-service
```

Остановка службы:

```
stop-service Служба
stop-service -displayname Отображаемое_имя
get-service Служба | stop-service
```

В этих примерах *Служба* — краткое, а *Отображаемое_имя* — понятное имя службы, например:

```
stop-service -displayname «DNS Client»
```

Хотя командлеты Start-Service, Suspend-Service, Resume-Service и Stop-Service не поддерживают параметр –ComputerName, их можно использовать для управления службами на удаленных компьютерах следующим образом:

```
get-service dnscache -computername engpc18 | stop-service
get-service dnscache -computername engpc18 | start-service

invoke-command -computername engpc18 -scriptblock {get-service dnscache |
stop-service }
invoke-command -computername engpc18 -scriptblock {get-service dnscache |
start-service }
```

Здесь Get-Service возвращает объект, представляющий службу на удаленном компьютере, который передается командлетам Start-Service, Suspend-Service, Resume-Service и Stop-Service. Учтите, что эти команды оповещают только об ошибках, но ничего не говорят о том, что заданная служба уже запущена, остановлена и т.д.

Прежде чем пытаться остановить или приостановить службу, необходимо проверить, можно ли это сделать. Для этого получить объект службы и проверьте его свойства CanPauseAndContinue и CanStop, например так:

```
$sname = read-host «Enter service name to stop»
$cname = read-host «Enter computer to work with»

$s = get-service $sname -computername $cname
if ($s.CanStop -eq $True) { $s | stop-service }
```

```
Enter service name to stop : dnscache
Enter computer to work with : engpc85
```

В этом примере имена службы и компьютера запрашиваются у пользователя, после чего команда получает объект соответствующей службы и, если служба допускает остановку, останавливает ее. Возможности этого примера легко расширить:

```
$cname = read-host «Enter computer to work with»
$sname = read-host «Enter service name to work with»

$s = get-service $sname -computername $cname
write-host «Service is:» $s.status -foregroundcolor green

$action = read-host «Specify action [Start|Stop|Pause|Resume]»

switch ($action) {

«start» { $s | start-service }
«stop» { if ($s.CanStop -eq $True) { $s | stop-service } }
«pause» { if ($s.CanPauseAndContinue -eq $True) { $s | pause-service } }
«resume» { $s | resume-service }

}
```

```
$su = get-service $sname -computername $cname
write-host «Service is:» $su.status -foregroundcolor green
```

```
Enter computer to work with: techpc12
Enter service name to work with: dnscache
Service is: Running
Specify action [Start|Stop|Pause|Resume]: stop
Service is:  Stopped
```

В этом примере имена службы и компьютера вводятся пользователем. Далее код получает соответствующий объект службы, выводит ее текущее состояние и запрашивает действие, которое требуется выполнить. После проверки свойств CanStop и CanPauseAndContinue над службой выполняется заданное действие и выводится ее обновленное состояние.

Совет Нужна возможность управления слу жбами из любого сеанса PowerShell? Вставьте показанный выше код в функцию и добавьте ее в свой профиль, после этого вам останется лишь вызывать эту функцию. Функция может быть такой:

```
function ms {
#вставьте сюда код
}
```

Теперь после загрузки профиля введите **ms** в командной строке PowerShell, и вы получите возможность управлять службами на любом из компьютеров сети вашей организации.

Настройка запуска служб

Службы Windows могут запускаться вручную либо автоматически, их также можно полностью отключать. Режим запуска служб настраивают командой такого вида:

```
set-service Служба -StartupType Type [-ComputerName Компьютеры]
```

или такого:

```
set-service -displayname Имя -StartupType Тип [-ComputerName Компьютеры]
```

где *Имя* — имя службы для настройки, *Тип* — новый режим запуска, а *Компьютеры* — имена нужных компьютеров. Допустимы следующие типы запуска:

- **Automatic** — служба запускается при загрузке системы. Если требуется запуск с задержкой, службе автоматически назначается режим Automatic (Delayed Start);
- **Manual** — служба запускается вручную через диспетчер управления службами, когда какой-либо процесс вызывает эту службу;
- **Disable** — служба отключена и не будет запущена при следующем запуске компьютера. Если служба уже запущена, она будет работать до выключения компьютера; при необходимости ее можно остановить вручную.

Чтобы настроить службу для автоматического запуска, введите, например

```
set-service dnscache -startuptype automatic
```

или

```
set-service dnscache -startuptype automatic -computername techpc85
```

Чтобы не вводить имена компьютеров вручную (через запятую), запишите их в текстовый файл (каждое имя на отдельной строке), после чего этот файл можно будет использовать как показано ниже:

```
set-service dnscache -startuptype automatic -computername (get-content
c:\data\clist.txt)
```

В этом примере список удаленных компьютеров загружается из файла CList.txt, расположенного в каталоге C:\Data.

Командлет Set-Service также позволяет запускать, останавливать и приостанавливать службы. Включая службу, имеет смысл сразу запустить ее, например, так:

```
set-service w3svc -startuptype automatic -status running
```

А при отключении службы ее можно сразу же остановить, например:

```
set-service w3svc -startuptype disabled -status stopped
```

Управление входом служб в систему и режимами восстановления

Время от времени администраторам требуется настраивать параметры входа в систему и восстановления служб. Проще всего сделать это с помощью утилиты Services Configuration (SC), поддерживающей «подкоманды» для работы со службами Windows. Исполняемый файл этой утилиты — sc.exe. В командной строке Windows ее можно запускать командой **sc**, но в PowerShell псевдоним *sc* по умолчанию закреплен за командлетом Set-Content, поэтому в командной строке PowerShell эту утилиту следует запускать командой **sc.exe**.

Настройка входа в систему для служб

Команда config утилиты SC позволяет настраивать службы Windows для входа в систему под заданной учетной записью. Так, чтобы настроить службу для входа под учетной записью LocalSystem, введите:

```
sc.exe Компьютер config Служба obj= LocalSystem
```

где *Компьютер* — UNC-имя компьютера, а *Служба* — имя службы, которая должна входить под учетной записью LocalSystem. Если у службы есть пользовательский интерфейс, добавьте к команде флаги **type= interact type= own**, например так:

```
sc.exe config vss obj= LocalSystem type= interact type= own
```

или так:

```
sc.exe \\techpc85 config vss obj= LocalSystem type= interact type= own
```

Примечание В показанных выше командах пробелы после знака равенства (=) обязательны. Заметьте также, что SC выводит только сообщения SUCCESS и FAILURE, и не показывает, что заданная служба уже настроена как требуется.

Флаг *type= interact* разрешает службе взаимодействие с Рабочим столом Windows, а флаг *type= own* запускает службу в отдельном процессе. Чтобы запустить службу, использующую общий процесс с другими службами, используйте флаг *type= share*:

```
sc.exe config dnscache obj= LocalSystem type= interact type= share
```

Совет Если вы не знаете, как запускается слу жба (в отдельном процессе или в общем), то сначала выясните это с помощью командлета Get-Service.

Службы могут входить и под именованными пользовательскими учетными записями, например:

```
sc.exe config Служба obj= [Домен\]Пользователь password= Пароль
```

где *Домен* — имя домена, в котором находится учетная запись пользователя (необязательный параметры), *Пользователь* — имя учетной записи пользователя, чьи полномочия требуется дать службе, а *Пароль* — пароль этой учетной записи. Рассмотрим следующий пример:

```
sc.exe config vss obj= adatum\backers password= TenMen55!
```

Эта команда настраивает службу Microsoft Visual SourceSafe (VSS) для входа под учетной записью пользователя Backers из домена Adatum и выводит сообщение SUCCESS либо FAILED. Причиной неудачного завершения команды может быть недопустимая или отсутствующая учетная запись либо неверный пароль.

Примечание Службу, у же нас троенную для взаимодейс твия с Рабо чим с толом и работы под учетной записью LocalSystem, можно настроить для работы под доменной учетной записью только с флагом *type= own*:

```
sc config Служба obj= [Домен\]Пользователь password= Пароль type= own.
```

Утилита SC предназначена для работы с отдельными компьютерами, но это можно изменить, используя возможности PowerShell:

```
$c = «\\techpc85»
sc.exe $c query dnscache
```

В этом примере имя компьютера не указано явно, а передается в переменной.

Имена компьютеров можно загружать и из файла, как показано ниже:

```
$computers = (get-content c:\data\unclist.txt)
foreach ($c in $computers) { sc.exe $c query dnscache }
```

Здесь список компьютеров загружается из файла UncList.txt, расположенного в каталоге C:\Data, после чего на каждом из компьютеров выполняется запрос службы.

Совет Поманите, что SC принимает UNC-имена к омпьютеров, то ес ть в тек стовом файле с именами следует писать, скажем, **\\techpc85** вместо **techpc85**.

Настройка восстановления служб

Команды Qfailure и Failure, поддерживаемые SC, позволяют просматривать и настраивать действия, выполняемые при сбоях служб. Например, можно настроить службу так, чтобы SCM перезапускал ее после сбоя либо запускал другое приложение, устраняющее причины сбоя.

Возможны разные настройки для первой, второй и всех последующих попыток восстановления. При каждом сбое текущее значение счетчика сбоев увеличивается на единицу; поддерживается параметр, задающий период до сброса этого счетчика. Например, можно сбрасывать счетчик сбоев службы по истечении 24 ч с момента последнего сбоя.

Пред настройкой параметров восстановления рекомендуется проверить текущие настройки с помощью команды SC Qfailure. Вот синтаксис этой команды:

```
sc.exe Компьютер qfailure Служба
```

где *Компьютер* — UNC-имя компьютера, а *Служба* — имя службы, например:

```
sc.exe qfailure vss

sc.exe \\techpc85 qfailure vss
```

Вывод команды содержит список действий, предпринимаемых при сбое службы, в порядке их очередности. В следующем примере служба VSS настроена для перезапуска при первых двух сбоях и перезапуске компьютера после третьего сбоя:

```
[SC] QueryServiceConfig2 SUCCESS

SERVICE_NAME: vss
       RESET_PERIOD (in seconds)  : 86400
       REBOOT_MESSAGE             :
       COMMAND_LINE               :
        FAILURE_ACTIONS           : RESTART -- Delay = 1 milliseconds.
                                    RESTART -- Delay = 1 milliseconds.
                                    REBOOT -- Delay = 1000 milliseconds.
```

мечание Параметры восс тановления для нек оторых критически важных слу жб Windows настраиваются автоматически во время установки. Как правило, после сбоя эти службы перезапускаются, возможен также запуск некоторых программ и сценариев.

Вот общий синтаксис команд для настройки восстановления служб после сбоя:

```
sc.exe failure Служба reset= Период actions= Действия
```

где *Служба* — имя настраиваемой службы; *Период* — время (в секундах) до сброса счетчика; *Действия* — действия в случае сбоя и время их задержки (в миллисекундах). Допустимые значения этого параметра включают:

- **Take No Action** (пустая строка, «») — ОС не пытается восстановить службу после данного сбоя (но может делать это после предыдущего или последующего сбоев);
- **Restart The Service** — ОС останавливает службу и снова запускает ее после небольшой паузы;
- **Run A Program** — позволяет запустить программу или сценарий после сбоя службы. Выбрав это значение, необходимо указать полный путь к программе, которую нужно запустить, а также необходимые ей параметры командной строки;
- **Reboot The Computer** — перезагружает компьютер по истечении заданного периода после сбоя службы.

мечание При настройке восстановления критически важных слу жб рекомендуется при первом и втором сбоях перезапускать слу жбу, а при третьем — перезагружать сервер.

Работая с командой SC Failure, учтите следующее:

- **период до сброса задают в секундах**, обычно как число, кратное длительности часа (3600 с) или суток (86 400 с). Например, чтобы сбросить счетчик через два часа после сбоя, укажите значение 7200;
- **действия по восстановлению службы должны выполняться после задержки** (она задается в миллисекундах). Задержки перезапуска службы обычно короче — 1 мс, 1 с (1000 мс) или 5 с (5000 мс), а перезагрузки компьютера — длиннее, скажем 15 или 30 с (15000 или 30000 мс).
- **действия и задержки вводятся как одна строка**, значения разделяются слешем (/), например restart/1000/restart/1000/reboot/15000. В этом примере первые два сбоя вызывают перезапуск службы с 1-с задержкой, а третий — перезагрузку компьютера после 15-с задержки.

Рассмотрим следующие примеры:

```
sc.exe failure w3svc reset= 86400 actions= restart/1/restart/1/reboot/30000
```

Здесь первый и второй сбои практически сразу вызывают перезагрузку компьютера, а третий — после 30-секундной задержки; счетчик сбоев сбрасывается через сутки (86 400 с). Удаленные компьютеры также задают по UNC-именам и IP-адресам, как показано в предыдущих примерах.

Если выбрано действие Run, необходимо указать программу или команду для запуска (включая полный путь и необходимые параметры) как значение параметра Command=. Не забудьте взять команду в двойные кавычки, как показано ниже:

```
sc.exe failure w3svc reset= 86400 actions= restart/1/restart/1/run/30000
command= «c:\restart_w3svc.exe 15»
```

Подробнее об управлении службами

Хотя возможностей командлета Get-Service хватает для решения множества административных задач, он все же не дает информации, необходимой для настройки служб. А для этого необходимо знать, как минимум, текущий тип запуска службы и имя, учетной записи, под которой она запускается. Допустимы следующие типы запуска:

- **Automatic (Автоматически)** — SCM автоматически запускает службу во время загрузки системы;
- **Manual (Вручную)** — SCM запускает службу, как только процесс вызовет ее с помощью метода StartService;
- **Disabled (Отключена)** — служба отключена и не запускается.

Большинство служб работает под одной из следующих учетных записей:

- **NT Authority\LocalSystem** — «виртуальная» учетная запись для запуска системных процессов и задач системного уровня; входит в группу Administrators (Администраторы) и обладает полным доступом к компьютеру. соответственно, служба, работающая под этой учетной записью, также получает полный доступ к компьютеру. Многие службы работают под учетной записью LocalSystem, у некоторых из них есть привилегии для взаимодействия с Рабочим столом. Службы, нуждающиеся в альтернативном наборе привилегий, работают под учетными записями LocalService и NetworkService;
- **NT Authority\LocalService** — «виртуальная» учетная запись с ограниченными привилегиями, обладающая доступом только к локальной системе. Входит в группу Users (Пользователи), обладает теми же правами, что и NetworkService, но по отношению только к локальному компьютеру;
- **NT Authority\NetworkService** — «виртуальная» учетная запись для служб, требующих дополнительные привилегии и возможности и права на вход в локальную систему и в сеть. Входит в группу Users и обладает меньшими полномочиями, чем LocalSystem (но большими, чем LocalService). В частности, службам, работающим под этой учетной записью, разрешено взаимодействие через сеть с использованием учетных данных компьютера.

Узнать тип запуска, имя учетной записи и получить другие сведения о конфигурации служб можно средствами WMI) и с помощью класса Win32_Service. Этот класс позволяет создавать объекты, представляющие службы в WMI.

Чтобы получить полный список служб, настроенных на компьютере, введите **get-wmiobject -class win32_service**:

```
get-wmiobject -class win32_service |
format-table name, startmode, state, status
```

```
name                    startmode       state         status
----                    ---------       -----         ------
AppMgmt                 Manual        Stopped         OK
AudioEndpointBuilder    Auto          Running         OK
Audiosrv                Auto          Running         OK
BFE                     Auto          Running         OK
BITS                    Auto          Running         OK
Bonjour Service         Auto          Running         OK
Browser                 Auto          Running         OK
CertPropSvc             Manual        Stopped         OK
```

Чтобы изучить отдельную службу или группу служб, соответствующих заданному критерию, добавьте к команде параметр –Filter. Следующая команда проверяет конфигурацию службы Browser:

```
get-wmiobject -class win32_service -filter «name='browser'»
```

```
ExitCode  : 0
Name      : Browser
ProcessId : 1136
StartMode : Auto
State     : Running
Status    : OK
```

Отформатировав вывод как список, вы получите дополнительную информацию, включая значение свойств PathName (путь к исполняемому файлу службы и его параметры) и StartName (учетная запись, под которой работает служба). Вот соответствующий пример для службы DNS Client:

```
$s = get-wmiobject -class win32_service -filter «name='dnscache'»
$s | format-list *
```

```
Name                    : Dnscache
Status                  : OK
ExitCode                : 0
DesktopInteract         : False
ErrorControl            : Normal
PathName              : C:\Windows\system32\svchost.exe -k NetworkService
ServiceType             : Share Process
StartMode               : Manual
AcceptPause             : False
AcceptStop              : True
Caption                 : DNS Client
```

```
CheckPoint              : 0
CreationClassName       : Win32_Service
Description             : Caches Domain Name System (DNS) names.
DisplayName             : DNS Client
InstallDate             :
ProcessId               : 1536
ServiceSpecificExitCode : 0
Started                 : True
StartName               : NT AUTHORITY\NetworkService
State                   : Running
SystemCreationClassName : Win32_ComputerSystem
SystemName              : TechPC85
```

Командлет Get-WmiObject поддерживает параметр –ComputerName, который позволяет задавать удаленные компьютеры для работы:

```
get-wmiobject -class win32_service -computername fileserver86,
dcserver22, printserver31

get-wmiobject -class win32_service -computername (get-content
c:\data\clist.txt)
```

Примечание Если вы работаете с группой компьютеров, то сможете узнать, на каком именно компьютере настроена данная служба, проверив ее свойство SystemName.

Со свойствам объектов Win32_Service работают почти так же, как и со свойствами объектов Service. Например, чтобы получить список служб, настроенных для автоматического запуска, введите следующую команду:

```
get-wmiobject -class win32_service -filter «startmode='auto'» |
format-table name, startmode, state, status
```

Для просмотра всех работающих служб введите:

```
get-wmiobject -class win32_service -filter «state='running'» |
format-table name, startmode, state, status
```

Класс Win32_Service поддерживает следующие методы для управления службами, настройка которых разрешена пользователям:

- **Change()** — изменяет конфигурацию службы. Принимает следующие параметры (в порядке передачи): DisplayName, PathName, ServiceTypeByte, ErrorControlByte, StartMode, DesktopInteractBoolean, StartName, StartPassword, LoadOrderGroup, LoadOrderGroupDependenciesArray, and ServiceDependenciesArray.

Внимание! Настраивать службы средствами PowerShell следует очень осторожно и только после тщательного планирования, иначе можно внести изменения, чреватые нестабильной работой компьютера. Перед внесением изменений в конфигурацию служб непременно создайте точку восстановления системы (подробнее об этом — в главе 14).

- **ChangeStartMode()** — Изменяет режим запуска службы. Принимает единственный параметр — тип запуска (Manual, Automatic или Disabled).

мечание Некоторые службы по умолчанию настроены для автоматического запуска с задержкой. Установка типа запуска Automatic средствами объекта Win32_Service в действительности настраивает службу для автоматического запуска с задержкой. Учтите также, что отключение службы не останавливает ее, а лишь запрещает ее запуск при следующей загрузке компьютера. Чтобы гарантированно остановить службу, остановите ее явно после отключения.

мание! Перед изменением типа запуска службы следует выяснить, какие службы зависят от нее, чтобы случайно не нарушить работу других служб.

- **Delete()** — удаляет службу (если ее состояние допускает удаление). Не спешите удалять службы: возможно, достаточно просто отключить их; отключенные службы не работают, но их легко включить снова, если возникнет такая необходимость, тогда как удаленную службу придется переустанавливать.

мание! Удалять службы средствами PowerShell следует с величайшей осторожностью: PowerShell не предупреждает о внесении изменений, нарушающих нормальную работу компьютера.

- **InterrogateService()** — подключается к службе с использованием диспетчера управления службами (SCM). Если этот метод возвращает ноль, подключение и опрос службы закончились успешно, в противном случае подключиться к службе не удалось. Вызов этого метода для остановленных и приостановленных служб всегда завершается ошибкой.
- **PauseService()** — приостанавливает службу для диагностики и устранения неполадок (только если ее свойство AcceptPause равно True и служба находится в состоянии, допускающем приостановку).
- **ResumeService()** — возобновляет работу приостановленной службы.
- **StopService()** — останавливает службу для диагностики и устранения неполадок (только если ее свойство AcceptStop равно True и служба находится в состоянии, допускающем остановку).
- **StartService()** — запускает остановленную службу, включая службы, настроенные для запуска вручную.

Вышеперечисленные методы используются для управления службами через консоль PowerShell, открытую с администраторскими полномочиями. Так, для настройки типа запуска служит метод ChangeStartMode(), принимающий имя типа запуска:

```
$Объект_службы.ChangeStartMode(Тип_запуска)
```

где *$Объект_службы* — ссылка на объект Win32_Service, а *Тип_запуска* — требуемый тип запуска. Вот пример:

```
$s = get-wmiobject -class win32_service -filter «name='dnscache'»
$s.changestartmode(«automatic»)
```

```
__GENUS          : 2
__CLASS          : __PARAMETERS
__SUPERCLASS     :
__DYNASTY        : __PARAMETERS
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
ReturnValue      : 0
```

Здесь службе назначается тип запуска Automatic. Обратите внимание на поле ReturnValue: если оно равно 0, вызов завершен успешно, любое другое значение свидетельствует об ошибке. Ошибки обычно возникают, если у консоли PowerShell нет администраторских полномочий, указана неверная служба либо служба находится в состоянии, запрещающем ее настройку. Учтите, что ошибки при конфигурировании могут нарушить нормальную работу компьютера, поэтому любые изменения должны тщательно обдумываться перед внесением.

Примечание. Вышеописанные команды возвращают только SUCCESS либо FAILURE и ничего не говорят, если служба уже запущена, остановлена или настроена с заданными параметрами.

Вызовы методов объектов WMI обсуждались выше, для этого служит командлет Invoke-WMIMethod. При использовании этого командлета для вызова метода достаточно одной команды (вместо двух, когда сначала создается ссылка на объект WMI, а затем вызывается метод этого объекта. Например, вместо:

```
$s = get-wmiobject -class win32_service -filter «name='dnscache'»
$s.stopservice()
```

можно записать:

```
invoke-wmimethod -path «win32_service.name='dnscache'» -name stopservice
```

Вот как выглядит синтаксис Invoke-WmiMethod:

```
Invoke-WmiMethod [-ComputerName [Компьютеры]] [-Credential
[Учетные_данные]] [-Name] [Метод] [-ThrottleLimit [Предел]]
[-AsJob]

Invoke-WmiMethod [-InputObject [WMIObject]] [-Name] [Метод]
[-ThrottleLimit [Предел]] [-AsJob]

Invoke-WmiMethod [-Namespace [WMINamespace]] -Path [WMIPath]
[-ArgumentList [Объекты] [-Name] [Методы] [-ThrottleLimit
[Предел]] [-AsJob]
```

```
Invoke-WmiMethod [-EnableAllPrivileges] [-Authority [Домен]] [-Name]
[Метод] [-ThrottleLimit [Предел]] [-AsJob]

Invoke-WmiMethod [-Locale [Культура]] [-Name] [Метод] [-ThrottleLimit
[Предел]] [-AsJob]
```

Вот примеры управления службами с помощью методов класса Win32_Service и командлета Invoke-WmiMethod:

Запуск службы:

```
invoke-wmimethod -path «win32_service.name='Имя_службы'»
-name startservice

invoke-wmimethod -path «win32_service.displayname='Отображаемое_имя'»
-name startservice

get-wmiobject -class win32_service -filter «name='Имя_службы'» |
invoke-wmimethod -name startservice
```

Приостановка службы:

```
invoke-wmimethod -path «win32_service.name='Имя_службы'»
-name pauseservice

invoke-wmimethod -path «win32_service.displayname='Отображаемое_имя'»
-name pauseservice

get-wmiobject -class win32_service -filter «name='Имя_службы'» |
invoke-wmimethod -name pauseservice
```

Возобновление приостановленной службы:

```
invoke-wmimethod -path «win32_service.name='Отображаемое_имя'»
-name resumeservice

invoke-wmimethod -path «win32_service.displayname='Отображаемое_имя'»
-name resumeservice

get-wmiobject -class win32_service -filter «name='Имя_службы'» |
invoke-wmimethod -name resumeservice
```

Остановка службы:

```
invoke-wmimethod -path «win32_service.name='Имя_службы'»
-name stopservice

invoke-wmimethod -path «win32_service.displayname='Отображаемое_имя'»
-name stopservice

get-wmiobject -class win32_service -filter «name='Имя_службы'» |
invoke-wmimethod -name stopservice.
```

Прежде чем останавливать или приостанавливать службу, выясните, можно ли это сделать, проверив свойства AcceptPause и AcceptStop объекта Win32_Service.

Показанные выше методы с Get-WmiObject годятся для работы с единственным объектом Win32_Service, но не работают, если Get-WmiObject возвращает несколько объектов Win32_Service. Дело в том, что возвращаемые объекты хранятся в массиве, и требуется указать элемент массива, содержащий нужный объект. Вот пример:

```
$servs = get-wmiobject -class win32_service |
where-object {$_.name -match «client»}

foreach ($s in $servs) { $s.changestartmode(«automatic») }
```

```
ReturnValue      : 0

ReturnValue      : 0

ReturnValue      : 0
```

Этот код возвращает три объекта Win32_Service, представляющий службы, настроенные для автоматического запуска.

Управление компьютерами

У компьютеров имеются атрибуты, такие как имена и членство в группах, которыми могут управлять администраторы. Учетные записи компьютеров разрешается в любые контейнеры и подразделения (organizational unit, OU) Active Directory. Однако лучше использовать контейнеры Computers, Domain Controllers и созданные вами OU. Стандартное средство Windows для работы с учетными записями компьютеров — консоль Active Directory Users And Computers (Пользователи и компьютеры Active Directory). PowerShell также поддерживает множество команд для этой цели (только в Windows Vista, Windows Server 2008 и более высоких версиях).

Команды для управления компьютерами

Для управления компьютерами в PowerShell используются следующие команды:

- **Add-Computer** — добавляет компьютер к домену или рабочей группе. Компьютеры задают по NetBIOS-именам, IP-адресам и полным доменным именам. Для подключения к домену необходимо указать имя домена; если в домене нет учетной записи для подключаемого компьютера, она создается при подключения. Для завершения операции требуется перезагрузка. Для получения подробных результатов используйте параметры –Verbose и –PassThru.

```
Add-Computer [-OUPath Путь_AD] [-Server Домен\Имя_компьютера]
[[-ComputerName] Компьютеры] [-DomainName] Домен
[-Unsecure] [-PassThru] [-Reboot] [[-Credential] Объект_удостоверений]

Add-Computer [[-ComputerName] Компьютеры] [-WorkGroupName] Имя
[-PassThru] [-Reboot] [[-Credential] Объект удостоверений]
```

- **Remove-Computer** — удаляет компьютеры из домена или рабочей группы; при удалении компьютера из домена отключает его учетную запись. Для завершения операции требуется перезагрузка. Для отключения от домена необходимо ввести учетные данные уполномоченного пользователя.

```
Remove-Computer [[-ComputerName] Компьютеры]
[-PassThru] [-Reboot] [[-Credential] Объект_удостоверений]
```

- **Rename-Computer** — переименовывает компьютеры в составе рабочих групп и доменов (в этом случае переименовывается и учетная запись компьютера). Переименовывать контроллеры доменов запрещено. Для переименования удаленных компьютеров необходимо ввести учетные данные уполномоченного пользователя.

```
Rename-Computer [[-ComputerName] Имя_компьютера] [-NewComputerName]
Новое_имя [-Credential Объект_удостоверений] [-Reboot]
```

- **Restart-Computer** — перезагружает операционную систему, с параметром –Force делает это немедленно.

```
Restart-Computer [[-ComputerName] Компьютеры] [-AsJob]
[-Authentication Тип_проверки] [[-Credential] Объект_удоствоерений]
[-Force] [-Impersonation Тип_олицетворения] [-ThrottleLimit Предел]
```

- **Stop-Computer** — завершает работу компьютеров. Параметр –AsJob запускает команду в фоновом режиме (только если компьютер настроен для удаленного управления).

```
Stop-Computer [[-ComputerName] Компьютеры] [-AsJob]
[-Authentication Тип_проверки] [[-Credential] Объект_удостоверений]
[-Force] [-Impersonation Тип_олицетворения] [-ThrottleLimit Предел]
```

- **Test-Connection** — отправляет ICMP-запрос эхо-отклика (ping) компьютерам и возвращает отклик. Если ICMP-пакеты не блокируются брандмауэром, эта команда позволяет проверять связь с компьютерами в IP-сетях. Возможно явно указать опрашивающий и отвечающий компьютеры, период ожидания и число запросов.

```
Test-Connection [-Count Число] [-Delay Интервал]
[-TimeToLive Время_жизни] [[-Source] Опрашщивающие_ПК] [-Destination]
Отвечающие_ПК
```

```
[-AsJob] [-Authentication Тип_проверки] [-BufferSize Size] [-Credential
Объект_удостоверений] [-Impersonation Тип_олицетворения] [-ThrottleLimit
Предел]
```

Рекомендуется работать с этими командами в консоли PowerShell, открытой с администраторскими полномочиями. Впрочем, и в этом случае может потребоваться ввод учетных данных, что делается следующим образом:

```
$cred = get-credential
add-computer -domainname cpandl-credential $cred
```

При вызове Get-Credential консоль PowerShell запрашивает имя и пароль пользователя, сохраняя их в переменной $cred. В дальнейшем эти учетные данные будут использованы для аутентификации.

При использовании вышеперечисленных команд учтите следующее:

- параметр –Authentication задает уровень проверки подлинности для WMI-подключения. Значение по умолчанию — Packet, другие допустимые значения включают: Unchanged (уровень, использовавшийся в прежней команде), Default (аутентификация средствами Windows), None (без аутентификации COM), Connect (аутентификация COM на уровне подключения), Call (аутентификация COM на уровне вызова), Packet (аутентификация COM на уровне пакетов), PacketIntegrity (аутентификация COM на уровне защиты целостности пакетов) и PacketPrivacy (аутентификация COM на уровне конфиденциальности пакетов);
- параметр –Impersonation задает уровень олицетворения для WMI-подключения. Значение по умолчанию — Impersonate, другие допустимые значения включают: Default (уровень по умолчанию), Anonymous (скрывает учетные данные вызывающего субъекта), Identify (разрешает запрашивать учетные данные вызывающего субъекта) и Impersonate (разрешает объектам использовать удостоверения вызывающего субъекта).

Как видите, по умолчанию используется аутентификация COM на уровне пакетов и разрешено олицетворение вызывающего субъекта. Эти настройки подходят в большинстве случаев, но иногда требуется использовать аутентификацию Windows вместо аутентификации COM. В этом случае задайте для параметра –Authentication значение Default.

Командлет Test-Connection работает так же, как команда Ping. Он позволяет проверить связь с компьютером, заданным по имени или IP-адресу. Например, чтобы проверить связь с системой, которой назначен адрес IPv4 192.168.10.55, введите:

```
test-connection 192.168.10.55
```

А для проверки связи с адресом IPv6 FEC0::02BC:FF:BECB:FE4F:961D введите

```
test-connection FEC0::02BC:FF:BECB:FE4F:961D
```

Если связь имеется, команда вернет соответствующий отклик, в противном случае вы получите сообщение о тайм-ауте либо ошибке «Unable to Connect». Причина может быть в том, что компьютер отключен от сети, выключен либо находится за брандмауэром, блокирующим ICMP-пакеты.

Переименование компьютеров

Командлет Rename-Computer позволяет без труда переименовывать рабочие станции и рядовые серверы домена. При подключении к домену учетные записи рабочих станций и рядовых серверов также переименовываются. Однако нельзя использовать Rename-Computer для переименования контроллеров домена, серверов, на которых работают службы сертификации и другие службы, запрещающие изменять имя компьютера.

Общий синтаксис команды для переименования выглядит так:

```
rename-computer -ComputerName Имя_компьютера -NewComputerName Новое_имя
-reboot
```

где *Имя_компьютера* — текущее имя компьютера, а *Новое_имя* — его новое имя. Чтобы переименовать локальный компьютер, опустите параметр –ComputerName:

```
rename-computer -NewComputerName TechPC12 -reboot
```

В этом примере компьютер получает имя TechPC12. Поскольку для завершения переименования требуется перезагрузка, можно автоматически перезагрузить компьютер после смены имени, указав параметр -Reboot. При необходимости можно ввода учетных данных можно использовать следующий прием:

```
$cred = get-credential
rename-computer -NewComputerName TechPC12 -credential $cred -reboot
```

Присоединение компьютеров к домену

Любой пользователь, прошедший аутентификацию, может присоединить компьютер к домену с помощью командлета Add-Computer, при этом для компьютера будет создана учетная запись в домене, если это не сделано раньше. При подключении компьютера к домену между ним и доменом устанавливаются доверительные отношения, компьютер получает учетные данные, соответствующие его учетной записи в Active Directory, и добавляется в соответствующую группу Active Directory, обычно это группа Domain Computers (компьютеры домена). Если же подключаемый компьютер должен стать контроллером домена, он будет добавлен в группу Domain Controllers (Контроллеры домена).

Совет Перед присоединением компьютера к домену проверьте его сетевые настройки. Если они неверны, то сначала необходимо исправить их, и только потом пытаться присоединить компьютер к домену. Кроме того, если учетная запись для данного компьютера уже существует, только уполномоченный пользователь (и обладающий правами локального администратора на этом компьютере) либо администратор домена может присоединить компьютер.

Чтобы присоединить компьютер к домену, создать при этом его запись, войдите на компьютер и вызовите Add-Computer как показано ниже:

```
add-computer -DomainName Имя_домена -reboot
```

где *Имя_домен* — имя домена Active Directory, к которому требуется присоединить компьютер. Для присоединения требуется перезагрузка, поэтому имеет смысл указать параметр –Reboot, чтобы автоматически перезагрузить компьютер. Если не указано OU, компьютер попадет в подразделение по умолчанию. Рассмотрим пример:

```
$cred = get-credential
add-computer -domainname cpandl -credential $cred -reboot
```

Здесь локальный компьютер присоединяется к домену cpandl.com, где для него создается учетная запись в OU по умолчанию (Computers). Если имя этого компьютера — TechPC85, полный путь к его объекту примет следующий вид: CN=TechPC85,CN=Computers,DC=cpandl,DC=com.

Совет Для получения подробных результатов добавьте параметры –PassThru и –Verbose. Параметр –Server позволяет явно указать контроллер домена (в формате *Имя_домена\Имя_компьютера*, например CPANDL\DcServer14), к которому требуется присоединить компьютер. Если этот параметр не задан, используется любой из доступных контроллеров домена.

Параметр –OUPath позволяет задавать полное имя OU, к которому требуется подключить компьютер, например:

```
$cred = get-credential
add-computer -domainname cpandl -outpath ou=engineering,dc=cpandl,dc=com
-credential $cred -reboot
```

Здесь компьютер добавляется в подразделение Engineering OU в домене cpandl.com. Если имя этого компьютера — TechPC85, полный путь к его объекту примет вид CN=TechPC85,OU=Engineering,DC=cpandl,DC=com.

Для присоединения к домену удаленных компьютеров используется следующий синтаксис Add-Computer:

```
add-computer -DomainName Имя_домена -computername Компьютеры -reboot
```

где *Имя_доена* — имя домена Active Directory, к которому требуется присоединить компьютер, а *Компьютеры* — список имен присоединяемых компьютеров, разделенных запятыми. И в этом случае команда автоматически создает учетные записи для компьютеров (если это не было сделано раньше) и помещает их в OU, заданное параметром –OUPath (он принимает полное имя OU).

Рассмотрим следующий пример:

```
$cred = get-credential
add-computer -domainname cpandl -computername EngPC14, EngPC17 -outpath
ou=engineering,dc=cpandl,dc=com -credential $cred -reboot
```

Эти команды присоединяют компьютеры EngPC14 и EngPC15 к домену cpandl.com и создают для них учетные записи в OU Engineering.

Список присоединяемых компьютеров можно загружать и из файла:

```
add-computer -domainname cpandl -computername (get-content
c:\data\clist.txt)
```

В этом примере имена компьютеров, присоединяемых к домену cpandl.com, загружаются из файла C:\Data\CList.txt. Для переименования локального компьютера вместо именно введите «**.**» или **localhost**.

Присоединение компьютеров к рабочей группе

Командлет Add-Computer используется и для присоединения компьютеров к рабочим группам. Синтаксис соответствующей команды выглядит так:

```
add-computer -WorkgroupName Имя_группы -reboot
```

где *Имя_группы* — имя рабочей группы, к которой следует присоединить компьютер. Для завершения операции требуется перезагрузка, поэтому имеет смысл использовать параметр –Reboot.

Рассмотрим следующий пример:

```
$cred = get-credential
add-computer -workgroupname testing -credential $cred -reboot
```

Здесь локальный компьютер присоединяется к рабочей группе Testing. Для получения подробного вывода добавьте к команде параметры –PassThru и –Verbose.

Add-Computer можно использовать и для присоединения удаленных компьютеров, синтаксис соответствующей команды выглядит так:

```
add-computer -WorkgroupName Имя_группы -computername Компьютеры -reboot
```

где *Имя_группы* — имя рабочей группы, к которой присоединяют компьютеры, а *Компьютеры* — список присоединяемых компьютеров с запятыми-разделителями.

Рассмотрим следующий пример:

```
$cred = get-credential
add-computer -workgroupname testing -computername TestPC11, TestPC12
-credential $cred -reboot
```

Здесь компьютеры TestPC11 и TestPC12 присоединяют к рабочей группе Testing.

Список присоединяемых компьютеров можно загружать и из файла:

```
add-computer -workgroupname testing -computername (get-content
c:\data\clist.txt)
```

В этом примере список компьютеров загружается из файла C:\Data\CList.txt. Для переименования локального компьютера вместо его имени введите «.» или **localhost**.

Удаление компьютеров из доменов и рабочих групп

Удалять записи компьютеров из доменов и рабочих групп разрешено только уполномоченным пользователям. В результате этой операции доменная учетная запись отключается, доверительные отношения между этим компьютером и доменом разрываются, и компьютер получает идентификатор защиты, соответствующий таковому рабочей группы. В итоге компьютер попадает в рабочую группу по умолчанию (Workgroup).

Для удаления компьютеров из доменов и рабочих групп используется командлет Remove-Computer. Рассмотрим пример:

```
$cred = get-credential
remove-computer -credential $cred -reboot
```

Эти команды удаляют локальный компьютер из группы или домена, в котором он состоял, и переводят его в рабочую группу по умолчанию (Workgroup).

Для вывода из доменов и групп удаленных компьютеров используется команда следующего вида:

```
remove-computer -computername Компьютеры -reboot
```

где *Компьютеры* — список имен компьютеров с разделителями-запятыми.

Рассмотрим следующий пример:

```
$cred = get-credential
remove-computer -computername TestPC11, TestPC12 -credential $cred -reboot
```

Эти команды удаляют компьютеры TestPC11 и TestPC12 из группы или домена, в котором они состояли и переводят их в рабочую группу по умолчанию Workgroup.

Управление перезагрузкой и завершением работы компьютеров

Администраторам часто требуется перезагружать и выключать компьютеры. Есть два способа сделать это. Первый — использование командлетов Restart-Computer и Shutdown-Computer в командной строке PowerShell, данный прием работает как с локальными, так и с удаленными компьютерами. Второй способ — перезапуск и отключение систем по графику. Для этого применяют утилиту Schtasks либо сценарии.

Обычно Windows-компьютеры запускаются и выключаются без проблем, но иногда они все же «зависают» при этом. В таком случае попробуйте выяснить причину сбоя. Вот некоторые из возможных причин:

- система пытается выполнить сценарий запуска или завершения работы, который не завершается или «зависает» (в этом случае система просто ожидает тайм-аута этого сценария);
- сбой возникает из-за неверной инициализации, такие проблемы можно попробовать устранить с помощью утилиты System Configuration (Msconfig). Возможно, для этого придется отключить некоторые компоненты, службы или другие элементы файла инициализации;
- проблему вызывает антивирусная программа, которая, например, пытается проверить гибкий или другой диск во время выключения компьютера. Для устранения этой проблемы следует на время отключить антивирусную программу;
- сбои при включении и выключении компьютеров могут вызывать неполадки звуковой платы. В таком случае следует отключить сбойное устройство и перезагрузить компьютер. Если после этого сбой исчезнет, переустановите или обновите драйверы устройства, а также проверьте, не повреждены ли звуковые файлы, которые воспроизводятся при входе и выходе из Windows;
- причиной зависаний при запуске и завершении работы могут быть сбои сетевой платы. Попробуйте отключить сетевую плату и перезагрузить компьютер. Если это сработает, попробуйте переустановить или обновить драйверы сетевой платы;
- «зависание» могут вызывать и сбойные драйверы видеоплаты. Попробуйте войти на «зависший» компьютер и откатить установку последних видеодрайверов, а если не получится — переустановите драйверы видеоплаты.

Чтобы перезагрузить или выключить компьютер, войдите в его систему и введите, соответственно, **restart-computer** или **stop-computer**. Чтобы выполнить эту операцию немедленно, добавьте к команде параметр –Force.

Для удаленной перезагрузки или выключения компьютеров используются команды следующего вида:

```
restart-computer -computername Компьютеры
```

или

```
stop-computer -computername Компьютеры
```

где *Компьютеры* — список имен компьютеров с разделителями-запятыми. И в этом случае параметр –Force вызывает немедленную перезагрузку или выключение. Возможно, для выполнения этих операций вам придется ввести учетные данные, как в этом примере:

```
$cred = get-credential
stop-computer -computername TestPC11, TestPC12 -credential $cred
```

Эти команды выключают компьютеры TestPC11 и TestPC12 от имени указанной учетной записи.

Список компьютеров для этих команд можно загружать и из файла:

```
$cred = get-credential
restart-computer -computername (get-content c:\data\clist.txt) -credential
$cred -force
```

Эти команды перезагружают компьютеры, имена которых перечислены в файле C:\Data\CList.txt, от имени заданной учетной записи.

Создание точек восстановления системы и работа с ними

Если активна функция восстановления системы, компьютер автоматически создает через заданные интервалы «моментальные снимки» состояния системы, которые называются *точками восстановления* (restore points). Точка восстановления включает параметры Windows, список установленных программ и т.д. Если нормальная работа компьютера нарушилась после изменения его конфигурации, можно воспользоваться точкой восстановления, чтобы вернуть систему в состояние, в котором она была на момент создания точки восстановления. Предположим, что после установки последнего пакета обновлений для Microsoft Office приложения Office перестали запускаться, и отменить установку пакета не удается. В этом случае можно прибегнуть к точке восстановления, созданной до установки пакета обновлений.

Функция System Restore (Восстановление системы) автоматически создает несколько типов точек восстановления:

- **Scheduled** — плановые точки восстановления, генерируемые ОС с регулярными интервалами;
- **Windows Update** — создаются перед установкой обновлений Windows;
- **Application Install** — создаются перед установкой приложений;
- **Application Uninstall** — создаются перед удалением приложений;
- **Device Install** — создаются перед установкой устройств;
- **Device Uninstall** — создаются перед удалением устройств.

Перед операцией, способной нарушить работу компьютера, следует вручную создать точку восстановления системы.

System Restore независимо управляет точками восстановления на отдельных жестких дисках. Наблюдение за изменениями конфигурации следует включить для каждого диска, содержащего критически важные приложения (по умолчанию оно включено только для системного диска). Если изменения конфигурации не отслеживаются, установленные на нем программы не удастся восстановить в случае сбоя.

В Windows Vista и более высоких версиях Windows исходные версии измененных файлов и папок автоматически включаются в состав точек

восстановления. Таким образом, можно получить первоначальные версии любых файлов и папок, измененных после создания точки восстановления. Единственное исключение — системные файлы и папки, такие как C:\Windows.

Эта возможность выручает в случае изменения, удаления или повреждения файлов. Если функция System Restore включена для некоторого диска, Windows автоматически создает ежедневные копии измененных файлов и папок. Впрочем, точку восстановления можно в любой можно создать вручную.

Настройка восстановления системы

Для настройки восстановления системы через консоль PowerShell, открытую с администраторскими полномочиями, используются следующие команды:

- **Enable-ComputerRestore** — включает функцию System Restore для внутренних жестких дисков (внешние и сетевые диски не поддерживаются).

  ```
  Enable-ComputerRestore [-Drive] Диски
  ```

- **Disable-ComputerRestore** — отключает System Restore для диска. В результате при восстановлении системы данный диск игнорируется.

  ```
  Disable-ComputerRestore [-Drive] Диски
  ```

- **Get-ComputerRestorePoint** — получает заданные точки восстановления на локальном компьютере, а также выводит состояние последней попытки восстановления.

  ```
  Get-ComputerRestorePoint [-RestorePoint] Номер
  Get-ComputerRestorePoint -LastStatus
  ```

- **Checkpoint-Computer** — создает точку восстановления на локальном компьютере. Тип точки восстановления задается необязательным параметром –RestorePointType.

  ```
  Checkpoint-Computer [[-RestorePointType] Тип] [-Description] Описание
  ```

- **Restore-Computer** — восстанавливает на локальном компьютере заданную точку восстановления и перезагружает компьютер. Номер точки восстановления задается необязательным параметром –RestorePoint.

  ```
  Restore-Computer [-RestorePoint] Описание
  ```

За мониторинг изменений конфигурации и файлов приложений отвечает служба System Restore. Эта служба настроена для автоматического запуска и работает под учетной записью Local System. Служба System Restore сохраняет сведения для восстановления всех отслеживаемых дисков, для чего на системном томе должно быть не меньше 300 Мб свободного места. При необходимости System Restore резервирует дополнительное место для записи точек восстановления, занимая таким образом до 10% общей емкости диска,

но в любой момент может освободить зарезервированное место для нужд пользователей и приложений. При исчерпании доступного свободного места System Restore записывает новые точки восстановления поверх старых.

Включение и отключении System Restore

Чтобы включить функцию System Restore для тома, воспользуйтесь командлетом Enable-ComputerRestore. Вот его общий синтаксис:

```
Enable-ComputerRestore [-Drive] Диск
```

В этой команде параметр –Drive задает диски в следующем формате:

```
enable-computerrestore -drive «C:\», «D:\»
```

Включить System Restore для произвольного диска можно только после либо одновременно с включением этой функции для системного диска.

Чтобы отключить System Restore, вызовите командлет Disable-ComputerRestore:

```
Disable-ComputerRestore [-Drive] Диск
```

Параметр –Drive задает диски в следующем формате:

```
disable-computerrestore -drive «C:\», «D:\»
```

Отключить функцию восстановления для системного том можно только после ее отключения на всех остальных томах.

Эти команды не поддерживают параметр –ComputerName, но их все же можно вызывать на удаленных компьютерах с помощью командлета invoke-command (см. главу 6). Вот пример:

```
invoke-command -computername techpc24 -scriptblock
{ enable-computerrestore -drive «C:\», «D:\» }
```

Эта команда включает восстановление для дисков C и D компьютера TechPC25.

Создание и применение точек восстановления

Для ручного создания точек восстановления введите **checkpoint-computer** и укажите описание точки восстановления, как показано ниже:

```
checkpoint-computer «Modify PowerShell»
```

В этом примере создается точка восстановления «Modify PowerShell». Во время создания точки Windows PowerShell выводит индикатор хода операции. При желании можно указать тип точки восстановления с помощью параметра –RestorePointType:

- **APPLICATION_INSTALL** — точка, созданная перед установкой приложения (этот тип задан по умолчанию);

- **APPLICATION_UNINSTALL** — точка, созданная перед удалением приложения;
- **DEVICE_DRIVER_INSTALL** — точка, созданная перед изменением параметров драйвера устройства;
- **MODIFY_SETTINGS** — точка, созданная перед изменением параметров конфигурации.

Чтобы получить список доступных точек восстановления или вывести отдельную точку с заданным номером (всем точкам восстановления присваиваются последовательные номера), используйте командлет Get-ComputerRestorePoint.

Чтобы вывести полный список доступных точек восстановления, введите **get-computerrestorepoint**:

```
get-computerrestorepoint
```

```
CreationTime Description SequenceNumber  EventType RestorePointType
------------ ----------- --------------  ---------   ---------------
1/22/2009 9:56:36 AM   Windows Update     287   BEGIN_SYSTEM_C...
APPLICATION_INSTALL
1/23/2009 11:30:46 AM  Windows Update     288   BEGIN_SYSTEM_C...
APPLICATION_INSTALL
1/23/2009 11:38:24 AM  Windows Update     289   BEGIN_SYSTEM_C...
APPLICATION_INSTALL
```

Из этого вывода видно, что для каждой точки восстановления отображается время создания, описание, серийный номер, тип события и точки восстановления. Заметив в выводе этой команды нужную точку восстановления, запомните или запишите ее номер, чтобы в дальнейшем выбрать ее с помощью параметра –RestorePoint. Вот как, например, получить точку восстановления номер 289:

```
get-computerrestorepoint 289
```

Поскольку все параметры точек, показанные выше, хранятся в одноименных свойствах, можно отбирать нужные точки восстановления с помощью командлета Get-ComputerRestorePoint и конструкции Where-Object. Например, следующая команда получает все точки восстановления, созданные за последние три дня:

```
$date = (get-date).adddays(-3)
get-computerrestorepoint | where-object {$_.creationtime -gt $date}
```

А эта команда получает току восстановления с заданным описанием:

```
get-computerrestorepoint | where-object {$_.description -eq
«Modify PowerShell»}
```

Для получения описания точек восстановления нужно знать числовые коды их типов:

- 0 — точки, созданные перед установкой приложений и обновлений Windows;
- 1 — точки, созданные перед удалением приложений;
- 7 — плановые точки восстановления;
- 10 — точки, созданные перед установкой драйверов устройств;
- 12 — точки, созданные перед изменением настроек системы.

Так, следующая команда получает все точки восстановления, созданные перед установкой приложений:

```
get-computerrestorepoint | where-object {$_.restorepointtype -eq 0}
```

Восстановление системы с помощью System Restore

Чтобы использовать точку восстановления, введите **restore-computer** и серийный номер точки восстановления. Получить список точек восстановления позволяет командлет Get-ComputerRestorePoint, из его вывода можно узнать номер нужной точки. Следующая команда восстанавливает систему с использованием точки номер 353:

```
restore-computer 353
```

А эта — восстанавливает систему на компьютере EngPC85 с использованием точки восстановления номер 276:

```
invoke-command -computername engpc85 -scriptblock
{ restore-computer 276 }
```

Во время восстановления работа компьютера завершается, после чего он снова запускается, уже с использованием параметров, сохраненных в точке восстановления. Чтобы проверить состояние восстановления, после перезагрузки компьютера введите **get-computerrestorepoint -laststatus**. Вывод этой команды подтверждает успех операции либо явно сообщает о неудачном завершении восстановления, например:

```
The last restore was interrupted.
```

Если и после восстановления систем Windows продолжает работать со сбоями, попробуйте еще раз восстановить эту точку либо выберите другую, более раннюю точку восстановления.

Глава 14

Тонкая настройка производительности

В предыдущей главе я продемонстрировал базовые методы мониторинга и оптимизации работы Windows-компьютеров. Мониторинг — это регулярная проверка систем на предмет возникновения сбоев, а оптимизация — настройка системы для достижения и поддержания ее оптимальной производительности. В этой главе мы разберем более продвинутые приемы для:

- управления приложениями, процессами и производительностью;
- наблюдения за производительностью компьютеров и поддержания ее на требуемом уровне;
- диагностики и устранения проблем с оборудованием.

Управление приложениями, процессами и производительностью

Одна из важных задач администратора — наблюдение за работой сетевых систем, он также должен добиваться, чтобы вверенные ему системы работали если не вовсе без сбоев, то хотя бы с минимальным их числом. Как сказано в предыдущей главе, внимательное наблюдение за журналами событий позволяет вовремя выявлять неполадки в работе приложений, системы безопасности и ключевых служб. Для диагностики и устранения причин сбоев необходим более тщательный анализ журналов и другой информации. В идеальном случае обнаружение и устранение причин сбоя позволяет предотвратить его повторное появление.

Каждый раз, когда пользователь или операционная система запускает службу или приложение либо исполняет некоторую команду, Windows создает один или несколько процессов для исполнения соответствующей программы. Для наблюдения за процессами и управления им служат следующие команды:

- **Debug-Process** — выполняет отладку процессов, работающих на локальном компьютере.

```
Debug-Process [-Id] ID_процессов | -InputObject Объекты | -Name Имена
```

- **Get-Process** — выводит список процессов с указанием их имен, идентификаторов (process ID) и размера занятой памяти.

  ```
  Get-Process -Id ID_процессов | -InputObject Объекты | [[-Name] Имена]
  [-ComputerName Имена_компьютеров] [-FileVersionInfo] [-Module]
  ```

- **Start-Process** — запускает процессы на локальном компьютере. Программу, исполняемую в процессе, задают с помощью пути к исполняемому файлу или сценарию, либо к файлу документа, который может быть открыт определенной программой, например Microsoft Office Word или Office Excel. Если задан файл, не являющийся исполняемым, Start-Process запускает программу, сопоставленную данному типу файла с использованием действия по умолчанию (обычно — Open; чтобы задать другое действие, используйте параметр –Verb).

  ```
  Start-Process [-Verb {Edit|Open|Print|...}] [-WorkingDirectory
  Путь] [[-ArgumentList] Аргументы] [-FilePath] Путь_к_пр-ме/док-ту

  [-Credential Объект_удостоверений] [-LoadUserProfile {$True|$False}]
  [-NoNewWindow] [-PassThru] [-RedirectStandardError Путь]
  [-RedirectStandardInput Путь] [-RedirectStandardOutput
  Путь] [-UseNewEnvironment] [-Wait] [-WindowStyle
  {Normal|Hidden|Minimized|Maximized}]
  ```

- **Stop-Process** — останавливает процессы, заданные по имени или ID, также поддерживает фильтры, позволяющие задавать процессы для остановки по состоянию, номеру сеанса, утилизации процессорного времени, памяти и т. д.

  ```
  Stop-Process [-Id] ID_процессов | -InputObject Объекты | -Name Имена
  [-Force] [-PassThru]
  ```

- **Wait-Process** — останавливает прием ввода до завершения заданного процесса.

  ```
  Wait-Process -Id ID_процессов | -InputObject Объекты | [-Name] Имена
  [[-TimeOut] Время_ожидания]
  ```

В следующих разделах мы подробно рассмотрим применение этих команд, но сначала давайте разберем работу процессов и связанные с ней общие сбои.

Системные и пользовательские процессы

Для изучения процессов, работающих на локальных и удаленных компьютерах, используется командлет Get-Process и некоторые другие команды. Так, Get-Process выводит идентификатор процесса, его состояние и другие важные сведения. Get-Process поддерживает фильтры, которые позволяют получать информацию об отдельных процессах. Более подробную информацию о процессах дают классы Win32_Process и Win32_Service.

В общем, процессы, запущенные операционной системой, называются системными, а процессы, запущенные пользователями, — пользовательскими. Большинство пользовательских процессов работает в интерактивном режиме, то есть взаимодействуют с пользователем посредством мыши и клавиатуры. Активный (выбранный в данный момент) процесс или программа управляет клавиатурой и мышью, пока он не будет завершен либо не будет выбрана другая программа или приложение. Активный процесс также называют процессом «переднего плана» (foreground process).

Процессы могут работать и в фоновом режиме (background process), т.е. независимо от сеансов вошедших в систему пользователей. Фоновые процессы не контролируют клавиатуру, мышь и другие устройства ввода; как правило, их запускает операционная система. Пользователи также могут запускать фоновые процессы с помощью Планировщика, и такие процессы будут выполняться независимо от наличия в системе зарегистрированного пользователя. Так, если Планировщик запустил процесс, когда в системе зарегистрирован некоторый пользователь, то исполнение этого процесса продолжится и после выхода пользователя из системы.

Windows отслеживает все процессы, регистрируя их имена, идентификаторы (ID), приоритет и другие параметры, включая использование системных ресурсов. Имя образа совпадает с именем исполняемого файла, запустившего процесс, например Msdtc.exe или Svchost.exe. Идентификатор процесса — это номер, под которым процесс фигурирует в системе, например 1160. Базовый приоритет указывает, сколько системных ресурсов может получить данный процесс по отношению к другим работающим процессам. При распределении системных ресурсов процессам с более высоким приоритетом отдается предпочтение перед процессами с более низким приоритетом. Таким образом, приоритетным процессам не приходиться простаивать в ожидании выделения процессорного времени, обработки обращения к памяти или диску, в это время процессы с низким приоритетом ждут, когда высокоприоритетные процессы освободят эти ресурсы.

Только в идеальном мире все процессы работают «как часы» и никогда не сталкиваются с ошибками. Увы, в реальности нередко возникают сбои, и чаще всего — в самый неожиданный момент. Вот наиболее общие проблемы, связанные с процессами:

- процесс «зависает», прекращая реагировать на запросы. В этой ситуации пользователи обычно жалуются на то, что «программа не открывается», «тормозит» или их «выкидывает из программы»;
- процесс не освобождает ресурсы процессора. При этом замедляется работа системы, поскольку сбойный процесс забирает почти все процессорное время и не дает работать другим процессам;
- процесс занимает слишком много памяти. Это бывает, например, при утечках памяти, когда процесс не освобождает вовремя используемую им память. В результате система работает все медленнее и может вовсе

зависнуть. Утечки памяти также могут вызвать ошибки в работе других программ, исполняемых на том же компьютере.

При возникновении вышеперечисленных и других проблем процесс обычно перезапускают. Желательно также проверить журналы событий, чтобы найти причину сбоя. Обнаружив утечку памяти, сообщите о ней разработчикам программы, возможно, они выпустят обновление, в котором эта ошибка будет устранена. Приложение, в котором имеется утечка памяти, следует регулярно перезапускать, чтобы операционная система смогла освободить занятую им память.

Диагностика работающих процессов

Чтобы получить полный список работающих в системе процессов, введите get-process:

```
get-process -computername fileserver86
```

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    127       4    13288      16708    59             1240 audiodg
    696       6     1872       5248    94              588 csrss
    495       7    21912      19672   155              644 csrss
     95       4     3160       6572    75     0.08    2752 CtHelper
    134       5     3808       7900    77     0.08    2708 Ctxfihlp
    189       5     9160       8764    72     0.22    3044 CTxfispi
     58       2     1236       4244    45     0.31    3260 ehmsas
```

Поскольку параметр –ComputerName принимает списки имен, разделенные запятыми, такая команда позволяет опросить сразу несколько компьютеров, например:

```
get-process -computername fileserver86, dcserver22, printserver31
```

Чтобы не набирать список имен всякий раз вручную, сохраните его в текстовом файле (каждое имя на отдельной строке). Этот файл можно использовать в командах следующим образом:

```
get-process -computername (get-content c:\data\clist.txt)
```

Здесь список удаленных компьютеров загружается из файла Clist.txt в каталоге C:\Data.

Чтобы найти нужный процесс, укажите его имя или ID (можно с подстановочными знаками), как показано в следующем примере:

```
get-process win* -computername fileserver86
```

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    106       4     1560       4192    46              636 wininit
```

```
    143       4     2516       6692    56              804 winlogon
    485      26    45992      69220   441    83.34    5200 WINWORD
```

Эта команда ищет все процессы, имя которых начинается на «*win*».

Командлет Get-Process возвращает объекты, которые представляют процессы, соответствующие заданным критериям. Его стандартный вывод включает следующие поля:

- **CPU** — псевдоним свойства TotalProcessorTime.TotalSeconds; процессорное время (в секундах), использованное процессом;
- **Handles** — псевдоним свойства HandleCount; число описателей файлов, открытых процессом;
- **NPM** — псевдоним свойства NonpagedSystemMemorySize; размер занятой процессом виртуальной памяти, содержимое которой запрещено выкачивать в страничный файл на диске;
- **PM** — псевдоним свойства PagedMemorySize; размер выделенной процессу виртуальной памяти, содержимое которой разрешено выкачивать на диск;
- **VM** — псевдоним свойства VirtualMemorySize; суммарный размер зарезервированной для процесса и переданной ему виртуальной памяти;
- **WS** — псевдоним свойства WorkingSet; размер памяти, занятой процессом в настоящее время, включая закрытый и открытый рабочие наборы;
- **Id** — число-идентификатор процесса;
- **ProcessName** — имя процесса или запустившего его исполняемого файла.

Анализируя процессы, имейте в виду, что одно приложение может запустить сразу несколько процессов. Как правило, это дочерние процессы, порожденные первым (главным) процессом приложения; все вместе они образуют дерево процессов приложения. Чтобы корректно завершить приложение, необходимо завершить его главный процесс.

Чтобы увидеть все доступные свойства процесса, отформатируйте вывод команды следующим образом:

```
get-process winword -computername server12 | format-list *
```

```
Name                       : WINWORD
Path            : C:\Program Files\Microsoft Office\OFFICE11\WINWORD.EXE
Company                    : Microsoft Corporation
CPU                        : 121.046875
FileVersion                : 11.0.8237
ProductVersion             : 11.0.8237
Description                : Microsoft Office Word
Product                    : Microsoft Office 2003
Id                         : 5200
PriorityClass              : Normal
HandleCount                : 491
WorkingSet                 : 73142272
```

```
PagedMemorySize            : 47992832
PrivateMemorySize          : 47992832
VirtualMemorySize          : 463138816
TotalProcessorTime         : 00:02:01.0468750
BasePriority               : 8
Handle                     : 2752
MachineName                : server12
MainWindowHandle           : 1573996
MainWindowTitle            : process.doc - Microsoft Word
MainModule              : System.Diagnostics.ProcessModule (WINWORD.EXE)
MaxWorkingSet              : 1413120
MinWorkingSet              : 204800
Modules                    : {System.Diagnostics.ProcessModule (WINWORD.
EXE), System.Diagnostics.ProcessModule (ntdll.dll),...}
NonpagedSystemMemorySize   : 27312
NonpagedSystemMemorySize64 : 27312
PagedMemorySize64          : 47992832
PagedSystemMemorySize      : 754224
PagedSystemMemorySize64    : 754224
PeakPagedMemorySize        : 48001024
PeakPagedMemorySize64      : 48001024
PeakWorkingSet             : 73150464
PeakWorkingSet64           : 73150464
PeakVirtualMemorySize      : 464146432
PeakVirtualMemorySize64    : 464146432
PriorityBoostEnabled       : True
PrivateMemorySize64        : 47992832
PrivilegedProcessorTime    : 00:00:36.2343750
ProcessName                : WINWORD
ProcessorAffinity          : 15
Responding                 : True
SessionId                  : 1
StartInfo                  : System.Diagnostics.ProcessStartInfo
StartTime                  : 2/28/2009 10:22:34 AM
Threads                    : {5204, 5244, 5300, 5540...}
UserProcessorTime          : 00:01:24.8125000
VirtualMemorySize64        : 463138816
EnableRaisingEvents        : False
WorkingSet64               : 73142272
```

Этот вывод содержит исчерпывающие сведения о конфигурации процесса, наиболее важные свойства процессов перечислены в табл. 14-1.

Примечание По умолчанию многие свойства, характеризующие использование памяти, определены как 32-разрядные, но на 64-разрядных системах у этих свойств есть 64-разрядные версии, и только они дают точные значения на 64-разрядных системах, оснащенных более чем 4 Гб оперативной памяти.

Табл. 14-1. Важнейшие свойства, отображаемые Get-Process

Имя	Описание
BasePriority	Приоритет процесса, определяющий количество выделяемых ему системных ресурсов. Стандартные значения: Low (4), Below Normal (6), Normal (8), Above Normal (10), High (13) и Real-Time (24). Большинство процессов имеет приоритет Normal, наивысший приоритет — Real-Time
CPU	Значение TotalProcessorTime (в секундах)
Description	Описание процесса
FileVersion	Версия исполняемого файла, запустившего процесс
HandleCount	Число описателей файлов, открытых процессом; характеризует зависимость процесса от файловой системы. Некоторые процессы открывают описатели тысячами, и для каждого из них требуется некоторое количество системной памяти
Id	Число, идентифицирующее процесс во время выполнения
MinWorkingSet	Минимальный размер рабочего набора процесса
Modules	Исполняемые файлы и динамически подключаемые библиотеки, используемые процессом
NonpagedSystemMemory-Size/NonpagedSystem-MemorySize64	Размер занятой процессом невыгружаемой виртуальной памяти (оперативной памяти, в которой находятся объекты, которые нельзя выкачивать на диск). Процессы, которым требуется много невыгружаемой памяти, следует взять на заметку: при нехватке памяти такие процессы могут вызывать большое количество страничных ошибок
PagedSystemMemorySize/ PagedSystemMemorySize64	Размер переданной процессу выгружаемой виртуальной памяти (оперативной памяти, содержащей объекты, которые разрешено выкачивать на диск, пока они не используются). Размер используемой процессом памяти увеличивается вместе активностью процесса; обычно процессу требуется больше выгружаемой памяти, чем невыгружаемой
Path	Полный путь к исполняемому файлу процесса
PeakPagedMemorySize/ PeakPagedMemorySize64	Пиковый размер выгружаемой памяти, используемой процессом
PeakVirtualMemorySize/ PeakVirtualMemorySize64	Пиковый размер виртуальной памяти, используемой процессом
PeakWorkingSet/ PeakWorkingSet64	Максимальный размер используемой процессом памяти, включая закрытый и открытый рабочие наборы. Слишком высокое значение этого свойства может свидетельствовать об утечке памяти

Табл. 14-1. Важнейшие свойства, отображаемые Get-Process

Имя	Описание
PriorityBoostEnabled	Булево значение, указывает, включена ли функция PriorityBoost для процесса
PriorityClass	Класс приоритета процесса
PrivilegedProcessorTime	Время режима ядра, использованное процессом
ProcessName	Имя процесса
ProcessorAffinity	Привязка к процессору
Responding	Булево значение, показывает, ответил ли процесс на запросы во время проверки
SessionId	Идентификатор пользовательского сеанса, в котором работает процесс; соответствует значению в столбце ID в на вкладке Users в Task Manager (Диспетчере задач)
StartTime	Дата и время запуска процесса
Threads	Число потоков в процесса. Многие серверные приложения — многопоточные, что позволяет им обрабатывать несколько клиентских запросов одновременно. Некоторые приложения поддерживают динамическое управление числом одновременно исполняемых потоков для оптимизации производительности. Если потоков слишком много, производительность системы снижается из-за частого переключения контекста
TotalProcessorTime	Суммарное количество процессорного времени, использованного процессом после запуска. Если процесс использует слишком много процессорного времени, соответствующее приложение может быть настроено неверно, либо этот процесс «завис», не освободив процессор
UserProcessorTime	Время режима пользователя, использованное процессом
VirtualMemorySize/ VirtualMemorySize64	Суммарный размер виртуальной памяти, зарезервированной для процесса и переданной ему. Содержимое виртуальной памяти хранится на диске, поэтому она работает медленнее оперативной. Чтобы повысить быстродействие приложения, настройте его так, чтобы оно использовало больше оперативной памяти (для этого в системе должно быть ОЗУ достаточного размера, в противном случае может замедлиться работа других процессов)

(*см. след. стр.*)

Табл. 14-1. Важнейшие свойства, отображаемые Get-Process

Имя	Описание
WorkingSet/WorkingSet64	Количество памяти, используемой процессом в данный момент, включая открытый и закрытый рабочие наборы. Закрытым рабочим набором называется память, которую процесс использует единолично и не может разделять с другими процессами, а открытым рабочим набором — память, разделяемая с другими процессами. Постоянно увеличивающееся использование памяти процессом, не снижающееся до исходного уровня, может свидетельствовать об утечке памяти

Фильтрация сведений о процессах

Направляя вывод Get-Process командлету Where-Object, можно изучать значения любых доступных свойств процессов. Так, например, удается вывести только «зависшие» процессы либо процессы, использующие много процессорного времени.

Для фильтрации процессов по значениям их свойств используют следующие операторы (табл. 14-2):

- **–Eq** — равно;
- **–Ne** — не равно;
- **–Gt** — больше;
- **–Lt** — меньше;.
- **–Ge** — больше или равно;.
- **–Le** — меньше или равно;
- **–Match** — шаблон искомой строки; если значение свойства соответствует заданному шаблону, соответствующий процесс отображается командой.

Табл. 14-2. Операторы и допустимые значения для фильтрации вывода Get-Process

Свойство	Операторы	Допустимые значения
BasePriority	–eq, –ne, –gt, –lt, –ge, –le	0–24
HandleCount	–eq, –ne, –gt, –lt, –ge, –le	Положительные целые числа
MachineName	–eq, –ne	Любая допустимая строка
Modules	–eq, –ne, –match	Имя DLL
Privileged-ProcessorTime	–eq, –ne, –gt, –lt, –ge, –le	Время в формате чч:мм:сс
ProcessID	–eq, –ne, –gt, –lt, –ge, –le	Положительные целые числа
ProcessName	–eq, –ne	Любая допустимая строка
Responding	–eq, –ne	$True, $False
SessionID	–eq, –ne, –gt, –lt, –ge, –le	Действительный номер сеанса

Табл. 14-2. Операторы и допустимые значения для фильтрации вывода Get-Process

Свойство	Операторы	Допустимые значения
Username	–eq, –ne	Имя пользователя или имя в формате *домен**пользователь*
UserProcessorTime	–eq, –ne, –gt, –lt, –ge, –le	Время в формате чч:мм:сс
WorkingSet	–eq, –ne, –gt, –lt, –ge, –le	Целое число килобайтов

По умолчанию Get-Process просматривает все процессы, независимо от их состояния. Проверяя булево значение свойства Responding, можно отбирать процессы, которые отвечают на запросы либо, наоборот, перестали отвечать на них (т. е. «зависли»):

```
get-process | where-object {$_.responding -eq $False}
get-process | where-object {$_.responding -eq $True}
```

В первом примере выводятся все «зависшие» процессы, а во втором — все процессы, которые работают нормально.

Высокоприоритетные процессы используют больше процессорного времени, чем остальные, поэтому при анализе производительности компьютера желательно изучить такие процессы. Большинство процессов имеет приоритет Normal (значение — 8). В следующем примере отбираются процессы с приоритетом больше 8:

```
get-process | where-object {$_.basepriority -gt 8}
```

```
  Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
  -------  ------    -----      ----- -----   ------     -- -----------
      655       5     1872       5232    94              588 csrss
      493       7    22108      19876   155              644 csrss
      791      12     4820       2420    69              692 lsass
      280       8     3216       7732    51              680 services
       28       1      364        812     4              516 smss
      106       4     1560       4192    46              636 wininit
      143       4     2528       6700    56              804 winlogon
```

Также нередко требуется выявить процессы, которые потребляют много процессорного времени. В следующем примере отбираются процессы, использовавшие больше 30 минут времени режима ядра:

```
get-process | where-object {$_.privilegedprocessortime -gt «00:30:00»}
```

```
  Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
  -------  ------    -----      ----- -----   ------     -- -----------
      553      32    52924      80096   487   3026.42   5200 W3SVC
```

Просмотр служб и связанных с ними процессов

Запрашивая класс Win32_Service с помощью Get-Process, можно узнать, какие службы работают в том или ином процессе. ID процесса, в котором работает служба, входит в стандартный вывод такого запроса. Вот пример для службы Windows Search:

```
get-wmiobject -class win32_service -filter «name='wsearch'»
```

```
ExitCode  : 0
Name      : WSearch
ProcessId : 2532
StartMode : Auto
State     : Running
Status    : OK
```

Свойство ProcessId объекта Win32_Service содержит подробные сведения о процессе, в котором работает служба:

```
$s = get-wmiobject -class win32_service -filter «name='wsearch'»
get-process -id $s.processid
```

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
   1294      15    43768      40164   162            2532 SearchIndexer
```

Ту же самую информацию дает команда

```
get-wmiobject -class win32_service -filter «name='wsearch'» |
foreach ($a) {get-process -id $_.processid}
```

По умолчанию вывод Get-Process форматируется в виде таблицы, но его можно отформатировать и как список. И еще один важный момент: Get-Process группирует выводимые службы по именам соответствующих исполняемых файлов. Так, SearchIndexer.exe — имя файла, запускающего службу Windows Search.

Определение процессов, соответствующих службами, серьезно упрощает управление компьютерами. Например, при подозрении на сбои службы World Wide Web Publishing Service (W3svc) следует начать наблюдение за процессами, в которых работает эта служба, отслеживая:

- состояние процесса (отвечает ли он на запросы);
- использование процессом памяти (выгружаемой и виртуальной), размер рабочего набора;
- использованное процессорное время в режиме ядра и режиме пользователя.

Это позволит выявить «зависшие» процессы, утечки памяти и процессы, использующие чрезмерное количество процессорного времени.

Просмотр DLL, используемых процессами

Свойство Modules, выводимое командлетом Get-Process содержит список DLL, используемых заданным процессом. Однако в стандартном выводе список может оказаться неполным. Рассмотрим следующий пример:

```
get-process dwm | format-list modules
```

```
Modules : {System.Diagnostics.ProcessModule (Dwm.exe),
System.Diagnostics.ProcessModule (ntdll.dll),
System.Diagnostics.ProcessModule (kernel32.dll),
System.Diagnostics.ProcessModule (ADVAPI32.dll)...}
```

Совет Управляющая переменная $FormatEnumerationLimit задает число значений в выводимых списках. По умолчанию выводится 4 элемента, поэтому предыдущая команда выводит только четыре DLL. Для просмотра полного списка DLL следует увеличить значение этой переменной хотя бы до 30.

По умолчанию конфигурация PowerShell разрешает выводить только первых четырех значений свойства Modules. Чтобы увидеть полный список, сохраните объект Process в переменной и обратитесь к свойству Modules, как показано ниже:

```
$p = get-process dwm
$p.modules
```

```
    Size(K) ModuleName              FileName
    ------- ----------              --------
         96 Dwm.exe                 C:\Windows\system32\Dwm.exe
       1180 ntdll.dll               C:\Windows\system32\ntdll.dll
        876 kernel32.dll            C:\Windows\system32\kernel32.dll
        792 ADVAPI32.dll            C:\Windows\system32\ADVAPI32.dll
        776 RPCRT4.dll              C:\Windows\system32\RPCRT4.dll
        300 GDI32.dll               C:\Windows\system32\GDI32.dll
        628 USER32.dll              C:\Windows\system32\USER32.dll
        680 msvcrt.dll              C:\Windows\system32\msvcrt.dll
       1296 ole32.dll               C:\Windows\system32\ole32.dll
        564 OLEAUT32.dll            C:\Windows\system32\OLEAUT32.dll
        252 UxTheme.dll             C:\Windows\system32\UxTheme.dll
        120 IMM32.dll               C:\Windows\system32\IMM32.dll
        800 MSCTF.dll               C:\Windows\system32\MSCTF.dll
         96 dwmredir.dll            C:\Windows\system32\dwmredir.dll
         28 SLWGA.dll               C:\Windows\system32\SLWGA.dll
       1188 urlmon.dll              C:\Windows\system32\urlmon.dll
        352 SHLWAPI.dll             C:\Windows\system32\SHLWAPI.dll
        276 iertutil.dll            C:\Windows\system32\iertutil.dll
         40 WTSAPI32.dll            C:\Windows\system32\WTSAPI32.dll
        232 slc.dll                 C:\Windows\system32\slc.dll
```

```
          36 LPK.DLL                    C:\Windows\system32\LPK.DLL
         500 USP10.dll                  C:\Windows\system32\USP10.dll
        1656 comctl32.dll               C:\Windows\WinSxS\x86_microsoft....
        1984 milcore.dll                C:\Windows\system32\milcore.dll
          28 PSAPI.DLL                  C:\Windows\system32\PSAPI.DLL
         132 NTMARTA.DLL                C:\Windows\system32\NTMARTA.DLL
         296 WLDAP32.dll                C:\Windows\system32\WLDAP32.dll
         180 WS2_32.dll                 C:\Windows\system32\WS2_32.dll
          24 NSI.dll                    C:\Windows\system32\NSI.dll
          68 SAMLIB.dll                 C:\Windows\system32\SAMLIB.dll
```

Следующая команда даст аналогичный результат:

```
get-process dwm | foreach ($a) {$_.modules}
```

Зная DLL, загруженные в процесс, проще выяснить причину его «зависания» либо чрезмерного использования системных ресурсов. Так, можно проверить версии DLL, чтобы убедиться в том, что они верны и совместимы, сверив их с информацией в Базе знаний Майкрософт (Microsoft Knowledge Base) или документации производителя.

Чтобы найти процесс, в который загружена определенная DLL, следует указать имя искомой DLL. Предположим, вы подозреваете, что «зависание» процессов вызывает старый драйвер очереди печати, Winspool.drv. В этом случае нужно выяснить, в какие процессы загружен Winspool.drv (вместо Winspool32.drv) и проверить их состояние и утилизацию ресурсов. Вот общий синтаксис соответствующей команды:

```
get-process | where-object {$_.modules -match «Имя_DLL»}
```

где *Имя_DLL* — имя искомой DLL (нечувствительно к регистру символов). Вот пример команды для поиска процессов, использующих файл Winspool.drv:

```
get-process | where-object {$_.modules -match «winspool.drv»}
```

```
 Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
 -------  ------    -----      ----- -----   ------     -- -----------
     147       5     2696       8108    77     0.11   2956 DrgToDsc
     787      23    38444      49148   218    11.97   2132 explorer
     114       5     4708       7164    69     0.17    580 IAAnotif
     288      10    11516      18312   118     1.39    672 IntelHCTAgent
      71       4     2316       7068    69     2.13   5448 notepad
      68       3     2844       4544    57     0.03   2804 rundll32
     543      32    52304      79096   480   283.83   5200 WINWORD
```

Останов процессов

Для останова процессов служит командлет Stop-Process. Он останавливает процессы, заданные по ID (с помощью параметра –Id) или по имени (с по-

мощью параметра –Name). Параметр –Id не поддерживает подстановочные знаки, а –Name — поддерживает их.

По умолчанию Stop-Process запрашивает подтверждение при попытке останова процесса, не принадлежащему текущему пользователю. Если у вас есть соответствующие разрешения, можно отменить запрос предупреждения с помощью параметра –Force.

Можно остановить сразу несколько процессов, указав их имена или ID. Однако будьте осторожны с именами: Stop-Process останавливает все процессы с заданным именем. То есть, если в системе работает три экземпляра Svchost, все они будут остановлены вызовом Stop-Process с данным именем.

Совет Учтите, что некоторые приложения запускают несколько процессов. Как правило, при останове главного процесса останавливается все дерево процессов приложения.

Разберем использование Stop-Process на примерах:

Останов процесса с ID = 1106:

```
stop-process 1106
```

Останов всех процессов с именем W3SVC:

```
stop-process -name w3svc
```

Останов процессов 1106, 1241 и 1546:

```
stop-process 1106, 1241, 1546
```

Останов процесса 891 без запроса подтверждения:

```
stop-process -force -id 891
```

Для выборочного останова процессов используйте Stop-Process в паре с Get-Process. Например, так можно остановить не все (что делается по умолчанию, если задан параметр –Name), а только «зависшие» экземпляры Winword, либо только процессы, в которые загружена определенная DLL.

При останове процессов будьте осторожны, чтобы случайно не остановить один из критических системных процессов, таких как Lsass, Wininit или Winlogon. Таким процессам назначаются идентификаторы, значения которых меньше 1000, поэтому на всякий случай не останавливайте процессы с ID 999 и меньше.

Разберем использование Get-Process и Stop-Process на примерах:

Останов «зависших» экземпляров winword:

```
get-process -name winword | where-object {$_.responding -eq $false}
| stop-process
```

Останов всех «зависших» процессов с id > 999:

```
get-process | where-object {$_.id -gt 999} | where-object
{$_.responding -eq $false} | stop-process
```

Останов всех процессов, использующих библиотеку winspool.drv:

```
get-process | where-object {$_.modules -match «winspool.drv»} |
stop-process
```

Хотя Stop-Process не поддерживает параметр –ComputerName, для управления процессами на удаленных компьютерах можно использовать следующий прием:

```
get-process w3svc -computername engpc18 | stop-process

invoke-command -computername engpc18 -scriptblock { get-process w3svc |
stop-process }
```

Здесь Get-Process получает объект Process, представляющий процесс на удаленном компьютере, который затем останавливается с помощью Stop-Process. Учтите, что эта команда сообщает только о сбоях и ничего не говорит, если процесс не был найден, либо его не удалось остановить.

Более детальный анализ процессов

Наряду с Get-Process для получения информации о работающих процессах применяют командлет Get-WmiObject вместе с классом Win32_Process. Команда get-wmiobject -class win32_process выводит подробные сведения о всех работающих на компьютере процессах. Возможно также получение сведений об отдельных процессах, заданных по имени образа, например:

```
get-wmiobject -class win32_process -filter «name='searchindexer.exe'»
```

```
Caption                    : SearchIndexer.exe
CommandLine                :
CreationClassName          : Win32_Process
CreationDate               : 20090228102000.542640-480
CSCreationClassName        : Win32_ComputerSystem
CSName                     : TECHPC22
Description                : SearchIndexer.exe
ExecutablePath             :
ExecutionState             :
Handle                     : 2532
HandleCount                : 1323
InstallDate                :
KernelModeTime             : 15156250
MaximumWorkingSetSize      :
MinimumWorkingSetSize      :
Name                       : SearchIndexer.exe
OSCreationClassName        : Win32_OperatingSystem
OtherOperationCount        : 34235
OtherTransferCount         : 2763510
PageFaults                 : 34156
```

```
PageFileUsage              : 43696
ParentProcessId            : 680
PeakPageFileUsage          : 44760
PeakVirtualSize            : 174481408
PeakWorkingSetSize         : 41128
Priority                   : 8
PrivatePageCount           : 44744704
ProcessId                  : 2532
QuotaNonPagedPoolUsage     : 16
QuotaPagedPoolUsage        : 177
QuotaPeakNonPagedPoolUsage : 29
QuotaPeakPagedPoolUsage    : 183
ReadOperationCount         : 9223
ReadTransferCount          : 45734098
SessionId                  : 0
Status                     :
ThreadCount                : 16
UserModeTime               : 55156250
VirtualSize                : 169123840
WindowsVersion             : 6.0.6001
WorkingSetSize             : 41414656
WriteOperationCount        : 3845
WriteTransferCount         : 27617602
ProcessName                : SearchIndexer.exe
```

Объекты Win32_Process предоставляют некоторые сведения о процессах, которые невозможно извлечь из объектов Process, например подробности об операциях чтения-записи. В остальном выводимая информация идентична той, что дает Get-Process, но в некоторых случаях она отображается по-другому.

У объектов Process имеется свойство StartTime, а у Win32_Process — свойство CreationDate. Значение свойства StartTime выводится как DateTime, а значение CreationDate — как строка DateTime. Свойство StartTime позволяет отобрать процессы, проработавшие дольше заданного периода. Следующая команда отбирает процессы, проработавшие дольше суток:

```
$yesterday = (get-date).adddays(-1)
get-process | where-object {$_.starttime -gt $yesterday}
```

То же самое позволяет сделать и этот код:

```
get-process | where-object {$_.starttime -gt (get-date).adddays(-1)}
```

Аналогично можно использовать свойство CreationDate, например:

```
$yesterday = (get-date).adddays(-1)

get-wmiobject -class win32_process | where-object {$_.creationdate -gt
$yesterday}
```

Объекты Win32_Process поддерживают свойство *Threadcount* — число потоков, связанных с процессом. Вот как можно узнать, сколько потоков работает в процессе:

```
get-wmiobject -class win32_process -filter «name='searchindexer.exe'» |
format-list name, threadcount
```

```
name        : SearchIndexer.exe
threadcount : 17
```

Сходную информацию дает свойство Threads объектов Process:

```
$p = get-process –name searchindexer
write-host «Number of threads: » ($p.threads).count
```

```
Number of threads: 17
```

Возможна также работа с отдельными объектами Thread. Так, в следующем примере выводятся сведения о потоках с использованием объектов Thread:

```
$p = get-process –name searchindexer
$p.threads
```

```
BasePriority            : 8
CurrentPriority         : 10
Id                      : 2536
IdealProcessor          :
PriorityBoostEnabled    :
PriorityLevel           :
PrivilegedProcessorTime :
StartAddress            : 0
StartTime               :
ThreadState             : Wait
TotalProcessorTime      :
UserProcessorTime       :
WaitReason              : Executive
ProcessorAffinity       :

BasePriority            : 8
CurrentPriority         : 9
. . .
```

Чтобы получить подробности об отдельном потоке, укажите его объект с помощью индекса в массиве объектов Threads. Например, ссылка на объект, представляющий первый поток, выглядит так: $p.threads[0].

У каждого потока есть такие характеристики, как базовый приоритет, текущий приоритет, идентификатор, начальный адрес и состояние. Для потоков, ожидающих другие процессы или потоки, также указывается причина

ожидания: Executive (ожидание компонента ядра операционной системы) или UserRequest (ожидание компонента режима пользователя).

Объекты Win32_Process поддерживают ряд методов для работы с процессами:

- **GetOwner()** — получает учетную запись, под которой работает процесс;
- **GetOwnerSid()** — получает идентификатор защиты учетной записи, под которой работает процесс;
- **Terminate()** — останавливает процесс.

Вот общий синтаксис вызова первых двух методов:

```
$Объект_процесса.GetOwner()
```

и

```
$Объект_процесса.GetOwnerSid()
```

где *$Объект_процесса* — ссылка на объект Win32_Process. Вот пример:

```
$p = get-wmiobject -class win32_process -filter "name='notepad.exe'"
$p.getowner()
```

```
Domain           : CPANDL
ReturnValue      : 0
User             : WILLIAMS
```

Эта команда получает сведения о владельце экземпляра процесса Notepad и выводит значения свойств Domain и User, представляющих имена домена и учетной записи пользователя-владельца процесса. А так выводят идентификатор защиты процесса:

```
$p.getownersid()
```

```
Sid              : S-1-5-21-4857584848-3848484848-8484884848-1111
```

Метод для останова процесса вызывают так:

```
$p.terminate()
```

```
ReturnValue      : 0
```

В этом примере важно возвращаемое значение (ReturnValue). Если оно равно 0, команда выполнена успешно; любое другое значение свидетельствует об ошибке. Ошибки обычно возникают, если процесс пытается остановить пользователь, не являющийся его владельцем и не имеющий разрешений на останов. В этом случае необходимо ввести учетные данные уполномоченного пользователя либо использовать консоль PowerShell, открытую с администраторскими правами.

Вышеописанные методы работают, если Get-WmiObject возвращает единственный объект Win32_Process, но не годятся, если команда возвращает несколько таких объектов. Причина в том, что группа объектов-процессов

возвращается в виде массива, поэтому необходимо явно указать нужный экземпляр Win32_Process. Вот пример:

```
$procs = get-wmiobject -class win32_process -filter "name='notepad.exe'"
foreach ($p in $procs) { $p.getowner() }
```

```
Domain           : CPANDL
ReturnValue      : 0
User             : WILLIAMS

Domain           : CPANDL
ReturnValue      : 0
User             : WILLIAMS
```

Здесь выводятся владельцы всех экземпляров процесса Notepad (данный прием сработает, даже если в системе работает единственный экземпляр Notepad).

Мониторинг производительности

Мониторинг производительности систем позволяет выявлять и устранять ситуации, чреватые нарушением нормальной работы систем. Для этих целей PowerShell поддерживает несколько команд, наиболее востребованные из них освещаются в этом разделе.

Команды для мониторинга производительности

Для мониторинга производительности используются следующие команды:

- **Get-Counter** — получат объекты, представляющие текущее значение счетчика производительности, непосредственно от инструментария Windows для мониторинга производительности. Поддерживается вывод наборов и отдельных счетчиков производительности, установка размера и интервала выборки данных, а также учетных данных пользователей, уполномоченных для просмотра сведений о производительности.

  ```
  Get-Counter [-MaxSamples Размер_выборки] [-Counter] Путь_к_счетчику
  [-SampleInterval Интервал] {AddtlParams}

  Get-Counter -ListSet Имя_набора {AddtlParams}

  {AddtlParams}
  [-Credential Объект_уч._данных] [-ComputerName Компьютеры]
  ```

- **Export-Counter** — экспортирует журналы счетчиков производительности в формате BLG (двоичный формат журнала; задан по умолчанию), CSV (формат с разделителями-запятыми) или TSV (формат с разделителями-табуляторами). Поддерживается экспорт данных, возвращаемых командлетами Get-Counter и Import-Counter (только в Windows 7, Windows Server 2008 R2 и выше).

```
Export-Counter [-FileFormat Формат] [-Path] Путь
-InputObject Объекты_выбранных_данных {AddtlParams}

{AddtlParams}
[-Force {$True | $False}] [-Circular {$True | $False}]
[-MaxSize МаксРазмер_в_байтах]
```

- **Import-Counter** — импортирует журналы счетчиков производительности и создает на их основе представляющие выборки данных счетчиков объекты, объекты, идентичные тем, что возвращает Get-Counter. Поддерживается импорт файлов в форматах BLG, CSV и TSV. При использовании BLG разрешается импортировать до 32 файлов одной командой. Для импорта части содержимого журнала служат параметры Import-Counter (см. примеры ниже).

```
Import-Counter [-Path] Путь {AddlParams}
Import-Counter -ListSet Набор_счетчиков [-Path] Путь
Import-Counter [-Summary {$True | $False}]

{AddtlParams}
[-Counter Счетчики] [-MaxSamples Число_выборок]
[-StartTime Дата_время] [-EndTime Дата_время]
```

Командлет Get-Counter поддерживает мониторинг и вывод заданных параметров производительности в реальном времени. Требуемые сведения о производительности задаются с применением следующих компонентов:

- **объекты производительности** — представляют системные компоненты, обладающие количественными характеристиками. Ими могут быть как физические (ОЗУ, процессоры, страничные файлы) так и логические (логические диски, очередь печати) компоненты операционной системы либо элементы программ (процессы и потоки);
- **экземпляры объектов производительности** — представляют объекты производительности, связанные с экземплярами компонентов системы. Например, на многопроцессорном компьютере или компьютере с несколькими жесткими дисками с каждым процессором или диском связан свой экземпляр объекта производительности. Возможен мониторинг всех либо отдельных экземпляров объектов производительности (т.е. одного процессора либо всех процессоров компьютера одновременно);
- **счетчики производительности** — представляют количественные характеристики объектов производительности. Например, для измерения утилизации страничного файла служит счетчик %Usage.

В операционной системе Windows, установленной в конфигурации по умолчанию, доступно множество объектов производительности. По мере добавления служб, приложений и компонентов число объектов производительности увеличивается. Например, при установке службы DNS на сервере появляются объекты для мониторинга производительности службы DNS.

Сбор сведений о производительности

Командлет Get-Counter позволяет записывать данные о производительности в простые файлы и файлы журналов, главное — правильно указать нужные счетчики производительности. Вот общий синтаксис пути к счетчику производительности:

*Компьютер**Объект**Счетчик*

где *Компьютер* — имя или IP-адрес компьютера, *Объект* — имя объекта производительности, а *Счетчик* — имя счетчика производительности. Например, для мониторинга размера доступной памяти на компьютере Dbserver79 можно ввести:

```
get-counter «\\dbserver79\memory\available mbytes»

   Timestamp             CounterSamples
   ---------             --------------
   2/27/2010 4:26:54 PM  \\dbserver79\memory\available mbytes : 1675
   2/27/2010 4:26:55 PM  \\dbserver79\memory\available mbytes : 1672
```

мечание В этом примере путь необходимо взять в двойные кавычки, поскольку в нем есть пробелы. Это требуется не всегда, но лучше взять себе за правило заключать пути в кавычки.

Указывать имя компьютера в составе пути к счетчику не обязательно. Если имя компьютера пропущено, Get-Counter использует параметр –ComputerName (он же позволяет опрашивать группы компьютеров), если же этот параметр не задан, подразумевается локальный компьютер. Тем не менее, рекомендуется освоить формат указания полных путей, так как именно они заносятся в трассировки и журналы производительности. Если опустить имя компьютера, путь принимает следующий вид:

*Объект**Счетчик*

Следующая команда проверяет размер доступной памяти на компьютерах, заданных параметром –ComputerName:

```
get-counter -computername fileserver12, dbserver18, dcserver21
«\memory\available mbytes»
```

При работе с удаленными компьютерами нередко требуется вводить альтернативные учетные данные. Вот как это можно сделать:

```
$cred = get-credential
get-counter -computername fileserver12, dbserver18, dcserver21
«\memory\available mbytes» -credential $cred
```

При вызове командлета Get-Credential PowerShell запрашивает у пользователя учетные данные и записывает их в переменную $cred. В дальнейшем эти учетные данные используются для аутентификации на удаленных компьютерах.

Чтобы отслеживать все счетчики объекта, введите вместо имени счетчика звездочку (*):

```
get-counter «\\dbserver79\Memory\*»
```

В этом примере отслеживаются все счетчики объекта Memory.

Если объект существует в нескольких экземплярах, например Processor или LogicalDisk, следует указать нужный экземпляр. В этом случае синтаксис полного пути принимает следующий вид:

*Компьютер**Объект*(*Экземпляр*)*Счетчик*

Экземпляр указывают в скобках после имени объекта производительности. Чтобы наблюдать сразу за всеми экземплярами, введите _Total как имя экземпляра, либо, если требуется определенный экземпляр, укажите его идентификатор. Например, для мониторинга счетчика Processor\% Processor Time у всех процессоров в системе можно использовать команду

```
get-counter «\\dbserver79\Processor(_Total)\% Processor Time»
```

либо, для мониторинга определенного процессора:

```
get-counter «\\dbserver79\Processor(0)\% Processor Time»
```

Здесь экземпляр Processor(0) представляет первый процессор в системе.

Get-Counter поддерживает ряд параметров. Параметр –MaxSamples задает число выборок данных; –SampleInterval — интервал между выборками (значение по умолчанию — 1 секунда); –ListSet — выводит счетчики для заданных объектов.

По умолчанию Get-Counter выводит данные на консоль; чтобы направить их в журнал, перенаправьте вывод командлету Export-Counter. По умолчанию Export-Counter экспортирует данные счетчиков в журналы двоичного формата. Параметр –FileFormat позволяет задать для журнала формат CSV (текстовый файл с разделителями-запятыми), TSV (текстовый файл с разделителями-табуляторами) либо BLG (двоичный формат). Рассмотрим следующий пример:

```
get-counter «\\dbserver79\Memory\*» | export-counter -fileformat tsv
-path .\dbserver79.txt
```

Эта команда отслеживает все счетчики объекта Memory и записывает их данные в текстовый файл Dbserver79.txt с разделителями-запятыми, расположенный в текущем рабочем каталоге. Чтобы использовать эти данные в дальнейшем, вызовите кмандлет Import-Counter. Для этого введите import-counter -path, затем путь к файлу с данными. Команда import-counter -summary -path (+ путь к файлу с данными) отображает сводку данных, а import-counter -listset * -path (+ путь к файлу с данными) — список отслеживаемых счетчиков. При желании можно указать интересующий вас период сбора данных с помощью параметров –StartTime и –EndTime. Рассмотрим пример:

```
$startdate = (get-date).adddays(-1)
$enddate = (get-date)
import-counter -path .\data.txt -starttime $startdate -endtime $enddate
```

Эти команды анализируют данные из расположенного в текущем каталоге файла Data.txt, собранные за сутки между текущим временем сегодняшнего и вчерашнего дня.

Чтобы получить список счетчиков, поддерживаемых объектом производительности, введите get-counter -listset, затем имя нужного объекта. Так, например, можно узнать, какие счетчики поддерживает объект производительности Memory:

```
get-counter -listset Memory
```

```
Counter: {\Memory\Page Faults/sec, \Memory\Available Bytes,
\Memory\Committed Bytes, \Memory\Commit Limit...}
CounterSetName     : Memory
MachineName        : EngPC85
CounterSetType     : SingleInstance
Description        : The Memory performance object  consists of counters
that describe the behavior of physical and virtual memory on the
computer.  Physical memory is the amount of random access memory on the
computer.  Virtual memory consists of the space in physical memory and
on
disk.  Many of the memory counters monitor paging, which is the movement
of pages of code and data between disk and physical memory.  Excessive
paging, a symptom of a memory shortage, can cause delays which interfere
with all system processes.
Paths: {\Memory\Page Faults/sec, \Memory\Available Bytes,
\Memory\Committed Bytes, \Memory\Commit Limit...}
PathsWithInstances : {}
```

Как и любые другие команды PowerShell, команды этого примера возвращают объекты, с которыми можно работать. Чтобы получить полный список счетчиков с указанием полного пути к каждому из них, опросить свойство Paths:

```
$c = get-counter -listset Memory
$c.paths
```

```
\Memory\Page Faults/sec
\Memory\Available Bytes
\Memory\Committed Bytes
\Memory\Commit Limit
```

либо используйте следующую команду:

```
get-counter -listset memory | foreach ($a) {$_.paths}
```

Получить список счетчиков с учетом экземпляров объектов производительности позволяет свойство PathsWithInstances. Вот как с ним работают:

```
$d = get-counter -listset PhysicalDisk
$d.pathswithinstances
```

```
\PhysicalDisk(0 E: C:)\Current Disk Queue Length
\PhysicalDisk(1 W:)\Current Disk Queue Length
\PhysicalDisk(2 D:)\Current Disk Queue Length
\PhysicalDisk(3 I:)\Current Disk Queue Length
\PhysicalDisk(4 J:)\Current Disk Queue Length
\PhysicalDisk(5 K:)\Current Disk Queue Length
\PhysicalDisk(6 L:)\Current Disk Queue Length
\PhysicalDisk(7 N:)\Current Disk Queue Length
\PhysicalDisk(8 O:)\Current Disk Queue Length
\PhysicalDisk(9 P:)\Current Disk Queue Length
\PhysicalDisk(10 Q:)\Current Disk Queue Length
\PhysicalDisk(_Total)\Current Disk Queue Length
```

Тот же самый результат даст и следующая команда:

```
get-counter -listset PhysicalDisk | foreach ($a) {$_.pathswithinstances}
```

Все эти примеры выводят длинный список счетчиков, сгруппированный по экземплярам объектов производительности. Вывод можно записать в текстовый файл:

```
get-counter -listset PhysicalDisk > disk-counters.txt
```

Далее полученный файл можно отредактировать, оставив в нем только нужные счетчики. Используется такой файл следующим образом:

```
get-counter (get-content .\disk-counters.txt) | export-counter -path
c:\perflogs\disk-check.blg
```

В этом примере Get-Counter читает список отслеживаемых счетчиков из файла Disk-Counters.txt и записывает их в двоичный файл Disk-Check.blg в каталоге C:\Perflogs.

По умолчанию Get-Counter будет выбирать данные раз в секунду, пока вы не нажмете Ctrl+C. Это удобно, если вы можете просматривать данные счетчиков в консоли PowerShell. Однако чаще всего у администраторов нет на это времени. В этом случае нужно настроить длительность и интервал выборки данных.

Для этого служат параметры –MaxSamples и –SampleInterval, соответственно. Например, чтобы заставить Get-Counter записать в журнал 100 значений счетчика с интервалом 120 секунд, можно ввести такую команду:

```
get-counter (get-content .\disk-counters.txt) -sampleinterval 120
-maxsamples 100 | export-counter -path c:\perflogs\disk-check.blg
```

Диагностика и устранение проблем с производительностью

Командлеты Get-Process и Get-Counter дают всю необходимую информацию для выявления и устранения большинства проблем, связанных с производительностью, но обычно диагностика и поиск причин таких проблем требует более тщательного анализа.

Мониторинг процессов и утилизации системных ресурсов

При работе с процессами администраторам часто требуется знать текущую картину распределения системных ресурсов, в частности, памяти. Один из способов получения такой картины — вывод значений ключевых счетчиков объекта Memory с помощью командлета Get-Counter. Как сказано выше, вывести поддерживаемые им счетчики можно командой:

```
get-counter -listset memory | foreach ($a) {$_.paths}
```

Большинство счетчиков объекта Memory выводят последнее моментальное, а не среднее значение соответствующего параметра.

Листинг 14-1 содержит пример получения моментальной картины распределения памяти с помощью Get-Counter. В этом примере нужные счетчики загружаются из файла Perf.txt и опрашиваются с интервалом в 30 с, а результаты сохраняются в файле SaveMemData.txt. Чтобы составить представление об использовании страничного файла, содержимое этого файла можно импортировать в электронные таблицы или таблицу в документе Word.

Я выбрал именно эти счетчики (см. листинг 14-1), поскольку они дают детальную картину использования памяти в системе. Чтобы собрать сведения об использовании памяти в разное время суток, сохраните показанную ниже команду в виде сценария и запускайте его по графику.

Листинг 14-1. Получение моментальной картины использования памяти

```
Команда

get-counter (get-content .\perf.txt) -maxsamples 5 -sampleinterval 30 >
SaveMemData.txt

Файл Perf.txt

\memory\% Committed Bytes In Use
\memory\Available MBytes
\memory\Cache Bytes
\memory\Cache Bytes Peak
\memory\Committed Bytes
```

```
\memory\Commit Limit
\memory\Page Faults/sec
\memory\Pool Nonpaged Bytes
\memory\Pool Paged Bytes

Примерный вывод

Timestamp                CounterSamples
---------                --------------
2/28/2009 5:04:37 PM     \\techpc22\memory\% committed bytes in use :
                         22.9519764760423
                         \\techpc22\memory\available mbytes :
                         1734
                         \\techpc22\memory\cache bytes :
                         390168576
                         \\techpc22\memory\cache bytes peak :
                         390688768
                         \\techpc22\memory\committed bytes :
                         1650675712
                         \\techpc22\memory\commit limit :
                         7191867392
                         \\techpc22\memory\page faults/sec :
                         3932.45999649944
                         \\techpc22\memory\pool nonpaged bytes :
                         70017024
                         \\techpc22\memory\pool paged bytes :
                         154710016

2/28/2009 5:05:07 PM     \\techpc22\memory\% committed bytes in use :
                         23.2283134955779
                         \\techpc22\memory\available mbytes :
                         1714
                         \\techpc22\memory\cache bytes :
                         389664768
                         \\techpc22\memory\cache bytes peak :
                         390701056
                         \\techpc22\memory\committed bytes :
                         1670549504
                         \\techpc22\memory\commit limit :
                         7191867392
                         \\techpc22\memory\page faults/sec :
                         617.601067565369
                         \\techpc22\memory\pool nonpaged bytes :
                         70008832
                         \\techpc22\memory\pool paged bytes :
                         154791936
```

При подозрении на проблемы с использованием памяти следует получить ключевые сведения о процессах с помощью Get-Process:

```
get-process | format-table –property ProcessName,
BasePriority, HandleCount, Id, NonpagedSystemMemorySize,
PagedSystemMemorySize, PeakPagedMemorySize, PeakVirtualMemorySize,
PeakWorkingSet, SessionId, Threads, TotalProcessorTime,
VirtualMemorySize, WorkingSet, CPU, Path
```

Эта команда выводит значения свойств в порядке, в котором они заданы (в виде списка с разделителями-запятыми; чтобы изменить порядок значений, просто отредактируйте этот список). При желании можно сохранить вывод в файл, как показано ниже:

```
get-process | format-table –property ProcessName,
BasePriority, HandleCount, Id, NonpagedSystemMemorySize,
PagedSystemMemorySize, PeakPagedMemorySize, PeakVirtualMemorySize,
PeakWorkingSet, SessionId, Threads, TotalProcessorTime,
VirtualMemorySize, WorkingSet, CPU, Path > savedata.txt
```

Если вывод нужно отображать на консоли PowerShell, настройте ее ширину так, чтобы она составляла не менее 180 символов, иначе данные будет трудно читать.

Мониторинг подкачки страниц памяти

Нередко администратору требуются сведения о страничных ошибках, для устранения которых системе требуется обращаться к диску, и страничных ошибках, не требующих обращения к диску. Страничная ошибка возникает, когда система не может найти по заданному адресу страницу памяти, запрошенную процессом. Если страница находится в оперативной памяти, но по другому адресу, такая страничная ошибка называется «мягкой» (soft page fault), а если страницу приходится подкачивать с жесткого диска — «жесткой» (hard page fault).

Для мониторинга страничных ошибок в реальном времени введите:

```
get-counter «\memory\Page Faults/sec» –sampleinterval 5
```

```
Timestamp                 CounterSamples
---------                 --------------
2/28/2009 6:00:01 PM      \\techpc22\memory\page faults/sec :
                          172.023153991804
2/28/2009 6:00:06 PM      \\techpc22\memory\page faults/sec :
                          708.944308818821
2/28/2009 6:00:11 PM      \\techpc22\memory\page faults/sec :
                          14.5375722784541
```

Эта команда выводит данные о страничных ошибках раз в пять секунд, отображая общее число ошибок в секунду. Чтобы прервать исполнение Get-

Counter, нажмите Ctrl+C. Для мониторинга страничных ошибок также используются следующие счетчики объекта Memory:

- Cache Faults/sec (Ошибок кэш-памяти/сек);
- Demand Zero Faults/sec (Ошибок запроса обнуления/сек);
- Page Reads/sec (Чтений страниц/сек);
- Page Writes/sec (Операций вывода страниц/сек);
- Write Copies/sec (Запись копий страниц/сек);
- Transition Faults/sec (Ошибок транзита/сек);
- Transition Pages RePurposed/sec (Переходных многоцелевых страниц/сек).

Особого внимания заслуживают счетчики Page Reads/sec и Page Writes/sec, которые предоставляют сведения о «жестких» страничных ошибках. Если разработчикам программ интересно знать причины страничных ошибок, то администраторам важнее знать их число. Большинство процессоров легко справляется с множеством «мягких» страничных ошибок, ведь для устранения такой ошибки системе достаточно «заглянуть» в другую область оперативной памяти, чтобы взять оттуда искомую страницу. Другое дело «жесткие» страничные ошибки: для ее устранения приходится читать запрошенную страницу с диска, что вызывает существенные задержки. Если мониторинг показывает множество «жестких» страничных ошибок, имеет смысл увеличить размер ОЗУ либо уменьшить размер памяти, кэшируемой системой и приложениями на жестком диске.

Наряду с вышеописанными счетчиками объекта Memory, для мониторинга страничного файла используется ряд других счетчиков. Так, для мониторинга компонентов, существующих в нескольких экземплярах (например, на компьютере с несколькими жесткими дисками или страничными файлами), используются соответствующие экземпляры объектов производительности. Разрешается отслеживать сразу все экземпляры объекта, получая по ним (например, по всем установленным на компьютере жестким дискам) единую сводку; для этого укажите **_Total** вместо имени экземпляра. Чтобы наблюдать за определенным экземпляром объекта производительности, укажите его индекс.

Листинг 14-2 иллюстрирует использование Get-Counter для получения «моментального снимка» использования страничных файлов. В этом примере список отслеживаемых счетчиков производительности загружается из файла PagePerf.txt. Далее значения указанных счетчиков записываются в файл SavePageData.txt пять раз с интервалом 30 с. Чтобы было проще составить представление об использовании страничных файлов, импортируйте содержимое SavePageData.txt в электронные таблицы либо в документ Word.

Листинг 14-2. Мониторинг использования страничного файла

```
Команды

get-counter (get-content .\pageperf.txt) -maxsamples 5 `
-sampleinterval 30 > SavePageData.txt

Файл PagePerf.txt

\memory\Pages/Sec
\Paging File(_Total)\% Usage
\Paging File(_Total)\% Usage Peak
\PhysicalDisk(_Total)\% Disk Time
\PhysicalDisk(_Total)\Avg. Disk Queue Length
```

Мониторинг памяти и рабочих наборов отдельных процессов

Командлет Get-Process предоставляет сведения об использовании памяти процессами:

```
get-process -id ID_процесса
```

где *ID_процесса* — идентификатор нужного процесса. Эта команда отображает, сколько памяти используется процессом в данный момент. Например, для процесса с ID 1072 вывод Get-Process может быть таким:

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    493      13    15520      13452    77              1072 svchost
```

В этом примере процесс использует 13 452 кб памяти. Наблюдая за процессом, можно выяснить, увеличивается ли со временем размер занятой им памяти. Если процесс использует все больше и больше памяти по сравнению со средним эталонным значением, в этом процессе возможна утечка памяти.

На листинге 14-3 показан исходный код сценария PowerShell, отслеживающего использование памяти процессами. Сценарий принимает ID процесса как первый параметр, если этот параметр не задан, генерируется сообщение об ошибке.

Листинг 14-3. Мониторинг использования памяти через командную строку PowerShell

```
Сценарий MemUsage.ps1

$p = read-host «Enter process id to track»
$n = read-host «Enter number of minutes to track»
for ($c=1; $c -le $n; $c++) {get-process -id $p; start-sleep -seconds 60}
```

Примерный вывод

```
Enter process id to track: 1072
Enter number of minutes to track: 1

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    497      13    15548      13464    77            1072 svchost
    494      13    15520      13452    77            1072 svchost
    495      13    15520      13452    77            1072 svchost
    493      13    15520      13452    77            1072 svchost
    495      13    15548      13464    77            1072 svchost
    495      13    15520      13452    77            1072 svchost
```

Из листинга 14-3 видно, что размер используемой процессами памяти немного варьируется во времени, но тенденции к его увеличению нет. Следовательно, в нем, скорее всего, нет утечек памяти, но для полной уверенности необходимо более длительное наблюдение за этим процессом.

Чтобы получить подробную информацию об использовании памяти отдельными процессами, вызывайте Get-Process следующим образом:

```
get-process Процесс | format-table -property `
NonpagedSystemMemorySize, PagedSystemMemorySize, VirtualMemorySize, `
PeakVirtualMemorySize, MinWorkingSet, WorkingSet, PeakWorkingSet
```

где *Процесс* — имя процесса (имя исполняемого файла без расширения .exe или .dll). В сценариях PowerShell (см., например, листинг 14-4) можно комбинировать вызовы Get-Process и Start-Sleep для регулярного мониторинга просмотра утилизации памяти процессами.

Листинг 14-4. Вывод подробной информации об использовании памяти

Сценарий DetMemUsage.ps1

```
$p = read-host «Enter process name to track»
$n = read-host «Enter number of minutes to track»
for ($c=1; $c -le $n; $c++) { get-process $p | format-table -property `
NonpagedSystemMemorySize, PagedSystemMemorySize, VirtualMemorySize, `
PeakVirtualMemorySize, MinWorkingSet, WorkingSet, PeakWorkingSet
start-sleep -seconds 60}
```

Примерный вывод

```
   Nonpaged       Paged     Virtual       Peak         Min  Working        Peak
     System      System  MemorySize    Virtual  WorkingSet      Set  WorkingSet
 MemorySize  MemorySize             MemorySize
 ----------  ----------  ---------  ----------  ---------  --------- ----------
       4776       96368   52891648   161480704             6946816     7282688
       8424      137056   61505536   161480704             8986624     9039872
```

13768	121136	137351168	161480704	38670336	73658368
13792	128904	82386944	161480704	13889536	73658368
14320	167912	187904000	258859008	74432512	138919936
44312	235704	221302784	429953024	65249280	278482944
25288	156520	91754496	429953024	1 5536128	278482944
30112	159376	123875328	429953024	23248896	278482944
25296	118424	86568960	429953024	20758528	278482944
2248	48088	24174592	429953024	2924544	278482944
7112	105160	55832576	429953024	6393856	278482944
5368	110960	63991808	429953024	7655424	278482944
1472	30400	15618048	429953024	2330624	278482944

Этот сценарий-пример выводит значения следующих свойств:

- **NonPagedSystemMemorySize** — невыгружаемая память, выделенная процессу;
- **PagedSystemMemorySize** — выгружаемая память, выделенная процессу;
- **VirtualMemorySize** — виртуальная память, переданная процессу и зарезервированная для него;
- **PeakVirtualMemorySize** — пиковый размер выгружаемой памяти, занятой процессом;
- **WorkingSet** — память, выделенная процессу операционной системой;
- **PeakWorkingSet** — пиковый размер памяти, используемой процессом.

Значения этих свойств дают представление об использовании памяти отдельным процессом. Ключевой параметр — размер рабочего набора, он показывает, сколько памяти отведено для процесса операционной системой. Если рабочий набор необратимо увеличивается со временем, это может свидетельствовать об утечке памяти в процессе, способной снизить производительность системы в целом.

Из листинга 14-4 видно, что за время наблюдения количество памяти, отведенное процессу, заметно изменяется. Скорее всего, этот процесс просто обслуживает множество клиентских или системных запросов, и тогда использование памяти должно снизиться примерно до исходного уровня, в противном случае налицо проблема с использованием памяти процессом.

Поскольку память — типичное «узкое место» как серверов, так и рабочих станций, этот раздел почти полностью посвящен приемам диагностике проблем с памятью. Именно память следует проверять в первую очередь при возникновении проблем с производительностью. Впрочем, не исключены и другие «узкие места», связанные, например, с процессорами, жесткими дисками и сетевыми компонентами.

Станек Уильям Р.

Windows PowerShell 2.0

Справочник администратора

Совместный проект издательства «Русская Редакция» и издательства «БХВ-Петербург».

Подписано в печать 21.01.2010 г. Формат 70x100/16. Усл. физ. л. 26.
Тираж 1500 экз. Заказ №

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано по технологии CtP в ОАО «Печатный двор» им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15