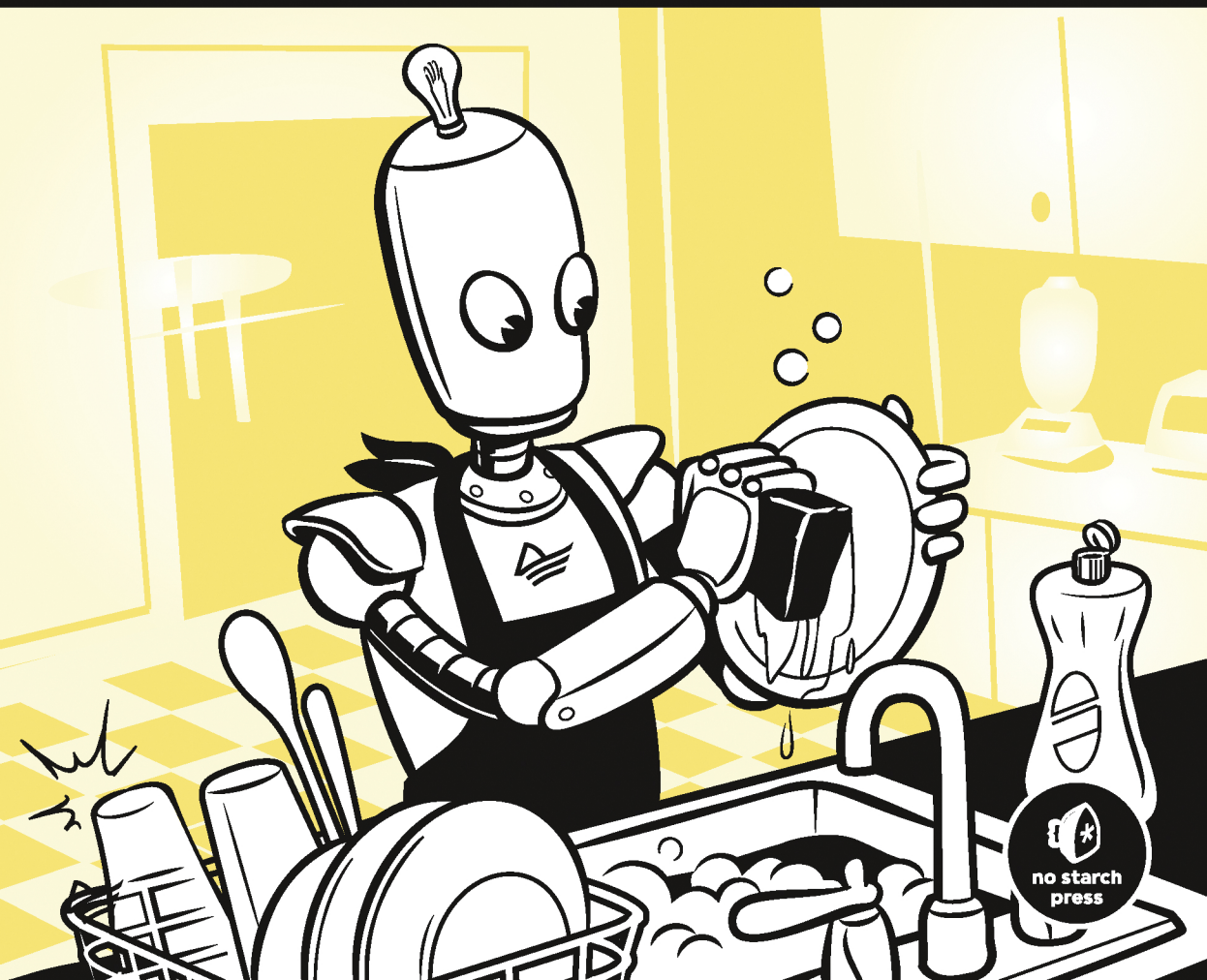
POWERSHELL®
ДЛЯ СИСАДМИНОВ

АДАМ БЕРТРАМ



POWERSHELL® FOR SYSADMINS

Workflow Automation Made Easy

by Adam Bertram

no starch press

San Francisco

АДАМ БЕРТРАМ

POWERSHELL®
ДЛЯ СИСАДМИНОВ

ПИТЕР®

Санкт-Петербург • Москва • Минск

2021

ББК 32.973.2-018.2
УДК 004.451
Б52

Бертрам Адам

Б52 PowerShell для сисадминов. — СПб.: Питер, 2021. — 416 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1732-1

PowerShell® — это одновременно язык сценариев и командная оболочка, которая позволяет управлять системой и автоматизировать практически любую задачу. В книге «PowerShell для сисадминов» обладатель Microsoft MVP Адам Бертрам aka «the Automator» покажет, как использовать PowerShell так, чтобы у читателя наконец-то появилось время на игрушки, йогу и котиков.

Вы научитесь:

- Комбинировать команды, управлять потоком выполнения, обрабатывать ошибки, писать сценарии, запускать их удаленно и тестировать их с помощью фреймворка тестирования Pester.
- Анализировать структурированные данные, такие как XML и JSON, работать с популярными сервисами (например, Active Directory, Azure и Amazon Web Services), создавать системы мониторинга серверов.
- Создавать и проектировать модули PowerShell.
- Использовать PowerShell для удобной, полностью автоматизированной установки Windows.
- Создавать лес Active Directory, имея лишь узел Hyper-V и несколько ISO-файлов.
- Создавать бесчисленные веб- и SQL-серверы с помощью всего нескольких строк кода!

Реальные примеры помогают преодолеть разрыв между теорией и работой в настоящей системе, а легкий авторский юмор упрощает чтение.

Перестаньте полагаться на дорогое ПО и невнятные советы из сети!

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.2
УДК 004.451

ISBN 978-1593279189 англ.

PowerShell for Sysadmins: Workflow Automation Made Easy

ISBN 978-1-59327-918-9, published by No Starch Press.

ISBN 978-5-4461-1732-1

Краткое содержание

Часть III.
Создаем свой модуль

Оглавление

Часть II. Автоматизация повседневных задач

Книга посвящается всем тем, кто ставит под сомнение статус-кво, противостоит корпоративной культуре в духе «мы всегда так поступали» и всегда предлагает лучшее решение задач.

Об авторе

Адам Бертрам (Adam Bertram) — опытный ИТ-специалист и эксперт в области интернет-бизнеса с 20-летним стажем. Предприниматель, ИТ-инфлюенсер, специалист Microsoft MVP, блогер, тренинг-менеджер и автор материалов по контент-маркетингу, сотрудничающий со многими ИТ-компаниями. Также Адам основал популярную платформу TechSnips для развития навыков ИТ-специалистов (techsnips.io).

О научном редакторе

Джеффри Хикс (Jeffry Hicks) — опытный ИТ-специалист с почти 30-летним стажем. Большую часть этого времени был консультантом по вопросам ИТ-инфраструктур и специализировался на серверных технологиях Microsoft с упором на автоматизацию и эффективность. Джеффри несколько раз получал награду Microsoft MVP. Он показал преимущества PowerShell и автоматизации ИТ-специалистам во всем мире, сейчас выступает в качестве независимого автора, преподавателя и консультанта.

Благодарности

Я не смог бы написать эту книгу и достичь всего того, что у меня есть, без поддержки моей жены Миранды. Время — это самый ценный ресурс, и благодаря ей у меня его больше, чем у многих других. Миранда — настоящий генеральный директор в нашей семье. Она каким-то образом справлялась с двумя дочерьми, поддерживала порядок в доме и кормила нас все те годы, пока я был с головой погружен в работу, стараясь обеспечивать семью. Я бы не смог добиться всего, что у меня есть, если бы она не поддерживала меня и наших детей.

Также хочу поблагодарить Джеффри Сновера (Jeffrey Snover) за создание языка сценариев PowerShell, который по-настоящему изменил мою жизнь. Благодарю Джеффа Хикса (Jeff Hicks), Дона Джонса (Don Jones) и Джейсона Хелмика (Jason Helmick) за то, что они вдохновили меня активно влиться в жизнь сообщества. Спасибо компании Microsoft за поддержку тех безумцев, которые стремятся достигнуть большего, с помощью программы MVP и других инициатив.

Введение

За время работы в сфере ИТ я занимал самые разные должности: находился на передовой в службе поддержки, выезжал к пользователям в качестве техника порой только для того, чтобы перезагрузить компьютер, обслуживал серверы в качестве системного администратора, разрабатывал решения будучи системным инженером, а также изучал разницу между маршрутизацией OSPF и RIP в качестве сетевого инженера.

И лишь открыв для себя PowerShell, я понял, как сильно могу увлечься определенной технологией. PowerShell изменил мою жизнь во многих отношениях и стал решающим этапом на моем карьерном пути. Этот язык помог мне стать незаменимым сотрудником на работе, знающим, как сэкономить бесчисленные рабочие часы моей команды, и принес мне мою первую шестизначную зарплату. Язык PowerShell настолько крут, что я решил поделиться им со всем миром. С тех пор в течение пяти лет подряд я получал престижную награду Microsoft MVP.

В этой книге я покажу вам, как использовать язык PowerShell для автоматизации тысяч задач, как создавать собственные инструменты вместо покупки готовых продуктов и как объединять в работе разные инструменты. Возможно, вы не станете активным членом сообщества PowerShell, но я гарантирую, что изучение этого языка даст вам именно те навыки, в которых нуждаются многие компании.

Почему именно PowerShell?

Язык Microsoft PowerShell, который когда-то назывался Monad (см. https://www.jsnover.com/Docs/MonadManifesto.pdf), в 2003 году позиционировался как более интуитивный способ автоматизации задач по сравнению с VBScript. Сейчас PowerShell представляет собой универсальный язык автоматизации, сценариев и разработки. Он был создан с целью стереть барьер между людьми, занимающимися сценариями, автоматизацией и операционной деятельностью. PowerShell должен был дать пользователям инструмент для автоматизации задач с помощью сценариев без необходимости изучать программирование. Это делает язык особенно полезным для системных администраторов, которым недостает опыта в разработке программного обеспечения. Если вы системный администратор, у которого не хватает времени на решение всех задач, то PowerShell может стать прекрасным союзником.

PowerShell сегодня — это повсеместно используемый кроссплатформенный язык написания сценариев и разработки с открытым исходным кодом. Вы можете использовать его не только для поддержки полностью настроенных ферм серверов, но и для создания текстового файла или настройки раздела реестра. Тысячи программных продуктов и сервисов используют PowerShell благодаря постоянно растущему уровню его внедрения среди ИТ-специалистов, разработчиков, инженеров DevOps, администраторов баз данных и системных инженеров.

Для кого эта книга

Эта книга предназначена для ИТ-специалистов и системных администраторов, которым надоело постоянно использовать один и тот же интерфейс и выполнять одну и ту же задачу в пятисотый раз за этот год. Также она будет полезна для инженеров DevOps, которые испытывают затруднения с автоматизацией новых серверных сред, выполнением автоматических тестов или автоматизацией конвейера непрерывной интеграции / непрерывной доставки (CI/CD).

Не получится назвать отрасль, для которой PowerShell был бы полезен больше всего. Традиционная должность пользователя PowerShell в «магазине Windows» — системный администратор Microsoft, однако PowerShell хорошо вписывается в набор инструментов любого сотрудника в сфере ИТ. Если вы работаете в ИТ, но не считаете себя разработчиком, эта книга для вас.

О книге

В этой книге использован практический подход со множеством примеров и реальных кейсов. Вместо того чтобы *рассказывать*, что такое переменная, я вам ее *покажу*. Если вы ищете традиционный учебник, то эта книга не для вас.

Я не буду разбирать PowerShell по частям и рассматривать каждый элемент по отдельности, поскольку в жизни вы так делать не будете. Например, я не ожидаю, что вы знаете определение *функции* или *цикла for* — вместо этого я буду при любой удобной возможности так комбинировать элементы, чтобы дать вам наиболее целостное представление о задаче и способах ее решения.

Книга разделена на три части. **Часть I «Основы»** дает знания, необходимые новичкам в PowerShell для общения с опытными специалистами разработки. Если вы владеете PowerShell на среднем или более высоком уровне, то можете сразу перейти к главе 8.

В **главах 1–7** рассматривается непосредственно язык PowerShell. Вы изучите основы, например, как найти помощь и новые команды, а также некоторые понятия программирования, общие для других языков, — переменные, объекты, функции, модули и основы обработки ошибок.

В **главе 8** объясняется, как использовать инструменты удаленной работы PowerShell для подключения и выполнения команд на других компьютерах.

В **главе 9** мы познакомимся с популярной платформой тестирования PowerShell под названием Pester, которую вы будете использовать на протяжении всей книги.

В **части II «Автоматизация повседневных задач»** вы примените знания, полученные в части I, чтобы начать автоматизировать часто встречающиеся задачи.

В **главах 10–13** рассказывается, как анализировать структурированные данные, а также обращаться с инструментами, с которыми работают многие ИТ-администраторы, например с Active Directory, Azure и Amazon Web Services (AWS).

В **главе 14** показано, как создать инструмент для инвентаризации серверов, который можно использовать в собственной среде.

В **части III «Создаем свой модуль»** вы сконцентрируетесь на создании единого модуля PowerShell под названием PowerLab, чтобы продемонстрировать возможности языка. Мы рассмотрим, каким должен быть хороший дизайн модуля, а также изучим передовые методы работы с функциями. Даже если

вы считаете себя опытным разработчиком сценариев PowerShell, вы наверняка узнаете что-то новое из третьей части.

В **главах 15–20** объясняется, как использовать PowerShell для автоматизации всей лабораторной или тестовой серверной среды на примере настройки виртуальных машин Hyper-V, настраивать операционные системы, а также разворачивать и настраивать серверы IIS и SQL.

Хочу верить, что эта книга поможет вам освоить PowerShell. Если вы новичок, я надеюсь, она придаст вам смелости приступить к автоматизации, а если вы опытный сценарист — то, скорее всего, узнаете новые для себя приемы.

Пора писать сценарии!

От издательства

В локализированной версии PowerShell некоторые приведенные в книге строки вывода консоли переведены на русский язык. Однако большая часть пока что все равно существует только на английском, поэтому мы намеренно оставили оригинальные листинги. Если вы хотите узнать об истории выпусков модулей и командлетов PowerShell, ознакомьтесь с информацией по ссылке https://docs.microsoft.com/ru-ru/powershell/scripting/whats-new/cmdlet-versions?view=powershell-5.1.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ЧАСТЬ I

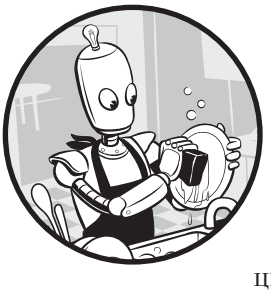
ОСНОВЫ

Как гласит старая пословица, прежде чем начать ходить, сперва нужно научиться ползать. Это же правило работает и с PowerShell. В частях II и III этой книги вы узнаете, как можно создать несколько мощных инструментов. Но прежде вам нужно изучить основы языка. Если вы уже опытный пользователь PowerShell, можете пропустить первую часть — возможно, вы узнаете из нее что-то новое, однако эти крупицы знаний не стоят потраченного времени.

Если же вы новичок в PowerShell, то эта часть для вас. Мы изучим язык PowerShell и некоторые его конструкции, которые вы будете постоянно использовать. Мы рассмотрим все основные моменты, начиная от базовых понятий программирования, таких как переменные и функции, и заканчивая написанием сценариев, их удаленным запуском и тестированием с помощью пока неизвестной вам системы Pester. Поскольку в этой части мы лишь рассматриваем основы, то пока что не будем создавать множество инструментов — для этого предназначены части II и III. Здесь же мы рассмотрим небольшие примеры, чтобы лучше понять язык. Вы получите первое представление о том, на что способен PowerShell. Давайте начнем!

1

Начало работы



Само название *PowerShell* отсылает к двум вещам. Одна из них — оболочка командной строки, установленная по умолчанию во всех последних версиях Windows начиная с Windows 7. Недавно она появилась и на операционных системах Linux и macOS с помощью PowerShell Core. Вторая — это язык сценариев. Вместе они относятся к единой структуре, которую можно использовать для автоматизации всего — от одновременной перезагрузки ста серверов до создания полной системы автоматизации, управляющей целым дата-центром.

В первых главах этой книги мы будем использовать консоль PowerShell для знакомства с основами. После того как вы их освоите, мы перейдем к более сложным темам, включая написание сценариев, функций и пользовательских модулей.

В этой главе мы рассмотрим некоторые базовые команды, а также методы поиска и чтения справочной информации.

Открытие консоли PowerShell

В этой книге мы используем версию PowerShell v5.1, которая уже встроена в Windows 10. В новых версиях PowerShell есть больше функций и меньше багов, но базовый синтаксис и функции PowerShell не претерпели значительных изменений со времен версии 2.

Чтобы открыть PowerShell в Windows 10, введите команду `PowerShell` в меню «Пуск». Вы сразу же увидите нужный вариант. При открытии приложения должна появиться синяя консоль и мигающий курсор, как показано на рис. 1.1.

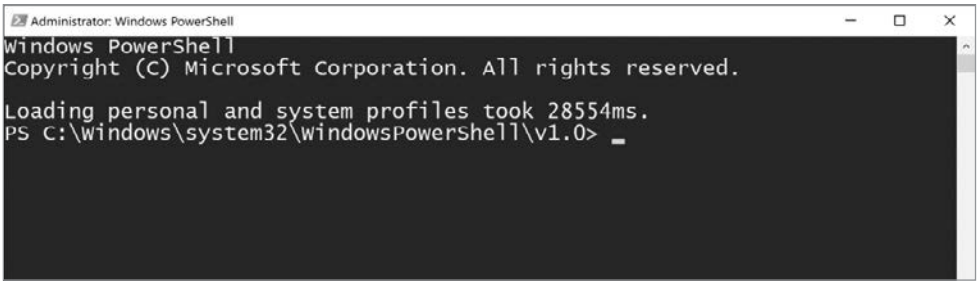


Рис. 1.1. Консоль PowerShell

Мигающий курсор означает, что PowerShell готов принимать от вас команды. Обратите внимание, что ваша строка — приглашение ввода, начинающаяся с `PS`, вероятно, будет отличаться от моей, так как в ней указано ваше текущее местоположение в системе. Как видно из заголовка моей консоли, я щелкнул по значку PowerShell правой кнопкой мыши и запустил его от имени администратора. Это дает мне полные права, а запуск происходит в каталоге C:\Windows\system32\WindowsPowerShell\v1.0.

Использование команд DOS

После открытия PowerShell можно приступать к его изучению. Если вы когда-либо использовали командную строку Windows, cmd.exe, для вас станет приятным сюрпризом то, что все команды, к которым вы привыкли (например, `cd`, `dir` и `cls`), также работают в PowerShell. По сути, эти «команды» среды DOS на самом деле являются не командами, а их *псевдонимами*, благодаря которым PowerShell и понимает эти команды. Но пока не обязательно вникать в эту разницу — просто считайте их своими друзьями из среды DOS!

Давайте попробуем ввести некоторые из этих команд. Если вы находитесь в строке приглашения `PS>` и хотите проверить содержимое определенного каталога, сначала перейдите в этот каталог с помощью команды `cd` (сокращение от *change directory*). С помощью этой команды вы перейдете в каталог *Windows*:

```
PS> cd .\Windows\
PS C:\Windows>
```

ИСПОЛЬЗОВАНИЕ АВТОЗАПОЛНЕНИЯ

Обратите внимание, что в этой команде я указал каталог *Windows* с точкой и обратной косой чертой по обе стороны: `.\Windows\`. На самом деле, вам не нужно вводить все это, потому что в консоли PowerShell есть отличная функция автозаполнения, которая позволяет вам с помощью клавиши Tab просмотреть доступные команды.

Например, если вы наберете `Get-`, а затем нажмете клавишу Tab, среда начнет предлагать вам все команды, которые начинаются с `Get-`. С помощью многократного нажатия этой клавиши можно перебрать все возможные команды, а Shift+Tab возвращает вас назад. Автозаполнение можно использовать и в параметрах, о чем мы с вами поговорим в разделе «Изучаем команды PowerShell» на примере команды `Get-Content` с последующим нажатием Tab. В этом случае вместо прокрутки команд PowerShell начнет предлагать возможные параметры для команды `Get-Content`.

Оказавшись в папке C:\Windows, можно использовать команду `dir` для вывода списка содержимого вашего текущего каталога, как показано в листинге 1.1.

Листинг 1.1. Вывод содержимого текущего каталога с помощью команды dir

```
PS C:\Windows> dir


    Directory: C:\Windows

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         3/18/2019  4:03 PM                addins
d-----          8/9/2019 10:28 AM                ADFS
d-----         7/24/2019  5:39 PM                appcompat
d-----         8/19/2019 12:33 AM                AppPatch
d-----         9/16/2019 10:25 AM                AppReadiness
--пропуск--
```

С помощью команды `cls` можно очистить окно консоли. Если вы знакомы с командной строкой cmd.exe, попробуйте другие известные вам команды и посмотрите, как они ведут себя в PowerShell. Большинство из них работает — но не все. Если вам любопытно, какие команды cmd.exe работают в этой консоли, то после ее запуска можно ввести команду `Get-Alias`: она выведет вам многие старомодные команды cmd.exe, к которым вы привыкли.

```
PS> Get-Alias
```

Таким образом вы сможете увидеть все встроенные псевдонимы и сопоставляемые с ними команды PowerShell.

Изучаем команды PowerShell

Как и почти во всех языках, в PowerShell есть *команды* — общий термин для именованных исполняемых выражений. Командой может быть что угодно — от устаревшего инструмента ping.exe до ранее упомянутой `Get-Alias`. Можно даже создавать собственные команды. Однако если вы попытаетесь использовать несуществующую команду, то получите печально известную ошибку с текстом красного цвета, как показано в листинге 1.2.

Листинг 1.2. При вводе неизвестной команды отображается ошибка

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

Чтобы увидеть список всех команд, которые PowerShell знает по умолчанию, можно выполнить команду `Get-Command`. Наверняка вы заметили общую закономерность: имена большинства команд сформированы по схеме *глагол-существительное*. Это уникальная черта PowerShell. Чтобы язык оставался как можно более интуитивно понятным, в Microsoft задали специальные правила образования имен команд. Хотя это не обязательный подход, его настоятельно рекомендуется соблюдать при создании собственных команд.

Команды PowerShell бывают нескольких видов: командлеты, функции, псевдонимы и иногда внешние сценарии. Большинство встроенных команд от Microsoft — это *командлеты*, которые обычно представляют собой команды из других языков, например C#. Выполнив команду `Get-Command`, как показано в листинге 1.3, вы увидите поле `CommandType`.

Листинг 1.3. Отображение типа команды Get-Alias

```
PS> Get-Command -Name Get-Alias
CommandType     Name              Version     Source
-----------     ----              -------     ------
Cmdlet          Get-Alias         3.1.0.0     Microsoft.PowerShell.Utility
```

Функции — это команды, напротив, написанные на PowerShell. Вы будете писать функции для выполнения задач, а командлеты оставьте на долю разработчиков программного обеспечения. Командлеты и функции — это наиболее распространенные типы команд, с которыми вы будете работать в PowerShell.

Вы будете использовать команду `Get-Command`, чтобы исследовать изобилие командлетов и функций, имеющихся в PowerShell. Однако как вы, возможно, уже убедились, ввод `Get-Command` без параметров заставит вас сидеть и ждать, пока консоль не переберет все возможные варианты.

У многих команд в PowerShell есть *параметры* — значения, которые вы задаете (или *передаете*) команде, чтобы задать ее поведение. Например, у `Get-Command` есть параметры, позволяющие возвращать только определенные команды вместо полного списка. В выводе `Get-Command` вы могли заметить такие распространенные глаголы, как `Get`, `Set`, `Update` и `Remove`. Если вы решили, что все команды `Get` *получают* информацию, а другие ее изменяют, то вы совершенно правы. PowerShell работает по принципу «что видишь, то и получишь». Команды имеют интуитивно понятные названия и обычно делают именно то, чего вы от них ожидаете.

Поскольку вы — новичок, вам точно не следует ничего менять в системе. Вам просто нужна информация из разных источников. С помощью параметра `Verb` в `Get-Command` можно сократить этот огромный список команд только до тех, которые, например, начинаются с `Get`. Для этого введите следующее в командной строке:

```
PS> Get-Command -Verb Get
```

Возможно, вы заметили, что команд все равно выводится многовато, поэтому можно сократить список результатов еще больше, добавив параметр `Noun`, и задать для него существительное `Content`, как показано в листинге 1.4.

Листинг 1.4. Вывод команд, содержащих глагол Get и существительное Content

```
PS> Get-Command -Verb Get -Noun Content
CommandType     Name              Version     Source
-----------     ----              -------     ------
Cmdlet          Get-Content       3.1.0.0     Microsoft.PowerShell.Management
```

Если теперь выбирать стало практически не из чего, можно использовать только параметр `Noun` без параметра `Verb`, как показано в листинге 1.5.

Листинг 1.5. Вывод команд, содержащих существительное Content

```
PS> Get-Command -Noun Content
CommandType     Name              Version   Source
-----------     ----              -------   ------
Cmdlet          Add-Content       3.1.0.0   Microsoft.PowerShell.Management
Cmdlet          Clear-Content     3.1.0.0   Microsoft.PowerShell.Management
Cmdlet          Get-Content       3.1.0.0   Microsoft.PowerShell.Management
Cmdlet          Set-Content       3.1.0.0   Microsoft.PowerShell.Management
```

Как видно из этих примеров, `Get-Command` позволяет разделять глаголы и существительные. Если вы предпочитаете определить всю команду как единое целое, вы можете использовать параметр `Name` и указать полное имя команды, как показано в листинге 1.6.

Листинг 1.6. Поиск командлета Get-Content по имени команды

```
PS> Get-Command -Name Get-Content
CommandType     Name              Version   Source
-----------     ----              -------   ------
Cmdlet          Get-Content       3.1.0.0   Microsoft.PowerShell.Management
```

Как я уже говорил ранее, у многих команд в PowerShell есть параметры, которые задают их поведение. С помощью справочной системы PowerShell можно узнать о том, какие параметры есть у команд.

Подсказки

Документация PowerShell ничуть не уникальна, но сам подход к интеграции документации и справочного содержимого в язык — это поистине произведение искусства. В этом разделе вы узнаете, как вывести в консоль справку по командам, получить больше информации о языке в разделах «О программе», а также обновлять документацию с помощью команды `Update-Help`.

Отображение документов

Подобно команде `man` в Linux, в PowerShell есть команда `help` и командлет `Get-Help`. Если вам интересно узнать, что делает один из этих командлетов `Content`, можно передать его имя команде `Get-Help`, чтобы вывести стандартные разделы справки: `SYNOPSIS`, `SYNTAX`, `DESCRIPTION`, `RELATED LINKS` и `REMARKS`. В этих разделах описываются функции команды и способы найти дополнительную информацию о ней и о связанных командах. В листинге 1.7 представлена документация по команде `Add-Content`.

Листинг 1.7. Страница справки, выводимая командой Add-Content

```
PS> Get-Help Add-Content

NAME
    Add-Content

SYNOPSIS
    Appends content, such as words or data, to a file.

--пропуск--
```

Передача только имени команды в `Get-Help` — это, конечно, практично, но в параметре `Examples` можно найти самую ценную информацию. Тут выводятся примеры реального использования команды в различных сценариях. Попробуйте ввести `Get-Help` *<ИмяКоманды>* `-Examples`: вы заметите, что почти у всех встроенных команд есть примеры, объясняющие их действие.

Сделаем это, например, для командлета `Add-Content`, как показано в листинге 1.8.

Листинг 1.8. Вывод примеров использования команды Add-Content

```
PS> Get-Help Add-Content -Examples

NAME
    Add-Content

SYNOPSIS
    Appends content, such as words or data, to a file.


    -------------------------- EXAMPLE 1 --------------------------

    C:\PS>Add-Content -Path *.txt -Exclude help* -Value "END"

    Description

    -----------

    This command adds "END" to all text files in the current directory,
    except for those with file names that begin with "help."
--пропуск--
```

Если вам нужна дополнительная информация, то у командлета `Get-Help` также есть параметры `Details` и `Full`, которые дадут вам подробное описание действий команды.

Изучение общих аспектов

В системе PowerShell помимо информации об отдельных командах есть *разделы About*, представляющие собой фрагменты справки для более широких тем и конкретных команд. Например, в этой главе вы изучаете некоторые базовые команды PowerShell. В Microsoft создали раздел About, в котором дается общее объяснение этих команд. Чтобы открыть его, введите команду `Get-Help about_Core_Commands`, как показано в листинге 1.9.

Листинг 1.9. Справка по командам ядра PowerShell

```
PS> Get-Help about_Core_Commands
TOPIC
    about_Core_Commands

SHORT DESCRIPTION
    Lists the cmdlets that are designed for use with Windows PowerShell
    providers.

LONG DESCRIPTION
    Windows PowerShell includes a set of cmdlets that are specifically
    designed to manage the items in the data stores that are exposed by Windows
    PowerShell providers. You can use these cmdlets in the same ways to manage
    all the different types of data that the providers make available to you.
    For more information about providers, type "get-help about_providers".

    For example, you can use the Get-ChildItem cmdlet to list the files in a
    file system directory, the keys under a registry key, or the items that
    are exposed by a provider that you write or download.
    The following is a list of the Windows PowerShell cmdlets that are designed
    for use with providers:

--пропуск--
```

Чтобы получить полный список справок About, используйте подстановочный знак для параметра `Name`. *Подстановочный знак* в PowerShell — это звездочка (`*`). Она может использоваться в качестве заполнителя для нуля или более символов. Вы можете использовать подстановочный знак с параметром `Name` команды `Get-Help`, как показано в листинге 1.10.

Листинг 1.10. Использование подстановочного знака в параметре Name команды Get-Help

```
PS> Get-Help -Name About*
```

Добавляя подстановочный знак к команде `About`, вы просите PowerShell искать все возможные темы, которые начинаются с *About*. Если совпадений окажется

несколько, PowerShell выведет список с краткой информацией о каждом. Чтобы получить полную информацию по одному из вариантов, необходимо передать его напрямую в `Get-Help`, как было показано ранее в листинге 1.9.

Хотя у `Get-Help` есть параметр `Name`, можно передать аргумент параметра непосредственно команде, введя `-Name`, как показано в листинге 1.10. Это действие известно как использование *позиционного параметра*, который определяет передаваемое значение на основе его (как вы уже догадались) позиции в команде. Позиционные параметры — это клише, имеющееся во многих командах PowerShell и позволяющее вам меньше нажимать на клавиши.

ЗАПУСК POWERSHELL ОТ ИМЕНИ АДМИНИСТРАТОРА

Иногда приходится запускать консоль PowerShell от имени *администратора*. Обычно это происходит, когда вам нужно изменить файлы, значения реестра или что-то еще, что находится за пределами вашего профиля пользователя. Например, упомянутая ранее команда `Update-Help` меняет файлы системного уровня и не может быть корректно выполнена пользователем, не являющимся администратором.

Вы можете запустить PowerShell от имени администратора, щелкнув правой кнопкой мыши в Windows PowerShell и выбрав пункт «Запуск от имени администратора», как показано на рис. 1.2.

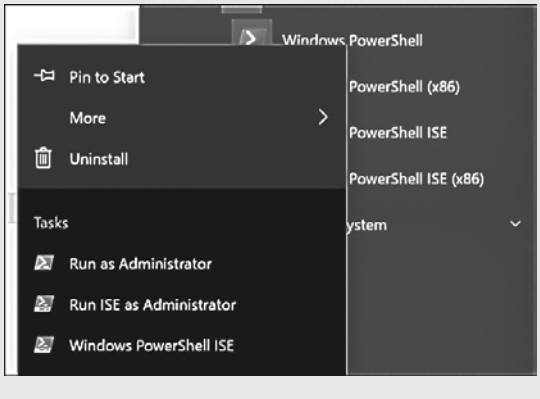


Рис. 1.2. Запуск PowerShell от имени администратора

Обновление документов

Справочная система в PowerShell — отличная вещь для всех, кто хочет узнать больше о языке, но одно ключевое качество делает ее еще лучше: она динамична! Любая документация со временем устаревает. Она изначально поставляется с продуктом, в работе которого со временем появляются ошибки, выпускаются новые функции. При этом документация остается прежней. В PowerShell эта проблема решена с помощью *обновляемой справки*, которая позволяет встроенным и сторонним командлетам или функциям ссылаться на URI в интернете, где размещается актуальная документация. Просто введите команду `Update-Help`, и PowerShell начнет анализировать справку по вашей системе и сравнивать ее с различными онлайн-ресурсами.

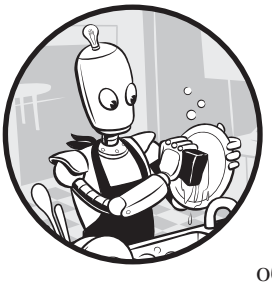
Обратите внимание: хотя обновляемая справка включена во все встроенные командлеты PowerShell, для сторонних команд она не требуется. Кроме того, актуальность документации зависит от разработчика. PowerShell предоставляет разработчикам инструменты для создания более качественного справочного контента, но это не отменяет необходимости поддерживать актуальность репозитория, где хранятся файлы справки. Наконец, иногда при запуске команды `Update-Help` вы можете получать сообщение об ошибке, если расположение, в котором хранится справка, больше не доступно. Короче говоря, не стоит ожидать, что `Update-Help` *всегда* будет выдавать вам последнюю справку по каждой команде в PowerShell.

Итоги

Из этой главы вы узнали о нескольких командах, которые помогут вам начать работу с PowerShell. Начиная что-то новое, вы не знаете заранее, чего вам пока недостает. Для самостоятельного изучения темы вам нужна крупица знаний. Освоив основы команд PowerShell и то, как можно использовать `Get-Command` и `Get-Help`, вы получили инструменты, необходимые для начала изучения PowerShell. Впереди вас ждет большое увлекательное путешествие!

2

Основные понятия PowerShell



В этой главе рассматриваются четыре основных понятия, связанных с PowerShell: переменные, типы данных, объекты и структуры данных. Эти понятия фундаментальны практически для каждого распространенного языка программирования, но в PowerShell есть еще кое-что, отличающее его от других языков: в нем все является объектом.

Возможно, сейчас для вас это не имеет особого значения, но держите в уме эту особенность, пока будете читать эту главу. Под конец вы поймете, насколько это важно.

Переменные

Переменная — это место для хранения *значений*. Можно представить переменную как некий цифровой ящик. Например, можно поместить некоторое значение в этот ящик, если нужно использовать его несколько раз. Так вам не придется снова и снова вводить в код одно и то же число — можно просто поместить его в переменную и вызывать ее всякий раз, когда вам требуется значение. Но, как вы могли догадаться из названия, настоящая сила переменных заключается в их способности изменяться: вы можете класть в этот ящик всякие штуки, заменять, просматривать и показывать их, прежде чем положить обратно.

Как вы позже убедитесь, это свойство позволяет создавать код, работающий в различных ситуациях, а не прописанный под один конкретный сценарий. Этот раздел описывает основные способы использования переменных.

Отображение и изменение переменной

Все переменные в PowerShell начинаются со знака доллара (`$`), который сообщает PowerShell, что вы вызываете именно переменную, а не командлет, функцию, файл сценария или исполняемый файл. Например, если вы хотите вывести значение переменной `MaximumHistoryCount`, нужно приписать к ее названию знак доллара и вызвать ее, как показано в листинге 2.1.

Листинг 2.1. Вызов переменной $MaximumHistoryCount

```
PS> $MaximumHistoryCount
4096
```

`$MaximumHistoryCount` — это встроенная переменная, которая определяет максимальное количество команд, которое PowerShell сохраняет в своей истории. По умолчанию оно равно 4096.

Вы можете изменить значение переменной: введите ее имя, начиная со знака доллара, а затем используйте знак равенства (`=`) и введите новое значение, как показано в листинге 2.2.

Листинг 2.2. Изменение значения переменной $MaximumHistoryCount

```
PS> $MaximumHistoryCount = 200
PS> $MaximumHistoryCount
200
```

Мы изменили значение переменной `$MaximumHistoryCount` на `200`, и теперь PowerShell будет хранить в своей истории только предыдущие 200 команд.

В листингах 2.1 и 2.2 используется уже существующая переменная. В целом, переменные в PowerShell делятся на две большие группы: определяемые пользователем и встроенные. Давайте сначала рассмотрим первую из них.

Пользовательские переменные

Использовать можно только существующие переменные. Попробуйте ввести в консоль PowerShell команду `$color`, как показано в листинге 2.3.

Листинг 2.3. Обращение к не определенной ранее переменной вызывает ошибку

```
PS> $color
The variable '$color' cannot be retrieved because it has not been set.

At line:1 char:1
+ $color
+ ~~~~
     + CategoryInfo          : InvalidOperation: (color:String) [],
RuntimeException
     + FullyQualifiedErrorId : VariableIsUndefined
```

ВКЛЮЧЕНИЕ СТРОГОГО РЕЖИМА

Если после выполнения листинга 2.3 вы не получили сообщение об ошибке, и в консоли не выводится никаких данных, попробуйте выполнить следующую команду, чтобы включить строгий режим:

```
PS> Set-StrictMode -Version Latest
```

Включение этого режима заставляет PowerShell выдавать информацию об ошибке, если вы нарушаете правила написания кода. Например, строгий режим вынуждает PowerShell возвращать ошибку при обращении к несуществующему свойству объекта или к не определенной ранее переменной. Считается, что лучше включать этот режим при написании сценариев, поскольку так вы будете писать более чистый и предсказуемый код. При простом запуске интерактивного кода из консоли PowerShell эта функция обычно не используется. Для получения дополнительной информации о строгом режиме запустите команду `Get Help Set-StrictMode Examples`.

В листинге 2.3 вы попытались обратиться к переменной `$color` еще до того, как она была создана, что и привело к ошибке. Чтобы создать переменную, нужно *объявить* ее, то есть сказать системе, что она существует, а затем *присвоить* ей значение (иначе говоря, *инициализировать* ее). Можно выполнить оба эти действия одновременно, как показано в листинге 2.4, где создается переменная `$color` со значением `blue`. Вы можете присвоить значение переменной с помощью того же метода, что и для изменения значения `$MaximumHistoryCount`: для этого нужно ввести имя переменной, затем знак равенства, а после — значение.

Листинг 2.4. Создание переменной color со значением blue

```
PS> $color = 'blue'
```

Когда переменная создана и получила значение, вы сможете ссылаться на нее, введя имя переменной в консоли (листинг 2.5).

Листинг 2.5. Проверка значения переменной

```
PS> $color
blue
```

Значение переменной не изменится, если что-то или кто-то не инициирует изменение явным образом. Можно вызывать переменную `$color` множество раз, и она всегда будет возвращать значение `blue`, пока не будет переопределена.

Когда вы используете знак равенства для определения переменной (листинг 2.4), вы делаете то же самое, что и команда `Set-Variable`. Аналогично, когда вы вводите в консоли переменную и получаете значение как в листинге 2.5, вы делаете то же, что и с помощью команды `Get-Variable`. В листинге 2.6 воспроизведены листинги 2.4 и 2.5, но с использованием этих команд.

Листинг 2.6. Создание переменной и отображение ее значения с помощью команд Set-Variable и Get-Variable

```
PS> Set-Variable -Name color -Value blue

PS> Get-Variable -Name color

Name                           Value
----                           -----
Color                          blue
```

Вы также можете использовать переменную `Get-Variable`, чтобы вывести все доступные переменные (как показано в листинге 2.7).

Листинг 2.7. Использование команды Get-Variable для вывода всех переменных

```
PS> Get-Variable
Name                           Value
----                           -----
$                              Get-PSDrive
?                              True
^                              Get-PSDrive
args                           {}
color                          blue
--пропуск--
```

Эта команда выводит список всех переменных, в настоящее время находящихся в памяти. Обратите внимание, что некоторые из них вы не определяли. О них поговорим в следующем разделе.

Встроенные переменные

Ранее я говорил, что существуют встроенные переменные, предварительно созданные в PowerShell. Хотя у вас есть возможность изменять значения некоторых из них, как в листинге 2.2, я обычно не рекомендую этого делать, поскольку последствия такого шага могут быть непредсказуемыми.

В общем, со встроенными переменными стоит обращаться как с *доступными только для чтения*. (Возможно, сейчас самое время вернуть переменной `$MaximumHistoryCount` прежнее значение 4096!)

В этом разделе рассматривается несколько автоматических переменных, которые вы, вероятно, будете использовать: `$null`, `$LASTEXITCODE` и переменные настройки.

Переменная $null

Переменная `$null` немного странная: она ничего не представляет. Присвоение значения `$null` другой переменной позволяет создать эту переменную, но не присваивает ей реального значения, как показано в листинге 2.8.

Листинг 2.8. Присвоение переменной значения $null

```
PS> $foo = $null
PS> $foo
PS> $bar
The variable '$bar' cannot be retrieved because it has not been set.
At line:1 char:1
+ $bar
+ ~~~~
     + CategoryInfo          : InvalidOperation: (bar:String) [], RuntimeException
     + FullyQualifiedErrorId : VariableIsUndefined
```

Здесь мы назначили переменной `$foo` значение `$null`. Когда мы попробуем вызвать переменную `$foo`, на экране ничего не отобразится, но и ошибки не появится, поскольку PowerShell знает такую переменную.

Вы можете увидеть, какие переменные распознает PowerShell, передав их в параметр команды `Get-Variable`. В листинге 2.9 видно, что PowerShell знает переменную `$foo`, но не знает переменную `$bar`.

Вы можете спросить: а зачем нам значение `$null`? Оказывается, оно на удивление удобно. Например, как вы далее убедитесь, мы нередко присваиваем

переменной это значение в виде ответа на что-нибудь, например, на результат выполнения определенной функции. Если при проверке переменной ее значение по-прежнему равно `$null`, значит, в функции что-то пошло не так и нужно исправить ошибку.

Листинг 2.9. Использование Get-Variable для поиска переменных

```
PS> Get-Variable -Name foo
Name                           Value
----                           -----
foo


PS> Get-Variable -Name bar
Get-Variable : Cannot find a variable with the name 'bar'.
At line:1 char:1
+ Get-Variable -Name bar
+ ~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (bar:String) [Get-Variable],
                              ItemNotFoundException
    + FullyQualifiedErrorId :
                 VariableNotFound,Microsoft.PowerShell.Commands.GetVariableCommand
```

Переменная LASTEXITCODE

Еще одна часто используемая встроенная переменная — `$LASTEXITCODE`. PowerShell позволяет запускать внешние исполняемые приложения вроде старого доброго ping.exe, которое проверяет отклик веб-сайта. Когда внешние приложения завершают работу, они заканчиваются с *кодом выхода*, или *кодом возврата*, который имеет определенное значение. Как правило, 0 означает удачный исход, а что-либо другое — сбой или аномалию. Для приложения ping.exe ответ 0 означает, что связь с узлом удалось успешно проверить, а 1 — что ничего не получилось.

При запуске приложения ping.exe, как показано в листинге 2.10, вы увидите ожидаемый результат, но не код выхода. Дело в том, что код выхода скрыт внутри переменной `$LASTEXITCODE`. В переменную `$LASTEXITCODE` всегда записывается код выхода последнего выполненного приложения. Код в листинге 2.10 проверяет связь с google.com, возвращает код выхода, а затем проверяет связь с несуществующим доменом и снова возвращает код выхода.

Значение `$LASTEXITCODE` равняется 0, когда вы пингуете google.com, и 1, когда пингуете несуществующий сайт dfdfdfdfd.com.

Листинг 2.10. Использование ping.exe для демонстрации переменной $LASTEXITCODE

```
PS> ping.exe -n 1 dfdfdfdfd.com

Pinging dfdfdfdfd.com [14.63.216.242] with 32 bytes of data:
Request timed out.

Ping statistics for 14.63.216.242:
    Packets: Sent = 1, Received = 0, Lost = 1 (100% loss),
PS> $LASTEXITCODE
1
PS> ping.exe -n 1 google.com

Pinging google.com [2607:f8b0:4004:80c::200e] with 32 bytes of data:
Reply from 2607:f8b0:4004:80c::200e: time=47ms

Ping statistics for 2607:f8b0:4004:80c::200e:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 47ms, Maximum = 47ms, Average = 47ms
PS> $LASTEXITCODE
0
```

Переменные предпочтений

В PowerShell есть встроенные переменные, называемые *переменными предпочтений*. Они управляют поведением по умолчанию различных выходных потоков: `Error`, `Warning`, `Verbose`, `Debug` и `Information`.

Чтобы вывести список всех переменных предпочтения, запустите команду `Get-Variable` для всех переменных, оканчивающихся на `Preference`, как показано ниже:

```
PS> Get-Variable -Name *Preference
Name                           Value
----                           -----
ConfirmPreference              High
DebugPreference                SilentlyContinue
ErrorActionPreference          Continue
InformationPreference          SilentlyContinue
ProgressPreference             Continue
VerbosePreference              SilentlyContinue
WarningPreference              Continue
WhatIfPreference               False
```

Эти переменные можно использовать для настройки выводов PowerShell. Например, если вы когда-либо совершали ошибку и в результате видели страшный красный текст, то это был выходной поток `Error`. Выполните следующую команду, чтобы сгенерировать сообщение об ошибке:

```
PS> Get-Variable -Name 'doesnotexist'
Get-Variable : Cannot find a variable with the name 'doesnotexist'.
At line:1 char:1
+ Get-Variable -Name 'doesnotexist'
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (doesnotexist:String) [Get-Variable],
                              ItemNotFoundException
    + FullyQualifiedErrorId :
                    VariableNotFound,Microsoft.PowerShell.Commands.
GetVariableCommand
```

Вы должны были получить аналогичное сообщение, так как для потока ошибок такое поведение задано по умолчанию. Если вы по какой-либо причине предпочли бы не видеть ничего вместо такого текста, можно задать переменной `$ErrorActionPreference` значение `SilentlyContinue` или `Ignore`, и тогда PowerShell не будет выводить текст ошибки:

```
PS> $ErrorActionPreference = 'SilentlyContinue'
PS> Get-Variable -Name 'doesnotexist'
PS>
```

Как вы видите, текст ошибки не выводится. Игнорирование ошибок обычно не приветствуется, поэтому, прежде чем продолжить работу, снова измените значение `$ErrorActionPreference` на `Continue`. Для получения дополнительных сведений о переменных предпочтений ознакомьтесь с содержимым `about_help`, запустив команду **`Get-Help about_Preference_Variables`**.

Типы данных

Переменные в PowerShell бывают разных форм или *типов*. Подробный разговор о типах данных PowerShell выходит за рамки этой главы. Вам достаточно знать, что в PowerShell есть несколько типов данных, к которым относятся логические значения, строки и целые числа, и что можно изменять тип данных переменной без получения сообщений об ошибках. Приведенный ниже код должен работать без ошибок:

```
PS> $foo = 1
PS> $foo = 'one'
PS> $foo = $true
```

Дело в том, что PowerShell может определять типы данных, основываясь на переданном значении переменной. Внутреннее устройство этого процесса слишком сложно и потому не приводится в этой книге, но вам следует знать основные типы данных и принципы их взаимодействия.

Логические значения

Практически в каждом языке программирования есть *логические значения* True или False (1 или 0, истина или ложь). Логические значения используются для обозначения некоторых двоичных вещей, например, включен свет или нет. В PowerShell логические значения называются *bool* и представлены двумя встроенными переменными — `$true` и `$false`. Эти встроенные переменные неизменны и жестко закодированы в PowerShell. В листинге 2.11 показано, как задать переменной значение `$true` или `$false`.

Листинг 2.11. Создание переменной типа bool

```
PS> $isOn = $true
PS> $isOn
True
```

В главе 4 вы найдете гораздо больше логических значений.

Целые числа и числа с плавающей точкой

Числа в PowerShell представляются двумя основными способами — в виде целых чисел и чисел с плавающей точкой.

Целочисленные типы

Целочисленные типы данных содержат только целые числа, а дробные числа округляются до ближайшего целого числа. Целочисленные типы данных бывают *знаковыми* и *беззнаковыми*. Знаковые типы данных могут хранить как положительные, так и отрицательные числа. В беззнаковых типах хранятся значения без знака.

По умолчанию PowerShell хранит целые числа в виде 32-разрядного типа `Int32` со знаком. Количество разрядов определяет, насколько большое (или маленькое) число может хранить переменная. В нашем случае это любое значение в диапазоне от –2 147 483 648 до 2 147 483 647. Для чисел вне этого диапазона можно использовать 64-битный знаковый тип `Int64`, который имеет диапазон от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807.

В листинге 2.12 показан пример того, как PowerShell работает с типами `Int32`.

Давайте рассмотрим каждый шаг. Пока не думайте о синтаксисе, а смотрите на выходные данные. Сначала мы создаем переменную `$num` и присваиваем ей значение 1 ❶. Затем мы проверяем тип `$num` ❷ и видим, что PowerShell

интерпретирует 1 как `Int32`. Затем задаем `$num` десятичное значение ❸ и еще раз проверяем тип, но теперь PowerShell считает переменную как тип `Double`. Это связано с тем, что PowerShell изменяет тип переменной в зависимости от ее значения. Однако можно обрабатывать переменную как имеющую строго определенный тип, *преобразовав* ее с помощью синтаксиса `[Int32]` перед именем `$num` ❹. Как видите, теперь PowerShell вынужден рассматривать 1,5 как целое число и потому округляет его до 2.

Листинг 2.12. Использование типа Int для хранения различных значений

```
❶ PS> $num = 1
  PS> $num
  1
❷ PS> $num.GetType().name
  Int32
❸ PS> $num = 1.5
  PS> $num.GetType().name
  Double
❹ PS> [Int32]$num
  2
```

Теперь посмотрим на тип `Double`.

Типы с плавающей точкой

Тип `Double` относится к более широкому классу переменных, известных как переменные *с плавающей точкой*. Его можно использовать для представления целых чисел, но чаще всего переменные с плавающей точкой используются для десятичных. Другой основной тип переменных с плавающей точкой — `Float`. Я не буду вдаваться во внутреннее устройство типов `Float` и `Double`. Вам важно знать, что хотя типы `Float` и `Double` и подходят для дробных чисел, они могут быть неточными, что показано в листинге 2.13.

Листинг 2.13. Ошибки точности у типов с плавающей точкой

```
PS> $num = 0.1234567910
PS> $num.GetType().name
Double
PS> $num + $num
0.2469135782
PS> [Float]$num + [Float]$num
0.246913582086563
```

Как видите, по умолчанию PowerShell использует тип `Double`. Однако обратите внимание на то, что происходит, когда вы прибавляете переменную `$num` к самой себе, приведя обе к типу `Float`, — вы получаете странный ответ. Опять

же, в этой книге мы не будем вдаваться в причины, но имейте в виду, что при использовании `Float` и `Double` могут возникать подобные ошибки.

Строки

Этот тип переменных мы уже видели. Когда мы определяли переменную `$color` в листинге 2.4, мы не просто писали `$color = blue`, а заключали значение в одинарные кавычки. Это указывает PowerShell на то, что значение представляет собой набор букв, или *строку*. Если вы попытаетесь присвоить значение `blue` без кавычек, PowerShell вернет ошибку:

```
PS> $color = blue
blue : The term 'blue' is not recognized as the name of a cmdlet, function, script
file, or operable program. Check the spelling of the name, or if a path was
included, verify that the path is correct and try again.
At line:1 char:10
+ $color = blue
+          ~~~~
    + CategoryInfo      : ObjectNotFound: (blue:String) [], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
```

При отсутствии кавычек PowerShell интерпретирует слово `blue` как команду и пытается ее выполнить. Поскольку такой команды не существует, PowerShell возвращает сообщение об ошибке. Чтобы правильно определить строку, вам необходимо заключить ее значение в кавычки.

Объединение строк и переменных

В строках могут быть не только слова, но также фразы и предложения. Например, вы можете присвоить переменной `$sentence` такое значение:

```
PS> $sentence = "Today, you learned that PowerShell loves the color blue"
PS> $sentence
Today, you learned that PowerShell loves the color blue
```

Возможно, вам понадобится использовать то же предложение, но со словами *PowerShell* и *blue* в качестве значений переменных. Например, что если у вас есть переменные `$name`, `$language` и `$color`? В листинге 2.14 они определены с использованием других переменных.

Листинг 2.14. Вставка переменных в строки

```
PS> $language = 'PowerShell'
PS> $color = 'blue'

PS> $sentence = "Today, you learned that $language loves the color $color"
```

```
PS> $sentence
Today, you learned that PowerShell loves the color blue
```

Обратите внимание на использование двойных кавычек. Заключение предложения в одинарные кавычки не дает ожидаемого результата:

```
PS> 'Today, $name learned that $language loves the color $color'
Today, $name learned that $language loves the color $color
```

И это не просто непонятная ошибка. В PowerShell существует важное различие между одинарными и двойными кавычками.

Использование двойных и одинарных кавычек

Если вы задаете значение переменной в виде простой строки, можно использовать одинарные или двойные кавычки, как показано в листинге 2.15.

Листинг 2.15. Изменение значений переменных с помощью одинарных и двойных кавычек

```
PS> $color = "yellow"
PS> $color
yellow
PS> $color = 'red'
PS> $color
red
PS> $color = ''
PS> $color
PS> $color = "blue"
PS> $color
blue
```

Как видите, для простых строк не имеет значения, какие кавычки использовать при определении. Так почему это было важно, когда в нашей строке были переменные? Ответ связан с понятием *интерполяции переменных*, или *расширения переменных*. Обычно, когда вы вводите в консоль `$color` и нажимаете ENTER, PowerShell *интерполирует*, или *расширяет*, эту переменную. Эти сложные термины означают, что PowerShell считывает значение внутри переменной, то есть «открывает ящик», чтобы вы могли заглянуть внутрь него. Когда вы используете двойные кавычки для вызова переменной, происходит то же самое: переменная открывается, что видно в листинге 2.16.

Листинг 2.16. Поведение переменной внутри строки

```
PS> "$color"
blue
PS> '$color'
$color
```

Однако обратите внимание на то, что происходит при использовании одинарных кавычек: консоль выводит саму переменную, а не ее значение. Одиночные кавычки сообщают PowerShell, что вы имеете в виду *именно то*, что вы вводите, будь то слово вроде *blue* или переменная с названием `$color`. Для PowerShell это не имеет значения — что написано в кавычках, то и выводится. Поэтому, когда вы используете переменную в одинарных кавычках, PowerShell не знает, как расширить ее значение. Вот почему следует использовать двойные кавычки при вставке переменных в составные строки.

О логических значениях, целых числах и строках можно сказать гораздо больше. Но пока давайте немного притормозим и рассмотрим нечто более общее — объекты.

Объекты

В PowerShell *всё* является объектом. С технической точки зрения *объект* — это отдельный экземпляр определенного шаблона, который называется классом. *Класс* определяет, что именно объект содержит внутри себя. Класс объекта определяет его *методы*, то есть действия, которые можно производить с этим объектом. Другими словами, методы — это то, что объект умеет. Например, у объекта списка может быть метод `sort()`, который сортирует список при вызове. Аналогично, класс объекта определяет его *атрибуты*, то есть переменные объекта. Атрибуты — это вся информация об объекте. Если говорить о списках, то у вас может быть атрибут `length`, в котором хранится количество элементов в списке. В определении класса иногда заданы значения атрибутов по умолчанию, но чаще всего вы сами их задаете.

Но все это очень абстрактно. Рассмотрим более конкретный пример: автомобиль. На этапе проектирования автомобиль — это просто план. Он определяет то, как автомобиль должен выглядеть, какой у него должен быть двигатель, шасси и т. д. В плане также указывается, что автомобиль должен уметь делать после завершения сборки — двигаться вперед и задним ходом, а также открывать и закрывать панорамную крышу. Этот план — своего рода класс автомобиля.

Каждый автомобиль происходит из класса, и к нему добавляются все свойства и методы класса. Один автомобиль может быть синим, другой автомобиль той же модели — красным, а третий может иметь другую трансмиссию. Эти атрибуты являются свойствами отдельно взятого экземпляра. При этом все они по-прежнему будут уметь двигаться вперед и назад, а также

открывать и закрывать панорамную крышу. Эти действия — методы автомобиля.

Теперь, когда вы представляете, как работают объекты, давайте уже испачкаем руки и поработаем с PowerShell.

Проверка атрибутов

Для начала давайте создадим простой объект, чтобы вы могли его разобрать и открыть для себя множество сторон объектов PowerShell. В листинге 2.17 мы создадим простой строковый объект с именем `$color`.

Листинг 2.17. Создание строкового объекта

```
PS> $color = 'red'
PS> $color
red
```

Обратите внимание, что при вызове переменной `$color` выводится только ее значение. Но переменные, поскольку они являются объектами, обычно содержат больше информации помимо значения. У них тоже есть атрибуты.

Чтобы просмотреть атрибуты объекта, воспользуйтесь командой `Select-Object` и параметром `Property`. Параметру `Property` передадим звездочку, как показано в листинге 2.18, чтобы PowerShell выдал все найденные значения.

Листинг 2.18. Исследование атрибутов объекта

```
PS> Select-Object -InputObject $color -Property *

Length
------
     3
```

Как видите, у строки `$color` есть только один атрибут — `Length`.

Вы можете напрямую ссылаться на атрибут `Length`, используя запись через точку: пишется имя объекта, точка, а затем имя атрибута, к которому вы хотите получить доступ (см. листинг 2.19).

Листинг 2.19. Использование точечной записи для проверки атрибута объекта

```
PS> $color.Length
3
```

Такое использование атрибутов скоро станет вам как родное.

Использование командлета Get-Member

При использовании команды `Select-Object` вы обнаружили, что у строки `$color` есть всего один атрибут. Но мы помним, что у объектов иногда бывают еще и методы. Чтобы увидеть все методы и атрибуты строкового объекта, воспользуйтесь командлетом `Get-Member` (листинг 2.20). Этот командлет станет вашим лучшим другом на долгое время. С его помощью можно быстро вывести список всех свойств и методов конкретного объекта, которые вместе называются *элементами* объекта.

Листинг 2.20. Использование команды Get-Member для исследования свойств и методов объекта

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

   TypeName: System.String

Name          MemberType        Definition
----          ----------        ----------
Clone         Method            System.Object Clone(), System.Object
                                ICloneable.Clone()
CompareTo     Method            int CompareTo(System.Object value),
                                int CompareTo(string strB), int IComparab...
Contains      Method            bool Contains(string value)
CopyTo        Method            void CopyTo(int sourceIndex, char[] destination,
                                int destinationIndex, int co...
EndsWith      Method            bool EndsWith(string value),
                                bool EndsWith(string value,
                                System.StringCompari...
Equals        Method            bool Equals(System.Object obj),
                                bool Equals(string value), bool Equals(string...
--пропуск--
Length        Property          int Length {get;}
```

А вот это уже другой разговор! Оказывается, с вашим простым строковым объектом связано довольно много методов. Вообще их гораздо больше, но здесь показаны не все. Количество методов и свойств объекта зависит от его родительского класса.

Вызов методов

Вы можете ссылаться на методы с помощью точечной записи. Однако в отличие от атрибутов, название метода всегда сопровождается набором открывающих и закрывающих круглых скобок и может включать один или несколько параметров.

Например, предположим, что вы хотите удалить символ из переменной `$color`. Сделать это можно с помощью метода `Remove()`. Давайте выделим метод `Remove()` переменной `$color` с помощью кода из листинга 2.21.

Листинг 2.21. Демонстрация строкового метода Remove()

```
PS> Get-Member -InputObject $color –Name Remove
Name   MemberType Definition
----   ---------- ----------
Remove Method     string Remove(int startIndex, int count),
                  string Remove(int startIndex)
```

Как видите, у нас есть два определения. Это означает, что метод можно использовать двумя способами: либо с параметрами `startIndex` и `count`, либо с одним только `startIndex`.

Итак, чтобы удалить второй символ из переменной `$color`, нужно написать номер символа, с которого вы хотите начать удаление — этот номер называется *индексом*. Индексы начинаются с 0, поэтому у первой буквы индекс 0, у второй — 1 и т. д. Наряду с индексом вы можете указать количество символов, которые нужно удалить, используя запятую для разделения аргументов, как показано в листинге 2.22.

Листинг 2.22. Вызов методов

```
PS> $color.Remove(1,1)
Rd
PS> $color
red
```

С помощью индекса 1 вы указали PowerShell, что хотите удалить часть символов начиная со второго. Второй аргумент ставит задачу удалить только один символ. То есть в результате получится `Rd`. Обратите внимание, что метод `Remove()` не изменяет значение строковой переменной навсегда. Если вы хотите сохранить это изменение, нужно присвоить вывод метода `Remove()` переменной, как показано в листинге 2.23.

Листинг 2.23. Присвоение вывода метода Remove() строке

```
PS> $newColor = $color.Remove(1,1)
PS> $newColor
Rd
```

В этих примерах мы использовали один из простейших типов объектов — строку. В следующем разделе мы познакомимся с более сложными объектами.

ПРИМЕЧАНИЕ

Если вы хотите узнать, возвращает ли метод объект (как это делает метод `Remove()`) или же он изменяет существующий объект, можно посмотреть его описание. Как видно в листинге 2.21, перед определением `Remove()` стоит слово `string`. Это означает, что функция возвращает новую строку. Функции со словом `void`, стоящим в начале, обычно изменяют существующие объекты. В главе 6 мы поговорим об этом более подробно.

Структуры данных

Структура данных — это способ организовать несколько элементов данных. Как и сами данные, их структуры в PowerShell являются объектами, хранящимися в переменных. Они бывают трех основных типов: массивы, объекты ArrayLists и хеш-таблицы.

Массивы

До сих пор я описывал переменную как ящик. Но если простая переменная (например, `Float`) — это один ящик, то *массив* представляет собой целый набор ящиков, склеенных вместе, — список элементов, представленных одной переменной.

Часто приходится использовать несколько связанных переменных — скажем, стандартный набор цветов. Вместо того чтобы хранить каждый цвет как отдельную строку и затем ссылаться на каждую их них, гораздо эффективнее хранить все эти цвета в единой структуре данных. В этом разделе показано, как создавать, считывать, изменять и расширять массивы.

Определение массивов

Давайте сперва определим переменную под названием `$colorPicker` и присвоим ей массив, содержащий четыре цвета в виде строк. Для этого используется знак «собака» (`@`), за которым следуют четыре строки (разделенные запятыми) в круглых скобках, как показано в листинге 2.24.

Знак `@`, за которым следует открывающая скобка и ноль или более элементов, разделенных запятой, сообщает PowerShell, что вы хотите создать массив.

Листинг 2.24. Создание массива

```
PS> $colorPicker = @('blue','white','yellow','black')
PS> $colorPicker
blue
white
yellow
black
```

Обратите внимание, что после вызова переменной `$colorPicker` PowerShell выводит каждый элемент массива на новой строке. В следующем разделе вы узнаете, как обращаться к каждому элементу по отдельности.

Чтение элементов массива

Чтобы получить доступ к элементу в массиве, нужно указать имя массива, за которым следует пара квадратных скобок (`[]`), где указывается индекс нужного элемента. Как и в случае со строковыми символами, нумерация начинается с 0, поэтому у первого элемента индекс 0, у второго — 1 и т. д. В PowerShell при использовании индекса –1 возвращается последний элемент.

В листинге 2.25 мы обратимся к нескольким элементам в нашем массиве.

Листинг 2.25. Считывание элементов массива

```
PS> $colorPicker[0]
blue
PS> $colorPicker[2]
yellow
PS> $colorPicker[3]
black
PS> $colorPicker[4]
Index was outside the bounds of the array.
At line:1 char:1
+ $colorPicker[4]
+ ~~~~~~~~~~~~~~~
    + CategoryInfo          : OperationStopped: (:) [], IndexOutOfRangeException
    + FullyQualifiedErrorId : System.IndexOutOfRangeException
```

Если вы попытаетесь указать индекс, которого нет в массиве, PowerShell выдаст ошибку.

Чтобы получить доступ к нескольким элементам в массиве одновременно, можно использовать *оператор диапазона* (`..`) между двумя числами. При его использовании PowerShell вернет эти два числа и все, что находятся между ними, например:

```
PS> 1..3
1
2
3
```

Чтобы использовать оператор диапазона для доступа к нескольким элементам в массиве, можно задать диапазон индексов, как показано ниже:

```
PS> $colorPicker[1..3]
white
yellow
black
```

Теперь, когда вы узнали, как получить доступ к элементам в массиве, давайте посмотрим, как их изменить.

Изменение элементов в массиве

Если вы хотите изменить элемент в массиве, не нужно переопределять весь массив. Вместо этого вы можете сослаться на элемент с помощью индекса и использовать знак равенства для присвоения нового значения, как показано в листинге 2.26.

Листинг 2.26. Изменение элементов в массиве

```
PS> $colorPicker[3]
black
PS> $colorPicker[3] = 'white'
PS> $colorPicker[3]
white
```

Внимательно проверьте правильность индекса, выведя элемент в консоль, прежде чем изменять его.

Добавление элементов в массив

Вы можете добавлять элементы в массив с помощью оператора сложения (+), как показано в листинге 2.27.

Листинг 2.27. Добавление одного элемента в массив

```
PS> $colorPicker = $colorPicker + 'orange'
PS> $colorPicker
blue
white
yellow
white
orange
```

Обратите внимание, что мы ввели `$colorPicker` по обе стороны от знака равенства. Это связано с тем, что вы просите PowerShell раскрыть переменную `$colorPicker`, а затем добавляете новый элемент.

Метод `+` вполне рабочий, но есть и более быстрый и удобный способ. Можно использовать знаки сложения и равенства вместе, создавая оператор `+=` (листинг 2.28).

Листинг 2.28. Использование оператора «+=» для добавления элемента в массив

```
PS> $colorPicker += 'brown'
PS> $colorPicker
blue
white
yellow
white
orange
brown
```

С помощью оператора `+=` *этот элемент добавляется в существующий массив*. Такое сокращение избавляет вас от необходимости вводить имя массива дважды. Этот способ гораздо более распространен, чем использование полного синтаксиса.

Также можно добавлять одни массивы к другим. Предположим, вы хотите добавить розовый и голубой цвета в массив `$colorPicker`. В листинге 2.29 мы определяем другой массив с этими двумя цветами и добавляем их так же, как и в листинге 2.28.

Листинг 2.29. Одновременное добавление нескольких элементов в массив

```
PS> $colorPicker += @('pink','cyan')
PS> $colorPicker
blue
white
yellow
white
orange
brown
pink
cyan
```

Добавление нескольких элементов одновременно может сэкономить много времени, особенно если вы создаете массив с большим количеством элементов. Обратите внимание, что PowerShell обрабатывает любой набор значений, разделенных запятыми, как массив, поэтому знак `@` или круглые скобки уже не нужны.

К сожалению, для удаления элемента из массива нет эквивалента оператора +=. Удаление элементов работает сложнее, чем вы думаете, и поэтому мы не будем пока на этом останавливаться. Чтобы понять почему, читайте дальше!

Объекты ArrayList

При добавлении элементов в массив происходит нечто странное. Каждый раз, когда вы это делаете, вы фактически создаете новый массив из вашего старого (интерполированного) массива и нового элемента. То же самое происходит, когда вы удаляете элемент из массива: PowerShell уничтожает ваш старый массив и создает новый. Это связано с тем, что все массивы в PowerShell имеют фиксированный размер. Нельзя напрямую изменить размер существующего массива, поэтому приходится создавать новый. Когда вы будете работать с небольшими массивами вроде рассмотренных ранее, вы даже не заметите этого. Но при работе с *огромными* массивами, состоящими из десятков или сотен тысяч элементов, скачки в производительности будут довольно значительными.

Когда нужно удалить или добавить множество элементов в массив, я предлагаю использовать другую структуру данных под названием *ArrayList*. Объекты ArrayList ведут себя практически так же, как и обычный массив PowerShell, но с одним важным отличием — у них нет фиксированного размера. Они могут динамически подстраиваться под добавленные или удаленные элементы, обеспечивая тем самым гораздо более высокую производительность при работе с большими объемами данных.

Определение объекта ArrayList выглядит точно так же, как определение массива. Единственное, нужно напрямую указать, что это именно ArrayList. В листинге 2.30 снова создается массив с цветами, но теперь он приводится к типу `System.Collections.ArrayList`.

Листинг 2.30. Создание ArrayList

```
PS> $colorPicker = [System.Collections.ArrayList]@('blue','white','yellow',
'black')
PS> $colorPicker
blue
white
yellow
black
```

Как и в случае с массивом, при вызове ArrayList каждый элемент отображается на отдельной строке.

Добавление элементов в ArrayList

Чтобы добавить или удалить элемент из ArrayList, не уничтожая его, вы можете использовать методы `Add()` и `Remove()`. В листинге 2.31 используется метод `Add()`, при этом в круглые скобки вводится новый элемент.

Листинг 2.31. Добавление одного элемента в ArrayList

```
PS> $colorPicker.Add('gray')
4
```

Обратите внимание на результат: вы получили число `4` — это индекс добавленного элемента. Как правило, вы не будете его использовать, поэтому можете передать вывод метода `Add()` в переменную `$null`, чтобы ничего не выводилось, как показано в листинге 2.32.

Листинг 2.32. Отправка вывода в $null

```
PS> $null = $colorPicker.Add('gray')
```

Существует несколько способов отменить вывод результатов команд PowerShell, но присвоение `$null` считается лучшим из них, поскольку она не может быть переназначена.

Удаление элементов из ArrayList

Аналогично можно удалять элементы с помощью метода `Remove()`. Например, если вы хотите удалить значение `gray` из ArrayList, введите значение в скобках метода, как показано в листинге 2.33.

Листинг 2.33. Удаление элемента из ArrayList

```
PS> $colorPicker.Remove('gray')
```

Обратите внимание, что для удаления элемента не обязательно знать его порядковый номер. Можно ссылаться на элемент по его фактическому значению — в нашем случае `gray`. Если в массиве есть несколько элементов с одинаковым значением, PowerShell удалит тот из них, который находится ближе всего к началу списка.

На этих простых примерах достаточно сложно заметить скачки производительности. Однако по сравнению с массивами списки ArrayList намного лучше работают с большими объемами данных. Как и в случае с большинством решений в программировании, каждую конкретную ситуацию нужно рассматривать отдельно и выбирать между массивом и ArrayList.

Эмпирическое правило таково: чем больше набор элементов, с которыми вы работаете, тем лучше будет использовать ArrayList. Если вы работаете с небольшими массивами, содержащими менее 100 элементов, вы не заметите большой разницы.

Хеш-таблицы

Массивы и ArrayList отлично работают, когда вам нужны данные, связанные лишь с некоей позицией в списке. Но иногда вам нужно нечто более прямолинейное: способ сопоставить элементы друг с другом. Например, у вас может быть список пользовательских имен, которые вы хотите сопоставить с настоящими именами. В этом случае можно использовать *хеш-таблицу* (или *словарь*) — структуру данных PowerShell, которая содержит список *пар «ключ — значение»*. Вместо использования числового индекса вы даете PowerShell входное значение, называемое *ключом*, а он возвращает связанное с ним *значение*. В примере ниже мы хотим проиндексировать хеш-таблицу, используя имя пользователя, а система вернет его настоящее имя.

В листинге 2.34 мы определили хеш-таблицу под названием `$users`, которая содержит информацию о трех пользователях.

Листинг 2.34. Создание хеш-таблицы

```
PS> $users = @{
abertram = 'Adam Bertram'
raquelcer = 'Raquel Cerillo'
zheng21 = 'Justin Zheng'
}
PS> $users
Name                           Value
----                           -----
abertram                       Adam Bertram
raquelcer                      Raquel Cerillo
zheng21                        Justin Zheng
```

PowerShell не даст вам определить хеш-таблицу с повторяющимися ключами. Каждый ключ должен указывать ровно на одно значение, которое, в свою очередь, может быть массивом или даже другой хеш-таблицей!

Чтение элементов из хеш-таблиц

Чтобы обратиться к конкретному значению в хеш-таблице, используется его ключ. Существует два способа сделать это. Допустим, вы хотите узнать

настоящее имя пользователя `abertram`. Вы можете использовать любой из двух способов, показанных в листинге 2.35.

Листинг 2.35. Доступ к значению хеш-таблицы

```
PS> $users['abertram']
Adam Bertram
PS> $users.abertram
Adam Bertram
```

У этих методов есть небольшие различия, но на данном этапе вы можете выбрать тот, который вам больше нравится.

Вторая команда в листинге 2.35 обращается к атрибуту `$users.abertram`. PowerShell добавляет каждый ключ к атрибутам объекта. Если вы хотите посмотреть все ключи и значения в хеш-таблице, нужно обратиться к атрибутам `Keys` и `Values`, как показано в листинге 2.36.

Листинг 2.36. Чтение ключей и значений хеш-таблицы

```
PS> $users.Keys
abertram
raquelcer
zheng21
PS> $users.Values
Adam Bertram
Raquel Cerillo
Justin Zheng
```

Если вы хотите увидеть *все* свойства хеш-таблицы (или любого объекта), выполните следующую команду:

```
PS> Select-Object -InputObject $yourobject -Property *
```

Добавление и изменение элементов хеш-таблицы

Чтобы добавить элемент в хеш-таблицу, можно использовать метод `Add()` или создать новый индекс, используя квадратные скобки и знак равенства. Оба способа показаны в листинге 2.37.

Листинг 2.37. Добавление элемента в хеш-таблицу

```
PS> $users.Add('natice', 'Natalie Ice')
PS> $users['phrigo'] = 'Phil Rigo'
```

Теперь в вашей хеш-таблице есть пять пользователей. Но что произойдет, если изменить одно из значений?

Изменяя хеш-таблицу, всегда нужно проверять существование нужной вам пары «ключ — значение». Чтобы проверить наличие ключа, можно использовать метод `ContainsKey()`, который есть у каждой хеш-таблицы, созданной в PowerShell. Если такой ключ есть, система вернет `True`; в противном случае она вернет `False`, как показано в листинге 2.38.

Листинг 2.38. Проверка наличия элемента в хеш-таблице

```
PS> $users.ContainsKey('johnnyq')
False
```

Убедившись, что в хеш-таблице есть нужный ключ, вы можете изменить его значение, используя простой знак равенства, как показано в листинге 2.39.

Листинг 2.39. Изменение значения в хеш-таблице

```
PS> $users['phrigo'] = 'Phoebe Rigo'
PS> $users['phrigo']
Phoebe Rigo
```

Итак, вы можете добавлять элементы в хеш-таблицу двумя способами. Как можно будет увидеть в следующем разделе, существует лишь один способ удалить элемент из хеш-таблицы.

Удаление элементов из хеш-таблицы

Подобно ArrayList, у хеш-таблиц есть метод `Remove()`. Просто вызовите его и передайте ему ключ элемента, который нужно удалить, как показано в листинге 2.40.

Листинг 2.40. Удаление элемента из хеш-таблицы

```
PS> $users.Remove('natice')
```

Один из ваших пользователей должен исчезнуть, но можно вызвать хеш-таблицу, чтобы убедиться в этом. Помните, что вы можете использовать атрибут `Keys`, чтобы посмотреть названия ключей.

Создание пользовательских объектов

До сих пор мы создавали и использовали типы объектов, встроенные в PowerShell. В большинстве случаев можно использовать их и тем самым экономить время на создание собственных объектов. Но иногда вам придется

создавать настраиваемые объекты со свойствами и методами, которые вы сами определяете.

В листинге 2.41 командлет `New-Object` используется для определения нового объекта типа `PSCustomObject`.

Листинг 2.41. Создание пользовательского объекта с помощью New-Object

```
PS> $myFirstCustomObject = New-Object -TypeName PSCustomObject
```

В этом примере используется команда `New-Object`, но вы можете сделать то же самое, используя знак равенства и операцию приведения, как показано в листинге 2.42. Это позволяет определить хеш-таблицу, в которой ключи являются именами атрибутов, а значения — значениями атрибутов, а затем привести ее к типу `PSCustomObject`.

Листинг 2.42. Создание пользовательского объекта с помощью ускорителя типа PSCustomObject

```
PS> $myFirstCustomObject = [PSCustomObject]@{OSBuild = 'x'; OSVersion = 'y'}
```

Обратите внимание, что в листинге 2.42 для разделения определений ключа и значения используется точка с запятой (`;`).

Как только вы создали пользовательский объект, можно использовать его так же, как и любой другой. В листинге 2.43 наш пользовательский объект передается командлету `Get_Member`, чтобы убедиться, что он относится к типу `PSCustomObject`.

Листинг 2.43. Исследование свойств и методов пользовательского объекта

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

   TypeName: System.Management.Automation.PSCustomObject

Name        MemberType   Definition
----        ----------   ----------
Equals      Method       bool Equals(System.Object obj)
GetHashCode Method       int GetHashCode()
GetType     Method       type GetType()
ToString    Method       string ToString()
OSBuild     NoteProperty string OSBuild=OSBuild
OSVersion   NoteProperty string OSVersion=Version
```

Как видите, у вашего объекта уже есть кое-какие заранее существующие методы (например, метод, который возвращает тип объекта!), а также атрибуты, которые вы определили в листинге 2.42.

Давайте попробуем обратиться к этим атрибутам через точку:

```
PS> $myFirstCustomObject.OSBuild
x
PS> $myFirstCustomObject.OSVersion
y
```

Выглядит круто! В остальной части книги мы будем использовать много объектов `PSCustomObject`. Это мощный инструмент, позволяющий создавать гораздо более гибкий код.

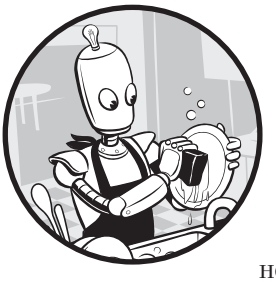
Итоги

Сейчас у вас должно быть общее представление об объектах, переменных и типах данных. Если вы все еще не разобрались с этими понятиями, пожалуйста, перечитайте эту главу. Это одни из самых основных вещей, которые мы будем рассматривать. Их глубокое понимание поможет вам легче воспринимать информацию из этой книги.

В следующей главе мы рассмотрим два способа объединения команд в PowerShell — конвейеры и сценарии.

3

Объединение команд



До сих пор мы использовали консоль PowerShell для вызова команд по одной. Для простого кода это не проблема: вы запускаете нужную команду, а затем еще одну, если требуется. Но для более крупных проектов необходимость вызывать каждую команду по отдельности занимает слишком много времени.

К счастью, вы можете комбинировать команды, чтобы вызывать их как единое целое. В этой главе вы изучите два способа комбинирования команд — с помощью конвейера PowerShell и путем сохранения кода во внешних сценариях.

Запуск службы Windows

Чтобы показать, зачем стоит комбинировать команды, начнем с простого примера, который мы выполним по старинке. Будем использовать две команды: `Get-Service`, которая запрашивает службы Windows и возвращает информацию о них, и `Start-Service`, запускающую эти службы. Как показано в листинге 3.1, мы используем команду `Get-Service`, чтобы убедиться, что служба существует, а затем используем команду `Start-Service` для ее запуска.

Мы запускаем службу `Get-Service`, только чтобы убедиться, что все работает без ошибок. Скорее всего, служба уже запущена. Если это так, `Start-Service` просто вернет консоли управление.

Листинг 3.1. Поиск и запуск службы с помощью параметра Name

```
PS> $serviceName = 'wuauserv'
PS> Get-Service -Name $serviceName
Status   Name               DisplayName
------   ----               -----------
Running  wuauserv           Windows Update
PS> Start-Service -Name $serviceName
```

Когда вы запускаете только одну службу, выполнение таких команд не слишком напрягает. Но представьте, как вам это надоест, если речь пойдет о работе с сотнями сервисов. Давайте посмотрим, как упростить эту задачу.

Использование конвейера

Первый способ упростить код — объединить команды в цепочку с помощью *конвейера* PowerShell. Это инструмент, который позволяет передавать выходные данные одной команды непосредственно во входные данные другой. Чтобы использовать конвейер, добавьте *оператор вертикальной черты* (|) между двумя командами, например:

```
PS> команда1 | команда2
```

В этом примере выходные данные `команды1` *передаются по конвейеру* в `команду2`, становясь ее входными данными. Последняя команда конвейера будет выведена на консоль.

Во многих языках сценариев оболочки, включая cmd.exe и bash, используется конвейер. В PowerShell он уникален тем, что передает между командами объекты, а не простые строки. Позже в этой главе вы увидите, как это происходит, а пока давайте перепишем код из листинга 3.1, используя конвейер.

Передача объектов между командами

Чтобы отправить вывод команды `Get-Service` в `Start-Service`, используйте код из листинга 3.2.

Листинг 3.2. Подсоединение существующих служб к команде Start-Service

```
PS> Get-Service -Name 'wuauserv' | Start-Service
```

В листинге 3.1 мы использовали параметр `Name`, чтобы указать команде `Start-Service`, какую именно службу нужно запустить. А в этом примере уже не

нужно указывать какие-либо параметры, потому что PowerShell сделает это за вас. Система просмотрит выходные данные `Get-Service`, поймет, какие значения следует передать в `Start-Service`, и сопоставит значения с параметрами, которые `Start-Service` принимает на вход.

При желании вы можете переписать листинг 3.2 вообще без параметров:

```
PS> 'wuauserv' | Get-Service | Start-Service
```

PowerShell отправляет строку `wuauserv` в команду `Get-Service`, а вывод `Get-Service` — в команду `Start-Service`, — и вам вообще не нужно что-либо указывать! Мы объединили три разные команды в одну строку, но нужно все равно повторно вводить эту строку для каждой службы, которую вы хотите запустить. В следующем разделе вы увидите, как с помощью одной строки запустить сколько угодно служб.

Передача массивов между командами

В текстовом редакторе, например в «Блокноте», создайте текстовый файл под названием Services.txt, в котором будут имена служб `Wuauserv` и `W32Time`, как показано на рис. 3.1.

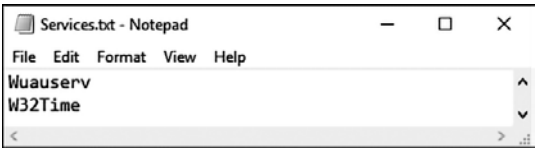


Рис. 3.1. Файл Services.txt с именами служб Wuauserv и W32Time, перечисленными на отдельных строках

В этом файле приведен список служб для запуска. Для простоты я указал всего две службы, но их может быть сколько угодно. Чтобы отобразить файл в окне PowerShell, используйте параметр `Path` командлета `Get-Content`:

```
PS> Get-Content -Path C:\Services.txt
Wuauserv
W32Time
```

Команда `Get-Content` читает файл построчно, добавляя каждую строку в массив, который затем возвращает. В листинге 3.3 мы используем конвейер для передачи массива, возвращаемого `Get-Content`, команде `Get-Service`.

Листинг 3.3. Отображение списка служб в окне PowerShell путем передачи содержимого файла Services.txt по конвейеру в команду Get-Service

```
PS> Get-Content -Path C:\Services.txt | Get-Service
Status   Name               DisplayName
------   ----               -----------
Stopped  Wuauserv           Windows Update
Stopped  W32Time            Windows Time
```

Команда `Get-Content` читает текстовый файл и выдает массив. Но вместо того, чтобы отправлять через конвейер сам массив, PowerShell *разворачивает* его и прогоняет каждый элемент через конвейер по одному.

Это позволяет выполнять одну и ту же команду для каждого элемента в массиве. Поместив каждую службу для запуска в текстовый файл и добавив приписку `| Start-Service` к команде, как в листинге 3.3, вы получите команду, которая сможет запустить сколько угодно служб.

Ограничений на количество команд, которые вы можете «склеить» с помощью конвейера, нет. Но если вы заметите, что их число превысило пять, вам, возможно, придется пересмотреть подход. Обратите внимание, что конвейер — это мощный инструмент, но он будет работать не всегда: большинство команд PowerShell принимают только определенные типы входных данных из конвейера, а некоторые не принимают вообще никаких. В следующем разделе на примере привязки параметров мы подробнее разберем, как PowerShell обрабатывает входные данные конвейера.

Поговорим о привязке параметров

Когда вы передаете команде параметры, PowerShell инициирует процесс, известный как *«привязка параметров»*, во время которого он сопоставляет каждый переданный в команду объект с параметрами, заданными создателем команды. Чтобы команда PowerShell могла принимать данные из конвейера, человек, который пишет команду — будь то сотрудник Microsoft или вы, — должен явно встроить поддержку конвейера для одного или нескольких параметров. Если вы попытаетесь передать информацию из конвейера в команду, которая не поддерживает его ни для одного параметра, или если PowerShell не найдет подходящую привязку, вы получите ошибку. Например, попробуйте выполнить следующую команду:

```
PS> 'string' | Get-Process
Get-Process : The input object cannot be bound to any parameters for the command...
--пропуск--
```

Вы увидите, что команда не принимает ввод из конвейера. Чтобы уточнить возможность использования конвейера для конкретной команды, можно просмотреть полное содержимое справки по команде, используя параметр `Full` в команде `Get-Help`. Давайте воспользуемся `Get-Help`, чтобы почитать о команде `Get-Service`, которую мы использовали в листинге 3.1.

```
PS> Get-Help -Name Get-Service –Full
```

На экране появится довольно длинная справка. Прокрутите ее вниз до раздела `PARAMETERS`. В этом разделе приведена информация о каждом параметре в более подробном виде, чем если бы вообще не использовался параметр `Detailed` или `Full`. В листинге 3.4 дана информация о параметре `Name` команды `Get-Service`.

Листинг 3.4. Информация о параметре Name команды Get-Service

```
-Name <string[]>
        Required?                   false
        Position?                   0
        Accept pipeline input?      true (ByValue, ByPropertyName)
        Parameter set name          Default
        Aliases                     ServiceName
        Dynamic?                    false
```

Информации тут много, но нас интересует поле `Accept pipeline input?` (в русской локализированной версии PowerShell «Принимать входные данные конвейера?»). Очевидно, это поле сообщает вам, принимает ли параметр входные данные конвейера. Если это не так, вы увидите рядом с этим полем значение `False`. Обратите внимание, что здесь есть и дополнительная информация: этот параметр принимает входные данные конвейера способом `ByValue` и `ByPropertyName`. Сравните эту информацию с параметром `ComputerName` той же команды в листинге 3.5.

Листинг 3.5. Информация о параметре ComputerName команды Get-Service

```
-ComputerName <string[]>
        Required?                   false
        Position?                   Named
        Accept pipeline input?      true (ByPropertyName)
        Parameter set name          (all)
        Aliases                     Cn
        Dynamic?                    false
```

Параметр `ComputerName` позволяет указать, на каком компьютере вы хотите запустить команду `Get-Service`. Обратите внимание, что этот параметр также принимает на вход строку. Так каким же образом PowerShell знает, что вы

имеете в виду имя службы, а не имя компьютера, если команда выглядит так, как показано ниже?

```
PS> 'wuauserv' | Get-Service
```

PowerShell сопоставляет входные данные конвейера с параметрами двумя способами. Первый — с помощью метода `ByValue`, когда PowerShell проверяет тип переданного объекта и интерпретирует его соответствующим образом. Поскольку команда `Get-Service` указывает, что принимает параметр `Name` методом `ByValue`, она будет интерпретировать любую переданную строку как `Name`, если не указано иное. Поскольку передаваемые через `ByValue` параметры зависят от типа входных данных, так можно передать только один параметр.

Второй способ подбора параметра из конвейера — с помощью `ByPropertyName`. В этом случае PowerShell проверит переданный объект, и если у него есть атрибут с соответствующим именем (в нашем случае `ComputerName`), то он рассмотрит значение этого атрибута и примет его в качестве параметра. Итак, если вы хотите передать в `Get-Service` имя службы и имя компьютера, вы можете создать объект `PSCustomObject` и передать его, как показано в листинге 3.6.

Листинг 3.6. Передача пользовательского объекта в команду Get-Service

```
PS> $serviceObject = [PSCustomObject]@{Name = 'wuauserv'; ComputerName = 'SERV1'}
PS> $serviceObject | Get-Service
```

Глядя на спецификации параметров команды и используя хеш-таблицу аккуратного хранения наиболее необходимых из них, вы сможете использовать конвейер для объединения всевозможных видов команд. Но когда вы начнете писать более сложный код на PowerShell, вам понадобится нечто большее, чем конвейер. В следующем разделе вы узнаете, как сохранять код PowerShell в виде сценариев.

Написание сценариев

Сценарии — это внешние файлы, в которых хранится последовательность команд. Запустить сценарий можно с помощью одной строки в консоли PowerShell. Как видно из листинга 3.7, чтобы запустить сценарий, достаточно ввести в консоли путь к нему.

Листинг 3.7. Запуск сценария из консоли

```
PS> C:\FolderPathToScript\script.ps1
Hello, I am in a script!
```

Хотя в сценарии нет ничего такого, чего нельзя было бы сделать в консоли, гораздо проще вызвать сценарий одной командой, чем набирать несколько тысяч! Не говоря уже о том, что, если вы вдруг захотите что-то изменить в своем коде либо совершите ошибку, вам придется вводить эти команды заново. Как вы позже убедитесь, сценарии позволяют писать сложный и надежный код. Но прежде чем начать писать сценарии, вы должны изменить некоторые настройки PowerShell, чтобы можно было их запускать.

Настройка политики выполнения

По умолчанию PowerShell не позволяет запускать сценарии. Если вы попытаетесь запустить внешний сценарий в настроенном по умолчанию PowerShell, появится ошибка, показанная в листинге 3.8.

Листинг 3.8. Ошибка, возникающая при попытке запустить сценарий

```
PS> C:\PowerShellScript.ps1
C:\PowerShellScript.ps1: File C:\PowerShellScript.ps1 cannot be loaded because
running scripts is disabled on this system. For more information, see about
_Execution_Policies at http://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ C:\PowerShellScript.ps1
+ ~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : SecurityError: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
```

Это раздражающее сообщение об ошибке возникает из-за *политики выполнения* PowerShell — специальной меры безопасности, которая определяет, какие сценарии можно запускать. Политика выполнения имеет четыре варианта настройки:

Restricted — эта настройка используется по умолчанию. Она не позволяет запускать сценарии.

AllSigned — эта настройка позволяет запускать только те сценарии, которые были криптографически подписаны доверенной стороной (подробнее об этом позже).

RemoteSigned — эта настройка позволяет запускать любой сценарий, созданный вами либо загруженный, при условии, что он был криптографически подписан доверенной стороной.

Unrestricted — эта настройка позволяет запускать любые сценарии.

ПОДПИСЬ СЦЕНАРИЯ

Подпись сценария — это зашифрованная строка, которая добавляется в конец сценария в качестве комментария. Эти подписи генерируются сертификатом, установленным на вашем компьютере. Когда вы задаете политику на `AllSigned` или `RemoteSigned`, вы сможете запускать только сценарии с правильными подписями. Наличие подписи источника позволяет PowerShell понять, что источник сценария надежен и что автор сценария не выдает себя за кого-то другого. Подпись сценария выглядит примерно так:

```
# SIG # Начало блока подписи
# MIIEMwYJKoZIhvcNAQcCoIIEJDCCBCACAQExCzAJBgUrDgMCGgUAMGkGCisGAQQB
# gjcCAQSgWzBZMDQGCisGAQQBgjcCAR4wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR
# AgEAAgEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQU6vQAnx5sqw2
--пропуск--
# m5ugggI9MIICOTCCAaagAwIBAgIQyLeyGZcGA4ZOGqK7VF45GDAJBgUrDgMCHQUA
# Dxoj+2keS9sRR6XPl/ASs68LeF8o9cM=
# SIG # Конец блока подписи
```

Необходимо подписывать все сценарии, которые вы будете создавать и выполнять в профессиональной среде. Здесь мы не будем вдаваться в подробности, но один из лучших ресурсов, на мой взгляд, раскрывающих это понятие, – это серия статей известного гуру безопасности Карлоса Переза (Carlos Perez) «PowerShell Basics — Execution Policy and Code Signing» на www.darkoperator.com/blog/2013/3/5/powershell-basics-execution-policy-part-1.html.

Чтобы узнать, какая политика выполнения у вас настроена в данный момент, выполните команду из листинга 3.9.

Листинг 3.9. Вывод текущей политики выполнения с помощью команды Get-ExecutionPolicy

```
PS> Get-ExecutionPolicy
Restricted
```

Скорее всего, вы увидите надпись `Restricted`. В целях изучения этой книги измените политику выполнения на `RemoteSigned`. Это позволит вам запускать любые ваши сценарии, а также гарантирует, что внешние сценарии будут выполняться, только если они из надежных источников. Чтобы изменить политику выполнения, используйте команду `Set-ExecutionPolicy` и укажите нужную политику, как показано в листинге 3.10. Обратите внимание, что нужно запустить эту команду от имени администратора (см. главу 1, где мы говорили о запуске команд от имени администратора). Эту команду достаточно

выполнить один раз, так как настройка сохраняется в реестре. Если вы находитесь в крупной среде Active Directory, настройку можно выполнить одновременно на нескольких компьютерах с помощью групповой политики.

Листинг 3.10. Изменение политики выполнения с помощью команды Set-ExecutionPolicy

```
PS> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned

Execution Policy Change
The execution policy helps protect you from scripts that you do not trust.
Changing the execution policy might expose you to the security risks described
in the about_Execution_Policies help topic at
http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution
policy?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "N"): A
```

Выполните команду `Get-ExecutionPolicy` еще раз, чтобы убедиться, что вы успешно изменили политику на `RemoteSigned`. Как я сказал ранее, вам не нужно задавать политику выполнения при каждом запуске PowerShell. Заданное значение `RemoteSigned` останется до тех пор, пока вы снова не внесете изменения.

Создание сценариев в PowerShell

Теперь, когда мы настроили политику выполнения, пришла пора написать сценарий и выполнить его в консоли. Вы можете писать сценарии PowerShell в любом удобном для вас текстовом редакторе (в Emacs, Vim, Sublime Text, Atom и даже в «Блокноте»), но удобнее всего пользоваться интегрированной средой сценариев PowerShell (ISE) или редактором кода Microsoft Visual Studio Code. Технически ISE устарела, но зато она предустановлена в Windows, поэтому вы наверняка начнете именно с нее.

Использование PowerShell ISE

Чтобы запустить PowerShell ISE, выполните команду из листинга 3.11.

Листинг 3.11. Открытие PowerShell ISE

```
PS> powershell_ise.exe
```

Перед вами откроется окно интерактивной консоли, показанное на рис. 3.2.

Чтобы создать сценарий, выберите команду File ▸ New (Файл ▸ Создать). Над консолью должна появиться белая панель, как показано на рис. 3.3.

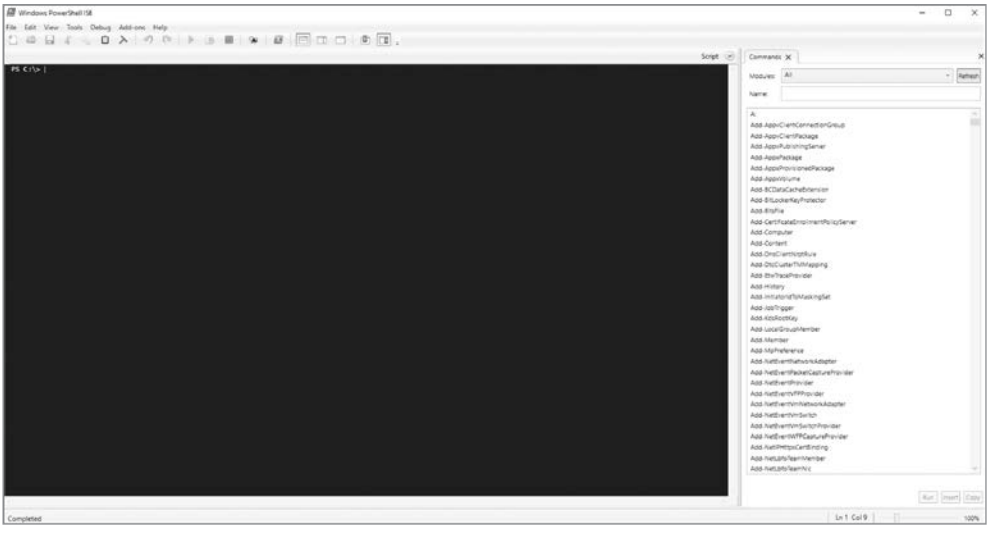


Рис. 3.2. Интегрированная среда сценариев PowerShell

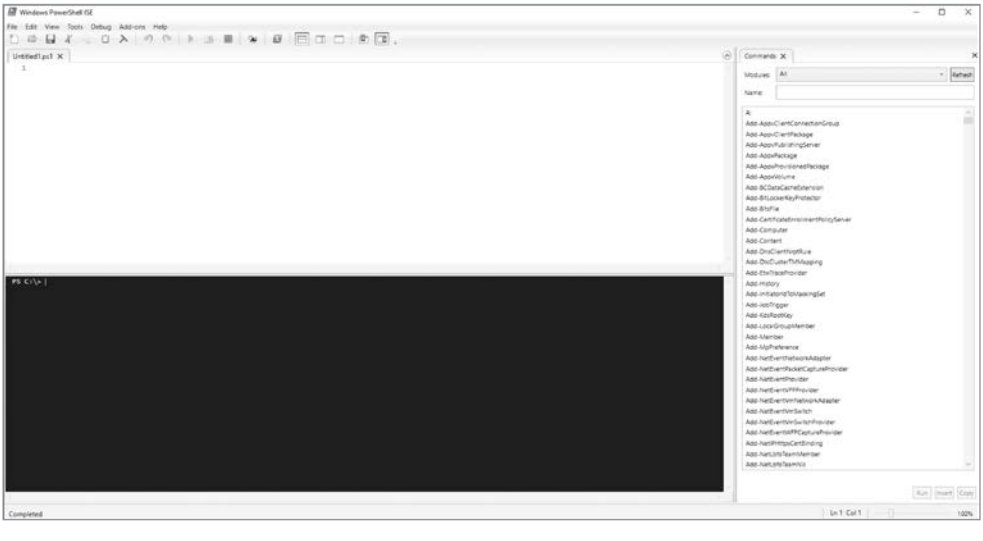


Рис. 3.3. PowerShell ISE с открытым сценарием

Выберите команду File ▸ Save (Файл ▸ Сохранить) и сохраните новый файл под именем WriteHostExample.ps1. Я сохранил свой сценарий в корневой папке диска C:, поэтому он находится по адресу C:\WriteHostExample.ps1. Обратите внимание, что у вашего файла будет расширение .ps1 — это говорит вашей системе, что это сценарий PowerShell.

Текст сценария мы будем писать в белом поле. PowerShell ISE позволяет редактировать и запускать сценарий в одном окне, что избавит вас от множества скучных переходов туда-обратно при редактировании. У PowerShell ISE есть еще множество функций, но я не буду здесь на них останавливаться.

Сценарии PowerShell — это простые текстовые файлы. Не имеет значения, какой текстовый редактор вы используете, главное — придерживаться правильного синтаксиса PowerShell.

Написание первого сценария

Используя любой редактор, добавьте в свой сценарий строку из листинга 3.12.

Листинг 3.12. Первая строка вашего сценария

```
Write-Host 'Hello, I am in a script!'
```

Обратите внимание, что в начале строки нет приписки PS>. Это позволяет понять, в консоли вы работаете или в редакторе.

Чтобы запустить этот сценарий, перейдите в консоль и введите путь к вашему сценарию, как показано в листинге 3.13.

Листинг 3.13. Выполнение сценария WriteHostExample.ps1 в консоли

```
PS> C:\WriteHostExample.ps1
Hello, I am in a script!
```

Мы прописали полный путь к сценарию WriteHostExample.ps1, чтобы запустить его. Если вы уже находитесь в каталоге с нужным вам сценарием, вы можете использовать точку для обозначения текущего рабочего каталога, например .\WriteHostExample.ps1.

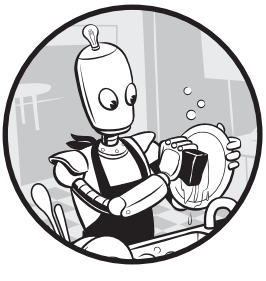
Готово, поздравляю — вы создали свой первый сценарий! Выглядит не особо впечатляюще, но это большой шаг в верном направлении. К концу книги мы будем писать собственные модули PowerShell в сценариях с сотнями строк кода.

Итоги

В этой главе мы узнали о двух важных методах объединения команд — о конвейере и сценариях. Мы узнали, как можно изменить политику выполнения, а также немного развеяли мифы вокруг конвейеров с помощью знаний о привязке параметров. Мы заложили основу для создания более мощных сценариев, но нам нужно рассмотреть еще несколько ключевых понятий, прежде чем дойдем до этого этапа. В главе 4 вы узнаете, как сделать свой код значительно более надежным, используя структуры потока управления, а именно операторы `if/then` и циклы `for`.

4

Поток управления



Немного повторим. В главе 3 мы узнали, как можно комбинировать команды с помощью конвейера и внешних сценариев. В главе 2 рассмотрели переменные и как их использовать для хранения значений. Одним из основных преимуществ работы с переменными является возможность писать с их помощью код, который работает не со значением, а со «смыслом». Вместо того чтобы работать, например, с числом 3, вы будете работать с общим понятием `$serverCount`. За счет этого вы можете писать код, который работает одинаково, будь у вас один, два или тысяча серверов. Совместите эту способность с возможностью сохранять код в сценариях, которые можно запускать на разных компьютерах, и вы сможете начать решать задачи гораздо большего масштаба.

Однако в жизни порой имеет значение, работаете ли вы с одним сервером, с двумя или с тысячей. Пока что у вас нет подходящего способа учитывать это, и ваши сценарии работают просто «сверху вниз», не имея возможности адаптироваться в зависимости от определенных значений. В этой главе мы будем использовать поток управления и условную логику для написания сценариев, которые будут выполнять различные команды в зависимости от значений, с которыми они работают. К концу главы вы узнаете, как использовать операторы `if/then` и `switch`, а также различные циклы, чтобы придать вашему коду столь необходимую гибкость.

Немного о потоке управления

Мы напишем сценарий, который считывает содержимое файла, хранящегося на нескольких удаленных компьютерах. Чтобы продолжить работу, загрузите файл под названием App_configuration.txt из прилагаемых к книге материалов по ссылке github.com/adbertram/PowerShellForSysadmins/ и поместите его в корень диска C:\ на нескольких удаленных компьютерах. Если у вас нет удаленных компьютеров, пока просто продолжайте читать. В этом примере я буду использовать серверы с именами SRV1, SRV2, SRV3, SRV4 и SRV5.

Чтобы получить доступ к содержимому файла, воспользуемся командой Get-Content и укажем путь к файлу в значении аргумента параметра Path, как показано ниже:

```
Get-Content -Path "\\servername\c$\App_configuration.txt"
```

Для начала сохраним все имена наших серверов в массиве и запустим эту команду для каждого сервера. Откройте новый файл .ps1 и введите в него код из листинга 4.1.

Листинг 4.1. Извлечение содержимого файла с нескольких серверов

```
$servers = @('SRV1','SRV2','SRV3','SRV4','SRV5')
Get-Content -Path "\\$($servers[0])\c$\App_configuration.txt"
Get-Content -Path "\\$($servers[1])\c$\App_configuration.txt"
Get-Content -Path "\\$($servers[2])\c$\App_configuration.txt"
Get-Content -Path "\\$($servers[3])\c$\App_configuration.txt"
Get-Content -Path "\\$($servers[4])\c$\App_configuration.txt"
```

Теоретически, этот код должен работать без проблем. Но в этом примере предполагается, что у вас что-то идет не так. Что делать, если сервер SRV2 не работает? А если кто-то забыл положить App_configuration.txt на SRV4? А может, кто-то изменил путь к файлу? Вы можете написать отдельный сценарий для каждого сервера, но это решение не будет масштабироваться, особенно когда вы начнете добавлять все больше и больше серверов. Вам нужен код, который будет работать в зависимости от ситуации.

Суть идеи *потока управления* в том, что он позволяет выполнять различные наборы инструкций в зависимости от заранее определенной логики. Представьте, что ваши сценарии выполняются по определенному пути. Пока что ваш путь прост — от первой строки кода до последней. Однако вы можете добавлять на этом пути развилки, возвращаться в места, где уже побывали, или перескакивать через них. Разветвляя пути выполнения вашего сценария,

вы наделяете его большей гибкостью, что позволяет обрабатывать множество ситуаций с помощью одного сценария.

Мы начнем с рассмотрения самого простого типа потока управления — условного оператора.

Использование условных операторов

В главе 2 мы узнали, что существуют логические значения: истина и ложь. Логические значения позволяют создавать *условные операторы*, которые ставят задачу PowerShell выполнить определенный блок кода в зависимости от того, имеет ли выражение (называемое *условием*) значение `True` или `False`. Условие — это вопрос с вариантами ответов да/нет. У вас больше пяти серверов? Работает ли сервер 3? Существует ли путь к файлу? Чтобы начать использовать условные операторы, давайте посмотрим, как преобразовать такие вопросы в выражения.

Построение выражений с помощью операторов

Логические выражения можно писать с помощью *операторов сравнения*, которые сравнивают значения. Чтобы использовать оператор сравнения, нужно поместить его между двумя значениями, например:

```
PS> 1 -eq 1
True
```

В этом случае оператор `-eq` позволяет определить равнозначность двух значений.

Ниже приведен список наиболее распространенных операторов сравнения, которые мы будем использовать:

`-eq` сравнивает два значения и возвращает `True`, если они равны.

`-ne` сравнивает два значения и возвращает `True`, если они не равны.

`-gt` сравнивает два значения и возвращает `True`, если первое больше второго.

`-ge` сравнивает два значения и возвращает `True`, если первое больше или равно второму.

`-lt` сравнивает два значения и возвращает `True`, если первое меньше второго.

-le сравнивает два значения и возвращает `True`, если первое меньше или равно второму.

-contains возвращает `True`, если второе значение является частью первого. Например, этот оператор позволяет определить, находится ли значение внутри массива.

В PowerShell есть и более продвинутые операторы сравнения. Здесь мы не будем на них останавливаться, но я рекомендую вам почитать о них в документации Microsoft по ссылке https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_comparison_operators/ или в разделе справки PowerShell (см. главу 1).

Вы можете использовать приведенные выше операторы для сравнения переменных и значений. Но выражение не обязательно должно быть сравнением. Иногда команды PowerShell можно использовать как условия. В предыдущем примере мы хотели узнать доступность сервера. С помощью командлета `Test-Connection` можно проверить наличие связи с сервером. Обычно в выходных данных командлета `Test-Connection` содержится много разной информации, но с помощью параметра `Quiet` вы можете заставить команду вернуть `True` или `False`, а с помощью параметра `Count` можно ограничить тест одной попыткой.

```
PS> Test-Connection -ComputerName offlineserver -Quiet -Count 1
False

PS> Test-Connection -ComputerName onlineserver -Quiet -Count 1
True
```

Чтобы узнать, отключен ли сервер, вы можете использовать оператор `-not` для преобразования выражения в противоположное:

```
PS> -not (Test-Connection -ComputerName offlineserver -Quiet -Count 1)
True
```

Теперь, когда вы познакомились с основными выражениями, давайте рассмотрим простейший условный оператор.

Оператор if

Оператор `if` работает просто: если выражение *X* истинно, то сделайте *Y*. Вот и все!

Чтобы использовать оператор в выражении, пишется ключевое слово `if`, за которым следуют круглые скобки, содержащие условие. После выражения

следует блок кода, выделенный фигурными скобками. PowerShell выполнит этот блок кода только в том случае, если это выражение будет иметь значение `True`. Если выражение `if` имеет значение `False` либо вообще ничего не возвращает, блок кода не будет выполнен. Синтаксис оператора `if/then` показан в листинге 4.2.

Листинг 4.2. Синтаксис оператора if

```
if (условие) {
    # выполняемый код, если условие истинно
}
```

В этом примере есть немного нового синтаксиса: символ решетки (`#`) обозначает *комментарий* — это текст, который PowerShell игнорирует. Вы можете использовать комментарии, чтобы оставить полезные примечания и описания для себя или кого-нибудь, кто позже будет читать ваш код.

Теперь давайте еще раз посмотрим на код, показанный в листинге 4.1. Я расскажу вам о том, как использовать оператор `if`, чтобы не пытаться достучаться до неработающего сервера. В предыдущем разделе мы уже видели, что команду `Test-Connection` можно использовать в качестве выражения, которое возвращает `True` или `False`, поэтому сейчас давайте упакуем `Test-Connection` в оператор `if`, а затем воспользуемся командой `Get-Content`, чтобы не пытаться обращаться к неработающему серверу. Сейчас мы поменяем код только для первого сервера, как показано в листинге 4.3.

Листинг 4.3. Использование оператора if для выборочного обращения к серверам

```
$servers = @('SRV1','SRV2','SRV3','SRV4','SRV5')
if (Test-Connection -ComputerName $servers[0] -Quiet -Count 1) {
    Get-Content -Path "\\$($servers[0])\c$\App_configuration.txt"
}
Get-Content -Path "\\$($servers[1])\c$\App_configuration.txt"
--пропуск--
```

Поскольку у вас есть `Get-Content` в операторе `if`, вы не столкнетесь с какими-либо ошибками, если попытаетесь получить доступ к неработающему серверу; если тест завершится неудачно, ваш сценарий будет знать, что не следует пытаться считать файл. Код попытается получить доступ к серверу, только если он *уже* знает, что тот включен. Но обратите внимание, что этот код срабатывает только в том случае, если условие истинно. Достаточно часто вам нужно будет задать одно поведение сценария для истинного условия и другое для ложного. В следующем разделе вы увидите, как определить поведение для ложного условия с помощью оператора `else`.

Оператор else

Чтобы предоставить вашему оператору `if` альтернативу, можно использовать ключевое слово `else` после закрывающей скобки блока `if`, за которым будет следовать еще одна пара фигурных скобок, содержащая блок кода. Как показано в листинге 4.4, мы будем использовать оператор `else`, чтобы вернуть в консоль ошибку, если первый сервер не отвечает.

Листинг 4.4. Использование оператора else для запуска кода, если условие не истинно

```
if (Test-Connection -ComputerName $servers[0] -Quiet -Count 1) {
    Get-Content -Path "\\$($servers[0])\c$\App_configuration.txt"
} else {
    Write-Error -Message "The server $($servers[0]) is not responding!"
}
```

Оператор `if/else` отлично работает, когда у вас есть две взаимоисключающие ситуации. В данном случае сервер либо подключен, либо нет, то есть нам нужно всего две ветви кода. Давайте посмотрим, как работать с более сложными ситуациями.

Оператор elseif

Оператор `else` охватывает противоположную ситуацию: если `if` не срабатывает, значит, выполните это в любом случае. Такой подход работает для двоичных условий, то есть когда сервер либо работает, либо нет. Но иногда приходится иметь дело с большим числом вариантов. Например, предположим, что у вас есть сервер, на котором нет нужного вам файла, и вы сохранили имя этого сервера в переменной `$problemServer` (добавьте в свой сценарий эту строку кода!). Это означает, что вам нужна дополнительная проверка, позволяющая узнать, является ли сервер, который вы опрашиваете в данный момент, проблемным. Это можно реализовать с помощью вложенных операторов `if`, как показано в коде ниже:

```
if (Test-Connection -ComputerName $servers[0] -Quiet -Count 1) {
    if ($servers[0] -eq $problemServer) {
        Write-Error -Message "The server $servers[0] does not have the right
                             file!"
    } else {
        Get-Content -Path "\\$servers[0]\c$\App_configuration.txt"
    }
} else {
    Write-Error -Message "The server $servers[0] is not responding!"
}
--пропуск--
```

Но есть и более аккуратный способ реализовать ту же логику — с помощью оператора `elseif`, который позволяет вам проверить дополнительное условие, перед тем как вернуться к коду в блоке `else`. Синтаксис блока `elseif` идентичен синтаксису блока `if`. Итак, чтобы проверить проблемный сервер с помощью оператора `elseif`, запустите код из листинга 4.5.

Листинг 4.5. Использование блока elseif

```
if (-not (Test-Connection -ComputerName $servers[0] -Quiet -Count 1)) { ❶
    Write-Error -Message "The server $servers[0] is not responding!"
} elseif ($servers[0] -eq $problemServer) ❷
    Write-Error -Message "The server $servers[0] does not have the right file!"
} else {
    Get-Content -Path "\\$servers[0]\c$\App_configuration.txt" ❸
}
--пропуск--
```

Обратите внимание, что мы не просто добавили оператор `elseif`, а заодно изменили логику кода. Теперь мы можем проверить, не находится ли сервер в автономном режиме, с помощью оператора `-not` ❶. Затем, как только мы определили сетевой статус сервера, мы проверяем, является ли он проблемным ❷. Если это не так, мы используем оператор `else` для запуска поведения по умолчанию — извлечения файла ❸. Как видите, существует несколько способов структурировать код описанным образом. Важно то, что код работает и что он читабелен для человека со стороны, будь то ваш коллега, видящий его впервые, или вы сами спустя некоторое время после написания.

Вы можете объединить в цепочку сколько угодно операторов `elseif`, что позволяет учитывать самые разные сочетания обстоятельств. Однако операторы `elseif` являются взаимоисключающими: когда один из `elseif` принимает значение `True`, PowerShell запускает только его код и не проверяет остальные случаи. В листинге 4.5 это не вызвало никаких проблем, так как вам нужно было проверить сервер на предмет «проблемности» только после проверки работоспособности, но в дальнейшем я советую вам держать в уме эту особенность.

Операторы `if`, `else` и `elseif` отлично подходят для реализации ответа кода на простые вопросы типа «да/нет». В следующем разделе вы узнаете, как работать с более сложной логикой.

Оператор switch

Давайте немного подкорректируем наш пример. Допустим, у нас есть пять серверов, и на каждом сервере путь к нужному файлу различается. Исходя из

того что вы знаете сейчас, вам придется создать отдельный оператор `elseif` для каждого сервера. Это сработает, но существует и более удобный метод.

Обратите внимание, что теперь мы будем работать с другим типом условия. Если раньше нам нужны были ответы на вопросы типа «да/нет», то теперь мы хотим получить конкретное значение одной вещи. Это сервер `SRV1`? `SRV2`? И так далее. Если бы вы работали только с одним или двумя конкретными значениями, оператор `if` подошел бы, но в данном случае оператор `switch` сработает гораздо лучше.

Оператор `switch` позволяет выполнять различные фрагменты кода в зависимости от некоторого значения. Он состоит из ключевого слова `switch`, за которым следует выражение в скобках. Внутри блока `switch` находится серия операторов, построенных по следующему принципу: сначала указывается значение, за ним следует набор фигурных скобок, содержащих блок кода, и наконец ставится блок `default`, как указано в листинге 4.6.

Листинг 4.6. Шаблон для оператора switch

```
switch (выражение) {
    значениевыражения {
        # Код
    }
    значениевыражения {
    }
    default {
        # Код, который выполняется при отсутствии совпадений
    }
}
```

Оператор `switch` может содержать практически неограниченное количество значений. Если выражение оценивается как значение, выполняется соответствующий код внутри блока. Важно то, что, в отличие от `elseif`, после выполнения одного блока кода PowerShell продолжит проверять и остальные условия, если не указано иное. Если ни одно из значений не подойдет, PowerShell выполнит код, указанный в блоке `default`. Чтобы прекратить перебор условий в операторе `switch`, используйте ключевое слово `break` в конце блока кода, как показано в листинге 4.7.

Листинг 4.7. Использование ключевого слова break в операторе switch

```
switch (выражение) {
    значениевыражения {
        # Код
        break
    }
--пропуск--
```

Ключевое слово `break` позволяет сделать условия в операторе `switch` взаимоисключающими. Вернемся к нашему примеру с пятью серверами и одним и тем же файлом, имеющим разные пути. Вы знаете, что сервер, с которым вы работаете, может иметь только одно значение (то есть он не может одновременно называться и `SRV1`, и `SRV2`), поэтому вам нужно использовать операторы `break`. Ваш сценарий должен выглядеть примерно так, как показано в листинге 4.8.

Листинг 4.8. Проверка различных серверов с помощью оператора switch

```
$currentServer = $servers[0]
switch ($currentServer) {
    $servers[0] {
        # Check if server is online and get content at SRV1 path.
        break
    }
    $servers[1] {
        ## Check if server is online and get content at SRV2 path.
        break
    }
    $servers[2] {
        ## Check if server is online and get content at SRV3 path.
        break
    }
--пропуск--
```

Вы можете переписать этот код, используя только операторы `if` и `elseif` (и я действительно рекомендую вам попробовать это!). В любом случае, какой бы метод вы ни выбрали, вам потребуется одна и та же структура для каждого сервера в списке, а это означает, что ваш сценарий будет довольно длинным — просто представьте, что вам придется протестировать пятьсот серверов вместо пяти. В следующем разделе вы узнаете, как избавиться от этой проблемы, используя одну из самых фундаментальных структур потока управления — цикл.

Использование циклов

Существует хорошее практическое правило для работы за компьютером: не повторяйся (don't repeat yourself, DRY). Если вы обнаружите, что выполняете одну и ту же работу, то, скорее всего, существует способ ее автоматизировать. То же самое и с написанием кода: если вы используете одни и те же строки снова и снова, вероятно, существует решение получше.

Один из способов избежать повторов — использовать циклы. *Цикл* позволяет многократно выполнять код, до тех пор пока не изменится некоторое заданное

условие. *Условие остановки* может запускать цикл заданное количество раз, либо до тех пор, пока не изменится некоторое логическое значение, либо определить бесконечное выполнение цикла. Каждый проход цикла мы будем называть *итерацией*.

PowerShell предлагает пять типов циклов: `foreach`, `for`, `do/while`, `do/until` и `while`. В этом разделе мы обсудим каждый тип цикла, отметим их уникальные черты и выделим лучшие ситуации для их использования.

Цикл foreach

Мы начнем с наиболее используемого цикла в PowerShell — с цикла `foreach`. Цикл `foreach` просматривает список объектов и выполняет одно и то же действие для каждого, завершая работу, когда объекты закончатся.

Список объектов обычно представлен массивом. Когда вы запускаете цикл по списку объектов, это будет проходом (или *итерацией*) по списку.

Цикл `foreach` полезен в тех случаях, когда вам нужно выполнить одну и ту же задачу над множеством разных, но при этом связанных объектов. Вернемся к листингу 4.1 (повторим его здесь):

```
$servers = @('SRV1','SRV2','SRV3','SRV4','SRV5')
Get-Content -Path "\\$($servers[0])\c$\App_configuration.txt"
Get-Content -Path "\\$($servers[1])\c$\App_configuration.txt"
Get-Content -Path "\\$($servers[2])\c$\App_configuration.txt"
Get-Content -Path "\\$($servers[3])\c$\App_configuration.txt"
Get-Content -Path "\\$($servers[4])\c$\App_configuration.txt"
```

Забудем на минуту обо всех наворотах из предыдущего раздела и вместо этого просто реализуем логику в цикле `foreach`. В отличие от других циклов в PowerShell, `foreach` можно использовать тремя способами: в виде оператора `foreach`, командлета `ForEach-Object` или метода `foreach()`. Эти методы схожи в применении, но вам все равно нужно понимать их различия. В следующих трех разделах мы перепишем листинг 4.1, используя каждый из трех способов реализации цикла `foreach`.

Оператор foreach

Первый тип `foreach`, который мы рассмотрим, — это оператор `foreach`. В листинге 4.9 представлен код из листинга 4.1, реализованный в виде цикла.

Листинг 4.9. Использование оператора foreach

```
foreach ($server in $servers) {
    Get-Content -Path "\\$server\c$\App_configuration.txt"
}
```

Видно, что за оператором `foreach` следуют круглые скобки, в которых последовательно перечислены три элемента: переменная, ключевое слово `in` и объект или массив, по которому выполняется итерация. Передаваемая в оператор переменная может иметь любое название, но я рекомендую использовать как можно более содержательные имена.

Когда оператор проходит по списку, PowerShell *копирует* изучаемый в данный момент объект в переменную. Обратите внимание, что, поскольку переменная является просто копией, вы не можете напрямую изменить элемент в исходном списке. Это легко проверить: выполните код, указанный ниже.

```
$servers = @('SRV1','SRV2','SRV3','SRV4','SRV5')
foreach ($server in $servers) {
    $server = "new $server"
}
$servers
```

У вас должно получиться что-то вроде этого:

```
SRV1
SRV2
SRV3
SRV4
SRV5
```

Ничего не изменилось! А все потому, что вы изменяете только копию исходной переменной в массиве. Это один из недостатков использования любого типа цикла `foreach`. Чтобы напрямую изменить исходное содержимое списка, который вы прогоняете через цикл, потребуется другой тип цикла.

Командлет ForEach-Object

Как и оператор `foreach`, командлет `ForEach-Object` проходит по набору объектов и выполняет над ними некоторое действие. Но поскольку `ForEach-Object` — это командлет, нужно будет передать ему этот набор объектов и необходимые действия в качестве параметров.

В листинге 4.10 показано, как реализовать код из листинга 4.9 с помощью командлета `ForEach-Object`.

Листинг 4.10. Использование командлета ForEach-Object

```
$servers = @('SRV1','SRV2','SRV3','SRV4','SRV5')
ForEach-Object -InputObject $servers -Process {
    Get-Content -Path "\\$_\c$\App_configuration.txt"
}
```

Реализация в данном случае слегка отличается от предыдущей, поэтому давайте подробно расмотрим этот пример. Обратите внимание, что `ForEach-Object` принимает параметр `InputObject`. Здесь используется массив `$servers`, но вы можете использовать любой объект, например строку или целое число. В двух последних вариантах PowerShell выполнит всего одну итерацию. Командлет также принимает параметр `Process` — блок сценария, содержащий код, который нужно запустить для каждого элемента внутри входного объекта. (*Блок сценария* — это набор операторов, которые вы передаете в командлет как единое целое.)

Возможно, вы заметили в листинге 4.10 еще кое-что странное. Вместо использования переменной `$server`, как в случае с оператором `foreach`, мы использовали синтаксис `$_`. Эта приписка указывает на текущий объект в конвейере. Основное различие между оператором `foreach` и командлетом `ForEach-Object` заключается в том, что командлет принимает входные данные конвейера. На практике `ForEach-Object` почти всегда используется путем передачи параметра `InputObject` через конвейер, например:

```
$servers | ForEach-Object -Process {
    Get-Content -Path "\\$_\c$\App_configuration.txt"
}
```

Командлет `ForEach-Object` может значительно сэкономить вам время.

Метод foreach()

Последний тип цикла `foreach`, который мы рассмотрим, — это метод объекта `foreach()`, впервые добавленный в версию PowerShell v4. Метод `foreach()` определен для всех массивов в PowerShell и может использоваться для того же, что и `foreach` и `ForEach-Object`. Он принимает параметр блока сценария, который содержит код, выполняемый в каждой итерации. Как и в случае с `ForEach-Object`, мы будем использовать приписку `$_` для захвата объекта текущей итерации, как показано в листинге 4.11.

Листинг 4.11. Использование метода foreach()

```
$servers.foreach({Get-Content -Path "\\$_\c$\App_configuration.txt"})
```

Метод `foreach()` работает значительно быстрее предыдущих двух, и эта разница будет заметна при обработке больших наборов данных. Я рекомендую вам по возможности использовать именно этот метод.

Цикл `foreach` отлично подходит, когда вам нужно выполнить некую задачу для каждого объекта. Но предположим, что вы хотите сделать что-нибудь попроще. Что, если вы хотите выполнить некоторую задачу определенное количество раз?

Цикл for

Чтобы выполнить код заданное количество раз, используется цикл `for`. В листинге 4.12 показан синтаксис основного цикла `for`.

Листинг 4.12. Простой цикл for

```
for (❶$i = 0; ❷$i -lt 10; ❸$i++) {
❹ $i
}
```

Цикл `for` состоит из четырех частей: объявление *переменной итерации* ❶, условие продолжения выполнения цикла ❷, действие, выполняемое над переменной итерации после каждого успешного цикла ❸ и код, который нужно выполнить ❹. В этом примере цикл начинается с инициализации переменной `$i` со значением 0. Затем мы проверяем, имеет ли переменная `$i` значение меньше 10; если это так, выполняется код в фигурных скобках, который выводит значение `$i`. После выполнения кода значение `$i` увеличивается на 1 ❸, затем снова проверяется, не достигло ли значение 10 ❷. Этот процесс повторяется до тех пор, пока `$i` не перестанет быть меньше 10, то есть пока не выполнится 10 итераций.

Цикл `for` можно использовать для выполнения задачи любое количество раз — просто задайте нужное условие ❷. Однако у цикла `for` существует гораздо больше применений. Одно из самых эффективных — это управление элементами в массиве. Ранее вы видели, что с помощью цикла `foreach` такие элементы изменить *невозможно*. А теперь давайте попробуем сделать это снова, используя цикл `for`:

```
$servers = @('SERVER1','SERVER2','SERVER3','SERVER4','SERVER5')
for ($i = 0; $i –lt $servers.Length; $i++) {
    $servers[$i] = "new $server"
}
$servers
```

Попробуйте запустить этот сценарий. Имена серверов должны измениться.

Цикл `for` особенно полезен при выполнении действий, для которых нужно несколько элементов в массиве. Предположим, что ваш массив `$servers` упорядочен определенным образом и вы хотите знать порядок серверов. Для этого вы можете использовать цикл `for`:

```
for (❶$i = 1; $i -lt $servers.Length; $i++) {
    Write-Host $servers[$i] "comes after" $servers[$i-1]
}
```

Обратите внимание, что на этот раз переменная итерации имеет начальное значение 1 ❶. Так сценарий не будет пытаться обратиться к серверу с номером 0, что может привести к ошибке.

Как вы сможете позже убедиться, цикл `for` — это мощный инструмент, у которого существует множество применений, помимо тех, что мы здесь рассмотрели. А пока давайте перейдем к следующему типу цикла.

Цикл while

Цикл `while` — это самый простой цикл: делай что-то до тех пор, пока некоторое условие истинно. Чтобы получить представление о синтаксисе цикла `while`, давайте перепишем цикл `for` из листинга 4.12 так, как показано в листинге 4.13.

Листинг 4.13. Простой счетчик с использованием цикла while

```
$counter = 0
while ($counter -lt 10) {
    $counter
    $counter++
}
```

Как видите, для использования цикла `while` нужно просто поместить условие выполнения в круглые скобки, а код, который вы хотите запустить, — в фигурные.

Цикл `while` лучше всего использовать, когда количество итераций цикла заранее *не известно*. Предположим, у вас есть сервер на Windows (снова назовем его `$problemServer`), который часто выходит из строя. Но на нем есть файл, который вам нужен, и вы не хотите проверять состояние сервера каждые пять минут. Вы можете использовать цикл `while` для автоматизации этого процесса, как показано в листинге 4.14.

Листинг 4.14. Использование цикла while для работы с проблемным сервером

```
while (Test-Connection -ComputerName $problemServer -Quiet -Count 1) {
    Get-Content -Path "\\$problemServer\c$\App_configuration.txt"
    break
}
```

Используя цикл `while` вместо цикла `if`, вы можете постоянно проверять состояние сервера. Затем, когда нужный файл будет загружен, можно использовать `break` для выходя из цикла, чтобы сценарий перестал проверять сервер. Ключевое слово `break` можно использовать в любом цикле. Это особенно важно при использовании одного из наиболее распространенных подвидов цикла `while` — цикла `while($true)`. Если у вас указано `$true` в качестве условия, ваш цикл `while` будет работать вечно, пока вы не остановите его с помощью оператора `break` или ввода с клавиатуры.

Циклы do/while и do/until

Циклы `do/while` и `do/until` работают подобно циклу `while`, но противоположны друг другу: цикл `do/while` делает что-то, *пока* условие истинно, а цикл `do/until` делает что-то, пока условие *не станет* истинным.

Пустой цикл `do/while` выглядит следующим образом:

```
do {
    } while ($true)
```

Как видите, перед условием `while` приписано слово `do`. Основное различие между циклом `while` и циклом `do/while` заключается в том, что `do/while` будет выполнять код *до* проверки условия.

Это может быть удобно в определенных ситуациях, особенно если вы постоянно получаете входной сигнал от какого-то источника и хотите его обработать. Например, предположим, что вы хотите предложить пользователю выбрать лучший язык программирования. Для этого вы можете использовать код, представленный в листинге 4.15. Здесь вы будете использовать цикл `do/until`.

Листинг 4.15. Использование цикла do/until

```
do {
    $choice = Read-Host -Prompt 'What is the best programming language?'
} until ($choice -eq 'PowerShell')
Write-Host -Object 'Correct!'
```

Циклы `do/while` и `do/until` *очень* похожи. Часто вы можете за счет этого сходства выполнить одни и те же действия, используя каждый цикл, просто изменив условие, как в примерах выше.

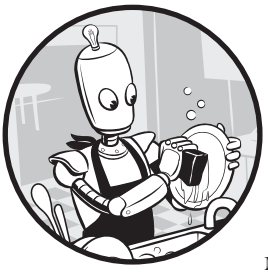
Итоги

Вы узнали много нового из этой главы. Вы изучили понятие потоков управления и то, как использовать условную логику для добавления в код альтернативных путей. Вы рассмотрели различные типы операторов потока управления, включая `if` и `switch`, и циклы `foreach`, `for` и `while`. Наконец, вы получили практический опыт использования PowerShell в задаче проверки серверов и доступа к содержащимся на них файлам.

Вы можете использовать условную логику для обработки некоторых ошибок, но все равно всегда существует вероятность что-то упустить. В главе 5 мы подробно рассмотрим ошибки и некоторые методы, которые вы можете использовать для их обработки.

5

Обработка ошибок



Мы уже рассмотрели варианты использования переменных и структур потока управления для написания гибкого кода, способного реагировать на возможные проблемы — нерабочие сервера, лежащие не на своем месте файлы и т. п. Некоторые из этих проблем вполне ожидаемы, и при случае вы сможете с ними разобраться. Но предвидеть каждую ошибку попросту невозможно. Всегда найдется что-то, что сломается. Лучшее, что вы можете сделать, — это написать код, который будет ломаться «как положено».

Это так называемая *обработка ошибок*, популярная у разработчиков методика. Она гарантирует, что код будет «готов» к возможным ошибкам и сможет их устранить или обработать. В этой главе мы познакомимся с некоторыми основными приемами обработки ошибок. Сперва поговорим о самих ошибках и определим разницу между завершающими и незавершающими ошибками. После этого мы изучим конструкцию `try/catch/finally` и, наконец, изучим встроенные переменные ошибок PowerShell.

Работа с исключениями и ошибками

В главе 4 мы уже поговорили о потоке управления, а также о том, как добавить в код разветвление пути выполнения. Если в ходе выполнения возникает проблема, код выходит из нормального потока. Событие, которое нарушает нормальный поток работы, мы будем называть *исключением*. Ошибки бывают

разные: деление на ноль, попытка получить доступ к элементу с индексом вне массива или открыть отсутствующий файл — все это заставит PowerShell *поднимать исключение*.

Если после создания исключения вы не будете его останавливать, оно будет сопровождено дополнительной информацией и выдано пользователю в виде ошибки. В PowerShell они бывают двух типов. Первый тип — это *завершающие ошибки*, то есть те, которые останавливают выполнение кода. Например, предположим, что у вас есть сценарий Get-Files.ps1, который находит список файлов в определенной папке и выполняет это же действие для каждого из файлов. Если сценарий не сможет найти папку (например, если кто-то переместил или переименовал ее), вам следует вернуть завершающую ошибку, поскольку код не сможет продолжить работу без доступа ко всем файлам. Но что, если поврежден только один из файлов?

Если вы попытаетесь получить доступ к поврежденному файлу, поднимется другое исключение. Но, поскольку вы выполняете одно и то же независимое действие для каждого файла, один поврежденный элемент не должен помешать вам обработать все остальные. В таком случае вам нужно будет написать код, который будет обрабатывать исключение, вызванное единственным поврежденным файлом, как *незавершающую ошибку*, недостаточно серьезную для остановки выполнения остальной части кода.

Стандартный способ справляться с подобными случаями — это вывести сообщение об ошибке, содержащее полезные сведения, и продолжить выполнение остальной части программы. В нескольких встроенных командах PowerShell это уже реализовано. Например, предположим, вы хотите проверить состояние служб Windows под названиями `bits`, `foo` и `lanmanserver`. Вы можете проверить их все с помощью единственной команды `Get-Service`, как показано в листинге 5.1.

Листинг 5.1. Незавершающая ошибка

```
PS> Get-Service bits,foo,lanmanserver
Get-Service : Cannot find any service with service name 'foo'.
At line:1 char:1
+ Get-Service bits,foo,lanmanserver
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
+ CategoryInfo          : ObjectNotFound: (foo:String) [Get-Service],
                          ServiceCommandException
+ FullyQualifiedErrorId : NoServiceFoundForGivenName,
                          Microsoft.PowerShell.Commands.GetServiceCommand
Status  Name               DisplayName
------  ----               -----------
Running bits               Background Intelligent Transfer Ser...
Running lanmanserver       Server
```

Никакой службы `foo`, разумеется, не существует, о чем PowerShell и сообщает вам. Но обратите внимание, что PowerShell получил статус остальных служб, при этом не прекратив выполнение после столкновения с ошибкой. Эту незавершающую ошибку можно заменить на завершающую, чтобы остальная часть кода перестала выполняться.

Важно понимать, что решение о преобразовании исключения в незавершающую или завершающую ошибку остается за разработчиком. Чаще всего, как показано в листинге 5.1, это решение принимает за вас тот, кто написал используемый вами командлет. Во многих случаях, если командлет поднимает исключение, он возвращает незавершающую ошибку и выводит сообщение о ней в консоль, позволяя вашему сценарию работать дальше. В следующем разделе мы увидим несколько способов превратить незавершающие ошибки в завершающие.

Обработка незавершающих ошибок

Допустим, вы хотите написать простой сценарий, который будет открывать папку, где, как вы знаете, лежит несколько текстовых файлов, и выводить первую строку каждого из них. Если такой папки не существует, вам нужно, чтобы сценарий немедленно завершился и сообщил об ошибке. Иначе, если вы обнаружите какие-либо другие ошибки, вам нужно, чтобы сценарий продолжил работу и при этом сообщил о них.

Мы начнем с написания сценария, который будет выдавать завершающую ошибку. В листинге 5.2 приведена первая попытка написать нечто подобное. (Я мог бы сжать код до чего-то более лаконичного, но в учебных целях я постарался сделать каждый шаг как можно более понятным.)

Листинг 5.2. Проба пера — сценарий Get-Files.ps1

```
$folderPath = '.\bogusFolder'
$files = Get-ChildItem -Path $folderPath
Write-Host "This shouldn't run."
$files.foreach({
    $fileText = Get-Content $files
    $fileText[0]
})
```

Функция `Get-ChildItem` позволяет вернуть все файлы, содержащиеся внутри пути, который вы задаете, — в данном случае это `bogusFolder`. Если вы запустите данный сценарий, то увидите следующий результат:

```
Get-ChildItem : Cannot find path 'C:\bogusFolder' because it does not exist.
At C:\Get-Files.ps1:2 char:10
+ $files = Get-ChildItem -Path $folderPath
+          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
+ CategoryInfo : ObjectNotFound: (C:\bogusFolder:String) [Get-ChildItem],
  ItemNotFoundException
    + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.
GetChildItemCommand
This shouldn't run.
```

Как вы могли заметить, произошло две вещи: PowerShell вернул ошибку, содержащую тип возникшего исключения (`ItemNotFoundException`), а затем выполнился вызов `Write-Host`. Это означает, что перед вами незавершающая ошибка.

Чтобы превратить эту ошибку в завершающую, нужно использовать параметр `ErrorAction`. Он является *общим параметром* — то есть встроенным в каждый командлет PowerShell. Параметр `ErrorAction` определяет, что нужно сделать, если рассматриваемый командлет столкнется с незавершающей ошибкой. У этого параметра есть пять основных значений:

- **Continue**: выводит сообщение об ошибке и продолжает выполнение командлета. Является значением по умолчанию.
- **Ignore**: продолжает выполнение командлета без вывода ошибки и ее записи в переменную `$Error`.
- **Inquire**: выводит сообщение об ошибке и предлагает пользователю ввести данные перед продолжением.
- **SilentlyContinue**: продолжает выполнение командлета без вывода сообщения об ошибке, но записывает ее в переменную `$Error`.
- **Stop**: выводит сообщение об ошибке и останавливает выполнение командлета.

Подробнее о переменной `$Error` вы узнаете далее в этой главе. Сейчас вам нужно передать функции `Get-ChildItem` параметр `Stop`. Отредактируйте сценарий и снова запустите код. Вы должны получить тот же результат, но уже без строки `This shouldn't run`.

Параметр `ErrorAction` полезен для управления поведением ошибок в каждом конкретном случае. Чтобы изменить метод, с помощью которого PowerShell обрабатывает все незавершающие ошибки, можно использовать `$ErrorActionPreference` — встроенную переменную, которая управляет поведением этих ошибок, настроенным по умолчанию. У переменной `$ErrorActionPreference` по умолчанию установлено значение `Continue`.

Имейте в виду, что параметр `ErrorAction` имеет приоритет над значением `$ErrorActionPreference`.

В целом, я считаю, что лучше всегда устанавливать значение `Stop` для `$ErrorActionPreference`, чтобы полностью исключить саму возможность незавершающих ошибок. Это позволяет перехватывать все типы исключений, а также избавить себя от необходимости знать заранее, какие ошибки будут завершать сценарий, а какие нет. Вы можете добиться того же результата, используя параметр `ErrorAction` для каждой команды, чтобы более точно определить, какие команды будут возвращать завершающую ошибку. Однако я бы предпочел установить правило один раз и забыть о нем, чем каждый раз добавлять параметр к новой команде.

Теперь давайте посмотрим, как обрабатывать завершающие ошибки с помощью конструкции `try/catch/finally`.

Обработка завершающих ошибок

Чтобы предотвратить остановку программы из-за завершающих ошибок, их сперва нужно *захватить*. Это можно сделать с помощью конструкции `try/catch/finally`. В листинге 5.3 показан ее синтаксис.

Листинг 5.3. Синтаксис конструкции try/catch/finally

```
try {
    # initial code
} catch {
    # code that runs if terminating error found
} finally {
    # code that runs at the end
}
```

С помощью конструкции `try/catch/finally` вы словно «подстилаете соломы» для обработки ошибок. Блок `try` содержит исходный код, который надо запустить. Если возникает завершающая ошибка, PowerShell перенаправляет поток выполнения в блок `catch`. Независимо от того, выполнится ли код в блоке `catch`, код в блоке `finally` будет выполняться всегда. Обратите внимание, что блок `finally` является необязательным, в отличие от `try` или `catch`.

Чтобы лучше понять, на что способна конструкция `try/catch/finally`, давайте вернемся к нашему сценарию Get-Files.ps1. Мы будем использовать оператор `try/catch`, чтобы сообщение об ошибке было более понятным, как в листинге 5.4.

Листинг 5.4. Использование оператора try/catch для обработки завершающих ошибок

```
$folderPath = '.\bogusFolder'
try {
    $files = Get-ChildItem -Path $folderPath –ErrorAction Stop
    $files.foreach({
        $fileText = Get-Content $files
        $fileText[0]
    })
} catch {
    $_.Exception.Message
}
```

Когда завершающая ошибка захватывается в блок `catch`, объект этой ошибки сохраняется в переменной `$_`. В этом примере мы используем переменную `$_.Exception.Message`, чтобы вернуть только сообщение об исключении. В данном случае код должен вернуть что-то вроде `Cannot find path 'C:\bogusFolder' because it does not exist`. Объекты ошибок содержат и другую информацию, в том числе тип возникшего исключения, трассировку стека (историю выполнения кода до возникновения исключения) и многое другое. Однако сейчас наиболее полезная для вас информация содержится в атрибуте `Message`, поскольку он, как правило, включает основную информацию, необходимую для понимания произошедшего в коде.

Сейчас ваш код должен работать так, как вы от него ожидаете. Передача значения `Stop` в параметр `ErrorAction` гарантирует, что отсутствующая папка вернет завершающую ошибку, которая будет захвачена. Но что произойдет, если вы столкнетесь с ошибкой при попытке использовать функцию `Get-Content` для доступа к файлу?

В качестве эксперимента попробуйте запустить следующий код:

```
$filePath = '.\bogusFile.txt'
try {
    Get-Content $filePath
} catch {
    Write-Host "We found an error"
}
```

Сейчас вы должны получить сообщение об ошибке от PowerShell, а не то, которое вы сами написали в блоке `catch`. Это связано с тем, что функция `Get-Content` возвращает незавершающую ошибку, если элемент не найден, а конструкция `try/catch` ловит только завершающие. Это означает, что код в листинге 5.4 будет работать так, как задумано, — любые ошибки доступа к файлам не остановят выполнение программы, а просто вернутся в консоль.

Обратите внимание, что в этом коде мы не использовали блок `finally`. Он отлично подходит для размещения кода, который выполняет задачи чистки вроде отключения открытых подключений к базе данных, очистки сеансов удаленного взаимодействия PowerShell и т. д. В данном случае нам это не было нужно.

Изучение автоматической переменной $Error

На протяжении всей этой главы мы возвращали множество ошибок. Независимо от их типа, все они сохранялись в автоматической переменной PowerShell под названием `$Error`. Это встроенная переменная, в которой хранится массив всех ошибок, возвращенных в текущем сеансе PowerShell и расположенных в порядке времени их появления.

Чтобы посмотреть, как работает переменная `$Error`, давайте перейдем в консоль и запустим команду, которая, как мы уже знаем, вернет завершающую ошибку (листинг 5.5).

Листинг 5.5. Пример ошибки

```
PS> Get-Item -Path C:\NotFound.txt
Get-Item : Cannot find path 'C:\NotFound.txt' because it does not exist.
At line:1 char:1
+ Get-Item -Path C:\NotFound.txt
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
+ CategoryInfo : ObjectNotFound: (C:\NotFound.txt:String) [Get-Item],
                 ItemNotFoundException
    + FullyQualifiedErrorId :
                 PathNotFound,Microsoft.PowerShell.Commands.GetItemCommand
```

Теперь в том же сеансе PowerShell выведем переменную `$Error` (листинг 5.6).

Листинг 5.6. Переменная $Error

```
PS> $Error
Get-Item : Cannot find path 'C:\NotFound.txt' because it does not exist.
At line:1 char:1
+ Get-Item -Path C:\NotFound.txt
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
+ CategoryInfo : ObjectNotFound: (C:\NotFound.txt:String) [Get-Item],
                 ItemNotFoundException
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.
GetItemCommand
--пропуск--
```

Если текущий сеанс работы начался давно, вы, скорее всего, увидите длинный список ошибок. Чтобы получить доступ к конкретному элементу, можно

использовать обращение по индексу, как и при работе с любыми другими массивами. Новые ошибки добавляются в начало массива `$Error`, поэтому элемент `$Error[0]` будет содержать самую недавнюю ошибку, `$Error[1]` — предыдущую, и т. д.

Итоги

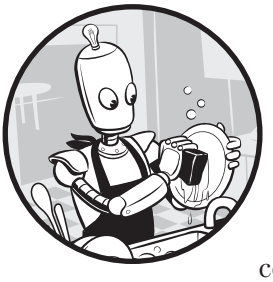
Обработка ошибок в PowerShell — это довольно обширная тема, и в этой главе мы лишь разобрали ее основы. Если вы хотите погрузиться в нее глубже, обратитесь к справке `about_try_catch_finally`, запустив команду `Get-Help about_try_catch_finally`. Еще один вариант — книга «Big Book of PowerShell Error Handling» Дэйва Вятта (Dave Wyatt) из DevOps Collective (https://leanpub.com/thebigbookofpowershellerrorhandling/).

Самое главное, что мы здесь усвоили, — это понимание разницы между завершающими и незавершающими ошибками, способы использования оператора `try/catch` и различные варианты параметра `ErrorAction`, которые помогут вам развить навыки, необходимые для обработки любых возможных ошибок, которые вам может выкинуть код.

До сих пор вы все делали в одном блоке кода. В следующей главе вы увидите, как разделить код на отдельные исполняемые модули под названием *функции*.

6

Пишем функции



До сих пор код наш код был относительно одномерным: у ваших сценариев была лишь одна задача. И хотя нет ничего плохого в сценарии, который всего лишь просматривает файлы в папке, при написании более серьезных инструментов PowerShell вам может понадобиться такой код, который будет выполнять несколько задач. Ничто не мешает вам добавлять в сценарий все больше элементов. Вы можете написать тысячу строк кода, который будет выполнять сотни задач, и расположить их все в одном непрерывном блоке. Но такой сценарий слишком беспорядочен и неудобен для чтения и работы. Вы можете выделить каждую задачу в отдельный сценарий, но пользоваться этим будет невозможно. Вам нужен один инструмент, который может делать все, что нужно, а не сотня инструментов для сотни задач.

Для этого мы будем выделять каждую задачу в отдельную *функцию* — именованный фрагмент кода, который выполняет одну задачу. Функция определяется один раз. Нужно просто один раз написать решение для некоторой задачи, затем сохранить его в функции, и теперь всякий раз, когда эта задача возникает, вам достаточно просто *вызывать* функцию для ее решения. Функции значительно повышают удобство использования и читаемость вашего кода, что сильно упрощает работу. В этой главе мы научимся создавать функции, добавлять к ним параметры, а также настраивать функции на прием входных данных конвейера. Но сначала разберемся с терминами.

Функции и командлеты

Само понятие функции звучит знакомо. Это, возможно, связано с тем, что функции немного похожи на командлеты, которые вы раньше использовали, вроде `Start-Service` и `Write-Host`. Они тоже представляют из себя именованные фрагменты кода, которые решают определенную задачу. Разница между функцией и командлетом заключается в том, *как именно* создается каждая из этих конструкций. Командлеты пишутся на другом языке (обычно на чем-то вроде C#), после чего компилируются и используются внутри PowerShell. А вот функции пишутся на самом языке сценариев PowerShell.

Чтобы посмотреть, какие команды являются командлетами, а какие — функциями, можно использовать командлет `Get-Command` и его параметр `CommandType`, как показано в листинге 6.1.

Листинг 6.1. Вывод доступных функций

```
PS> Get-Command -CommandType Function
```

С помощью этой команды отображаются все функции, загруженные в ваш текущий сеанс PowerShell или в модули, доступные PowerShell (в главе 7 мы поговорим о них более подробно). Чтобы просмотреть другие функции, нужно скопировать и вставить их в консоль, добавить их в доступный модуль или *сослаться через точку* (это мы также рассмотрим позже).

Раз уж мы с этим разобрались, давайте начнем писать функции.

Определение функции

Прежде чем использовать функцию, сперва необходимо ее определить. Для этого нужно использовать ключевое слово `function`, за которым следует задаваемое пользователем описательное имя, а после него — набор фигурных скобок. Внутри последних находится блок сценария, который должен выполнить PowerShell. В листинге 6.2 определяется и выполняется простая функция.

Листинг 6.2. Вывод сообщения в консоль с помощью простой функции

```
PS> function Install-Software { Write-Host 'I installed some software, Yippee!' }
PS> Install-Software
I installed some software, Yippee!
```

В новой функции `Install-Software` для вывода сообщения в консоль используется команда `Write-Host`. Определив функцию, можно использовать ее имя для выполнения кода внутри ее блока сценария.

Имена функций — важная деталь. Вы можете назвать функцию как угодно, но имя всегда должно описывать то, что она делает. Соглашение об именах функций в PowerShell следует синтаксису «глагол-существительное». Считается, что лучше всегда использовать этот синтаксис, если в другом нет острой необходимости. Можно использовать команду `Get-Verb`, чтобы просмотреть список рекомендуемых глаголов. В качестве существительного обычно выступает имя объекта, с которым вы работаете, в единственном числе (в данном случае `Software`).

Если вы хотите изменить поведение функции, вы можете переопределить ее, как показано в листинге 6.3.

Листинг 6.3. Переопределение функции Install-Software для изменения ее поведения

```
PS> function Install-Software { Write-Host 'You installed some software, Yay!' }
PS> Install-Software
You installed some software, Yay!
```

Теперь, когда вы переопределили функцию `Install-Software`, на экране появится немного другое сообщение.

Функции можно определять в сценарии либо вводить непосредственно в консоль. В листинге 6.2 у вас была небольшая функция, поэтому написать ее прямо в консоли было несложно. В большинстве случаев функции у вас будут объемными, и проще будет определить их в сценарии или модуле, который затем можно вызвать для загрузки функции в память. Как вы могли понять на примере листинга 6.3, необходимость вводить сотни строк функции каждый раз, когда вы хотите изменить ее, может немного раздражать.

В оставшейся части этой главы мы научим функцию `Install-Software` принимать параметры и входные данные конвейера. Я предлагаю вам открыть ваш любимый текстовый редактор и по ходу главы записывать функцию в файле `.ps1`.

Добавление параметров в функции

У функций в PowerShell может быть любое количество параметров. Когда вы создаете собственные функции, вы можете самостоятельно добавлять в них параметры и определять способ их работы. Параметры могут быть обязательными или нет. Они также могут принимать либо любое значение, либо некоторое из определенного списка возможных аргументов.

Например, вымышленное ПО, которое вы устанавливаете с помощью функции `Install-Software`, может иметь несколько версий, но пока что сама функция не дает пользователю указать версию для установки. Если бы эту функцию использовали только вы, то можно было бы просто переопределять ее под каждую конкретную версию. Однако это было бы пустой тратой времени, а также чревато ошибками, не говоря уже о том, что ваш код должен быть доступен и для других пользователей.

Добавление в функцию параметров позволяет внести в нее вариативность. Подобно тому как с помощью переменных вы пишете сценарии для обработки разных вариантов одной и той же ситуации, параметры позволяют вам создавать функцию, выполняющую одну задачу разными способами. В данном случае мы хотим установить несколько версий одного и того же ПО на разных компьютерах.

Давайте сначала добавим в функцию параметр, который позволит указать устанавливаемую версию.

Создание простого параметра

Для создания параметра функции требуется блок `param`, в котором перечисляются все ее параметры. Можно определить этот блок с ключевым словом `param`, за которым следуют круглые скобки, как показано в листинге 6.4.

Листинг 6.4. Определение блока параметров

```
function Install-Software {
    [CmdletBinding()]
    param()

    Write-Host 'I installed software version 2. Yippee!'
}
```

ПРИМЕЧАНИЕ

В этой книге мы будем давать примеры создания только расширенных функций. Существуют также базовые функции, но в настоящее время они обычно используются только для узконаправленных, нишевых случаев. Различия между ними незначительны, поэтому мы не будем вдаваться в подробности. Тем не менее, если вы видите под именем функции ссылку `[CmdletBinding()]` или параметр, определенный как `[Parameter()]`, то речь идет именно о расширенной функции.

Сейчас действие вашей функции не изменилось. Мы лишь выполнили ее подготовку, и теперь она готова к приему параметров. С помощью команды

`Write-Host` мы сымитируем установку программного обеспечения, чтобы вы могли сосредоточиться на написании функции.

После добавления блока `param` можно создать параметр, поместив его в круглые скобки блока, как показано в листинге 6.5.

Листинг 6.5. Создание параметра

```
function Install-Software {
    [CmdletBinding()]
    param(
   ❶ [Parameter()]
   ❷ [string] $Version
    )
  ❸ Write-Host "I installed software version $Version. Yippee!"
}
```

Внутри блока `param` сначала определяется пустой блок `Parameter` ❶. Это обязательный блок, но он ничего не делает (в следующем разделе я объясню, как его использовать).

Давайте лучше сосредоточимся на типе `[string]` ❷ перед именем параметра. Поместив его в квадратные скобки перед именем переменной, можно преобразовать параметр так, чтобы PowerShell всегда пытался преобразовать в строку любое значение, переданное этому параметру, если оно не было таким изначально. В данном случае все, что передано в переменной `$Version`, будет всегда рассматриваться как строка. Не обязательно приводить ваш параметр к типу, но я настоятельно рекомендую это делать, потому что так будет появляться гораздо меньше ошибок.

Кроме того, мы добавляем переменную `$Version` в оператор `print` ❸, и поэтому при запуске команды `Install-Software` с параметром `Version` и передачи номера версии получается результат, показанный в листинге 6.6.

Листинг 6.6. Передача параметра вашей функции

```
PS> Install-Software -Version 2
I installed software version 2. Yippee!
```

Теперь мы определили для своей функции рабочий параметр. Посмотрим, что мы сможем с ним сделать.

Атрибут параметра Mandatory

Блок `Parameter` можно использовать для управления различными *атрибутами параметра*, что позволяет изменять его поведение. Например, если вы хотите

убедиться, что объект, вызывающий функцию, должен передать заданный параметр, вы можете определить этот параметр как обязательный.

По умолчанию параметры не являются обязательными. Давайте заставим пользователя передать номер версии, используя ключевое слово `Mandatory` внутри блока `Parameter`, как показано в листинге 6.7.

Листинг 6.7. Использование обязательного параметра

```
function Install-Software {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory)]
        [string]$Version
    )

    Write-Host "I installed software version $Version. Yippee!"
}
Install-Software
```

После запуска кода вы должны получить следующий запрос:

```
cmdlet Install-Software at command pipeline position 1
Supply values for the following parameters:
Version:
```

После установки атрибута `Mandatory` выполнение функции без параметра будет остановлено до тех пор, пока пользователь не введет нужное значение. Функция будет ждать, пока пользователь не задаст значение для параметра `Version`. Как только это будет сделано, PowerShell выполнит функцию и продолжит работу. Чтобы избежать появления этого запроса, при вызове функции просто передайте значение в виде параметра с синтаксисом *-ИмяПараметра* — например, `Install-Software -Version 2`.

Значения параметров по умолчанию

Вы также можете присвоить определенному параметру значение по умолчанию. Это удобно, когда по большей части у параметра должно быть определенное значение. Например, если в 90% случаев требуется установить версию № 2 и вы не хотите каждый раз задавать ее при запуске функции, то можно присвоить параметру `$Version` значение 2 по умолчанию, как показано в листинге 6.8.

Наличие у параметра значения по умолчанию не мешает вам передавать свое — ваше значение просто переопределит заданное по умолчанию.

Листинг 6.8. Использование значения параметра по умолчанию

```
function Install-Software {
    [CmdletBinding()]
    param(
        [Parameter()]
        [string]$Version = 2
    )

    Write-Host "I installed software version $Version. Yippee!"
}
Install-Software
```

Добавление атрибутов проверки параметров

Мы можем не только делать параметры обязательными и задавать для них значения по умолчанию, но также ограничить их возможные значения с помощью *атрибутов проверки параметров*. Стоит по возможности ограничивать количество информации, которую пользователи (или даже вы!) передают функциям или сценариям, — это исключит ненужный код внутри функции. Предположим, что вы передаете значение 3 функции `Install-Software`, полагая, что такая версия существует. Сейчас наша функция подразумевает, что все пользователи знают обо всех существующих версиях, поэтому она не учитывает, что произойдет, если вы, например, укажете версию 4. В этом случае функция не сможет найти подходящую папку, ведь ее не существует.

В листинге 6.9 мы использовали строку `$Version` при указании пути. Если кто-то передаст значение, которое не образует имя существующей папки (например, `SoftwareV3` или `SoftwareV4`), код выполнится с ошибкой.

Листинг 6.9. Предполагаемые значения параметров

```
function Install-Software {
    param(
        [Parameter(Mandatory)]
        [string]$Version
    )
    Get-ChildItem -Path \\SRV1\Installers\SoftwareV$Version
}

Install-Software -Version 3
```

Ошибка получится вот такой:

```
Get-ChildItem : Cannot find path '\\SRV1\Installers\SoftwareV3'
because it does not exist.
At line:7 char:5
```

```
+     Get-ChildItem -Path \\SRV1\Installers\SoftwareV3
+     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (\\SRV1\Installers\SoftwareV3:String)
                              [Get-ChildItem], ItemNotFoundException
    + FullyQualifiedErrorId :
PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand
```

Вы можете написать код обработки ошибок с учетом этой проблемы либо же вообще предотвратить проблему, разрешив пользователю передавать только существующую версию ПО. Чтобы ограничить возможные варианты пользовательского ввода, добавляется проверка параметров.

Существуют разные виды проверки параметров, но в случае с вашей функцией `Install-Software` лучше всего сработает атрибут `ValidateSet`. Он позволяет указать список разрешенных для параметра значений. Если передаваться может только строка 1 или 2, убедитесь, что пользователь укажет только эти значения, иначе функция немедленно прекратит работу и сообщит ему о причинах.

Давайте добавим атрибуты проверки параметров внутри блока `param` прямо под блоком `Parameter`, как показано в листинге 6.10.

Листинг 6.10. Использование атрибута проверки параметра ValidateSet

```
function Install-Software {
    param(
        [Parameter(Mandatory)]
        [ValidateSet('1','2')]
        [string]$Version
    )
    Get-ChildItem -Path \\SRV1\Installers\SoftwareV$Version
}

Install-Software -Version 3
```

Набор элементов 1 и 2 добавляется в конечные скобки атрибута `ValidateSet`. Так вы сообщаете PowerShell, что единственными допустимыми значениями `Version` являются только 1 или 2. Если пользователь пытается передать значение, которого нет в наборе, он получит текст ошибки (листинг 6.11), сообщающий об ограниченном числе доступных вариантов.

Листинг 6.11. Передача значения параметра, которого нет в блоке ValidateSet

```
Install-Software : Cannot validate argument on parameter 'Version'. The argument
"3" does not belong to the set "1,2" specified by the ValidateSet attribute.
Supply an argument that is in the set and then try the command again.
At line:1 char:25
```

```
+ Install-Software -Version 3
+                          ~~~~
+ CategoryInfo          : InvalidData: (:) [Install-
                          Software],ParameterBindingValidationException
    + FullyQualifiedErrorId : ParameterArgumentValidationError,Install-Software
```

Для проверки чаще всего используется атрибут `ValidateSet`, но кроме него есть и другие. Чтобы получить полную информацию обо всех способах ограничения значений параметров, ознакомьтесь с разделом справки `Functions_Advanced_Parameters`, запустив команду `Get-Help about_Functions_Advanced_Parameters`.

Прием входных данных конвейера

До сих пор вы создавали функцию с параметром, который можно передать только с использованием типичного синтаксиса *-ИмяПараметра <Значение>*. Но в главе 3 вы узнали, что в PowerShell существует конвейер, который позволяет вам легко и просто передавать объекты от одной команды к другой. Вспомните, что у некоторых функций не было поддержки конвейера, но для собственных функций вы можете ее настраивать. Давайте добавим к нашей функции `Install-Software` возможности конвейера.

Добавление еще одного параметра

Сначала добавьте в ваш код еще один параметр с указанием компьютера, на который нужно установить программное обеспечение. Мы также добавим этот параметр в команду `Write-Host`, чтобы сымитировать установку. Добавление показано в листинге 6.12.

Листинг 6.12. Добавление параметра ComputerName

```
function Install-Software {
    param(
        [Parameter(Mandatory)]
        [string]$Version
        [ValidateSet('1','2')],

        [Parameter(Mandatory)]
        [string]$ComputerName
    )
    Write-Host "I installed software version $Version on $ComputerName. Yippee!"

}

Install-Software -Version 2 -ComputerName "SRV1"
```

Как и в случае с переменной `$Version`, мы добавили параметр `ComputerName` в блок `param`.

После добавления функции параметра `ComputerName` вы можете перебирать список имен компьютеров и передавать эти значения и версии в функцию `Install-Software`, например:

```
$computers = @("SRV1", "SRV2", "SRV3")
foreach ($pc in $computers) {
    Install-Software -Version 2 -ComputerName $pc
}
```

Но, как вы сами уже могли убедиться, от таких циклов `foreach` лучше отказаться и вместо этого использовать конвейер.

Организация совместимости функции с конвейером

К сожалению, если вы попытаетесь с ходу использовать конвейер, то получите ошибку. Прежде чем добавлять в функцию поддержку конвейера, нужно решить, какой тип ввода должна принимать ваша функция. Как вы уже знаете после прочтения главы 3, у функций PowerShell есть два типа конвейерного ввода: `ByValue` (весь объект) и `ByPropertyName` (отдельное свойство объекта). В нашем случае, поскольку список `$computers` содержит только строки, их можно передать через `ByValue`.

Чтобы добавить поддержку конвейера, необходимо добавить атрибут к нужному параметру, используя одно из двух ключевых слов: `ValueFromPipeline` или `ValueFrom PipelineByPropertyName`, как показано в листинге 6.13.

Листинг 6.13. Добавление поддержки конвейера

```
function Install-Software {
    param(
        [Parameter(Mandatory)]
        [string]$Version
        [ValidateSet('1','2')],

        [Parameter(Mandatory, ValueFromPipeline)]
        [string]$ComputerName
    )
    Write-Host "I installed software version $Version on $ComputerName. Yippee!"
}

$computers = @("SRV1", "SRV2", "SRV3")
$computers | Install-Software -Version 2
```

Снова запустите ваш сценарий — вы должны увидеть что-то вроде этого:

```
I installed software version 2 on SRV3. Yippee!
```

Обратите внимание, что функция `Install-Software` выполняется только для последней строки в массиве. В следующем разделе мы посмотрим, как это можно исправить.

Добавление блока process

Чтобы PowerShell выполнил эту функцию для всех поступающих в нее объектов, нужно включить в нее блок `process`. Внутри этого блока вы помещаете тот код, который хотите выполнять каждый раз, когда функция получает входные данные конвейера. Добавьте в ваш сценарий блок `process`, как показано в листинге 6.14.

Листинг 6.14. Добавление блока process

```
function Install-Software {
    param(
        [Parameter(Mandatory)]
        [string]$Version
        [ValidateSet('1','2')],

        [Parameter(Mandatory, ValueFromPipeline)]
        [string]$ComputerName
    )
    process {
        Write-Host "I installed software version $Version on $ComputerName. Yippee!"
    }
}

$computers = @("SRV1", "SRV2", "SRV3")
$computers | Install-Software -Version 2
```

Обратите внимание, что за ключевым словом `process` следует набор фигурных скобок, в которых находится код, выполняемый вашей функцией.

Благодаря блоку `process` вы увидите вывод всех трех серверов в массиве `$computers`:

```
I installed software version 2 on SRV1. Yippee!
I installed software version 2 on SRV2. Yippee!
I installed software version 2 on SRV3. Yippee!
```

В блоке `process` содержится основной код, который вы хотите выполнить. Также можно использовать блоки `begin` и `end` для ограничения того кода, который будет выполняться в начале и в конце вызова функции. Для получения

дополнительной информации о создании расширенных функций с блоками `begin`, `process` и `end` ознакомьтесь со справкой `_Functions_Advanced`, выполнив команду `Get-Help about_Functions_Advanced`.

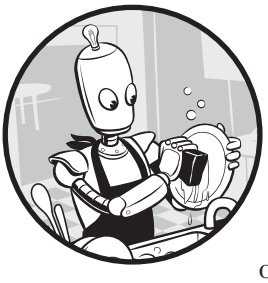
Итоги

Функции позволяют разделить код на отдельные блоки. Это не только помогает разбить вашу работу на мелкие фрагменты, которыми проще управлять, но и делает код более читабельным и удобным для тестирования. Если вы будете использовать для своих функций описательные имена, код станет «сам себе документацией», и любой, кто будет его читать, поймет, как и что работает.

В этой главе мы узнали основы работы с функциями — как можно их определять, уточнять параметры и их атрибуты, а также принимать ввод из конвейера. В следующей главе вы увидите, как можно объединять функции с помощью модулей.

7

Изучаем модули



В предыдущей главе мы изучили функции. Они позволяют разбить сценарий на управляемые единицы, чтобы код был более эффективным и читаемым. Но зачем нам ограничивать хорошую функцию рамками одного сценария или сеанса консоли? В этой главе мы поговорим о *модулях* — группах схожих функций, собранных вместе для использования в различных сценариях.

PowerShell-модуль в чистом виде представляет собой текстовый файл с расширением .psm1 и кое-какими дополнительными метаданными. Существуют и другие типы — *бинарные модули* и *динамические модули*, но эти понятия выходят за рамки данной книги.

Любая команда, которая явным образом не выполнялась в вашем сеансе, почти наверняка исходит из модуля. Многие из команд, которые вы использовали в этой книге, являются частью встроенных модулей Microsoft, которые идут в комплекте с PowerShell. Помимо них также бывают сторонние модули, в том числе те, которые вы сами создаете. Чтобы использовать модуль, его сначала нужно установить. Затем, если необходимо использовать команду из модуля, сам модуль должен быть импортирован в ваш сеанс. Начиная с версии PowerShell v3, среда автоматически импортирует модуль при попытке вызвать из него команду.

Мы начнем эту главу с изучения модулей, которые уже установлены в вашей системе. Затем мы рассмотрим структуру модуля и его составные части, а после этого узнаем, как загружать и устанавливать модули из PowerShell Gallery.

Изучение встроенных модулей

В комплекте с PowerShell идет немало встроенных модулей. В этом разделе вы увидите, как находить и импортировать модули из вашего сеанса.

Поиск модулей в сеансе

Просмотреть импортированные в текущий сеанс модули можно с помощью командлета `Get-Module` (который сам является частью модуля). Это команда, которая позволяет просмотреть все имеющиеся в системе модули, доступные в текущем сеансе.

Запустите новый сеанс PowerShell и выполните командлет `Get-Module`, как показано в листинге 7.1.

Листинг 7.1. Просмотр импортированных модулей с помощью команды Get-Module

```
PS> Get-Module

ModuleType Version  Name                              ExportedCommands
---------- -------  ----                              ----------------
Manifest   3.1.0.0  Microsoft.PowerShell.Management  {Add-Computer, Add-Content...
--пропуск--
```

Каждая строка выходных данных команды `Get-Module` представляет собой модуль, который был импортирован в текущий сеанс. Это означает, что все команды этого модуля можно использовать в любой момент. Модули `Microsoft.PowerShell.Management` и `Microsoft.PowerShell.Utility` по умолчанию импортируются в любой сеанс PowerShell.

Обратите внимание на столбец `ExportedCommands` в листинге 7.1. Это те команды из модуля, которые вы можете использовать. Все варианты можно посмотреть с помощью `Get-Command` с указанием имени модуля. Давайте проверим все экспортированные команды внутри модуля `Microsoft.PowerShell.Management`, как показано в листинге 7.2.

Листинг 7.2. Просмотр команд внутри модуля PowerShell

```
PS> Get-Command -Module Microsoft.PowerShell.Management

CommandType     Name                Version    Source
-----------     ----                -------    ------
Cmdlet          Add-Computer        3.1.0.0    Microsoft.PowerShell.Management
Cmdlet          Add-Content         3.1.0.0    Microsoft.PowerShell.Management
--пропуск--
```

Здесь перечислены все команды, которые были экспортированы из этого модуля — их можно вызвать и вне его. Некоторые авторы любят включать в свои модули функции, недоступные для пользователя. Любая функция, которая не экспортируется и работает только внутри сценария или модуля, называется *приватной функцией* или, как иногда говорят, *вспомогательной функцией*.

Использование команды `Get-Module` без каких-либо параметров вернет все импортированные модули, но что насчет модулей, которые были установлены, но не импортированы?

Поиск модулей на вашем компьютере

Чтобы вывести список всех модулей, которые установлены в системе и могут быть импортированы в ваш сеанс, можно использовать команду `Get-Module` с параметром `ListAvailable`, как показано в листинге 7.3.

Листинг 7.3. Использование команды Get-Module для вывода всех доступных модулей

```
PS> Get-Module -ListAvailable
   Directory: C:\Program Files\WindowsPowerShell\Modules

ModuleType Version   Name                ExportedCommands
---------- -------   ----                ----------------
Script     1.2       PSReadline          {Get-PSReadlineKeyHandler, Set-PS...


   Directory:\Modules


ModuleType Version   Name                ExportedCommands
---------- -------   ----                ----------------
Manifest   1.0.0.0   ActiveDirectory     {Add-ADCentralAccessPolicyMember...
Manifest   1.0.0.0   AppBackgroundTask   {Disable-AppBackgroundTaskDiagnosticLog...
--пропуск--
```

Параметр `ListAvailable` просит PowerShell проверить наличие в папках вложенных папок с файлами .psm1. PowerShell прочитает каждый из этих модулей в файловой системе и вернет вам имя, метаданные и список функций каждого модуля, которые можно использовать.

PowerShell ищет модули в нескольких местах на жестком диске, заданных по умолчанию типом модуля:

- **Системные модули**: почти все модули, которые идут в комплекте с PowerShell, будут расположены в папке C:\Windows\System32\WindowsPowerShell\1.0\Modules.

Этот путь обычно используется только для внутренних PowerShell-модулей. В принципе, вы можете размещать в этой папке и свои модули, но делать это не рекомендуется.

- **Модули, доступные всем пользователям**: модули также хранятся в папке C:\Program Files\WindowsPowerShell\Modules. Этот путь обычно называют путем *«Для всех пользователей»*, и именно здесь находятся все модули, которые должны быть доступны всем пользователям системы.
- **Модули текущего пользователя**: наконец, вы можете хранить модули в папке C:\Users\<ТекущийПользователь>\Documents\WindowsPowerShell\Modules. Внутри этой папки вы найдете все созданные или загруженные вами модули, которые доступны только текущему пользователю. Размещение модулей по этому пути позволяет разделить их на случай, если за компьютером работают несколько пользователей.

При вызове команды `Get-Module -ListAvailable` PowerShell считывает все пути к этим папкам и возвращает все расположенные там модули. Тем не менее это лишь пути для модулей по умолчанию — при необходимости вы можете добавить свои.

Можно создать новый путь к модулю с помощью переменной среды `$PSModulePath`, в которой определены все папки с модулями. Они разделены точкой с запятой, как показано в листинге 7.4.

Листинг 7.4. Переменная среды PSModulePath

```
PS> $env:PSModulePath
C:\Users\Adam\Documents\WindowsPowerShell\Modules;
C:\Program Files\WindowsPowerShell\Modules\Modules;
C:\Program Files (x86)\Microsoft SQL Server\140\Tools\PowerShell\Modules\
```

Вы можете добавить папку в переменную среды `PSModulePath`, выполнив небольшой синтаксический анализ строки. Этот метод немного сложноват, но все же им стоит воспользоваться:

```
PS> $env:PSModulePath + ';C;\MyNewModulePath'.
```

Имейте в виду, что в этом случае новая папка добавится только в текущий сеанс. Чтобы сделать это изменение постоянным, нужно использовать метод `SetEnvironment Variable()` в `.NET`-классе `Environment`:

```
PS> $CurrentValue = [Environment]::GetEnvironmentVariable("PSModulePath",
"Machine")
PS> [Environment]::SetEnvironmentVariable("PSModulePath", $CurrentValue + ";C:\
MyNewModulePath", "Machine")
```

Давайте теперь импортируем модули и посмотрим, как их можно использовать.

Импорт модулей

Если путь к папке модуля уже помещен в переменную среды `PSModulePath`, вам остается лишь импортировать его в текущий сеанс. Сейчас у PowerShell есть функция автоматического импорта, и, если у вас установлен модуль, просто вызовите нужную функцию, а PowerShell автоматически ее туда импортирует. Тем не менее важно понимать саму процедуру импорта.

Давайте воспользуемся встроенным модулем под названием `Microsoft.PowerShell.Management`. В листинге 7.5 мы запускаем команду `Get-Module` дважды: один раз в новом сеансе PowerShell и еще один после использования команды `cd`, псевдонима команды `Set-Location` из модуля `Microsoft.PowerShell.Management`. Посмотрим, что происходит.

Листинг 7.5. PowerShell автоматически импортирует модуль Microsoft.PowerShell.Management после использования команды cd

```
PS> Get-Module

ModuleType Version  Name                                ExportedCommands
---------- -------  ----                                ----------------
Manifest   3.1.0.0  Microsoft.PowerShell.Utility     {Add-Member, Add-Type...
Script     1.2      PSReadline                          {Get-PSReadlineKeyHandler...

PS> cd\
PS> Get-Module

ModuleType Version  Name                                ExportedCommands
---------- -------  ----                                ----------------
Manifest   3.1.0.0  Microsoft.PowerShell.Management  {Add-Computer, Add-Content...
Manifest   3.1.0.0  Microsoft.PowerShell.Utility     {Add-Member, Add-Type...
Script     1.2      PSReadline                          {Get-PSReadlineKeyHandler....
```

Как видите, модуль `Microsoft.PowerShell.Management` автоматически импортировался после использования команды `cd`. Функция автоматического импорта обычно справляется сама. Если команда внутри модуля по какой-то причине недоступна, то, вероятно, есть проблема с модулем.

Чтобы вручную импортировать модуль, используйте команду `Import-Module`, как показано в листинге 7.6.

Листинг 7.6. Ручной импорт, повторный импорт и удаление модуля

```
PS> Import-Module -Name Microsoft.PowerShell.Management
PS> Import-Module -Name Microsoft.PowerShell.Management -Force
PS> Remove-Module -Name Microsoft.PowerShell.Management
```

Обратите внимание, что в этом листинге также используются параметр `Force` и команда `Remove-Module`. Если модуль был изменен (скажем, если вы внесли изменения в пользовательский модуль), можно использовать команду `Import-Module` с параметром `Force` для выгрузки и повторного импорта модуля. В свою очередь, команда `Remove-Module` выгружает модуль из сеанса.

Компоненты PowerShell-модуля

Теперь давайте посмотрим, как устроены PowerShell-модули изнутри.

Файл .psm1

Любой текстовый файл с расширением `.psm1` может быть PowerShell-модулем. Но чтобы этот файл делал что-то полезное, в нем должны быть функции. Все функции внутри модуля лучше строить вокруг некоторой общей концепции. Например, в листинге 7.7 показаны некоторые функции, связанные с установкой программного обеспечения.

Листинг 7.7. Функции, связанные с установкой программного обеспечения

```
function Get-Software {
    param()
}

function Install-Software {
    param()
}

function Remove-Software {
    param()
}
```

Обратите внимание, что в имени каждой команды существительные повторяются — изменяется только глагол. При создании модулей так делать лучше всего. Если вам нужно изменить существительное, возможно, стоит разбить один модуль на несколько.

Манифест модуля

Помимо файла `.psm1` у модуля также есть манифест или файл `.psd1`. *Манифест модуля* — это текстовый файл, написанный в виде хеш-таблицы PowerShell. Это не обязательный, но рекомендуемый элемент. Эта хеш-таблица содержит метаданные о модуле.

Можно создать манифест модуля с нуля, но в PowerShell есть команда New-ModuleManifest, которая может сгенерировать для вас шаблон. Давайте воспользуемся командой New-ModuleManifest, чтобы создать манифест модуля для нашего программного пакета, как показано в листинге 7.8.

Листинг 7.8. Использование команды New-ModuleManifest для создания манифеста модуля

```
PS> New-ModuleManifest -Path 'C:\Program Files\WindowsPowerShell\Modules\
Software\Software.psd1'
-Author 'Adam Bertram' -RootModule Software.psm1
-Description 'This module helps in deploying software.'
```

Эта команда создает файл .psd1, который выглядит следующим образом:

```
#
# Module manifest for module 'Software'
#
# Generated by: Adam Bertram
#
# Generated on: 11/4/2019
#

@{

# Script module or binary module file associated with this manifest.
RootModule = 'Software.psm1'

# Version number of this module.
ModuleVersion = '1.0'

# Supported PSEditions
# CompatiblePSEditions = @()

# ID used to uniquely identify this module
GUID = 'c9f51fa4-8a20-4d35-a9e8-1a960566483e'

# Author of this module
Author = 'Adam Bertram'

# Company or vendor of this module
CompanyName = 'Unknown'

# Copyright statement for this module
Copyright = '(c) 2019 Adam Bertram. All rights reserved.'

# Description of the functionality provided by this module
Description = 'This modules helps in deploying software.'

# Minimum version of the Windows PowerShell engine required by this module
```

```
# PowerShellVersion = ''

# Name of the Windows PowerShell host required by this module
# PowerShellHostName = ''
--пропуск--
}
```

После запуска команды вы увидите, что там есть много полей, для которых я не указал параметры. Мы не будем углубляться в манифесты модулей. На данный момент вам достаточно знать о необходимости определять как минимум параметры `RootModule`, `Author`, `Description` и, возможно, `Version`. Все эти атрибуты необязательны, но вам стоит завести привычку добавлять как можно больше информации в манифест модуля.

Теперь, когда вы ознакомились с анатомией модуля, давайте посмотрим, как его загрузить и установить.

Работа с пользовательскими модулями

До сих пор мы работали только с модулями, установленными в PowerShell по умолчанию. В этом разделе мы узнаем, как искать, устанавливать и удалять пользовательские модули.

Поиск модулей

Одно из лучших свойств модулей — это возможность делиться ими: зачем тратить время на решение уже решенной задачи? Скорее всего, оно уже есть в PowerShell Gallery. *PowerShell Gallery* (https://www.powershellgallery.com) — это репозиторий тысяч модулей и сценариев PowerShell. Их может загружать любой пользователь, у которого есть учетная запись. В репозитории есть различные модули, которые создают как обычные люди, так и гигантские корпорации вроде Microsoft.

К счастью, вы также можете использовать Gallery из самой PowerShell. В PowerShell есть встроенный модуль `PowerShellGet`, где можно найти простые команды для взаимодействия с PowerShell Gallery. В листинге 7.9 используется командлет `Get-Command` для вывода команд `PowerShellGet`.

В модуле `PowerShellGet` есть команды для поиска, сохранения и установки чужих модулей, а также для публикации своих. Нам до этого пока далеко (мы даже еще ни одного модуля не написали), поэтому сосредоточимся на поиске и установке модулей из PowerShell Gallery.

Листинг 7.9. Команды PowerShellGet

```
PS> Get-Command -Module PowerShellGet

CommandType     Name                Version    Source
-----------     ----                -------    ------
Function        Find-Command        1.1.3.1    powershellget
Function        Find-DscResource    1.1.3.1    powershellget
Function        Find-Module         1.1.3.1    powershellget
Function        Find-RoleCapability 1.1.3.1    powershellget
Function        Find-Script         1.1.3.1    powershellget
Function        Get-InstalledModule 1.1.3.1    powershellget
Function        Get-InstalledScript 1.1.3.1    powershellget
Function        Get-PSRepository    1.1.3.1    powershellget
Function        Install-Module      1.1.3.1    powershellget
Function        Install-Script      1.1.3.1    powershellget
Function        New-ScriptFileInfo  1.1.3.1    powershellget
--пропуск--
```

Для поиска в PowerShell Gallery модуля с определенным именем воспользуйтесь командой `Find-Module`. Например, если вы ищете модули для управления инфраструктурой VMware, вы можете использовать подстановочные знаки с параметром `Name`, чтобы найти все модули в PowerShell Gallery, в которых есть слово «VMware», как показано в листинге 7.10.

Листинг 7.10. Использование команды Find-Module для поиска модулей, связанных с VMware

```
PS> Find-Module -Name *VMware*
Version      Name                        Repository    Description
-------      ----                        ----------    -----------
6.5.2.6...   VMware.VimAutomation.Core   PSGallery     This Windows...
1.0.0.5...   VMware.VimAutomation.Sdk    PSGallery     This Windows...
--пропуск--
```

Команда `Find-Module` ничего не загружает, а просто показывает вам, что есть в PowerShell Gallery. В следующем разделе мы посмотрим, как устанавливать модули.

Установка модулей

Если вы хотите установить модуль, воспользуйтесь командой `Install-Module`. Эта команда обычно принимает параметр `Name`, но вместо этого давайте воспользуемся конвейером и передадим объекты, которые возвращает команда `Find-Module`, непосредственно команде `Install-Module` (листинг 7.11).

Обратите внимание, что вы можете получить предупреждение о ненадежном репозитории.

Подобные предупреждения появляются, как вы могли догадаться, когда команда `Find-Module` работает с ненадежным репозиторием PowerShell. Чтобы избавиться от них, укажите PowerShell доверять всем пакетам внутри этого репозитория. В противном случае вам будет предложено запустить команду `Set-PSRepository`, как показано в листинге 7.11, чтобы вручную задать политику установки для этого репозитория.

Листинг 7.11. Установка модуля с помощью команды Install-Module

```
PS> Find-Module -Name VMware.PowerCLI | Install-Module
Untrusted repository You are installing the modules from an untrusted repository.
If you trust this repository, change its InstallationPolicy value by running the
Set-PSRepository cmdlet.
Are you sure you want to install the modules from
'https://www.powershellgallery.com/api/v2/'?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "N"): a
Installing package 'VMware.PowerCLI'
Installing dependent package 'VMware.VimAutomation.Cloud' ooooooooooooooooooooooooo
ooooooooooooooooooooooooooooooooooooooooo] Installing package 'VMware.VimAutomation.
Cloud'
Downloaded 1003175.00 MB out of 1003175.00 MB. [oooooooooooooooooooooooooooooooooooo
ooooooooooooooooooooooooooooooooooo]
```

Команда в листинге 7.11 загрузит модуль и поместит его в расположение модулей All Users в папке C:\Program Files. Чтобы проверить, ведет ли этот путь к модулю, можно использовать следующую команду:

```
PS> Get-Module -Name VMware.PowerCLI -ListAvailable | Select-Object
                                                      -Property ModuleBase

ModuleBase
----------
C:\Program Files\WindowsPowerShell\Modules\VMware.PowerCLI\6.5.3.6870460
```

Деинсталляция модулей

Новичков в PowerShell часто смущает разница между удалением и деинсталляцией модуля. Как говорилось в разделе «Импорт модулей» на с. 113, команда `Remove-Module` позволяет *удалить* модуль из сеанса PowerShell. Но она лишь выгружает модуль из сеанса, а не удаляет его с диска.

Для удаления — *деинсталляции* — модуля воспользуйтесь командлетом `Uninstall-Module`. С помощью команды из листинга 7.12 мы удаляем только что установленный модуль.

Листинг 7.12. Удаление модуля

```
PS> Uninstall-Module -Name VMware.PowerCLI
```

Команда `Uninstall-Module` позволяет удалять только модули, загруженные из PowerShell Gallery, — встроенные все равно останутся!

Создание собственного модуля

Пока что мы работали только с модулями, которые писали другие люди. Конечно же, одна из потрясающих особенностей модулей PowerShell заключается в том, что вы можете создавать свои модули и делиться ими со всем миром. В части III этой книги мы займемся практическим созданием модуля, ну а сейчас посмотрим, как превратить наш модуль `Software` в настоящий.

Как вы уже знаете, типичный PowerShell-модуль состоит из папки (*контейнер модуля*), файла .psm1 (самого модуля) и файла .psd1 (манифеста модуля). Если папка модуля находится в одном из трех возможных расположений (System, All Users или Current), PowerShell автоматически увидит это и выполнит импорт.

Сначала создадим папку для модуля. Она должна называться так же, как и сам модуль. Лично я стараюсь делать так, чтобы мои модули были доступны для всех пользователей в системе, поэтому располагаю их в All Users, например:

```
PS> mkdir 'C:\Program Files\WindowsPowerShell\Modules\Software'
```

Создайте папку с пустым файлом .psm1, который будет содержать ваши функции:

```
PS> Add-Content 'C:\Program Files\WindowsPowerShell\Modules\Software\Software.psm1'
```

Затем создайте манифест модуля, как было показано в листинге 7.8:

```
PS> New-ModuleManifest -Path 'C:\Program Files\WindowsPowerShell\Modules
\Software\Software.psd1'
-Author 'Adam Bertram' -RootModule Software.psm1
-Description 'This module helps in deploying software.'
```

На этом этапе PowerShell уже видит ваш модуль, но не экспортированные команды:

```
PS> Get-Module -Name Software -List

    Directory: C:\Program Files\WindowsPowerShell\Modules

ModuleType Version   Name                                ExportedCommands
---------- -------   ----                                ----------------
Script     1.0       Software
```

Добавим в файл .psm1 три функции, которые вы уже использовали. Посмотрим, распознает ли их PowerShell:

```
PS> Get-Module -Name Software -List

    Directory: C:\Program Files\WindowsPowerShell\Modules

ModuleType Version   Name                                ExportedCommands
---------- -------   ----                                ----------------
Script     1.0       Software                            {Get-Software...
```

PowerShell экспортировал все команды внутри вашего модуля и сделал их доступными для использования. Если вы хотите пойти дальше и выбрать команды для экспорта, то откройте манифест модуля и найдите ключ `FunctionsToExport`. В нем вы сможете определить через запятую все команды для экспорта. Делать это не обязательно, но такой метод позволяет более тонко настраивать экспорт функций модуля.

Поздравляю! Вы только что создали свой первый модуль! От него мало пользы, пока вы не добавили в него настоящую функциональность, но эту веселую задачу вам предстоит решить самостоятельно.

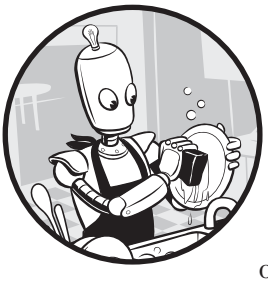
Итоги

В этой главе мы узнали о модулях — блоках кода, объединенных одной темой, которые экономят ваше время. Мы рассмотрели строение модуля, а также методику его установки, импорта, удаления и деинсталляции. Кроме того, мы создали свой собственный простой модуль!

В главе 8 вы узнаете, как работать с удаленными компьютерами с помощью инструментов удаленного взаимодействия PowerShell.

8

Удаленный запуск сценариев



Если вы единственный ИТ-специалист в небольшой организации, скорее всего, вы ответственны за несколько серверов. Когда у вас есть сценарий, который необходимо запустить, вы заходите на каждый сервер, открываете консоль PowerShell и запускаете его. Однако можно сэкономить кучу времени, если запустить всего один сценарий, который будет выполнять определенную задачу на каждом сервере. В этой главе вы узнаете, как удаленно запускать команды с помощью инструментов PowerShell.

Удаленное управление PowerShell — это инструмент, который позволяет пользователю удаленно запускать команды на одном или нескольких компьютерах одновременно. *Сеанс*, или, если точнее, сеанс `PSSession`, — это взаимодействие со средой PowerShell, запущенной на удаленном компьютере, с которого вы можете выполнять команды. Среди утилит Microsoft Sysinternals есть инструмент `psexec`, который работает по-другому, но у него та же идея: вы пишете код, который работает на вашем компьютере, а затем отправляете и выполняете его на удаленном.

Большую часть этой главы мы уделим понятию сессий: что они собой представляют, как их использовать и что делать, когда вы закончили с ними работать. Но сначала нам нужно понять кое-что о блоках сценариев.

ПРИМЕЧАНИЕ

Сотрудники корпорации Microsoft внедрили удаленное управление в версии PowerShell v2. Оно было создано на основе службы Windows Remote Management (WinRM). По этой причине удаленное управление PowerShell иногда ассоциируют с термином WinRM.

Работа с блоками сценариев

В удаленном управлении PowerShell широко используются блоки сценариев, которые, как и функции, представляют собой код, упакованный в единый исполняемый блок. Они отличаются от функций парой важных деталей: у них нет имени и их можно назначать переменным.

Чтобы увидеть эту разницу, давайте рассмотрим пример. Определим функцию `New-Thing`, которая вызывает команду `Write-Host` и выводит какой-нибудь текст в консоли (см. листинг 8.1).

Листинг 8.1. Определение функции New-Thing, которая выводит текст в окне консоли

```
function New-Thing {
    param()
    Write-Host "Hi! I am in New-Thing"
}

New-Thing
```

После запуска этого сценария вы увидите, что он выводит в консоль текст «Hi! I am in New-Thing!». Но обратите внимание, что для этого вам нужно было вызвать `New-Thing`, чтобы запустить функцию.

Вы можете воспроизвести результат вызова функции `New-Thing` с помощью блока сценария, присвоив для начала блок сценария переменной, как показано в листинге 8.2.

Листинг 8.2. Создание блока сценария и присвоение его переменной с именем $newThing

```
PS> $newThing = { Write-Host "Hi! I am in a scriptblock!" }
```

Для создания блока сценария поместите код в фигурные скобки. Мы сохранили наш блок сценария в переменной `$newThing`. Вам может показаться, что для выполнения этого блока сценария можно просто вызвать переменную, как показано в листинге 8.3.

Листинг 8.3. Создание и выполнение блока сценария

```
PS> $newThing = { Write-Host «Hi! I am in a scriptblock!» }
PS> $newThing
Write-Host «Hi! I am in a scriptblock!»
```

Но, как видите, PowerShell считывает содержимое переменной `$newThing` буквально, не понимая, что `Write-Host` — это команда, которую он должен выполнить. Вместо этого в консоли выводится содержимое блока сценария.

Чтобы PowerShell запустил код внутри блока, вам нужно использовать символ `&`, за которым следует имя переменной. В листинге 8.4 показан этот синтаксис.

Листинг 8.4. Выполнение блока сценария

```
PS> & $newThing
Hi! I am in a scriptblock!
```

Этот символ сообщает PowerShell, что штука в фигурных скобках — это код, который надо выполнить. Это один из способов выполнения блока кода. Однако ему не хватает настроек, доступных для команд, которые вам понадобятся при работе с PowerShell на удаленных компьютерах. В следующем разделе мы рассмотрим другой способ выполнения блоков сценария.

Использование команды Invoke-Command для выполнения кода на удаленных системах

Во время удаленного управления PowerShell вы будете использовать две основные команды: `Invoke-Command` и `New-PSSession`. В этом разделе вы узнаете о первой из них, а в следующем я расскажу о второй.

Команду `Invoke-Command` вы, вероятно, будете использовать чаще всего при удаленном управлении PowerShell. Существует два основных способа ее использования. Первый — для запуска так называемых *ad hoc-команд* — небольших одноразовых выражений, которые вы хотите выполнить. Второй способ — для работы в интерактивных сеансах. В этой главе мы рассмотрим оба варианта.

Примером ad hoc-команды является выполнение `Start-Service` для запуска службы на удаленном компьютере. Когда вы выполняете специальную команду с помощью `Invoke-Command`, PowerShell создает сеанс и разрывает его, как только команда будет выполнена. Это ограничивает разнообразие действий с `Invoke-Command`, поэтому в следующем разделе я расскажу о том, как можно создать собственные сеансы.

Пока что остановимся на том, как Invoke-Command работает с ad hoc-командами.

Откройте консоль PowerShell, введите Invoke-Command и нажмите клавишу Enter (листинг 8.5).

Листинг 8.5. Запуск Invoke-Command без параметров

```
PS> Invoke-Command

cmdlet Invoke-Command at command pipeline position 1
Supply values for the following parameters:
ScriptBlock:
```

Консоль сразу же попросит вас предоставить блок сценария. Мы напишем команду hostname, которая вернет имя того хоста, на котором выполняется команда.

Чтобы передать блок сценария с командой hostname в Invoke-Command, используйте обязательный параметр ComputerName, который сообщает Invoke-Command, на каком удаленном компьютере нужно запустить эту команду, как видно из листинга 8.6. (Обратите внимание: для того чтобы это сработало, мой компьютер и удаленный компьютер WEBSRV1 должны быть частью одного домена Active Directory (AD), при этом у моего компьютера должны быть права администратора на WEBSRV1.)

Листинг 8.6. Запуск простого примера Invoke-Command

```
PS> Invoke-Command -ScriptBlock { hostname } -ComputerName WEBSRV1
WEBSRV1
```

ПРИМЕЧАНИЕ

Если вы проделаете этот трюк на удаленном компьютере с операционной системой старше Windows Server 2012 R2, все может сработать не так, как ожидалось. В таком случае сначала включите удаленное управление PowerShell. Начиная с версии Server 2012 R2, удаленное взаимодействие PowerShell включено по умолчанию, а служба WinRM работает со всеми необходимыми портами брандмауэра и настроенными правами доступа. Но если вы используете более раннюю версию Windows, то включать управление придется вручную, поэтому сначала запустите команду Enable-PSRemoting на своем удаленном компьютере в сеансе консоли с повышенными привилегиями, и лишь потом запускайте Invoke-Command на более старом сервере. Вы также можете использовать команду Test-WSMan, чтобы убедиться, что удаленное взаимодействие PowerShell настроено и доступно.

Обратите внимание, что теперь вывод команды hostname — это имя удаленного компьютера: в моей системе удаленный компьютер называется WEBSRV1. Вы выполнили свою первую удаленную команду!

Запуск локальных сценариев на удаленных компьютерах

В предыдущем разделе мы попробовали выполнять блоки сценариев на удаленных компьютерах. Команду Invoke-Command также можно использовать для выполнения целых сценариев. Вместо параметра Scriptblock вы можете прописать параметр FilePath и указать путь к сценарию на вашем компьютере. При использовании параметра FilePath команда Invoke-Command считает содержимое сценария локально, а затем выполнит его код на удаленном компьютере. Сам сценарий на удаленном компьютере при этом не выполняется.

Для демонстрации предположим, что на вашем локальном компьютере есть сценарий GetHostName.ps1, расположенный в корневом каталоге C:\. В нем записана всего одна строка — hostname. Нужно запустить этот сценарий на удаленном компьютере, чтобы вернуть его имя хоста. Заметьте, что хотя наш сценарий предельно прост, команде Invoke-Command все равно, что там внутри. Она с радостью выполнит все, что найдет в сценарии.

Чтобы запустить сценарий, надо передать его файл в параметр FilePath команды Invoke-Command, как показано в листинге 8.7.

Листинг 8.7. Запуск локального сценария на удаленных компьютерах

```
PS> Invoke-Command -ComputerName WEBSRV1 -FilePath C:\GetHostName.ps1
WEBSRV1
```

Команда Invoke-Command запускает код из сценария GetHostName.ps1 на компьютере WEBSRV1 и возвращает результат в ваш локальный сеанс.

Удаленное использование локальных переменных

В удаленном управлении PowerShell предусмотрено много полезного, однако следует проявлять осторожность при использовании локальных переменных. Допустим, у вас есть путь к файлу на удаленном компьютере — C:\File.txt. Поскольку в какой-то момент этот путь может измениться, вы помещаете его в переменную, например в $serverFilePath:

```
PS> $serverFilePath = 'C:\File.txt'
```

Теперь вам может потребоваться указать путь C:\File.txt внутри блока сценария на удаленном хосте. В листинге 8.8 видно, что произойдет, если вы попытаетесь сослаться на переменную напрямую.

Листинг 8.8. Локальные переменные в удаленных сеансах не работают

```
PS> Invoke-Command -ComputerName WEBSRV1 -ScriptBlock { Write-Host "The value of
foo is $serverFilePath" }
The value of foo is
```

Обратите внимание, что у переменной `$serverFilePath` нет значения, потому что при выполнении блока сценария на удаленном компьютере этой переменной вообще не существует! Когда вы определяете переменную в сценарии или в консоли, эта переменная сохраняется в определенной *области выполнения*. Она является своего рода контейнером, который PowerShell использует для хранения информации во время сеанса. Ранее вы могли сталкиваться с областями выполнения, если пытались открыть две консоли PowerShell одновременно и использовать (безуспешно) переменные из первой консоли в другой.

Переменные, функции и другие конструкции по умолчанию не распространяются на несколько пространств выполнения. Однако существует несколько методов, позволяющих это сделать. Есть два основных способа передачи переменных на удаленный компьютер.

Передача переменных с помощью параметра ArgumentList

Чтобы получить значение переменной в удаленном блоке сценария, воспользуйтесь параметром `ArgumentList` команды `Invoke-Command`. Этот параметр позволяет передавать массив локальных значений `$args` в блок сценария, который в дальнейшем можно включать в код блока. Чтобы показать, как это работает, в листинге 8.9 мы передадим переменную `$serverFilePath`, которая содержит путь к файлу C:\File.txt, в удаленный блок сценария, а затем создадим ссылку на нее через массив `$args`.

Листинг 8.9. Использование массива $args для передачи локальных переменных в удаленный сеанс

```
PS> Invoke-Command -ComputerName WEBSRV1 -ScriptBlock { Write-Host "The value
of foo is $($args[0])" } -ArgumentList $serverFilePath
The value of foo is C:\File.txt
```

Как вы видите, значение переменной C:\File.txt теперь находится внутри блока сценария. Это происходит потому, что мы передали `$serverFilePath` в `ArgumentList` и заменили ссылку `$serverFilePath` внутри блока сценария на

`$args[0]`. Если вы хотите передать в блок сценария более одной переменной, добавьте другое значение к значению параметра `ArgumentList` и увеличьте ссылку `$args` на единицу там, где хотите сослаться на новую переменную.

Использование оператора $Using для передачи значений переменных

Другой способ передачи значений локальных переменных в удаленный блок сценария — использование оператора `$using`. Вам не понадобится использовать `ArgumentList`, если вы добавите оператор `$using` к любому имени локальной переменной. Прежде чем PowerShell отправит блок сценария на удаленный компьютер, он найдет оператор `$using` и раскроет все локальные переменные внутри этого блока.

Листинг 8.10 — это тот же листинг 8.9, но в котором теперь используется оператор `$using:serverFilePath` вместо `ArgumentList`.

Листинг 8.10. Использование оператора $using для ссылки на локальные переменные в удаленном сеансе

```
PS> Invoke-Command -ComputerName WEBSRV1 -ScriptBlock { Write-Host "The value
of foo is $using:serverFilePath" }
The value of foo is C:\File.txt
```

Нетрудно заметить, что результаты листингов 8.9 и 8.10 одинаковы.

Оператор `$using` требует меньше возни, при этом он интуитивно более понятен, но в будущем, когда вы начнете использовать Pester для тестирования сценариев, вам, возможно, придется вернуться к использованию параметра `ArgumentList`: при использовании оператора `$using` Pester не сможет оценить значение в переменной `$using`. При использовании параметра `ArgumentList` переменные, передаваемые в удаленный сеанс, определяются локально, и Pester способен их понять и интерпретировать. Сейчас все это звучит не очень понятно, но вы во всем разберетесь после прочтения главы 9. Ну а пока оператор `$using` работает просто прекрасно!

Теперь, когда у вас есть общее представление о командлете `Invoke-Command`, давайте проведем еще парочку сеансов.

Работа с сеансами

Как мы говорили ранее, в удаленном управлении PowerShell существует понятие *сеанса*. Когда вы создаете его удаленно, PowerShell открывает *локальный сеанс* на удаленном компьютере, который можно использовать для выполнения

команд. Углубляться в технические подробности устройства сеанса нам не требуется. Достаточно знать, что вы можете создавать сеанс, подключаться и отключаться от него, и он будет сохранять последнее состояние, в котором вы его оставили. Сеанс не завершится, пока не будет удален.

В предыдущем разделе, когда вы запускали командлет `Invoke-Command`, он начинал новый сеанс, выполнял код и сразу же завершал сеанс. В этом разделе вы увидите, как создавать то, что я называю *полными сеансами* — в них вы можете напрямую вводить команды. Использование командлета `Invoke-Command` хорошо подходит для выполнения команд один раз, но не слишком эффективно, когда их много и они даже не помещаются в один блок сценария. Например, если вы работаете над большим сценарием для локальной работы и он должен получить информацию из другого источника, использовать ее в сеансе удаленного взаимодействия, а затем получить информацию из этого сеанса для локального использования и, наконец, снова вернуться на локальный компьютер, то вам нужно будет создать сценарий, который многократно запускает командлет `Invoke-Command`. Что еще хуже, у вас возникнут дополнительные проблемы, если вам нужно будет задать переменную в удаленном сеансе и использовать ее позже. Командлет `Invoke-Command`, который вы использовали до сих пор, уже не поможет — вам понадобится сеанс, который сохранится после вашего ухода.

Создание нового сеанса

Чтобы создать полупостоянный сеанс на удаленном компьютере через удаленное управление PowerShell, необходимо создать полный сеанс с помощью команды `New-PSSession` — она создаст сеанс на удаленном компьютере и будет ссылаться на него на локальном.

Чтобы создать новый сеанс `PSSession`, используйте команду `New-PSSession` с параметром `ComputerName`, как показано в листинге 8.11. В этом примере мой компьютер находится в том же домене Active Directory, что и `WEBSRV1`, и я вошел в систему как пользователь домена с правами администратора на `WEBSRV1`. Чтобы подключиться с помощью параметра `ComputerName` (как показано в листинге 8.11), пользователь должен быть локальным администратором или, по крайней мере, входить в группу «Пользователи удаленного управления» на удаленном компьютере. Если вы не находитесь в домене Active Directory, то можете использовать параметр `Credential` команды `New-PSSession` для передачи объекта `PSCredential`. В этом объекте содержатся учетные данные для аутентификации на удаленном компьютере.

Листинг 8.11. Создание нового сеанса PSSession

```
PS> New-PSSession -ComputerName WEBSRV1

 Id Name    ComputerName  ComputerType   State   ConfigurationName     Availability
 -- ----    ------------  ------------   -----   -----------------     ------------
  3 WinRM3  WEBSRV1       RemoteMachine  Opened  Microsoft.PowerShell  Available
```

Как вы видите, команда `New-PSSession` возвращает сеанс. Когда сеанс будет задан, вы сможете входить и выходить из сеанса с помощью команды `Invoke-Command`, но вместо параметра `ComputerName`, как вы это делали с ad hoc-командой, вам придется использовать параметр `Session`.

В параметр `Session` нужно передать объект сеанса. Вы можете использовать команду `Get-PSSession`, чтобы вывести на экран все ваши текущие сеансы. В листинге 8.12 результат `Get-PSSession` будет сохранен в переменной.

Листинг 8.12. Поиск сеансов, созданных на локальном компьютере

```
PS> $session = Get-PSSession
PS> $session

 Id Name    ComputerName  ComputerType   State   ConfigurationName     Availability
 -- ----    ------------  ------------   -----   -----------------     ------------
  6 WinRM6  WEBSRV1       RemoteMachine  Opened  Microsoft.PowerShell  Available
```

Поскольку вы запускали команду `New-PSSession` всего один раз, листинг 8.12 также создаст всего один сеанс. Если у вас несколько сеансов, вы можете выбрать нужный для команды `Invoke-Command` с помощью параметра `Id` команды `Get-PSSession`.

Вызов команд в сеансе

Теперь, когда ваш сеанс находится в переменной, вы можете передать ее в `Invoke-Command` и запустить какой-нибудь код внутри сеанса, как показано в листинге 8.13.

Листинг 8.13. Использование существующего сеанса для вызова команд на удаленном компьютере

```
PS> Invoke-Command -Session $session -ScriptBlock { hostname }
WEBSRV1
```

Вы наверное заметили, что эта команда выполняется намного быстрее, чем раньше, когда мы передавали ей другую команду. Все дело в том, что теперь `Invoke-Command` не приходится создавать и закрывать новый сеанс. Когда вы создаете полный сеанс, он не только работает быстрее, но и получает доступ

к большему количеству функций. Например, как показано в листинге 8.14, вы можете задать переменные в удаленном сеансе, а затем вернуться в сеанс без потери этих переменных.

Листинг 8.14. Значения переменных сохраняются при последующих подключениях сеанса

```
PS> Invoke-Command -Session $session -ScriptBlock { $foo = 'Please be here
                                                    next time' }
PS> Invoke-Command -Session $session -ScriptBlock { $foo }
Please be here next time
```

Пока сеанс открыт, в удаленном сеансе можно делать все что угодно, и его состояние при этом не изменится. Но это действительно только для вашего текущего локального сеанса. Если вы запустите еще один процесс PowerShell, вы не сможете просто продолжить с того места, где остановились. Удаленный сеанс по-прежнему будет активен, но ссылка на этот удаленный сеанс на локальном компьютере исчезнет. В этом случае сеанс `PSSession` перейдет в отключенное состояние (мы рассмотрим это в следующем разделе).

Открытие интерактивных сеансов

В листинге 8.14 `Invoke-Command` используется для отправки команд на удаленный компьютер, он же и получает ответ. Выполнение таких удаленных команд похоже на выполнение неконтролируемого сценария. Этот вариант не такой интерактивный, как использование клавиш в консоли PowerShell. Если вы хотите открыть интерактивную консоль для сеанса, запущенного на удаленном компьютере, например для устранения неполадок, вы можете использовать команду `Enter-PSSession`.

Команда `Enter-PSSession` позволяет пользователю работать с сеансом в интерактивном режиме. Команда может либо создать собственный сеанс, либо использовать существующий, созданный с помощью `New-PSSession`. Если вы не укажете сеанс, команда `Enter-PSSession` создаст новый и будет ждать дальнейшего ввода, как показано в листинге 8.15.

Листинг 8.15. Начало интерактивного сеанса

```
PS> Enter-PSSession -ComputerName WEBSRV1
[WEBSRV1]: PS C:\Users\Adam\Documents>
```

Обратите внимание, что текст приглашения PowerShell изменился на `[WEBSRV1]: PS`. Теперь команды будут выполняться в удаленном сеансе. На этом этапе вы можете запустить любую команду, как если бы вы находились в консоли

удаленного компьютера. Такая интерактивная работа с сеансами — отличный способ отказаться от использования Remote Desktop Protocol (RDP) для вызова интерактивного интерфейса для выполнения таких задач, как устранение неполадок на удаленном компьютере.

Отключение от сеансов и повторное подключение к ним

Если вы закроете консоль PowerShell, снова откроете ее и попытаетесь использовать команду `Invoke-Command` в последнем рабочем сеансе, то получите сообщение об ошибке, показанное в листинге 8.16.

Листинг 8.16. Попытка выполнить команду в отключенном сеансе

```
PS> $session = Get-PSSession -ComputerName websrv1
PS> Invoke-Command -Session $session -ScriptBlock { $foo }
Invoke-Command : Because the session state for session WinRM6, a617c702-ed92
-4de6-8800-40bbd4e1b20c, websrv1 is not equal to Open, you cannot run a
command in the session. The session state is Disconnected.
At line:1 char:1
+ Invoke-Command -Session $session -ScriptBlock { $foo }
--пропуск--
```

Если PowerShell находит сеанс `PSSession` на удаленном компьютере, но при этом не может найти ссылку на локальном, это означает, что сеанс отключен. Вот что происходит, если вы неправильно отключаете ссылку на локальный сеанс в удаленном `PSSession`.

Вы можете отключить существующие сеансы с помощью команды `Disconnect-PSSession`. Можно очистить любые созданные ранее сеансы, если предварительно получить их с помощью команды `Get-PSSession`, а затем направить в команду `Disconnect-PSSession` (листинг 8.17). Как вариант, вы можете использовать параметр в `Disconnect-PSSession` для отключения одного сеанса за раз.

Листинг 8.17. Отключение PSSession

```
PS> Get-PSSession | Disconnect-PSSession

 Id Name    ComputerName  ComputerType   State        ConfigurationName     Availability
 -- ----    ------------  ------------   -----        -----------------     ------------
  4 WinRM4  WEBSRV1       RemoteMachine  Disconnected Microsoft.PowerShell  None
```

Чтобы правильно отключиться от сеанса, нужно передать имя удаленного компьютера в параметр `Session`. Это можно сделать либо вызовом его через *объект сессии* `Disconnect-PSSession -Session`, либо подтягиванием существующего сеанса в команду через `Get-PSSession`, как показано в листинге 8.17.

Если позже вы захотите снова подключиться к сеансу, то уже после отключения с помощью `Disconnect-PSSession` закройте консоль PowerShell и воспользуйтесь командой `Connect-PSSession`, как показано в листинге 8.18. Обратите внимание, что в этом случае вы можете подключаться только к отключенным сеансам, которые уже были созданы с вашей учетной записи. При этом вы не увидите сеансы, созданные другими пользователями.

Листинг 8.18. Повторное подключение к сеансу PSSession

```
PS> Connect-PSSession -ComputerName websrv1
[WEBSRV1]: PS>
```

Теперь вы можете запускать код на удаленном компьютере, словно вы и не закрывали консоль.

Если вы по-прежнему получаете сообщение об ошибке, возможно, все дело в несовпадающих версиях PowerShell. Отключенные сеансы работают, только если локальный компьютер и удаленный сервер работают на одинаковых версиях. Например, если на локальном компьютере установлена PowerShell 5.1, а на удаленном стоит версия, которая не поддерживает отключенные сеансы (например, PowerShell v2 или более ранняя), то ничего не выйдет. Нужно убедиться, что на локальном компьютере и на удаленном сервере установлена одна и та же версия PowerShell.

Чтобы это проверить, посмотрите значение переменной `$PSVersionTable`, которая содержит информацию о версиях (листинг 8.19).

Листинг 8.19. Проверка версии PowerShell на локальном компьютере

```
PS> $PSVersionTable

Name                           Value
----                           -----
PSVersion                      5.1.15063.674
PSEdition                      Desktop
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.15063.674
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
```

Чтобы проверить версию на удаленном компьютере, запустите на нем команду `Invoke-Command`, передав ей переменную `$PSVersionTable`, как показано в листинге 8.20.

Листинг 8.20. Проверка версии PowerShell на удаленном компьютере

```
PS> Invoke-Command -ComputerName WEBSRV1 -ScriptBlock { $PSVersionTable }

Name                           Value
----                           -----
PSRemotingProtocolVersion      2.2
BuildVersion                   6.3.9600.16394
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0}
PSVersion                      4.0
CLRVersion                     4.0.30319.34014
WSManStackVersion              3.0
SerializationVersion           1.1.0.1
```

Перед отключением от сеанса стоит проверить соответствие ваших версий, чтобы не потерять выполненную работу в удаленной системе.

Удаление сеансов с помощью команды Remove-PSSession

Каждый раз, когда команда `New-PSSession` создает новый сеанс, он существует как на удаленном сервере, так и на локальном компьютере. Вы также можете одновременно открыть несколько сеансов на разных серверах. При этом если часть из этих сеансов больше не используется, вы, вероятно, захотите их удалить. Это можно сделать с помощью команды `Remove-PSSession`, которая прерывает сеанс на стороннем компьютере и удаляет ссылку на локальный сеанс `PSSession`, если он есть. Пример показан в листинге 8.21.

Листинг 8.21. Удаление сеанса PSSession

```
PS> Get-PSSession | Remove-PSSession
PS> Get-PSSession
```

В этом примере мы снова запустили `Get-PSSession`, но ничего не получили. Это означает, что на вашем локальном компьютере нет активных сеансов.

Общие сведения об авторизации при удаленном управлении PowerShell

До сих пор мы обходили стороной вопрос аутентификации. По умолчанию, если ваш локальный и удаленный компьютеры находятся в одном домене и на обоих включено удаленное управление PowerShell, явная проверка подлинности не требуется. В любом другом случае вам нужно будет как-то выполнить аутентификацию.

Два самых распространенных способа аутентификации на удаленном компьютере через удаленное управление PowerShell — это использование Kerberos или CredSSP. Если вы находитесь в домене Active Directory, вы, вероятно, уже используете систему тикетов Kerberos, даже если вы сами об этом не знаете. В Active Directory и некоторых системах Linux используются *области* Kerberos — объекты, которые выдают тикеты клиентам. Затем эти тикеты представляются ресурсам и сравниваются с нужными (в Active Directory) на контроллерах домена.

Инструмент CredSSP не нуждается в Active Directory. CredSSP появился еще в Windows Vista и использует клиентскую службу учетных данных (CSP), позволяя приложениям делегировать учетные данные пользователя удаленным компьютерам. Для аутентификации двух систем CredSSP не требует никакой другой внешней системы вроде контроллеров домена.

В среде Active Directory удаленное управление PowerShell работает через протокол сетевой проверки подлинности Kerberos для выполнения вызовов в Active Directory, где выполняется вся внутренняя проверка подлинности. PowerShell использует учетную запись, в которую вы вошли локально как пользователь, и с ее помощью авторизуется на удаленном компьютере, как, в общем, и многие другие службы. В этом и состоит прелесть использования единых подписей.

Однако если вы не находитесь в среде Active Directory, такой тип аутентификации не подходит. Например, в случае если вам нужно подключиться к удаленному компьютеру через интернет или по локальной сети, и при этом вы используете учетные данные удаленного компьютера. PowerShell поддерживает множество методов удаленной аутентификации, но наиболее распространенным, помимо использования Kerberos, является инструмент CredSSP, который позволяет локальному компьютеру делегировать учетные данные пользователя удаленному компьютеру. Эта схема похожа на принцип работы Kerberos, но не требует использования Active Directory.

Обычно при работе в среде Active Directory вам не требуются другие типы аутентификации, но обратное тоже иногда случается, так что лучше быть готовым ко всему. В этом разделе мы рассмотрим общую задачу аутентификации и методы ее решения.

Проблема двойного перехода

Проблема *двойного перехода* появилась в тот момент, когда в Microsoft добавили функцию удаленного управления PowerShell. Она возникает при запуске кода

внутри удаленного сеанса, если вы затем пытаетесь получить доступ к его удаленным ресурсам. Например, если у вас в сети есть контроллер домена с именем DC и вы хотите извлечь файлы из папки C:\ с помощью административного общего ресурса C$, вы сможете без проблем просматривать общий ресурс удаленно с локального компьютера (листинг 8.22).

Листинг 8.22. Перечисление файлов через общий ресурс UNC

```
PS> Get-ChildItem -Path '\\dc\c$'


Directory: \\dc\c$


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        10/1/2019 12:05 PM                FileShare
d-----       11/24/2019 2:28 PM                inetpub
d-----       11/22/2019 6:37 PM                InstallWindowsFeature
d-----        4/16/2019 1:10 PM                Iperf
```

Проблемы возникают, когда вы создаете сеанс PSSession и пытаетесь повторно выполнить ту же команду, как видно из листинга 8.23.

Листинг 8.23. Попытка доступа к сетевым ресурсам в сеансе

```
PS> Enter-PSSession -ComputerName WEBSRV1
[WEBSRV1]: PS> Get-ChildItem -Path '\\dc\c$'
ls : Access is denied
--пропуск--
[WEBSRV1]: PS>
```

В этом случае PowerShell отказывает вам в доступе, даже если он есть у вашей учетной записи. Так происходит, потому что Kerberos использует свою обычную аутентификацию по умолчанию, а PowerShell, в свою очередь, не передает удаленно эти учетные данные другому сетевому ресурсу. Другими словами, двойной переход не работает. По соображениям безопасности PowerShell отказывается делегировать эти учетные данные, в результате чего возвращает сообщение «Access Denied».

Двойной прыжок с использованием CredSSP

В этом разделе вы узнаете способ обойти проблему двойного перехода. Именно *обойти*, а не *исправить*. В Microsoft предупреждали, что использование CredSSP влечет за собой проблемы с безопасностью, поскольку учетные данные, переданные на первый компьютер, автоматически используются для всех

подключений с него. Это значит, что если с исходным компьютером что-то случится, то эти учетные данные можно будет использовать для подключения к другим компьютерам в сети. Тем не менее вместо использования обходных путей вроде ограниченного делегирования Kerberos по признаку ресурсов, многие пользователи предпочитают подход CredSSP из-за простоты его использования.

Прежде чем внедрять CredSSP, его нужно включить как на клиенте, так и на сервере, используя команду `Enable-WsManCredSSP` в сеансе PowerShell с повышенными привилегиями. У команды есть параметр `Role`, который определяет, на чьей стороне будет включен CredSSP. Сначала включите CredSSP на стороне клиента, как показано в листинге 8.24.

Листинг 8.24. Включение поддержки CredSSP на клиентском компьютере

```
PS> Enable-WSManCredSSP ❶-Role ❷Client ❸-DelegateComputer WEBSRV1

CredSSP Authentication Configuration for WS-Management
CredSSP authentication allows the user credentials on this computer to be sent
to a remote computer. If you use CredSSP authentication for a connection to
a malicious or compromised computer, that machine will have access to your
username and password. For more information, see the Enable-WSManCredSSP Help
topic.
Do you want to enable CredSSP authentication?
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"): y


cfg         : http://schemas.microsoft.com/wbem/wsman/1/config/client/auth
lang        : en-US
Basic       : true
Digest      : true
Kerberos    : true
Negotiate   : true
Certificate : true
CredSSP     : true
```

Чтобы включить CredSSP у клиента, мы задаем значение `Client` ❶ для параметра `Role` ❷. Здесь также используется обязательный параметр `DelegateComputer` ❸, потому что PowerShell необходимо знать, каким компьютерам разрешено использовать делегируемые учетные данные. Вы можете передать в параметр `DelegateComputer` звездочку (*), чтобы разрешить делегирование всем компьютерам, но в целях безопасности лучше дать разрешение только компьютерам, с которыми вы работаете (в данном случае `WEBSRV1`).

После включения CredSSP на клиенте вам необходимо сделать то же самое на сервере (листинг 8.25). К счастью, для этого достаточно открыть новый удаленный сеанс без использования CredSSP, а затем включить CredSSP в рамках сеанса, вместо того чтобы использовать Microsoft Remote Desktop для доступа или вообще подходить к серверу.

Листинг 8.25. Включение поддержки CredSSP на сервере

```
PS> Invoke-Command -ComputerName WEBSRV1 -ScriptBlock {
                    Enable-WSManCredSSP -Role Server }

CredSSP Authentication Configuration for WS-Management CredSSP authentication
allows the server to accept user credentials from a remote computer. If you enable
CredSSP authentication on the server, the server will have access to the username
and password of the client computer if the client computer sends them. For more
information, see the Enable-WSManCredSSP Help topic.
Do you want to enable CredSSP authentication?
[Y] Yes [N] No [?] Help (default is "Y"): y

#text
-----
False
True
True
False
True
Relaxed
```

Таким образом, вы включили CredSSP на клиенте и на сервере. Клиент позволяет делегировать учетные данные пользователя удаленному серверу, а на удаленном сервере CredSSP включается сам. Теперь вы можете снова попытаться получить доступ к удаленным сетевым ресурсам из того удаленного сеанса (листинг 8.26). Обратите внимание: если вам когда-нибудь понадобится отменить включение CredSSP, используйте команду `Disable-WsmanCredSSP` для сброса изменений.

Листинг 8.26. Доступ к сетевым ресурсам через сеанс с аутентификацией CredSSP

```
PS> Invoke-Command -ComputerName WEBSRV1 -ScriptBlock { Get-ChildItem -Path
'\\dc\c$' }
❶-Authentication Credssp ❷-Credential (Get-Credential)

cmdlet Get-Credential at command pipeline position 1
Supply values for the following parameters:
Credential
```

```
    Directory: \\dc\c$


Mode           LastWriteTime    Length Name                      PSComputerName
----           -------------    ------ ----                      --------------
d-----    10/1/2019 12:05 PM           FileShare                 WEBSRV1
d-----   11/24/2019  2:28 PM           inetpub                   WEBSRV1
d-----   11/22/2019  6:37 PM           InstallWindowsFeature     WEBSRV1
d-----    4/16/2019  1:10 PM           Iperf                     WEBSRV1
```

Обратите внимание, что вы должны явно сообщить команде `Invoke-Command` (или `Enter-PSSession`) о намерении использовать аутентификацию CredSSP ❶, и обеим командам нужны учетные данные. Их вы получите с помощью команды `Get-Credential`, а не Kerberos ❷.

После выполнения `Invoke-Command` предоставьте `Get-Credential` имя пользователя и пароль для доступа к общему ресурсу к диску `c$` на DC — и теперь команда `Get-ChildItem` сработает правильно!

Итоги

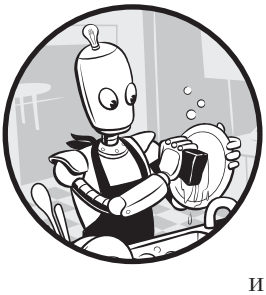
Удаленное управление PowerShell — это, безусловно, простейший инструмент для выполнения кода на удаленных системах. Как вы узнали из этой главы, функция удаленного управления PowerShell интуитивно понятна и проста в использовании. Как только вы поймете концепцию блока сценариев и кода внутри него, работа с удаленными блоками сценариев перестанет быть для вас проблемой.

В части III этой книги вы создадите свой собственный PowerShell-модуль. Для этого вы будете использовать удаленное взаимодействие PowerShell почти в каждой команде. Если у вас возникли проблемы при изучении этой главы, перечитайте ее еще раз либо же самостоятельно поэкспериментируйте. Пробуйте разные сценарии, ломайте, исправляйте, делайте все возможное, чтобы понять принцип работы удаленного управления PowerShell. Это один из самых важных навыков, который вы почерпнете из этой книги.

В главе 9 мы рассмотрим еще один важный навык — тестирование с использованием Pester.

9

Тестирование с помощью Pester



Это неизбежно! Рано или поздно вам придется тестировать свой код. Легко предположить, что ваш код идеален, но еще легче — доказать обратное. Тестируя ваши сценарии с помощью Pester, вы перестанете гадать и поймете, как все обстоит на самом деле.

Тестирование является неотъемлемой частью процесса традиционной разработки программного обеспечения уже на протяжении десятилетий. Однако такие понятия, как модульное, функциональное, интеграционное и приемочное тестирование, известные опытным разработчикам ПО, могут быть незнакомыми для сценаристов — тех из нас, кто занимается автоматизацией с помощью PowerShell, но не является разработчиком. Поскольку во многих организациях код PowerShell используется для запуска критически важных производственных систем, мы возьмем страницу из мира программирования и применим ее к PowerShell.

В этой главе вы узнаете, как создавать тесты для ваших сценариев и модулей, позволяющие с уверенностью сказать, что ваш код работает даже после внесения изменений. Это можно сделать с помощью тестовой среды, известной как Pester.

Знакомьтесь: Pester

Pester — это PowerShell-модуль для тестирования с открытым исходным кодом, доступный в PowerShell Gallery. Поскольку он эффективен и написан

на PowerShell, он де-факто стал стандартом для тестирования в этой среде. С его помощью вы можете создать несколько типов тестов, включая модульные, интеграционные и приемочные. Не беспокойтесь, если эти названия не говорят вам ни о чем. В этой книге мы будем использовать Pester только для проверки изменений окружающей среды — например, была ли создана виртуальная машина с правильным именем, установлен ли IIS или установлена ли соответствующая операционная система. Мы будем называть это *тестами инфраструктуры*.

Мы не будем рассказывать, как проверять вещи вроде наличия вызова функции, правильности установки переменной или возврата сценарием определенного типа объекта — это все относится к миру *модульного тестирования*. Если вам интересно узнать о модульном тестировании с помощью Pester и о том, как его использовать в различных ситуациях, ознакомьтесь с книгой «The Pester Book» (LeanPub, 2019, leanpub.com/pesterbook/): там вы найдете практически все, что нужно знать о тестировании в PowerShell.

Основы Pester

Чтобы использовать модуль Pester, его сперва нужно установить. Если ваш компьютер работает на Windows 10, Pester будет установлен по умолчанию, а пользователи с более ранней версией ОС могут скачать его из PowerShell Gallery. Даже если у вас установлена Windows 10, возможно, ваша версия Pester устарела, так что лучше все же скачать актуальную. Поскольку Pester есть в PowerShell Gallery, вы можете запустить команду `Install-Module -Name Pester` для его загрузки и установки. В установленном Pester будут все необходимые вам команды.

Стоит еще раз напомнить, что вы будете использовать Pester для написания и запуска тестов инфраструктуры для проверки любых ожидаемых изменений, которые сценарий выполняет в своей среде. Например, вы можете запустить тест инфраструктуры после создания нового пути к файлу с помощью команды `Test-Path` и проверить существование этого пути. Инфраструктурные тесты — это меры предосторожности, позволяющие подтвердить, что ваш код работает именно так, как задумано.

Файл Pester

По сути, сценарий тестирования Pester состоит из сценария PowerShell, имя которого заканчивается на .Tests.ps1. При этом вы можете назвать основной

сценарий как угодно: имена и структура теста — это уже ваше дело. Для примера назовем сценарий Sample.Tests.ps1.

В структуру сценария тестирования Pester входит один или несколько блоков `describe`, каждый из которых содержит (необязательные) блоки `context`, каждый из которых, в свою очередь, содержит блоки `it`, в каждом из которых есть утверждения. Звучит сложно, поэтому в листинге 9.1 я покажу вам пример.

Листинг 9.1. Базовая структура теста Pester

```
C:\Sample.Tests.ps1
    describe
        context
          it
            assertions
```

Давайте рассмотрим каждую из этих частей.

Блок describe

Блок `describe` — это место для группировки схожих тестов. В листинге 9.2 создан блок `describe` с названием IIS. С помощью него можно будет задействовать весь код для тестирования функций Windows, пулов приложений и веб-сайтов.

Базовый синтаксис блока `describe` включает само слово `describe`, за которым следует его имя в одинарных кавычках, а затем фигурные скобки.

Листинг 9.2. Блок describe в Pester

```
describe 'IIS' {

}
```

Эта структура похожа на привычное условие `if/then`, но все не так просто! Это блок сценария, который скрытно передается в функцию `describe`. Обратите внимание: если вы из тех, кто любит писать фигурные скобки в новой строке, то вам не повезло — открывающая фигурная скобка должна находиться в той же строке, что и ключевое слово `describe`.

Блок context

После создания блока `describe` вы можете добавить дополнительный блок `context`. Он позволяет группировать похожие блоки, что помогает организовать тесты при инфраструктурном тестировании. В листинге 9.3 мы добавим блок `context`, который будет содержать все тесты для функций Windows.

Расположение тестов таким образом в блоках `context` — хорошая идея, ведь так ими проще управлять.

Листинг 9.3. Блок context в Pester

```
describe 'IIS' {
    context 'Windows features' {
    }
}
```

Хотя блок `context` необязателен, позже он станет незаменимым, когда вы будете тестировать десятки или сотни компонентов!

Блок it

Теперь давайте добавим блок `it` внутрь блока `context`. Блок `it` — это самый маленький компонент, но в нем находится сам тест. Его синтаксис, показанный в листинге 9.4, выглядит как имя, за которым следует блок. Мы видели похожее в блоке `describe`.

Листинг 9.4. Блок describe с блоками context и it

```
describe 'IIS' {
    context 'Windows features' {
        it 'installs the Web-Server Windows feature' {

        }
    }
}
```

Обратите внимание, что до сих пор мы только лишь добавляли разные метки для теста. В следующем разделе мы добавим сам тест.

Утверждения

Внутри блока `it` находится одно или несколько утверждений. *Утверждение* — это, в общем-то, сам тест, а именно код, который сравнивает ожидаемое состояние с фактическим. Наиболее распространенное утверждение в Pester — утверждение `should`. У него есть различные операторы для взаимодействия, например `be`, `bein`, `belessthan` и т. д. Полный список доступных операторов приведен на вики-странице Pester (github.com/pester/Pester/wiki/).

В нашем примере с IIS проверим, был ли на сервере создан пул приложений с именем `test`. Для этого вам сначала необходимо узнать текущее состояние Windows-функции `Web-server` на сервере (мы назовем его `WEBSRV1`).

Покопавшись в доступных командах PowerShell с помощью `Get-Command` и ознакомившись со справкой команды `Get-WindowsFeature`, вы обнаружите, что код для этой цели выглядит как-то так:

```
PS> (Get-WindowsFeature -ComputerName WEBSRV1 -Name Web-Server).Installed
True
```

Вы знаете, что если функция `Web-server` установлена, свойство `Installed` вернет значение `True`; в противном случае оно вернет `False`. Можно предположить, что при запуске `Get-WindowsFeature` свойство `Installed` будет иметь значение `True`. Нужно проверить, будет ли так в действительности. Представьте этот сценарий внутри блока `it`, как показано в листинге 9.5.

Листинг 9.5. Проверка условия теста с помощью Pester

```
describe 'IIS' {
    context 'Windows features' {
        it 'installs the Web-Server Windows feature' {
            $parameters = @{
                ComputerName = 'WEBSRV1'
                Name         = 'Web-Server'
            }
            (Get-WindowsFeature @parameters).Installed | should be $true
        }
    }
}
```

Здесь мы создали рудиментарный тест Pester для проверки статуса установки функции Windows. Сначала вы вводите тест, а затем передаете результаты этого теста через конвейер в условие тестирования, которое в нашем случае задано как `should be $true`.

Написание тестов Pester — это достаточно обширная тема, так что я советую вам почитать книгу «The Pester Book» (leanpub.com/pesterbook/) или цикл статей на 4sysops (4sysops.com/archives/powershell-pester-testing-getting-started/). Информации оттуда должно быть достаточно для понимания тестов, которые я включил в эту книгу. Когда вы прочтете ее, написание собственных тестов Pester позволит вам проверить свои навыки работы с PowerShell.

Теперь у вас есть сценарий Pester. А раз так, то нужно его запустить!

Выполнение теста Pester

Самый распространенный способ выполнять тесты Pester — использовать команду `Invoke-Pester`. Эта команда является частью Pester-модуля и позволяет

тестировщику передать ему путь к сценарию тестирования, который затем будет интерпретирован и выполнен Pester, как показано в листинге 9.6.

Листинг 9.6. Запуск теста Pester

```
PS> Invoke-Pester -Path C:\Sample.Tests.ps1
Executing all tests in 'C:\Sample.Tests.ps1'

Executing script C:\Sample.Tests.ps1

  Describing IIS
    [+] installs the Web-Server Windows feature 2.85s
Tests completed in 2.85s
Tests Passed: 1, Failed: 0, Skipped: 0, Pending: 0, Inconclusive: 0
```

Команда `Invoke-Pester` выполнила сценарий Sample.Tests.ps1 и предоставила основную информацию: имя блока `describe`, результат теста, а также сводку всех тестов, выполненных в процессе. Стоит обратить внимание, что команда `Invoke-Pester` всегда отображает сводку состояния каждого выполненного теста. В нашем случае установка функции `Windows Web-server` прошла успешно, на что указывает символ «+» и зеленый цвет текста.

Итоги

В этой главе мы рассмотрели основы тестовой среды Pester. Мы загрузили и установили модуль Pester, а также создали простой тест. Это должно помочь вам понять устройство и принцип работы теста Pester. В следующих главах мы снова и снова будем использовать этот фреймворк. Мы добавим множество блоков `describe`, `it` и различных утверждений, но наша основная структура останется относительно неизменной.

На этом завершается наша последняя глава части I. Вы ознакомились с основным синтаксисом и понятиями, которые в дальнейшем будете использовать при написании сценариев с помощью PowerShell. А теперь давайте перейдем к части II, где мы получим практический опыт и начнем решать реальные задачи!

ЧАСТЬ II

АВТОМАТИЗАЦИЯ ПОВСЕДНЕВНЫХ ЗАДАЧ

Если первая часть вам показалась больше похожей на школьные упражнения, чем на нечто практическое, не волнуйтесь — я с вами согласен. Окунаться в реку сразу с головой нельзя, нужно сперва прощупать глубину. Именно это мы сделали в части I — познакомили с PowerShell новичков, а знатокам напомнили, что к чему.

Во второй части мы, наконец, перейдем к интересным вещам, а именно *применим* навыки из части I в реальных задачах. Вы узнаете, как использовать PowerShell для автоматизации некоторых типовых задач, с которыми многие технические специалисты сталкиваются каждый день. Если вы опытный технический специалист, вы, несомненно, уже были прежде знакомы с некоторыми из этих сценариев, когда сонно листали Active Directory, тратили уйму времени на копирование и вставку в листах Excel и нервничали в попытках настроить ПО с дистанционным управлением для одновременного подключения целой дюжины компьютеров, чтобы отчитаться перед менеджерами.

В этой части книги вы познакомитесь с инструментами для автоматизации подобных задач. Конечно, мы не можем охватить все — существует слишком уж много скучных вещей, которые нужно автоматизировать! Далее я привожу несколько распространенных сценариев, с которыми я сталкивался на протяжении двадцати лет работы в отрасли. Если мы не рассмотрели вашу задачу — не волнуйтесь! К концу этой книги у вас будет достаточно знаний, чтобы автоматизировать ее самостоятельно.

Часть II разбита на четыре темы, которые мы рассмотрим в пяти главах.

Работа со структурированными данными

Данные повсюду. Если вы работали с ним раньше, то знаете, что у них миллионы разных форматов: базы данных SQL, файлы XML, объекты JSON, CSV и т. д. У всякого типа данных есть своя структура, к каждой из которых нужен свой подход. В главе 10 вы узнаете, как читать, писать и изменять различные формы данных.

Автоматизация задач Active Directory

Active Directory (AD) — это *служба каталогов*. На высоком уровне ее можно представить как иерархическое представление того, к каким ИТ-ресурсам пользователь может получить доступ. AD — это версия службы каталогов Microsoft, и, как вы можете представить, она используется тысячами организаций по всему миру, что делает ее весьма подходящей для автоматизации.

Глава 11 посвящена основам управления различными объектами AD из консоли PowerShell. Когда вы привыкнете к командлетам AD, мы научимся использовать их для автоматизации повседневных задач на нескольких небольших примерах.

Управление облаком

Как и почти все современные технологии, PowerShell в значительной степени поддерживает работу в облаке. Понимание того, как PowerShell работает в облачных средах, включая Microsoft Azure и Amazon Web Services (AWS), откроет для вас новые блестящие возможности автоматизации. В главах 12 и 13 вы создадите виртуальные машины, веб-службы и многое другое. Вы даже увидите пример взаимодействия PowerShell с двумя облачными провайдерами. Поскольку PowerShell безразлично, какое облако мы используем (он работает независимо), у нас есть возможность управлять любым из них!

Создание сценария инвентаризации сервера

Как вы уже могли заметить, сложность материала в книге постепенно возрастает, поэтому вам понадобится прочный фундамент для прыжка в водоворот технической магии из части III. Именно об этом и рассказывает глава 14, в которой мы объединим все уже накопленные знания в едином проекте. Здесь вы узнаете, как объединить разрозненные источники информации

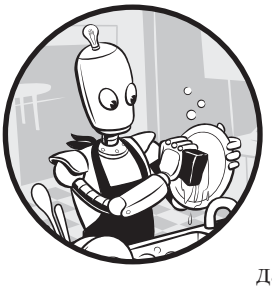
в единый отчет. Мы выполним запросы для компьютеров из AD и опросим их с помощью CIM/WMI для доступа к полезной информации, такой как имя, оперативная память, скорость процессора, операционная система, IP-адрес и многое другое.

Итоги

К концу этой части вы получили представление о тех видах повседневных задач, которые можно автоматизировать с помощью PowerShell. Автоматизировав пару из них самостоятельно, вы поймете, что нет никакой необходимости тратить деньги на дорогостоящее программное обеспечение или на модных консультантов для управления вашей средой. PowerShell может дополнять сотни продуктов — где есть желание (и PowerShell), там есть возможность.

10

Парсинг структурированных данных



PowerShell может считывать, обновлять и удалять данные из множества источников. Это происходит благодаря встроенной поддержке любого объекта .NET и практически каждого метода оболочки. Если ваши данные хранятся в какой-то структурированной форме, то работать с ними будет еще проще.

В этой главе мы сосредоточимся на нескольких распространенных формах структурированных данных, включая CSV, таблицы Microsoft Excel и JSON. Вы узнаете, как управлять каждым типом данных, используя собственные командлеты PowerShell и объекты .NET. К концу главы вы сможете профессионально использовать PowerShell для управления всеми видами структурированных данных.

CSV-файлы

Один из самых простых и распространенных способов хранения данных — использовать для этих целей CSV-файл. *CSV-файл* — это простой текстовый файл, в котором содержится таблица. Ее элементы разделены определенным общим символом — *разделителем* (чаще всего это запятые). У каждого CSV-файла основная структура одинакова: первая строка в CSV — это строка заголовка,

содержащая все заголовки столбцов таблицы. В следующих строках находится содержимое таблицы.

В этом разделе вы в основном будете работать с парой командлетов CSV: `Import-Csv` и `Export-Csv`.

Чтение CSV-файлов

Из всех задач обработки CSV, которые может выполнять PowerShell, почти наверняка наиболее распространенной является чтение. Учитывая то, насколько проста и эффективна структура CSV-файлов, неудивительно, что они используются компаниями и приложениями во всем технологическом мире — отсюда и популярность команды PowerShell `Import-Csv`.

Но что значит *чтение* CSV-файла? Хотя в нем есть вся необходимая информация, вы не можете импортировать ее прямо в программу. Обычно приходится читать файл и преобразовывать его в подходящие данные. Этот процесс называется *парсингом*. Команда `Import-Csv` парсит CSV-файл — считывает его, а затем преобразует данные в объекты PowerShell. Мы скоро обсудим способы использования `Import-Csv`, но сначала нужно понять принцип действия этой команды.

Начнем с простой таблицы, содержащей данные сотрудников вымышленной компании (рис. 10.1).

	A	B	C	D
1	First Name	Last Name	Department	Manager
2	Adam	Bertram	IT	Miranda Bertram
3	Barack	Obama	Executive Office	Michelle Obama
4	Miranda	Bertram	Executive Office	
5	Michelle	Obama	Executive Office	

Рис. 10.1. CSV-файл сотрудников

На рисунке — скриншот Excel, но нетрудно понять, как выглядят данные в виде CSV-файла с простым текстом. В нашем примере вы будете работать с файлом Employees.csv, который можно найти в ресурсах к этой главе. Посмотрите на листинг 10.1.

Здесь мы используем команду `Get-Content` для запроса нашего текстового файла (CSV). Ее можно использовать для чтения любых текстовых файлов.

Листинг 10.1. Чтение CSV-файла с помощью Get-Content

```
PS> Get-Content -Path ./Employees.csv –Raw

First Name,Last Name,Department,Manager
Adam,Bertram,IT,Miranda Bertram
Barack,Obama,Executive Office,Michelle Obama
Miranda,Bertram,Executive Office
Michelle,Obama,Executive Office
```

Мы видим, что это типичный CSV-файл со строкой заголовка и несколькими строками данных, разделенными на столбцы с помощью запятых. Обратите внимание, что прочитать файл можно с помощью командлета `Get-Content`. Так как CSV — это текстовый файл, команда `Get-Content` идеально подходит для его чтения (вообще-то это первый этап, происходящий с `Import-Csv`).

Но также учтите, что команда `Get-Content` возвращает информацию в виде простой строки, что происходит при использовании параметра `Raw`. В противном случае команда `Get-Content` возвращает массив строк, каждый элемент которого представляет строку в файле CSV:

```
PS> Get-Content ./Employees.csv -Raw | Get-Member

    TypeName: System.String
    --пропуск--
```

Хотя команда `Get-Content` может считывать данные, она *не понимает* само устройство CSV-файла. Она понятия не имеет о том, что в таблице есть строки заголовка или данных, и не знает, что делать с разделителями. Команда просто принимает и выдает содержимое. Вот почему у нас есть команда `Import-Csv`.

Использование команды Import-Csv для обработки данных

Чтобы увидеть, как работает команда `Import-Csv`, сравните вывод листинга 10.1 с выводом `Import-Csv` в листинге 10.2.

Первым делом бросается в глаза, что заголовки отделены линией от основных данных. Это означает, что команда `Import-Csv` считывает файл, обрабатывает верхнюю строку как строку заголовка и уже знает, как отделить ее от остального файла. Также заметно отсутствие запятых — когда команда читает и *распознает* CSV-файл, она знает, что разделитель используется для отделения элементов в таблице и не является частью ее данных.

Но что случится, если в коде появится случайный разделитель? Попробуйте поставить запятую в середине строки *Adam* в файле Employees.csv и запустите

код. Что произошло? Теперь все данные в строке *Adam* сдвинуты: *am* теперь значится как фамилия, *Bertram* — как отдел, а *IT* — как новый менеджер. Команда `Import-Csv` достаточно умна для понимания формата CSV, но она не может распознавать его содержимое — вот тут-то вы и вступаете в игру.

Листинг 10.2. Использование Import-Csv

```
PS> Import-Csv -Path ./Employees.csv

First Name Last Name Department       Manager
---------- --------- ----------       -------
Adam       Bertram   IT               Miranda Bertram
Barack     Obama     Executive Office Michelle Obama
Miranda    Bertram   Executive Office
Michelle   Obama     Executive Office

PS> Import-Csv -Path ./Employees.csv | Get-Member

   TypeName: System.Management.Automation.PSCustomObject

PS> $firstCsvRow = Import-Csv -Path ./Employees.csv | Select-Object –First
1
PS> $firstCsvRow | Select-Object -ExpandProperty 'First Name'
Adam
```

Превращение сырых данных в объекты

Команда `Import-Csv` не просто считывает CSV-файл и выводит его с необычным форматированием. Содержимое файла помещается в массив `PSCustomObjects`. Здесь каждый объект `PSCustomObject` содержит данные одной строки. У каждого объекта есть свойства, соответствующие заголовкам строки. Если же вам нужны данные из столбца этого заголовка, достаточно лишь обратиться к этому свойству. Зная всего лишь какой вид данных следует ожидать, команда `Import-Csv` может взять прежде неизвестную строку и превратить ее в простые и удобные объекты. Круто!

Данные в виде массива `PSCustomObjects` позволяют вам использовать их гораздо эффективнее. Допустим, вы хотите найти только сотрудников с фамилией *Bertram*. Поскольку каждая строка данных в CSV является объектом `PSCustomObject`, вы можете сделать это с помощью команды `Where-Object`:

```
PS> Import-Csv -Path ./Employees.csv | Where-Object { $_.'Last Name' -eq 'Bertram' }

First Name Last Name Department       Manager
---------- --------- ----------       -------
Adam       Bertram   IT               Miranda Bertram
Miranda    Bertram   Executive Office
```

Если вместо этого нужно вывести только те строки, в которых есть отдел Executive Office, то сделать это тоже проще простого! Мы используем тот же метод, заменив фамилию на отдел, а значение *Bertram* на *Executive Office*:

```
PS> Import-Csv -Path ./Employees.csv | Where-Object {$_.Department -eq
                                       'Executive Office' }

First Name Last Name Department       Manager
---------- --------- ----------       -------
Barack     Obama     Executive Office Michelle Obama
Miranda    Bertram   Executive Office
Michelle   Obama     Executive Office
```

А что произойдет, если в качестве разделителя использовать точку с запятой? Попробуйте изменить CSV-файл и посмотрите. Нехорошо, правда? Вам не обязательно использовать именно запятую в качестве разделителя, но `Import-Csv` по умолчанию принимает только их. Если вы хотите использовать другой разделитель, нужно указать его в команде `Import-Csv`.

Давайте это проверим: замените все запятые в нашем файле Employees.csv на знаки табуляции:

```
PS> (Get-Content ./Employees.csv -Raw).replace(',',"`t") | Set-Content
                ./Employees.csv
PS> Get-Content ./Employees.csv –Raw
First Name Last Name    Department Manager
Adam   Bertram IT Miranda Bertram
Barack Obama   Executive Office    Michelle Obama
Miranda Bertram Executive Office
Michelle    Obama   Executive Office
```

Если у вас есть файл с табуляцией вместо запятых, можно указать символ табуляции (в виде `` `t ``) в качестве нового разделителя с помощью параметра `Delimiter` (листинг 10.3).

Листинг 10.3. Использование параметра Delimiter в команде Import-Csv

```
PS> Import-Csv -Path ./Employees.csv -Delimiter "`t"

First Name Last Name Department       Manager
---------- --------- ----------       -------
Adam       Bertram   IT               Miranda Bertram
Barack     Obama     Executive Office Michelle Obama
Miranda    Bertram   Executive Office
Michelle   Obama     Executive Office
```

Обратите внимание, что мы получили такой же результат, как и в листинге 10.2.

Определяем собственный заголовок

Что делать, если у вас есть таблица данных, но вы хотите сделать строку заголовка более удобной для пользователя? Команда `Import-Csv` справится и с этим. Как и в случае с новым разделителем, ей достаточно просто передать параметр. В листинге 10.4 параметр `Header` используется для передачи серии строк, разделенных запятыми (новых заголовков).

Листинг 10.4. Использование параметра Header команды Import-Csv

```
PS> Import-Csv -Path ./Employees.csv -Delimiter "`t"
-Header 'Employee FName','Employee LName','Dept','Manager'

Employee FName Employee  LName Dept               Manager
-------------- --------------- ----               -------
First Name     Last Name       Department         Manager
Adam           Bertram         IT                 Miranda Bertram
Barack         Obama           Executive Office   Michelle Obama
Miranda        Bertram         Executive Office
Michelle       Obama           Executive Office
```

Как видите, после запуска команды у каждого объекта в строке данных будет новая метка в качестве имен свойств.

Создание CSV-файлов

С чтением существующих CSV-файлов мы разобрались. А как насчет создания собственных? Вы можете написать его от руки, но это потребует немалых затрат времени и энергии, особенно если речь идет о тысячах строк. К счастью, у PowerShell есть собственный командлет для создания CSV-файлов — `Export-Csv`. Его можно использовать для создания CSV-файлов из любого существующего объекта PowerShell. Вам достаточно указать, какие объекты использовать в качестве строк и где должен быть создан файл.

Давайте сначала рассмотрим второе требование. Допустим, вы выполнили несколько команд PowerShell и хотите сохранить вывод консоли в файле. Можно использовать команду `Out-File`, но в этом случае текст будет неструктурированным. А вам нужен красивый структурированный файл со строками заголовков и разделителями. Введите команду `Export-Csv`.

Давайте предположим, что вы хотите запросить все запущенные на вашем компьютере процессы и записать их название, издателя и описание. Можно использовать команду `Get-Process` и `Select-Object`, чтобы ограничить список свойств:

```
PS> Get-Process | Select-Object -Property Name,Company,Description

Name                 Company                       Description
----                 -------                       -----------
ApplicationFrameHost Microsoft Corporation         Application Frame Host
coherence            Parallels International GmbH  Parallels Coherence service
coherence            Parallels International GmbH  Parallels Coherence service
coherence            Parallels International GmbH  Parallels Coherence service
com.docker.proxy
com.docker.service   Docker Inc.
Docker.Service
--пропуск--
```

В листинге 10.5 показано, что происходит при сохранении вывода с помощью `Export-Csv`.

Листинг 10.5. Использование Export-Csv

```
PS> Get-Process | Select-Object -Property Name,Company,Description |
Export-Csv -Path C:\Processes.csv –NoTypeInformation
PS> Get-Content -Path C:\Processes.csv
"Name","Company","Description"
"ApplicationFrameHost","Microsoft Corporation","Application Frame Host"
"coherence","Parallels International GmbH","Parallels Coherence service"
"coherence","Parallels International GmbH","Parallels Coherence service"
"coherence","Parallels International GmbH","Parallels Coherence service"
"com.docker.proxy",,
"com.docker.service","Docker Inc.","Docker.Service"
```

Когда вы передаете вывод напрямую в команду `Export-Csv`, указываете путь к CSV (с помощью параметра `Path`) и используете параметр `NoTypeInformation`, то получаете CSV-файл с ожидаемой строкой заголовка и строками данных.

> **ПРИМЕЧАНИЕ**
>
> Параметр `NoTypeInformation` не является обязательным, но, если вы его не используете, в верхней строке вашего CSV-файла будет указан тип объекта, из которого он был получен. Если вы повторно импортируете CSV-файл обратно в PowerShell, это будет вам мешать. Строка выглядит примерно как `#TYPE Selected.System.Diagnostics.Process`.

Проект 1. Создание отчета об инвентаризации компьютеров

Для закрепления полученных знаний давайте займемся созданием мини-проекта. Пусть он будет основан на какой-нибудь распространенной ситуации.

Представьте на мгновение, что ваша компания приобрела другую и вы не знаете, какие серверы и ПК установлены в ее сети. Все, что у вас есть — это CSV-файл с IP-адресами и названием отдела, в котором находится каждое устройство. Вас попросили выяснить, что это за устройства, и предоставить руководству новый CSV-файл с результатами.

Что делать? Вообще это двухэтапный процесс: нужно прочитать имеющийся CSV-файл и затем создать собственный. В ваш CSV-файл нужно будет внести каждый IP-адрес, который вы обрабатываете, относящийся к нему отдел, статус ответа IP-адреса на пингование, а также имя DNS устройства.

Мы начнем с CSV-файла, который выглядит как пример ниже. IP-адреса определяются по маске 255.255.255.0, поэтому они идут вплоть до 192.168.0.254:

```
PS> Get-Content -Path ./IPAddresses.csv
"192.168.0.1","IT"
"192.168.0.2","Accounting"
"192.168.0.3","HR"
"192.168.0.4","IT"
"192.168.0.5","Accounting"
--пропуск--
```

Я создал сценарий Discover-Computer.ps1 (его можно найти в материалах к этой главе). По мере работы над проектом мы будем добавлять туда код.

Сначала вам нужно считать каждую строку в CSV-файле. Это делается с помощью команды `Import-Csv`, которая будет записывать каждую строку из нашего файла в переменную для дальнейшей обработки:

```
$rows = Import-Csv -Path C:\IPAddresses.csv
```

Теперь, когда у вас есть данные, нужно их использовать. С каждым IP-адресом мы будем выполнять два действия: пинговать его и определять имя хоста. Давайте попробуем выполнить эти действия на одной строке нашего CSV-файла, чтобы убедиться, что мы все пишем правильно.

В следующем листинге мы будем использовать команду `Test-Connection`, которая отправляет один ICMP-пакет на указанный вами IP-адрес (в данном случае это IP-адрес из первой строки нашего файла). Параметр `Quiet` сообщает команде, что нужно вернуть лишь значение `True` или `False`.

```
PS> Test-Connection -ComputerName $row[0].IPAddress -Count 1 –Quiet
PS> (Resolve-DnsName -Name $row[0].IPAddress -ErrorAction Stop).Name
```

Во второй строке мы запрашиваем имя хоста на том же IP-адресе с помощью команды `Resolve-DnsName`, которая возвращает несколько свойств. Поскольку

нас интересует только имя, мы заключаем всю команду в круглые скобки и используем запись через точку для выдачи свойства `Name`.

Убедившись, что с синтаксисом все в порядке, мы повторим этот код для каждой строки в CSV-файле. Это проще всего сделать с помощью цикла `foreach`:

```
foreach ($row in $rows) {
    Test-Connection -ComputerName $row.IPAddress -Count 1 –Quiet
    (Resolve-DnsName -Name $row.IPAddress -ErrorAction Stop).Name
}
```

Запустите код. Что при этом происходит? Вы получили кучу строк `True/False` с именами хостов, но теперь непонятно, с каким IP-адресом связана каждая из них. Вам нужно будет создать *хеш-таблицы* для каждой строки и задать собственные элементы. Также необходимо учесть возможное наличие и причины ошибок команды `Test-Connection` или `Resolve-DnsName`. В листинге 10.6 показан пример того, как это сделать.

Листинг 10.6. Мини-проект — обнаружение CSV-файла

```
$rows = Import-Csv -Path C:\IPAddresses.csv
foreach ($row in $rows) {
    try { ❶
        $output = @{ ❷
            IPAddress  = $row.IPAddress
            Department = $row.Department
            IsOnline   = $false
            HostName   = $null
            Error      = $null
        }
        if (Test-Connection -ComputerName $row.IPAddress -Count 1 -Quiet) { ❸
            $output.IsOnline = $true
        }
        if ($hostname = (Resolve-DnsName -Name $row.IPAddress
                                         -ErrorAction Stop).Name) { ❹
            $output.HostName = $hostName
        }
    } catch {
        $output.Error = $_.Exception.Message ❺
    } finally {
        [pscustomobject]$output ❻
    }
}
```

Давайте разберемся со всем этим. Сначала мы создаем хеш-таблицу, значения в которой будут соответствовать столбцам строки (по желанию вы также можете включить дополнительную информацию) ❷. Затем мы проверяем подключение к сети, пропинговав IP-адрес ❸. Если компьютер подключен, задаем

для переменной `IsOnline` значение `True`. Сделаем то же самое с `HostName` — проверим, можно ли его найти ❹, и обновим значение хеш-таблицы при положительном результате. Если возникают какие-либо ошибки, мы записываем их в значение `Error` хеш-таблицы ❺. Наконец, мы превращаем нашу хеш-таблицу в `PSCustomObject` и возвращаем ее независимо от наличия ошибок ❻. Обратите внимание, что мы запаковываем всю эту функцию в блок `try/catch` ❶, который будет выполнять код в блоке `catch`, если код в блоке `try` выдаст ошибку. Мы используем параметр `ErrorAction`, поэтому команда `Resolve-DnsName` выдаст исключение (ошибку), если произойдет что-то непредвиденное.

Запустив код, вы увидите такой результат:

```
HostName   :
Error      : 1.0.168.192.in-addr.arpa : DNS name does not exist
IsOnline   : True
IPAddress  : 192.168.0.1
Department : HR

HostName   :
Error      : 2.0.168.192.in-addr.arpa : DNS name does not exist
IsOnline   : True
IPAddress  : 192.168.0.2
Department : Accounting
--пропуск--
```

Поздравляю — мы выполнили большую часть работы! Теперь мы знаем, с какими данными выдачи связан каждый IP-адрес. Осталось только записать вывод в Csv-файл. Как мы уже поняли, это можно сделать с помощью команды `Export-Csv`. Мы просто перенаправим созданный `PSCustomObject` в `Export-Csv`, и результат будет выводиться непосредственно в CSV-файл, а не в консоль.

Обратите внимание, что в дальнейшем мы будем использовать параметр `Append`: по умолчанию команда `Export-CSV` перезаписывает CSV-файл, а с параметром `Append` строка просто добавляется в конец файла вместо его перезаписи.

```
PS> [pscustomobject]$output |
Export-Csv -Path C:\DeviceDiscovery.csv -Append
-NoTypeInformation
```

После запуска сценария вы увидите, что CSV-файл будет точно таким же, как и результат, который вы получили в консоли PowerShell:

```
PS> Import-Csv -Path C:\DeviceDiscovery.csv

HostName   :
```

```
Error      : 1.0.168.192.in-addr.arpa : DNS name does not exist
IsOnline   : True
IPAddress  : 192.168.0.1
Department : HR

HostName   :
Error      :
IsOnline   : True
IPAddress  : 192.168.0.2
Department : Accounting
```

Теперь у вас есть файл DeviceDiscovery.csv (или как там вы его назвали), в котором есть строки для каждого IP-адреса из исходного CSV-файла, все значения из него, а также те значения, которые вы обнаружили с помощью команд `Test-Connection` и `Resolve-DnsName`.

Таблицы Excel

Трудно представить себе компанию, в которой бы не использовались электронные таблицы Excel. Скорее всего, если вы получите проект сценария, в нем уже будет такая таблица. Но, прежде чем мы займемся Excel, давайте обозначим нашу позицию: по возможности лучше им не пользоваться!

В CSV-файле можно хранить данные столь же эффективно, как и в простой электронной таблице Excel, но с первым в PowerShell работать намного проще. Формат таблиц Excel закрытый, поэтому их нельзя читать с помощью PowerShell при отсутствии соответствующей внешней библиотеки. Если в вашей книге Excel только один лист, упростите себе работу и сохраните ее в формате CSV. Конечно, это не всегда возможно, но, если ситуация позволяет, лучше поступить именно так. Доверьтесь мне.

Но что делать, когда сохранить файл в формате CSV невозможно? В этом случае можно воспользоваться уже готовым модулем. Раньше считывание файлов .xls или .xlsx в PowerShell требовало особого подхода к разработке. Вам нужно было устанавливать Excel и получать доступ к *COM-объектам* — сложным программным компонентам, которые сводят на нет все удовольствие от работы в PowerShell. К счастью, другие люди уже проделали всю эту тяжелую работу за вас, поэтому мы просто возьмем крутой модуль `ImportExcel` Дуга Финке (Doug Finke). Для работы с этим свободно распространяемым модулем не требуется установка Excel, а в использовании он намного проще, чем COM-объекты.

Сперва вам нужно установить модуль. `ImportExcel` доступен в PowerShell Gallery и устанавливается с помощью `Install-Module ImportExcel`. Давайте посмотрим, на что он способен.

Создание таблиц Excel

Для начала вам необходимо создать электронную таблицу Excel. Разумеется, это можно было бы сделать и обычным способом, открыв Excel и все такое прочее, но разве это весело? Давайте создадим простую электронную таблицу с одним листом с помощью PowerShell (ведь сперва нужно научиться ползать, а потом уже ходить). Для этого мы воспользуемся командой `Export-Excel`. Как и команда `Export-Csv`, `Export-Excel` считывает имена свойств всех полученных объектов, а затем создает из них строку заголовка и строки данных.

Самый простой способ использовать `Export-Excel` — передать команде один или несколько объектов, как если бы вы использовали `Export-Csv`. Давайте создадим книгу Excel с одним листом, в которой будут содержаться все запущенные процессы на компьютере.

Ввод `Get-Process | Export-Excel .\Processes.xlsx` дает нам таблицу, показанную на рис. 10.2.

	A	B	C	D	E	F	G	H
1	Name	SI	Handles	VM	WS	PM	NPM	Path
2	ApplicationFrameHost	1	315	2.19919E+12	26300416	7204864	17672	C:\WINDOWS\system32\ApplicationFrameHost.exe
3	coherence	0	120	62164992	5607424	1769472	7968	C:\Program Files (x86)\Parallels\Parallels Tools\Services\coherence.exe
4	coherence	1	113	82739200	5255168	1818624	7736	C:\Program Files (x86)\Parallels\Parallels Tools\Services\coherence.exe
5	coherence	1	130	78884864	5672960	1802240	8896	C:\Program Files (x86)\Parallels\Parallels Tools\Services\WOW\coherence.exe
6	com.docker.localhost-forwarder	1	343	35518222336	9560064	31272960	7592	C:\Program Files\Docker\Docker\Resources\com.docker.localhost-forwarder.exe
7	com.docker.proxy	1	74	35512229888	9891840	19812352	6632	C:\Program Files\Docker\Docker\Resources\com.docker.proxy.exe
8	com.docker.service	0	517	650498048	40833024	27496448	43120	C:\Program Files\Docker\Docker\com.docker.service
9	conhost	0	105	2.19908E+12	5926912	1486848	7056	C:\WINDOWS\system32\conhost.exe
10	conhost	0	105	2.19909E+12	5943296	1544192	7192	C:\WINDOWS\system32\conhost.exe

Рис. 10.2. Электронная таблица Excel

Раз уж вы сразу не преобразовали таблицу в CSV, то, вероятно, вы работаете с чем-то посложнее единственного листа. Давайте добавим в нашу книгу еще пару листов. Для этого воспользуйтесь параметром `WorksheetName`, как показано в листинге 10.7. Это позволит создать дополнительные листы с объектами, отправленными в `Export-Excel`.

Листинг 10.7. Добавление листов в книгу Excel

```
PS> Get-Process | Export-Excel .\Processes.xlsx -WorksheetName 'Worksheet2'
PS> Get-Process | Export-Excel .\Processes.xlsx -WorksheetName 'Worksheet3'
```

Тема создания таблицы с помощью команды `Export-Excel` вообще *гораздо* более сложная, но мы здесь не будем вдаваться в подробности, чтобы сохранить время (и пару деревьев на Земле). Если вам интересна эта тема, просмотрите справочную документацию по `Export-Excel` — там вы найдете кучу доступных параметров.

Чтение таблиц Excel

Теперь, когда у вас есть рабочая таблица, давайте прочитаем данные в ее строках. Для этого понадобится `Import-Excel`. Эта команда считывает лист в книге и возвращает один или несколько объектов `PSCustomObject` — по одному для каждой строки. Самый простой способ использовать команду — это указать путь к книге с помощью параметра `Path`.

В листинге 10.8 вы увидите, что команда `Import-Excel` возвращает объект, в котором имена столбцов являются атрибутами.

Листинг 10.8. Использование команды Import-Excel

```
PS> Import-Excel -Path .\Processes.xlsx

Name                       : ApplicationFrameHost
SI                         : 1
Handles                    : 315
VM                         : 2199189057536
WS                         : 26300416
PM                         : 7204864
NPM                        : 17672
Path                       : C:\WINDOWS\system32\ApplicationFrameHost.exe
Company                    : Microsoft Corporation
CPU                        : 0.140625
--пропуск--
```

По умолчанию команда `Import-Excel` возвращает только первый лист. В нашем примере у книги несколько листов, поэтому нам нужно каким-то образом просмотреть каждый. А теперь представьте, что в последний раз вы открывали эту таблицу достаточно давно и уже не помните имена листов. Ничего страшного. Мы будем использовать команду `Get-ExcelSheetInfo`, чтобы считать все листы в книге, как показано в листинге 10.9.

Листинг 10.9. Использование команды Get-ExcelSheetInfo

```
PS> Get-ExcelSheetInfo -Path .\Processes.xlsx

Name       Index Hidden  Path
----       ----- ------  ----
```

```
Sheet1          1 Visible C:\Users\adam\Processes.xlsx
Worksheet2      2 Visible C:\Users\adam\Processes.xlsx
Worksheet3      3 Visible C:\Users\adam\Processes.xlsx
```

Мы будем использовать этот вывод для извлечения данных из всех наших листов. Воспользуемся циклом `foreach` и вызовем команду `Import-Excel` для каждого рабочего листа внутри книги, как показано в листинге 10.10.

Листинг 10.10. Извлечение строк со всех листов

```
$excelSheets = Get-ExcelSheetInfo -Path .\Processes.xlsx
Foreach ($sheet in $excelSheets) {
    $workSheetName = $sheet.Name
    $sheetRows = Import-Excel -Path .\Processes.xlsx -WorkSheetName
    $workSheetName
  ❶ $sheetRows | Select-Object -Property *,@{'Name'='Worksheet';
                                              'Expression'={ $workSheetName }
}
```

Обратите внимание, что мы используем вычисляемое свойство с помощью команды `Select-Object` ❶. Обычно в параметре `Property` команды `Select-Object` используется простая строка, указывающая на нужное вам свойство. Однако при использовании вычисляемого свойства вам нужно передать команде `Select-Object` хеш-таблицу, содержащую имя возвращаемого свойства и выражение, которое выполняется, когда `Select-Object` получает ввод. Новое вычисленное свойство будет результатом этого выражения.

По умолчанию команда `Import-Excel` не добавляет имя листа в атрибуты объекта. Это означает, что вы не будете знать, какому листу принадлежит та или иная строка. Учитывая это, вам необходимо создать атрибут `Worksheet` для каждого объекта строки, чтобы позже вам было на что ссылаться.

Добавление данных в таблицы Excel

В предыдущем разделе мы создали книгу с нуля. Рано или поздно вам понадобится добавить в рабочий лист строки данных. К счастью, при использовании модуля `ImportExcel` эта задача решается довольно просто: вам достаточно использовать параметр `Append` в команде `Export-Excel`.

В качестве примера предположим, что вам нужно отследить историю выполнения процессов на своем компьютере. Вы хотите экспортировать все процессы, запущенные на вашем компьютере в течение некоторого времени, а затем сравнить результаты с помощью Excel. Для этого после экспорта нужно

добавить к каждой строке временную метку, чтобы видеть, когда была собрана информация о процессе.

Давайте добавим в нашу книгу еще один лист и назовем его «ProcessesOverTime». Мы будем использовать вычисляемое свойство, чтобы добавить временные метки в каждую строку процесса. Выглядеть это будет примерно так:

```
PS> Get-Process |
Select-Object -Property *,@{Name = 'Timestamp';Expression = { Get-Date -Format
'MM-dd-yy hh:mm:ss' }} |
Export-Excel .\Processes.xlsx -WorksheetName 'ProcessesOverTime'
```

Выполните эту команду, а затем откройте рабочую книгу «Processes». Вы увидите лист под названием `ProcessesOverTime`, в котором будет список всех запущенных на вашем компьютере процессов, а также дополнительный столбец с отметкой времени. Как вы догадались, в ней указано то время, когда была получена информация о процессе.

На этом этапе мы добавим в лист еще несколько строк, используя эту же команду, но с параметром `Append`. Ее можно запускать сколько угодно, и каждый раз она просто будет добавлять строки в лист:

```
PS> Get-Process |
Select-Object -Property *,@{Name = 'Timestamp';Expression = { Get-Date -Format
'MM-dd-yy hh:mm:ss' }} |
Export-Excel .\Processes.xlsx -WorksheetName 'ProcessesOverTime' -Append
```

После сбора данных вы сможете просмотреть свою книгу Excel и всю собранную информацию о процессах.

Проект 2. Создание инструмента мониторинга служб Windows

Давайте объединим ваши новые навыки, полученные в этом разделе, и поработаем над еще одним маленьким проектом. На этот раз мы создадим процесс для отслеживания состояний служб Windows с указанием времени и будем записывать данные на лист Excel. Затем мы создадим отчет, в котором будет видно, когда различные службы изменяли свое состояние. Иными словами, это будет инструмент мониторинга служб.

Первое, что вам нужно сделать — это выяснить, как обратиться к службам Windows и вернуть для каждой имя и состояние. Сделать это достаточно просто, запустив команду `Get-Service | Select-Object -Property Name,Status`. Затем нужно получить метку времени для каждой строки в листе Excel. Как и в уже

пройденном вами уроке, мы будем использовать для этого вычисляемое свойство (листинг 10.11).

Листинг 10.11. Экспорт состояния службы

```
PS> Get-Service |
Select-Object -Property Name,Status,@{Name = 'Timestamp';Expression =
{ Get-Date -Format 'MM-dd-yy hh:mm:ss' }} |
Export-Excel .\ServiceStates.xlsx -WorksheetName 'Services'
```

У вас получится книга Excel под названием ServiceStates.xlsx. У нее будет один лист с именем Services, который будет выглядеть примерно так, как показано на рис. 10.3.

	A	B	C
1	Name	Status	Timestamp
2	AdtAgent	Stopped	04-22-18 10:06:58
3	AJRouter	Stopped	04-22-18 10:06:58
4	ALG	Stopped	04-22-18 10:06:58
5	AppHostSvc	Running	04-22-18 10:06:58
6	AppIDSvc	Stopped	04-22-18 10:06:58
7	Appinfo	Stopped	04-22-18 10:06:58

Рис. 10.3. Книга Excel

Прежде чем снова запустить ту же команду, давайте попробуем изменить состояние парочки служб Windows. Это позволит нам отслеживать изменения. Остановите и запустите несколько служб, тем самым изменив их состояние. Затем выполните ту же команду из листинга 10.11, но на этот раз используя параметр `Append` команды `Export-Excel`. Так у вас появятся данные для работы. (Не забудьте использовать параметр `Append`, иначе команда перезапишет существующий рабочий лист!)

Теперь, когда у вас появились данные, пришло время подвести итоги. У Excel есть несколько способов сделать это, но для начала остановимся на сводной таблице. *Сводная таблица* — это способ объединения данных. В ней группируется одно или несколько свойств, а затем выполняются различные действия над соответствующими значениями этих свойств (подсчет, добавление и т. д.). С помощью этой таблицы вы сможете легко определить, какие службы изменили состояние и когда это произошло.

Мы будем использовать параметры `IncludePivotTable`, `PivotRows`, `PivotColumns` и `PivotData` для создания сводной таблицы (рис. 10.4).

	A	B	C	D
	Count of Timestamp	Column Labels		
	Row Labels	Stopped	Running	Grand Total
⊟	AdtAgent	4		4
	04-22-18 10:06:58	1		1
	04-22-18 10:11:28	1		1
	04-22-18 10:14:08	1		1
	04-22-18 10:14:53	1		1
⊟	AJRouter	4		4
	04-22-18 10:06:58	1		1
	04-22-18 10:11:28	1		1
	04-22-18 10:14:08	1		1
	04-22-18 10:14:53	1		1
⊟	ALG	4		4
	04-22-18 10:06:58	1		1
	04-22-18 10:11:28	1		1
	04-22-18 10:14:08	1		1
	04-22-18 10:14:53	1		1
⊟	AppHostSvc	2	2	4
	04-22-18 10:06:58		1	1
	04-22-18 10:11:28		1	1
	04-22-18 10:14:08	1		1
	04-22-18 10:14:53	1		1
⊟	AppIDSvc	4		4

Рис. 10.4. Сводная таблица состояния обслуживания

Как видно из листинга 10.12, мы будем считывать данные из рабочего листа Services и использовать эти данные для создания сводной таблицы.

Листинг 10.12. Создание сводной таблицы Excel с помощью PowerShell

```
PS> Import-Excel .\ServiceStates.xlsx -WorksheetName 'Services' |
Export-Excel -Path .\ServiceStates.xlsx -Show -IncludePivotTable
-PivotRows Name,Timestamp
-PivotData @{Timestamp = 'count'} -PivotColumns Status
```

У модуля `ImportExcel` в PowerShell есть ряд доступных параметров. Если вы хотите продолжить работу с этим набором данных, можете поэкспериментировать. Обратите внимание на репозиторий ImportExcel на GitHub (github.com/dfinke/ImportExcel) или попробуйте использовать другие данные. Если у вас есть данные, PowerShell может использовать и выводить их как угодно!

Данные в формате JSON

Если вы работали в ИТ-сфере в течение последних пяти лет, вы, вероятно, сталкивались с объектами JSON. *JavaScript Object Notation* (JSON), созданный в начале 2000-х годов, представляет собой машиночитаемый, но при этом понятный человеку язык иерархической организации наборов данных. Как

следует из названия, такое представление данных активно использовалось в приложениях JavaScript, а следовательно, и в веб-разработке.

Недавний всплеск числа онлайн-сервисов, пользующихся *REST API* (это технология, которую используют для передачи данных между клиентом и сервером), привел к параллельному всплеску использования JSON. Если вы что-то делаете в интернете, вам пригодится навык работы с файлами JSON, которыми можно легко управлять в PowerShell.

Чтение данных в формате JSON

Подобно чтению CSV-файлов, вы можете считать JSON-файлы в PowerShell двумя способами: с анализом или без него. Поскольку JSON — это простой текст, PowerShell по умолчанию обрабатывает его как строку. В качестве примера рассмотрим JSON-файл Employees.json, который прилагается в виде материала к этой главе:

```
{
    "Employees": [
        {
            "FirstName": "Adam",
            "LastName": "Bertram",
            "Department": "IT",
            "Title": "Awesome IT Professional"
        },
        {
            "FirstName": "Bob",
            "LastName": "Smith",
            "Department": "HR",
            "Title": "Crotchety HR guy"
        }
    ]
}
```

Если вам нужен только вывод в виде строки, вы можете использовать команду `Get-Content -Path Employees.json -Raw`. Но со строкой можно мало что сделать. Нужна какая-то структура. Для этого вам понадобится нечто, что понимает устройство JSON (точнее, способ представления отдельных узлов и их массивов в JSON), а также может анализировать файлы соответствующим образом. Вам понадобится командлет `ConvertFrom-Json`.

`ConvertFrom-Json` — это встроенный командлет PowerShell, который принимает на вход необработанный JSON и преобразует его в объекты PowerShell. В листинге 10.13 видно, что PowerShell теперь рассматривает `Employees` как атрибут.

Листинг 10.13. Преобразование JSON в объекты

```
PS> Get-Content -Path .\Employees.json -Raw | ConvertFrom-Json

Employees
---------
{@{FirstName=Adam; LastName=Bertram; Department=IT;
Title=Awesome IT Professional}, @{FirstName=Bob;
LastName=Smith; Department=HR; Title=Crotchety H...
```

Если вы посмотрите на атрибут `Employees`, то увидите, что все узлы сотрудников были проанализированы, причем каждый ключ — это заголовок столбца, а каждое значение представляет значение строки:

```
PS> (Get-Content -Path .\Employees.json -Raw | ConvertFrom-Json).Employees

FirstName LastName Department Title
--------- -------- ---------- -----
Adam      Bertram  IT         Awesome IT Professional
Bob       Smith    HR         Crotchety HR guy
```

Атрибут `Employees` теперь представляет собой массив объектов, которые можно запрашивать и с которыми можно работать, — в общем, работать с ним можно, как и с любым другим массивом.

Создание строк JSON

Допустим, у вас есть данные из множества источников и вы хотите преобразовать их все в формат JSON. Что делать? Здесь проявляются чудеса командлета `ConvertTo-Json` — он способен преобразовать любой объект PowerShell в JSON.

Давайте преобразуем CSV-файл, созданный вами ранее в этой главе, в файл Employees.json. Для начала придется импортировать CSV-файл:

```
PS> Import-Csv -Path .\Employees.csv -Delimiter "`t"

First Name Last Name Department       Manager
---------- --------- ----------       -------
Adam       Bertram   IT               Miranda Bertram
Barack     Obama     Executive Office Michelle Obama
Miranda    Bertram   Executive Office
Michelle   Obama     Executive Office
```

Чтобы выполнить преобразование, нужно передать полученный вывод в `ConvertTo-Json`, как показано в листинге 10.14.

Листинг 10.14. Преобразование объектов в JSON

```
PS> Import-Csv -Path .\Employees.csv -Delimiter "`t" | ConvertTo-Json
[
    {
        "First Name": "Adam",
        "Last Name": "Bertram",
        "Department": "IT",
        "Manager": "Miranda Bertram"
    },
    {
        "First Name": "Barack",
        "Last Name": "Obama",
        "Department": "Executive Office",
        "Manager": "Michelle Obama"
    },
    {
        "First Name": "Miranda",
        "Last Name": "Bertram",
        "Department": "Executive Office",
        "Manager": null
    },
    {
        "First Name": "Michelle",
        "Last Name": "Obama",
        "Department": "Executive Office",
        "Manager": null
    }
]
```

Как и следовало ожидать, для настройки преобразования нужно передать парочку параметров. Один из них — параметр `Compress`, который урезает количество выходных данных и удаляет все потенциально нежелательные разрывы строк:

```
PS> Import-Csv -Path .\Employees.csv -Delimiter "`t" | ConvertTo-Json -Compress
[{"First Name":"Adam","Last
Name":"Bertram","Department":"IT","Manager":"Miranda
Bertram"},{"First Name":"Barack","Last
Name":"Obama","Department":"Executive
Office","Manager":"Michelle Obama"},{"First
Name":"Miranda","Last Name":"Bertram","Department":"Executive
Office","Manager":null},{"First Name":"Michelle",
"Last Name":"Obama","Department":"Executive
Office","Manager":null}]
```

Если есть атрибут и его значение, функция `ConvertTo-Json` сможет работать. Атрибут всегда будет ключом узла, а его значение — значением узла.

Проект 3. Запрос и парсинг REST API

Теперь, когда вы знаете, как работать с данными JSON, давайте сделаем нечто более интересное: мы воспользуемся PowerShell для запроса REST API и анализа полученных результатов. Вы можете использовать практически любой REST API, но для некоторых требуется аутентификация, поэтому будем использовать тот, который ее не требует. Я нашел на сервисе postcodes.io вариант REST API, который позволяет вам запрашивать почтовые индексы Великобритании по различным критериям.

Мы будем использовать URI api.postcodes.io/random/postcodes. Когда вы обращаетесь к этому URI, он запрашивает службу API postcodes.io и возвращает случайный почтовый индекс в форме JSON. Чтобы запросить этот URI, мы будем использовать PowerShell-командлет `Invoke-WebRequest`:

```
PS> $result = Invoke-WebRequest -Uri 'http://api.postcodes.io/random/postcodes'
PS> $result.Content
{"status":200,"result":{"postcode":"IP12
2FE","quality":1,"eastings":641878,"northings":250383,"country
:"England","nhs_ha":"East of England","longitude":
1.53013518866685,"latitude":52.0988661618569,"european_elector
al_region":"Eastern","primary_care_trust":"Suffolk","region":"
East of England","lsoa":"Suffo
lk Coastal 007C","msoa":"Suffolk Coastal
007","incode":"2FE","outcode":"IP12","parliamentary_constituen
cy":"Suffolk Coastal","admin_district":"Suffolk Coa
stal","parish":"Orford","admin_county":"Suffolk","admin_ward":
"Orford & Eyke","ccg":"NHS Ipswich and East
Suffolk","nuts":"Suffolk","codes":{"admin_distri
ct":"E07000205","admin_county":"E10000029","admin_ward":"E0501
449","parish":"E04009440","parliamentary_constituency":"E14000
81","ccg":"E38000086","nuts"
:"UKH14"}}}
```

Теперь посмотрим, сможете ли вы преобразовать результат в объект PowerShell:

```
PS> $result = Invoke-WebRequest -Uri 'http://api.postcodes.io/random/postcodes'
PS> $result.Content | ConvertFrom-Json

status result
------ ------
   200 @{postcode=DE7 9HY; quality=1; eastings=445564;
       northings=343166; country=England; nhs_ha=East Midlands;
       longitude=-1.32277519314161; latitude=...

PS> $result = Invoke-WebRequest -Uri 'http://api.postcodes.io/random/postcodes'
PS> $contentObject = $result.Content | ConvertFrom-Json
PS> $contentObject.result
```

```
postcode                    : HA7 2SR
quality                     : 1
eastings                    : 516924
northings                   : 191681
country                     : England
nhs_ha                      : London
longitude                   : -0.312779792807334
latitude                    : 51.6118279308721
european_electoral_region   : London
primary_care_trust          : Harrow
region                      : London
lsoa                        : Harrow 003C
msoa                        : Harrow 003
incode                      : 2SR
outcode                     : HA7
parliamentary_constituency  : Harrow East
admin_district              : Harrow
parish                      : Harrow, unparished area
admin_county                :
admin_ward                  : Stanmore Park
ccg                         : NHS Harrow
nuts                        : Harrow and Hillingdon
codes                       : @{admin_district=E09000015;
                              admin_county=E99999999; admin_ward=E05000303;
                              parish=E43000205;
```

Вы можете без проблем преобразовать ответ в объект JSON. Для этого нужны две команды — `Invoke-WebRequest` и `ConvertFrom-Json`. Но было бы здорово обойтись всего одной, да? Оказывается, в PowerShell есть команда, которая сделает все за вас: `Invoke-RestMethod`.

Командлет `Invoke-RestMethod` аналогичен `Invoke-WebRequest`: он отправляет различные HTTP-команды веб-службам и возвращает ответ. Поскольку служба API postcodes.io не требует аутентификации, вы можете просто использовать параметр `Uri` в `Invoke-RestMethod`, чтобы получить ответ API:

```
PS> Invoke-RestMethod –Uri 'http://api.postcodes.io/random/postcodes'

status result
------ ------
   200 @{postcode=NE23 6AA; quality=1; eastings=426492;
       northings=576264; country=England; nhs_ha=North East;
       longitude=-1.5865793029774; latitude=55...
```

Мы видим, что команда `Invoke-RestMethod` возвращает код состояния HTTP и ответ от API в атрибуте `result`. А где тут JSON? Как и задумывалось, он уже преобразован в объект. Нет необходимости делать это вручную, так как можно использовать атрибут `result`:

```
PS> (Invoke-RestMethod -Uri 'http://api.postcodes.io/random/postcodes').result

postcode                    : SY11 4BL
quality                     : 1
eastings                    : 332201
northings                   : 331090
country                     : England
nhs_ha                      : West Midlands
longitude                   : -3.00873643515338
latitude                    : 52.8729967314029
european_electoral_region   : West Midlands
primary_care_trust          : Shropshire County
region                      : West Midlands
lsoa                        : Shropshire 011E
msoa                        : Shropshire 011
incode                      : 4BL
outcode                     : SY11
parliamentary_constituency  : North Shropshire
admin_district              : Shropshire
parish                      : Whittington
admin_county                :
admin_ward                  : Whittington
ccg                         : NHS Shropshire
nuts                        : Shropshire CC
codes                       : @{admin_district=E06000051;
                              admin_county=E99999999; admin_ward=E05009287;
                              parish=E04012256;
```

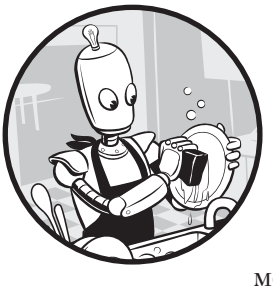
Работать с JSON в PowerShell проще простого. С помощью удобных командлетов PowerShell можно избежать сложного парсинга строк — мы просто передаем JSON или объект, который скоро будет преобразован в JSON, в конвейер и наблюдаем чудеса!

Итоги

В этой главе мы рассмотрели несколько способов структурирования данных, а также узнали, как работать с этими структурами в PowerShell. Встроенные командлеты PowerShell упрощают этот процесс, что позволяет пользоваться простыми командами вместо написания сложного кода. Но не позволяйте этой простоте обмануть вас: PowerShell может анализировать и обрабатывать практически любые данные. Даже если для некоторого типа данных изначально нет команды, ради сложных концептов можно углубиться в любые классы .NET благодаря базе .NET, существующей у PowerShell. В следующей главе мы поработаем с Microsoft Active Directory (AD). Там полно повторяющихся задач, поэтому с AD зачастую начинают изучение принципов работы с PowerShell. Мы проведем немало времени, работая с этим прекрасным ресурсом в остальной части книги.

11

Автоматизация Active Directory



Microsoft Active Directory (AD) — один из лучших продуктов для автоматизации с PowerShell. В организациях сотрудники постоянно приходят, уходят и перемещаются. Для отслеживания этого постоянно меняющегося потока необходима динамическая система, и тут на помощь приходит Microsoft Active Directory (AD). ИТ-специалисты часто выполняют в AD схожие и повторяющиеся задачи, что делает этот ресурс идеальным объектом для автоматизации.

В этой главе мы рассмотрим использование PowerShell для автоматизации нескольких сценариев с участием AD. Управлять его многочисленными объектами можно с помощью PowerShell, но мы рассмотрим три наиболее распространенных метода: через учетные записи пользователей, учетные записи компьютеров и группы. С этими типами объектов администратор AD, вероятно, работает постоянно.

Исходные требования

Пока мы рассматриваем примеры из этой главы, я могу сделать несколько предположений о вашей компьютерной среде.

Во-первых, вы работаете на компьютере с Windows, который уже является членом домена Active Directory. Существуют способы работать в AD с компьютера

рабочей группы, используя другие учетные данные, но это выходит за рамки данной главы.

Во-вторых, вы будете работать с доменом, в котором находится компьютер. Сложные вопросы междоменного взаимодействия и обмена полномочиями также выходят за рамки этой главы.

И последнее: вы зашли на свой компьютер с учетной записью AD, у которой есть соответствующие разрешения на чтение, изменение и создание общих объектов AD, таких как пользователи, компьютеры, группы и подразделения. Я выполняю эти упражнения с компьютера с учетной записью, которая входит в группу администраторов домена, то есть могу контролировать все в домене. Хотя делать это не слишком-то необходимо и, как правило, в производственной среде не рекомендуется, это позволяет нам рассмотреть разные темы, не беспокоясь о правах объектов.

Установка модуля ActiveDirectory в PowerShell

Как вам уже известно, выполнить задачу с помощью PowerShell можно поразному. Кроме того, нет особого смысла изобретать колесо, если вы можете использовать и комбинировать существующие инструменты, чтобы превратить их во что-то более крутое. В этой главе мы будем использовать только один модуль — `ActiveDirectory`. Хотя у него есть недостатки вроде непонятных параметров, необычного синтаксиса фильтрации и странного поведения при ошибках, это все же наиболее полный модуль для управления AD.

Модуль `ActiveDirectory` поставляется с пакетом программного обеспечения *Remote Server Administration Tools*. Это программный пакет, состоящий из множества инструментов. К сожалению, пока что найти модуль `ActiveDirectory` можно только там. Прежде чем продолжить работу с этой главой, я рекомендую вам загрузить и установить этот пакет.

Чтобы убедиться, что модуль ActiveDirectory у вас установлен, можно использовать команду `Get-Module`:

```
PS> Get-Module -Name ActiveDirectory -List
Directory: C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules

ModuleType Version   Name             ExportedCommands
---------- -------   ----             ----------------
Manifest   1.0.0.0   ActiveDirectory  {Add-ADCentralAccessPolicyMember,...
```

Если вы получите такой вывод, значит, `ActiveDirectory` установлен.

Запросы и фильтрация объектов AD

Как только вы убедитесь, что все необходимое установлено, можно приступать к работе.

Один из лучших способов разобраться с новым PowerShell-модулем — это поискать все его команды с глаголом `Get`. Эти команды всего лишь считывают информацию, поэтому риск случайно что-нибудь изменить минимален. Давайте сделаем это и поищем связанные с объектами команды, с которыми мы будем работать в этой главе. В листинге 11.1 показано, как вывести только те команды `ActiveDirectory`, которые начинаются с `Get` и так или иначе содержат слово «computer».

Листинг 11.1. Команды Get модуля ActiveDirectory

```
PS> Get-Command -Module ActiveDirectory -Verb Get -Noun *computer*

CommandType     Name                                Version    Source
-----------     ----                                -------    ------
Cmdlet          Get-ADComputer                      1.0.0.0    ActiveDirectory
Cmdlet          Get-ADComputerServiceAccount        1.0.0.0    ActiveDirectory

PS> Get-Command -Module ActiveDirectory -Verb Get -Noun *user*

CommandType     Name                                Version    Source
-----------     ----                                -------    ------
Cmdlet          Get-ADUser                          1.0.0.0    ActiveDirectory
Cmdlet          Get-ADUserResultantPasswordPolicy   1.0.0.0    ActiveDirectory

PS> Get-Command -Module ActiveDirectory -Verb Get -Noun *group*

CommandType     Name                                Version    Source
-----------     ----                                -------    ------
Cmdlet          Get-ADAccountAuthorizationGroup     1.0.0.0    ActiveDirectory
Cmdlet          Get-ADGroup                         1.0.0.0    ActiveDirectory
Cmdlet          Get-ADGroupMember                   1.0.0.0    ActiveDirectory
Cmdlet          Get-ADPrincipalGroupMembership      1.0.0.0    ActiveDirectory
```

Здесь можно увидеть несколько интересных команд. В этой главе мы будем использовать команды `Get-ADComputer`, `Get-ADUser`, `Get-ADGroup` и `Get-ADGroupMember`.

Фильтрация объектов

У многих команд `Get` модуля AD есть параметр `Filter`. Он похож на команду PowerShell `Where-Object`, поскольку фильтрует результат команды, но выполняет эту задачу иначе.

У параметра `Filter` свой собственный синтаксис, который может быть достаточно трудным для понимания, особенно при использовании сложных фильтров. Чтобы получить полную информацию о синтаксисе этого параметра, запустите команду `Get-Help about_ActiveDirectory_Filter`.

В этой главе мы не будем использовать сложную фильтрацию. Давайте воспользуемся параметром `Filter` и командой `Get-ADUser`, чтобы найти всех пользователей в домене, как показано в листинге 11.2. Однако будьте осторожны: если в вашем домене много учетных записей пользователей, вам придется подождать какое-то время.

Листинг 11.2. Поиск всех учетных записей пользователей в домене

```
PS> Get-ADUser -Filter *

DistinguishedName : CN=adam,CN=Users,DC=lab,DC=local
Enabled           : True
GivenName         :
Name              : adam
ObjectClass       : user
ObjectGUID        : 5e53c562-4fd8-4620-950b-aad8fbaa84db
SamAccountName    : adam
SID               : S-1-5-21-930245869-402111599-3553179568-500
Surname           :
UserPrincipalName :
--пропуск--
```

Как видите, параметр `Filter` принимает подстановочный знак строкового значения `*`. Этот символ указывает (большинству) команд `Get` выдать все, что они найдут. Иногда это может быть действительно необходимо, но все же в большинстве случаев вам не нужны *все* объекты. Однако при правильном использовании подстановочный знак — очень полезная вещь.

Допустим, вы хотите найти все учетные записи компьютеров в AD, начинающиеся с буквы C. Это можно выполнить запуском команды `Get-ADComputer -Filter 'Name -like "C*"'`, где `C*` — это любые возможные символы после *C*. Это работает и в обратную сторону. Скажем, если нужно найти фамилии, оканчивающиеся на *«son»*, следует запустить `Get-ADComputer -Filter 'Name -like "*son"'`.

Если вы хотите найти всех пользователей с фамилией *Jones*, можно запустить команду `Get-ADUser -Filter "surName -eq 'Jones'"`. Для поиска одного пользователя по имени и фамилии можно использовать `Get-ADUser -Filter "surName -eq 'Jones' -and givenName -eq 'Joe'`. Параметр `Filter` позволяет использовать различные операторы PowerShell, такие как `like` и `eq`, для создания фильтра,

который вернет только нужные результаты. Атрибуты Active Directory хранятся в базе данных AD в верблюжьем регистре, поэтому в фильтрах я использовал именно его, хотя прямой необходимости в этом нет.

Еще одна полезная команда для фильтрации объектов AD — это команда `Search-ADAccount`. У этой команды есть встроенная поддержка распространенных сценариев фильтрации, например поиск всех пользователей с истекшим паролем, поиск заблокированных пользователей или поиск компьютеров в сети. Ознакомьтесь со справкой по командлету `Search-AD Account` — там можно найти полный набор параметров.

В большинстве случаев синтаксис `Search-ADAccount` не требует пояснений. Различные параметры вроде `PasswordNeverExpires`, `AccountDisabled` и `AccountExpired` не требуют других параметров для работы.

Помимо этих параметров, у команды `Search-ADAccount` также есть другие, которые требуют дополнительных входных данных. Например, чтобы указать, сколько лет атрибуту `datetime`, или чтобы ограничить результаты определенными типами объектов (например, Users или Computers).

В качестве примера воспользуемся параметром `AccountInactive`. Допустим, вы хотите найти всех пользователей, которые не использовали свою учетную запись в течение 90 дней. Для этого предназначена `Search-ADAccount`. Воспользуйтесь синтаксисом из листинга 11.3 и параметрами `–UsersOnly` для фильтрации типа объекта и `–TimeSpan` для фильтрации объектов, которые не были активны в течение последних 90 дней. Так вы сможете быстро найти всех нужных пользователей.

Листинг 11.3. Использование Search-ADAccount

```
PS> Search-ADAccount -AccountInactive -TimeSpan 90.00:00:00 -UsersOnly
```

Командлет `Search-ADAccount` возвращает объекты типа `Microsoft.Active Directory.Management.ADUser`. Это тот же тип объекта, что и `Get-ADUser` и `Get-ADComputer`. Команда `Search-ADAccount` поможет при использовании `Get`, если вы не знаете, какой именно синтаксис использовать для параметра `Filter`.

Возврат отдельных объектов

Иногда вам нужно выдать определенный объект AD, для чего использование параметра `Filter` не требуется. Вместо этого воспользуйтесь параметром `Identity`.

`Identity` — это гибкий параметр, который позволяет указывать атрибуты объекта AD, делающие его уникальным. Таким образом, он возвращает только один объект. У каждой учетной записи пользователя есть уникальный атрибут `samAccountName`. Вы можете использовать параметр `Filter`, чтобы найти пользователя с определенным значением `samAccountName`. Это будет выглядеть следующим образом:

```
Get-ADUser -Filter "samAccountName -eq 'jjones'"
```

Однако гораздо проще использовать вместо этого параметр `Identity`:

```
Get-ADUser -Identity jjones
```

Проект 4. Поиск учетных записей пользователей, пароль которых не менялся в течение 30 дней

Теперь, когда вы знаете, как запрашивать объекты AD, давайте создадим небольшой сценарий и применим эти знания на практике. Представим задачу: вы работаете в компании, которая собирается внедрить новую политику срока действия пароля, а ваша задача — определить все учетные записи, пароль которых не менялся в течение последних 30 дней.

Давайте подумаем, какую команду лучше использовать. Сперва вы наверняка вспомните команду `Search-ADAccount`, о которой узнали в этой главе. `Search-ADAccount` имеет множество применений для поиска и фильтрации различных объектов, но не позволяет создать собственные фильтры. Чтобы выполнить более детальный поиск, вам придется создавать свой фильтр с помощью команды `Get-ADUser`.

Далее нужно понять, что нужно фильтровать. Вы знаете, что надо вывести учетные записи с паролем, неизменным в течение последних 30 дней, но, ограничившись лишь этим параметром, вы найдете больше учетных записей, чем нужно. Почему? Если вы не добавите свойство `Enabled`, вы найдете еще и старые учетные записи, которые уже не актуальны (это возможно, если кто-то покинул компанию или забросил учетную запись после потери привилегий). Итак, нам нужны активные компьютеры, которые не меняли свой пароль в течение последних 30 дней.

Начнем с фильтрации учетных записей пользователей. Это можно сделать с помощью `-Filter "Enabled -eq 'True'"`. Довольно просто. Следующий шаг — выяснить, как получить доступ к атрибуту, который сохраняется при установке пароля.

По умолчанию команда `Get-ADUser` не возвращает все свойства пользователя. Используя параметр `Properties`, вы можете указать, какие свойства вы хотите видеть: в нашем случае это `name` и `passwordlastset`. Обратите внимание, что у некоторых пользователей нет свойства `passwordlastset`: значит, они никогда не задавали свой собственный пароль.

```
PS> Get-AdUser -Filter * -Properties passwordlastset | select name,passwordlastset

name          passwordlastset
----          ---------------
adam          2/22/2019 6:45:40 AM
Guest
DefaultAccount
krbtgt        2/22/2019 3:03:32 PM
Non-Priv User 2/22/2019 3:12:38 PM
abertram
abertram2
fbar
--пропуск--
```

Теперь, когда у вас есть имя атрибута, вам нужно создать для него фильтр. Помните, что нам нужны только учетные записи, пароли которых менялись за последние 30 дней. Чтобы найти разницу в датах, вам нужны лишь две: самая старая (30 дней назад) и самая новая (сегодня). Узнать сегодняшнюю дату можно с помощью команды `Get-Date`. Затем можно использовать метод `AddDays`, чтобы уточнить дату, которая была 30 дней назад. Сохраним оба значения в переменных — позже мы ими воспользуемся.

```
PS> $today = Get-Date
PS> $30DaysAgo = $today.AddDays(-30)
```

Теперь, когда у вас есть даты, вы можете использовать их в фильтре:

```
PS> Get-ADUser -Filter "passwordlastset -lt '$30DaysAgo'"
```

Осталось лишь добавить в фильтр условие `Enabled`. В листинге 11.4 показано, как это сделать.

Листинг 11.4. Поиск включенных учетных записей пользователей, которые не меняли свой пароль в течение 30 дней

```
$today = Get-Date
$30DaysAgo = $today.AddDays(-30)
Get-ADUser -Filter "Enabled -eq 'True' -and passwordlastset –lt
'$30DaysAgo'"
```

Теперь у вас есть код для поиска всех активных пользователей Active Directory, которые установили свои пароли в течение последних 30 дней.

Создание и изменение объектов AD

Теперь, когда вы знаете, как находить объекты в AD, давайте научимся изменять и создавать их. В этом разделе будет две части: одна посвящена пользователям и компьютерам, а вторая — группам.

Пользователи и компьютеры

Чтобы просмотреть учетные записи пользователей и компьютеров, нужно воспользоваться командой `Set` — либо `Set-ADUser`, либо `Set-ADComputer`. Эти команды позволяют менять любой атрибут объекта. Обычно мы передаем объект из команды `Get` (как те, что рассмотрены в предыдущем уроке) в конвейер.

В качестве примера предположим, что сотрудница по имени Jane Jones вышла замуж и вас просят изменить фамилию ее учетной записи. Если атрибут идентификации учетной записи пользователя неизвестен, для его поиска можно использовать параметр `Filter` в команде `Get-ADUser`. Но сначала следует выяснить, в каком виде AD хранит имя и фамилию каждого пользователя. Затем можно использовать значения этих атрибутов для их передачи в параметр `Filter`.

Один из способов найти все доступные атрибуты в AD — воспользоваться .NET. С помощью объекта схемы можно найти класс пользователя и перечислить все его атрибуты:

```
$schema =[DirectoryServices.ActiveDirectory.ActiveDirectorySchema]::
                                              GetCurrentSchema()
$userClass = $schema.FindClass('user')
$userClass.GetAllProperties().Name
```

Найдем в списке доступных атрибутов `givenName` и `surName` для использования в параметре `Filter` с командой `Get-ADUser` и найдем учетную запись пользователя. Затем можно передать этот объект команде `Set-ADUser`, как показано в листинге 11.5.

Листинг 11.5. Изменение атрибутов объекта AD с помощью команды Set-ADUser

```
PS> Get-ADUser -Filter "givenName -eq 'Jane' -and surName –eq
'Jones'" | Set-ADUser -Surname 'Smith'
PS> Get-ADUser -Filter "givenName -eq 'Jane' -and surName –eq
'Smith'"

DistinguishedName : CN=jjones,CN=Users,DC=lab,DC=local
```

```
Enabled           : False
GivenName         : Jane
Name              : jjones
ObjectClass       : user
ObjectGUID        : fbddbd77-ac35-4664-899c-0683c6ce8457
SamAccountName    : jjones
SID               : S-1-5-21-930245869-402111599-3553179568-3103
Surname           : Smith
UserPrincipalName :
```

Также можно изменять несколько атрибутов одновременно. Оказывается, наша Джейн перешла в другой отдел и заодно получила повышение. Это не проблема. Вам просто нужно использовать параметры, соответствующие атрибутам AD:

```
PS> Get-ADUser -Filter "givenName -eq 'Jane' -and surname –eq
'Smith'" | Set-ADUser -Department 'HR' -Title Director
PS> Get-ADUser -Filter "givenName -eq 'Jane' -and surname –eq
'Smith'" -Properties GivenName,SurName,Department,Title

Department        : HR
DistinguishedName : CN=jjones,CN=Users,DC=lab,DC=local
Enabled           : False
GivenName         : Jane
Name              : jjones
ObjectClass       : user
ObjectGUID        : fbddbd77-ac35-4664-899c-0683c6ce8457
SamAccountName    : jjones
SID               : S-1-5-21-930245869-402111599-3553179568-3103
Surname           : Smith
Title             : Director
UserPrincipalName :
```

Наконец, можно создавать объекты AD с помощью команд `New-AD*`. Создание новых объектов AD работает аналогично изменению существующих, но в этом случае у вас не будет доступа к параметру `Identity`. Создать новую учетную запись компьютера в AD проще простого: запустите команду `New-ADComputer -Name FOO`. Аналогично можно создать учетную запись пользователя с помощью команды `New-ADUser -Name adam`. Заметно, что у команд `New-AD*` есть параметры, совпадающие с атрибутами AD, как и у команд `Set-AD*`.

Группы

Группы устроены сложнее, чем пользователи и компьютеры. Это своего рода контейнер для объектов AD. В этом смысле группа — это куча вещей. Но в то же время она является *одиночным* контейнером, то есть представляет собой единый объект AD, как пользователи или компьютеры. Поэтому вы можете

запрашивать, создавать и изменять группы так же, как делали это с пользователями и компьютерами, но с парой отличий.

Предположим, ваша организация создала новый отдел под названием «AdamBertram Lovers», и в нем полным-полно новых сотрудников. Теперь вам нужно создать группу с таким же названием. В листинге 11.6 показан пример ее создания. Мы будем использовать параметр `Description`, чтобы передать строку описания группы, и параметр `GroupScope`, чтобы у созданной группы была область `DomainLocal`. При необходимости можно было бы также выбрать область `Global` или `Universal`.

Листинг 11.6. Создание группы AD

```
PS> New-ADGroup -Name 'AdamBertramLovers'
-Description 'All Adam Bertram lovers in the company'
-GroupScope DomainLocal
```

Когда группа будет создана, ее можно будет изменять так же, как в случае с пользователем или компьютером. Например, чтобы изменить описание, вы можете написать следующим образом:

```
PS> Get-ADGroup -Identity AdamBertramLovers |
Set-ADGroup -Description 'More Adam Bertram lovers'
```

Конечно, основное различие между группами и пользователями/компьютерами состоит в том, что последние могут являться частью группы. Однако вы не можете использовать известные вам команды для добавления и изменения членов группы. Вместо этого воспользуйтесь командами `Add-ADGroupMember` и `Remove-ADGroupMember`.

Например, добавить Джейн в нашу группу поможет команда `Add-ADGroupMember`. Если Джейн захочет покинуть группу, вы можете удалить ее с помощью команды `Remove-ADGroupMember`. После введения команды появляется запрос с просьбой подтвердить удаление члена группы:

```
PS> Get-ADGroup -Identity AdamBertramLovers | Add-ADGroupMember Members 'jjones'
PS> Get-ADGroup -Identity AdamBertramLovers | Remove-ADGroupMember-Members 'jjones'

    Confirm
Are you sure you want to perform this action?
Performing the operation "Set" on target
"CN=AdamBertramLovers,CN=Users,DC=lab,DC=local".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend
[?]
Help (default is "Y"): a
```

Вы можете добавить параметр `Force`, если хотите пропустить эту проверку, но в какой-то момент такое решение может вас подвести!

Проект 5. Создание сценария приема сотрудников

Давайте закрепим новые знания и рассмотрим другой практический сценарий. Ваша компания наняла нового сотрудника. Теперь вы, как системный администратор, должны выполнить ряд действий: создать пользователя AD, создать учетную запись для его компьютера и добавить его в определенные группы. Мы создадим сценарий для автоматизации этого процесса.

Прежде чем вы начнете этот проект (да и вообще любой другой), важно выяснить, что именно сценарий будет делать, то есть описать его работу простым языком. В нашем случае вам необходимо создать пользователя AD, который будет:

- динамически создавать имя пользователя, используя для этого имя и фамилию;
- создавать и назначать пользователю случайный пароль;
- заставлять пользователя менять пароль при входе в систему;
- устанавливать атрибут отдела, используя для этого информацию об отделе будущего пользователя;
- назначать пользователю внутренний номер сотрудника.

Наконец, добавим учетную запись пользователя в группу, названную по имени отдела.

С перечнем задач мы определились, теперь давайте создадим сценарий. Уже готовый сценарий называется New-Employee.ps1, он находится в прилагаемых к книге файлах.

Мы хотим, чтобы этот сценарий можно было использовать многократно. В идеале он должен делать всю работу за нас. Значит, необходимо найти подходящий способ передачи входных данных в сценарий. Из требований ясно, что вам понадобятся имя, фамилия, отдел и номер сотрудника. В листинге 11.7 представлена схема сценария с определенными параметрами и блоком `try/catch`, которые позволяют обнаружить возможные завершающие ошибки. Вверху прописан оператор `#requires`, который при запуске будет проверять наличие модуля `ActiveDirectory` на компьютере.

Листинг 11.7. Основа сценария New-Employee.ps1

```
#requires -Module ActiveDirectory

[CmdletBinding()]
param (
    [Parameter(Mandatory)]
    [string]$FirstName,

    [Parameter(Mandatory)]
    [string]$LastName,

    [Parameter(Mandatory)]
    [string]$Department,

    [Parameter(Mandatory)]
    [int]$EmployeeNumber
)

try {

} catch {
    Write-Error -Message $_.Exception.Message
}
```

Теперь, когда вы создали базу, заполним блок `try`.

Сначала вам нужно создать пользователя AD в соответствии с требованиями, изложенными в нашем неформальном определении. Имя пользователя нужно *создавать динамически*. Для этого есть несколько способов: некоторые организации предпочитают, чтобы имя пользователя состояло из первой буквы имени и фамилии, другие используют полное имя и фамилию, а третьи вообще делают что-то совершенно другое. Допустим, в нашем случае это имя и фамилия. Если это имя пользователя занято, мы будем добавлять следующий символ до тех пор, пока не будет создано уникальное имя пользователя.

Давайте сначала разберемся с базовым случаем. Мы будем использовать встроенный метод `Substring` для каждого строкового объекта, чтобы получить первую букву имени. Затем мы объединим фамилию с этой буквой, используя *форматирование строки* — оно позволяет определить плейсхолдеры для многочисленных выражений в строке, которые будут заменяться значениями во время выполнения программы:

```
$userName = '{0}{1}' -f $FirstName.Substring(0, 1), $LastName
```

После создания имени пользователя необходимо уточнить его статус у AD — возможно, оно уже занято. Тут поможет команда `Get-ADUser`.

```
Get-ADUser -Filter "samAccountName -eq '$userName'"
```

Если эта команда что-нибудь вернет, значит, имя пользователя уже занято, и вам нужно попробовать создать другое. Таким образом, необходимо найти способ динамически генерировать новые имена, будучи готовым к тому, что имя уже занято. Для проверки использованных имен пользователей можно применить цикл `while`, который обусловлен вашим предыдущим вызовом `Get-ADUser`. Но вам понадобится еще одно условие на случай, если буквы в имени закончатся. Цикл не должен работать вечно, поэтому добавим еще условие для остановки: `$userName -notlie "$FirstName*"`.

Условие `while` выглядит так:

```
(Get-ADUser -Filter "samAccountName -eq '$userName'") –and
($userName -notlike "$FirstName*")
```

Теперь заполним оставшуюся часть цикла:

```
$i = 2
while ((Get-ADUser -Filter "samAccountName -eq '$userName'") –and
($userName -notlike "$FirstName*")) {
    Write-Warning -Message "The username [$($userName)] already exists.
    Trying another..."
    $userName = '{0}{1}' -f $FirstName.Substring(0, $i), $LastName
    Start-Sleep -Seconds 1
    $i++
}
```

В каждой итерации цикла мы будем добавлять дополнительный символ из имени к предлагаемому имени пользователя, беря подстроку от `0` до `i`, где `$i` — переменная счетчика, которая начинается с `2` (следующая позиция в строке) и увеличивается каждый раз при запуске цикла. К моменту завершения цикла `while` он либо найдет уникальное имя пользователя, либо исчерпает все параметры.

Если проверяемое имя пользователя *свободно*, вы его займете, как и намеревались. Если же такое имя *найдено*, вам нужно проверить еще несколько вещей. Следует посмотреть, существует ли *организационная единица (OU)* и группа, в которую вы помещаете учетную запись пользователя:

```
if (-not ($ou = Get-ADOrganizationalUnit -Filter "Name –eq '$Department'")) {
     throw "The Active Directory OU for department [$($Department)] could
     not be found."
} elseif (-not (Get-ADGroup -Filter "Name -eq '$Department'")) {
     throw "The group [$($Department)] does not exist."
}
```

После всех проверок необходимо создать учетную запись пользователя. Снова вернемся к словесному описанию: нужно *создать и назначить пользователю*

случайный пароль. Его следует генерировать при каждом запуске этого сценария. Для этого проще всего использовать статический метод `GeneratePassword` в объекте `System.Web.Security.Membership`, как показано ниже:

```
Add-Type -AssemblyName 'System.Web'
$password = [System.Web.Security.Membership]::GeneratePassword(
    (Get-Random Minimum 20 -Maximum 32), 3)
$secPw = ConvertTo-SecureString -String $password -AsPlainText -Force
```

Я решил сгенерировать пароль, состоящий из набора от 20 до 32 символов, но вообще это можно изменить. При желании вы также можете найти минимально допустимый пароль для AD, запустив команду `Get-ADDefaultDomain PasswordPolicy | Select-object -expand minPasswordLength`. Этот метод даже позволяет указать длину и сложность нового пароля.

Теперь, с паролем в виде защищенной строки у вас есть все значения параметров, необходимые для создания пользователя в соответствии с требованиями, которые я изложил ранее.

```
 $newUserParams             = @{
     GivenName              = $FirstName
     EmployeeNumber         = $EmployeeNumber
     Surname                = $LastName
     Name                   = $userName
     AccountPassword        = $secPw
     ChangePasswordAtLogon  = $true
     Enabled                = $true
     Department             = $Department
     Path                   = $ou.DistinguishedName
     Confirm                = $false
}
New-ADUser @newUserParams
```

После создания пользователя вам лишь останется добавить его в группу, соответствующую отделу, с помощью простой команды `Add-ADGroupMember`:

```
Add-ADGroupMember -Identity $Department -Members $userName
```

Не забудьте глянуть на сценарий New-Employee.ps1 в материалах, прилагаемых к книге, — там он реализован целиком.

Синхронизация с другими источниками данных

На крупных предприятиях десятки людей ежедневно создают и изменяют миллионы объектов Active Directory. При такой активности и количестве

данных проблемы неизбежны. Одна из самых серьезных проблем, с которыми вы можете столкнуться, — это синхронизация базы данных AD с остальной частью организации.

AD компании должна быть организована по образцу самой компании. Это означает, что у каждого отдела должна быть собственная группа в AD, у каждого офиса — свое подразделение, и т. д. Перед нами стоит сложная задача обеспечения непрерывной синхронизации AD с остальной частью организации. Это посильная задача для PowerShell.

С помощью PowerShell вы можете «связать» AD практически с любым другим источником информации, то есть постоянно считывать внешние источники данных и синхронизировать их с AD по мере необходимости.

Этот процесс примерно состоит из следующих этапов:

1. Запрос внешнего источника данных (базы данных SQL, CSV-файла и т. д.).
2. Получение объектов из AD.
3. Определение каждого объекта в источнике, для которого у AD есть уникальный атрибут для сопоставления. Обычно он называется *ID* (например, ID сотрудника или даже имя пользователя). Единственным важным условием является уникальность атрибута. Если совпадений не найдено, при желании можно создать или удалить объект из AD.
4. Нахождение единственно подходящего объекта.
5. Сопоставление внешних источников данных с атрибутами объекта AD.
6. Изменение существующего объекта AD либо создание нового.

В следующем разделе мы будем воплощать этот план в действие.

Проект 6. Создание сценария синхронизации

В этом разделе вы узнаете, как создать сценарий для синхронизации сотрудников из CSV-файла с AD. Здесь также будут использоваться команды из предыдущей главы. Перед началом я рекомендую вам взглянуть на Employees.csv и Invoke-AdCsvSync.ps1 из материалов книги и ознакомиться с файлами проекта.

Стандартизация — это ключ к созданию хорошего инструмента синхронизации AD. Я не имею в виду, что источники данных должны быть одними и теми же (технически это невозможно), но ваш сценарий должен опрашивать каждое хранилище данных одинаковым образом, и каждое хранилище данных должно возвращать один и тот же тип объектов. Самое сложное — это работать с двумя

источниками, использующими разные схемы. В этом случае, возможно, придется реализовать перевод путем сопоставления одного имени поля с другим (позже мы это сделаем).

Представим следующее: вы уже знаете, что у AD есть общие атрибуты, связанные с каждой учетной записью пользователя, — например, имя, фамилия и отдел. Мы назовем их *схемой* атрибутов. Однако всегда есть вероятность, что у исходного хранилища данных никогда не будет одинаковых атрибутов. И даже если они будут, у них все равно будут разные имена. Для решения этой проблемы необходимо настроить сопоставление между двумя хранилищами данных.

Сопоставление атрибутов источника данных

Простой и эффективный способ создать сопоставление — использовать хеш-таблицу. Ее ключом будет имя атрибута в первом хранилище данных, а значением — имя атрибута во втором хранилище. Приведу пример. Предположим, что вы работаете в компании под названием Acme. Acme хочет синхронизировать записи о сотрудниках из CSV-файла в AD. Если говорить точнее, мы хотим синхронизировать файл Employees.csv:

```
"fname","lname","dept"
"Adam","Bertram","IT"
"Barack","Obama","Executive Office"
"Miranda","Bertram","Executive Office"
"Michelle","Obama","Executive Office"
```

Зная заголовки CSV и имена свойств в AD, вы можете создать хеш-таблицу, где значение поля CSV будет ключом, а имя атрибута AD — значением:

```
$syncFieldMap = @{
     fname = 'GivenName'
     lname = 'Surname'
     dept = 'Department'
}
```

Так мы сможем выполнить преобразование между двумя схемами хранения данных. Также нам понадобится уникальный идентификатор для каждого сотрудника. Сейчас у нас нет уникального идентификатора, который мог бы соответствовать объекту AD в каждой строке CSV. Например, у вас может быть несколько Адамов, более одного сотрудника в ИТ-отделе или несколько человек с фамилией Бертрам. Значит, нужно создать собственный уникальный идентификатор. Чтобы упростить задачу, предположим, что в базе нет

двух сотрудников с одинаковыми именем и фамилией. В противном случае создаваемый идентификатор, вероятно, будет зависеть от вашей организационной схемы. Исходя из этого предположения, вы можете просто объединить соответствующие поля имени и фамилии каждого хранилища данных, чтобы создать временный уникальный идентификатор.

Его мы добавим в другую хеш-таблицу. Мы еще не выполнили конкатенацию, но уже подготовили основу:

```
$fieldMatchIds = @{
     AD = @('givenName','surName')
     CSV = @('fname','lname')
}
```

Теперь, когда мы реализовали способ сопоставления различных полей, можно включить этот код в пару функций, чтобы «заставить» два хранилища данных возвращать одни и те же свойства, позволяя сравнивать подобное с подобным.

Создание функций для возврата схожих свойств

Теперь, когда у нас есть хеш-таблицы, необходимо перенести имена полей и создать уникальные идентификаторы. Можно создать функцию для сопоставления обоих хранилищ данных, которая будет запрашивать CSV-файл и выводить оба атрибута. Создадим для этого функцию `Get-Acme EmployeeFromCsv`, которая показана в листинге 11.8. Я задал параметру `CsvFilePath` значение `C:\Employees.csv`, подразумевая, что наш CSV-файл находится именно там.

Листинг 11.8. Функция Get-AcmeEmployeeFromCsv

```
function Get-AcmeEmployeeFromCsv
{
[CmdletBinding()]
     param (
          [Parameter()]
          [string]$CsvFilePath = 'C:\Employees.csv',

          [Parameter(Mandatory)]
          [hashtable]$SyncFieldMap,

          [Parameter(Mandatory)]
          [hashtable]$FieldMatchIds
     )
     try {
          ## Read each key/value pair in $SyncFieldMap to create calculated
          ## fields which we can pass to Select-Object later. This allows us to
          ## return property names that match Active Directory attributes rather
```

```
        ## than what's in the CSV file.
    ❶   $properties = $SyncFieldMap.GetEnumerator() | ForEach-Object {
            @{
                Name = $_.Value
                Expression = [scriptblock]::Create("`$_.$($_.Key)")
            }
        }
        ## Create the unique ID based on the unique fields defined in
        ## $FieldMatchIds
    ❷   $uniqueIdProperty = '"{0}{1}" -f '
        $uniqueIdProperty = $uniqueIdProperty +=
        ($FieldMatchIds.CSV | ForEach-Object { '$_.{0}' -f $_ }) – join ','
        $properties += @{
            Name = 'UniqueID'
            Expression = [scriptblock]::Create($uniqueIdProperty)
        }
        ## Read the CSV file and "transform" the CSV fields to AD attributes
        ## so we can compare apples to apples
    ❸   Import-Csv -Path $CsvFilePath | Select-Object – Property $properties
    } catch {
        Write-Error -Message $_.Exception.Message
    }
}
```

Эта функция работает в три этапа: сначала сопоставляет атрибуты CSV-файла с атрибутами AD ❶, затем создает уникальный идентификатор и делает его атрибутом ❷ и, наконец, считывает CSV-файл, используя функцию `Select-Object` и вычисляемое свойство, чтобы вернуть нужные вам атрибуты ❸.

Как видно из следующего примера кода, можно передать хеш-таблицы `$syncFieldMap` и `$fieldMatchIds` своей новой функции `Get-AcmeEmployeeFromCsv` для возврата имен атрибутов, которые будут синхронизироваться с атрибутами Active Directory и с новым уникальным идентификатором:

```
PS> Get-AcmeEmployeeFromCsv -SyncFieldMap $syncFieldMap
-FieldMatchIds $fieldMatchIds

GivenName Department       Surname UniqueID
--------- ----------       ------- --------
Adam      IT               Bertram AdamBertram
Barack    Executive Office Obama   BarackObama
Miranda   Executive Office Bertram MirandaBertram
Michelle  Executive Office Obama   MichelleObama
```

Теперь следует создать функцию для выполнения запроса из AD. К счастью, на этот раз нам не нужно преобразовывать имена атрибутов — подойдут имена атрибутов AD. В этой функции мы будем лишь вызывать `Get-ADUser` и проверять наличие возврата нужных атрибутов, как показано в листинге 11.9.

Листинг 11.9. Функция Get-AcmeEmployeeFromAD

```
function Get-AcmeEmployeeFromAD
{
     [CmdletBinding()]
     param (
         [Parameter(Mandatory)]
         [hashtable]$SyncFieldMap,

         [Parameter(Mandatory)]
         [hashtable]$FieldMatchIds
     )

     try {
          $uniqueIdProperty = '"{0}{1}" -f '
          $uniqueIdProperty += ($FieldMatchIds.AD | ForEach Object
                                { '$_.{0}' -f $_ }) -join ','

         $uniqueIdProperty = @{ ❶
               Name = 'UniqueID'
               Expression = [scriptblock]::Create($uniqueIdProperty)
          }

          Get-ADUser -Filter * -Properties @($SyncFieldMap.Values) | Select-Object
                                           *,$uniqueIdProperty ❷
     } catch {
          Write-Error -Message $_.Exception.Message
     }
}
```

Перечислим особенности этого кода: сначала мы создаем уникальный идентификатор для выполнения сопоставлений ❶, затем запрашиваем список пользователей AD и возвращаем значения в хеш-таблице и ранее созданный уникальный идентификатор ❷.

После запуска видно, что программа возвращает учетные записи пользователей AD с нужными свойствами и уникальным ID.

Поиск совпадений в Active Directory

Теперь у нас есть две схожие функции, которые извлекают информацию из хранилищ данных и возвращают те же имена атрибутов.

Следующий шаг — найти все совпадения между CSV и AD. Чтобы упростить работу, воспользуемся кодом из листинга 11.10 для создания другой функции с именем `Find-UserMatch`, которая будет выполнять обе эти задачи и собирать оба набора данных. После получения данных функция будет искать совпадение в поле `UniqueID`.

Листинг 11.10. Поиск совпадений пользователей

```
function Find-UserMatch {
     [OutputType()]
     [CmdletBinding()]
     param
     (
         [Parameter(Mandatory)]
         [hashtable]$SyncFieldMap,

         [Parameter(Mandatory)]
         [hashtable]$FieldMatchIds
     )
     $adusers = Get-AcmeEmployeeFromAD -SyncFieldMap $SyncFieldMap -
FieldMatchIds $FieldMatchIds ❶

     $csvUsers = Get-AcmeEmployeeFromCSV -SyncFieldMap $SyncFieldMap -FieldMatchIds
                 $FieldMatchIds ❷

     $adUsers.foreach({
          $adUniqueId = $_.UniqueID
          if ($adUniqueId) { ❸
               $output = @{
                    CSVProperties = 'NoMatch'
                    ADSamAccountName = $_.samAccountName
               }
               if ($adUniqueId -in $csvUsers.UniqueId) { ❹
                   $output.CSVProperties = ($csvUsers.Where({$_.UniqueId
                                            -eq $adUniqueId})) ❺
               }
               [pscustomobject]$output
          }
     })
}
```

Давайте разберем этот код. Сначала мы получаем пользователей из AD ❶, а затем из CSV-файла ❷. Для каждого пользователя из AD мы проверяем заполненность атрибута `UniqueID` ❸. Если таковое имеется, проверяем, найдется ли совпадение между пользователями из CSV и AD ❹. При совпадении в настраиваемом объекте создаем атрибут с именем `CSVProperties`, который содержит все атрибуты, связанные с совпадающим пользователем ❺.

Если совпадение найдено, функция вернет `samAccountName` пользователя AD и все его атрибуты из CSV; в противном случае она вернет `NoMatch`. Возвращение атрибута `samAccountName` — это уникальный идентификатор в AD, который позволяет позже найти этого пользователя.

```
PS> Find-UserMatch -SyncFieldMap $syncFieldMap -FieldMatchIds $fieldMatchIds

ADSamAccountName CSVProperties
```

```
---------------- -------------
user             NoMatch
abertram         {@{GivenName=Adam; Department=IT;
                 Surname=Bertram; UniqueID=AdamBertram}}
dbddar           NoMatch
jjones           NoMatch
BSmith           NoMatch
```

Теперь у нас есть функция, которая позволяет находить совпадения между данными в AD и в CSV. Вы готовы начать приятную (но пугающую) работу по внесению массовых изменений в AD!

Изменение атрибутов Active Directory

Теперь вы можете узнать, какая строка CSV относится к каждой учетной записи пользователя AD. Можно использовать функцию `Find-UserMatch`, чтобы найти пользователя AD по его уникальному идентификатору, а затем обновить его информацию из AD, чтобы она соответствовала данным в CSV, как показано в листинге 11.11.

Листинг 11.11. Синхронизация CSV с атрибутами AD

```
## Find all of the CSV <--> AD user account matches
$positiveMatches = (Find-UserMatch).where({ $_.CSVProperties -ne 'NoMatch' })
foreach ($positiveMatch in $positiveMatches) {
     ## Create the splatting parameters for Set-ADUser using
     ## the identity of the AD samAccountName
     $setADUserParams = @{
          Identity = $positiveMatch.ADSamAccountName
     }

     ## Read each property value that was in the CSV file
     $positiveMatch.CSVProperties.foreach({
          ## Add a parameter to Set-ADUser for all of the CSV
          ## properties excluding UniqueId
          ## Find all of the properties on the CSV row that are NOT UniqueId
          $_.PSObject.Properties.where({ $_.Name -ne 'UniqueID' }).foreach({
               $setADUserParams[$_.Name] = $_.Value
          })
     })
     Set-ADUser @setADUserParams
}
```

Для создания надежного и гибкого сценария синхронизации AD нужно много поработать. По пути вы столкнетесь с множеством мелких деталей и сбоев, особенно когда будете создавать еще более сложные сценарии.

С темой синхронизации с PowerShell мы познакомились лишь самую малость. Если вы хотите узнать, насколько глубока кроличья нора, ознакомьтесь с модулем `PSADSync` в PowerShell Gallery (`Find-Module PSADSync`). Этот модуль был специально создан для задач вроде нашей, но он способен работать с гораздо более сложными случаями. Если вы немного растерялись во время этого упражнения, я настоятельно рекомендую пересмотреть код еще несколько раз. Единственный верный способ изучить PowerShell — это экспериментировать! Запустите код, расстройтесь из-за его неработоспособности, исправьте его и пробуйте снова.

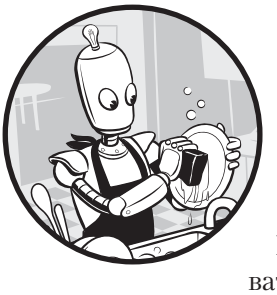
Итоги

В этой главе вы познакомились с PowerShell-модулем `ActiveDirectory`. Вы узнали, как создавать и обновлять списки пользователей, компьютеров и групп в AD. На нескольких практических примерах мы увидели, как можно использовать PowerShell для автоматизации утомительной работы с Active Directory.

В следующих двух главах мы переходим к облаку! Мы продолжим наше путешествие по автоматизации всего и вся и рассмотрим автоматизацию некоторых общих задач, которые выполняются в Microsoft Azure и в Amazon Web Services (AWS).

12

Работа с Azure



Сейчас компании стараются перевести все больше сервисов в облачное хранилище, поэтому специалистам по автоматизации нужно знать, как с ними работать. К счастью, благодаря PowerShell и его модулям взаимодействовать с облаком будет очень просто. В этой и следующей главах я покажу вам, как использовать PowerShell для автоматизации задач: в этой главе — с Microsoft Azure, а в следующей — с Amazon Web Services.

Исходные требования

Если вы будете работать с кодом из этой главы, вам понадобится несколько вещей. Для начала, у вас должна быть настроена подписка на Microsoft Azure. В этой главе вы будете работать с реальными облачными ресурсами, поэтому за учетную запись вам придется платить, но плата должна быть разумной. Пока вы надолго не оставляете включенной ни одну из ваших виртуальных машин, плата не будет превышать 10 долларов.

После настройки подписки Azure вам понадобится пакет модулей PowerShell Az. Он содержит сотни команд для выполнения задач практически во всех службах Azure. Его можно скачать, запустив в консоли команду `Install-Module Az` (убедитесь, что работаете от имени администратора). Обращаю ваше внимание, что я использую версию модуля Az 2.4.0. Если вы используете более позднюю версию, возможно, некоторые команды будут работать иначе.

Авторизация в Azure

В Azure предусмотрено несколько способов авторизации. В этой главе мы будем использовать субъект-службу. *Субъект-служба* — это идентификация приложения Azure, объект, представляющий приложение, которому назначаются различные разрешения.

Зачем ее создавать? Нам нужно авторизоваться в Azure с помощью автоматизированного сценария, не требующего взаимодействия с пользователем. Для этого в Azure требуется использовать субъект-службу или учетную запись организации. Я хочу, чтобы материалы книги были доступны всем, независимо от типа учетной записи, поэтому вы будете использовать субъект-службу для аутентификации в Azure.

Создание субъекта-службы

Как ни странно, первое, что вам нужно сделать для создания субъекта-службы, — это авторизоваться старомодным способом. Воспользуемся командой `Connect-AzAccount`, которая откроет окно, как на рис. 12.1.

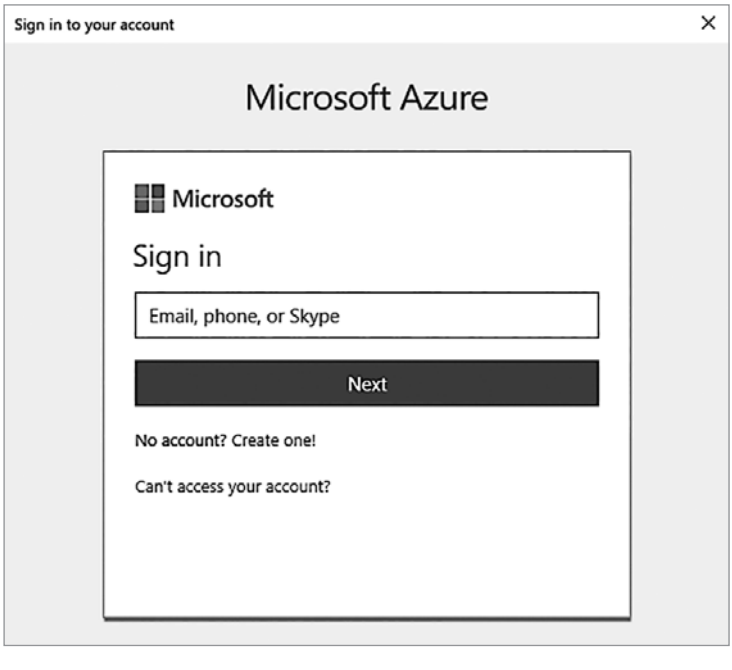


Рис. 12.1. Запрос учетных данных Connect-AzAccount

Укажите свое имя пользователя и пароль Azure, после чего окно закроется и появится результат, как в листинге 12.1.

Листинг 12.1. Результат выполнения команды Connect-AzAccount

```
PS> Connect-AzAccount

Environment           : AzureCloud
Account               : email
TenantId              : tenant id
SubscriptionId        : subscription id
SubscriptionName      : subscription name
CurrentStorageAccount :
```

Обязательно запишите ID подписки и клиента — позже они вам понадобятся. Если по какой-то причине вы не получили эти данные с помощью команды `Connect-AzAccount`, можно сделать это с помощью `Get-AzSubscription`.

Теперь, когда вы авторизовались в интерактивном режиме, можно приступить к созданию субъекта-службы. Это трехэтапный процесс: сначала вы создаете новое приложение Azure AD, затем саму субъект-службу и, наконец, назначаете для нее роль.

Создать приложение Azure AD можно с любым именем и URI (листинг 12.2). Вид URI пока не имеет значения, но он потребуется для создания приложения AD. Нужно убедиться, что у вас есть соответствующие права для создания приложения AD (см. docs.microsoft.com/en-us/azure/active-directory/develop/app-objects-and-service-principals).

Листинг 12.2. Создание приложения Azure AD

```
PS> ❶$secPassword = ConvertTo-SecureString -AsPlainText -Force -String 'password'
PS> ❷$myApp = New-AzADApplication -DisplayName AppForServicePrincipal
-IdentifierUris
'http://Some URL here' -Password $secPassword
```

На примере кода видно, что сначала с помощью пароля создается безопасная строка ❶. Потом создается новое приложение Azure AD ❷ — это требуется для субъекта-службы.

Затем используется команда `New-AzADServicePrincipal` для создания субъекта-службы, как показано в листинге 12.3. В нем мы ссылаемся на приложение из листинга 12.2.

Наконец, необходимо назначить субъекту-службе нужную роль. В листинге 12.4 назначается роль `Contributor` (Участник), чтобы гарантировать, что субъект-служба будет иметь доступ для выполнения всех задач из этой главы.

Листинг 12.3. Создание субъекта-службы Azure с помощью PowerShell

```
PS> $sp = New-AzADServicePrincipal -ApplicationId $myApp.ApplicationId
PS> $sp

ServicePrincipalNames : {application id, http://appforserviceprincipal}
ApplicationId         : application id
DisplayName           : AppForServicePrincipal
Id                    : service principal id
Type                  : ServicePrincipal
```

Листинг 12.4. Назначение роли для субъекта-службы

```
PS> New-AzRoleAssignment -RoleDefinitionName Contributor -ServicePrincipalName
$sp.ServicePrincipalNames[0]

RoleAssignmentId   : /subscriptions/subscription id/providers
                     /Microsoft.Authorization/roleAssignments/assignment id
Scope              : /subscriptions/subscription id
DisplayName        : AppForServicePrincipal
SignInName         :
RoleDefinitionName : Contributor
RoleDefinitionId   : id
ObjectId           : id
ObjectType         : ServicePrincipal
CanDelegate        : False
```

Таким образом, мы создали субъект-службу и задали ей роль.

Осталось лишь сохранить зашифрованный пароль в виде защищенной строки для приложения на диске. Вы можете сделать это с помощью команды `ConvertFrom-SecureString`. Команда `ConvertFrom-SecureString` (дополнение к `ConvertTo-SecureString`) преобразует зашифрованный текст из защищенной строки PowerShell в общую строку, что позже позволит ее использовать:

```
PS> $secPassword | ConvertFrom-SecureString | Out-File -FilePath C:\
AzureAppPassword.txt
```

После сохранения пароля на диске вы будете готовы к настройке неинтерактивной авторизации на Azure.

Неинтерактивная авторизация с помощью команды Connect-AzAccount

Команда `Connect-AzAccount` предлагает вручную ввести имя пользователя и пароль. При создании сценариев следует избегать интерактивности, ведь вы точно не хотите полагаться на кого-то, кто будет сидеть за компьютером и вводить ваш пароль! К счастью, объект `PSCredential` можно передать в `Connect-AzAccount`.

Мы напишем небольшой сценарий для реализации неинтерактивной авторизации. Сначала создадим объект `PSCredential`, содержащий идентификатор и пароль приложения Azure:

```
$azureAppId = 'application id'
$azureAppIdPasswordFilePath = 'C:\AzureAppPassword.txt'
$pwd = (Get-Content -Path $azureAppIdPasswordFilePath | ConvertTo-SecureString)
$azureAppCred = (New-Object System.Management.Automation.PSCredential
                                                    $azureAppId,$pwd)
```

Помните идентификаторы подписки и клиента, которые вы записали ранее? Их также нужно будет передать в `Connect-AzAccount`:

```
$subscriptionId = 'subscription id'
$tenantId = 'tenant id'
Connect-AzAccount -ServicePrincipal -SubscriptionId $subscriptionId -TenantId
                                                                    $tenantId
-Credential $azureAppCred
```

Теперь все готово для неинтерактивной аутентификации! Все настроено и сохранено, поэтому проходить авторизацию больше не придется.

Если вам нужен готовый код, загрузите сценарий AzureAuthentication.ps1 из материалов книги.

Создание виртуальной машины Azure и всех зависимостей

Настало время настроить виртуальную машину Azure. Это одна из самых популярных служб Azure, поэтому навык их создания может стать большим преимуществом для любого, кто работает в этой среде.

Когда-то давно, когда я впервые подписывался на Azure, чтобы просто попробовать работать с виртуальными машинами, я думал, что все можно настроить с помощью одной команды: просто пишешь `New-AzureVm`, и вуаля! — можно баловаться с новехонькой виртуальной машиной. Я еще никогда так не ошибался.

Я даже понятия не имел, сколько требуется зависимостей, чтобы виртуальная машина заработала. Вы заметили, насколько мал раздел, посвященный зависимостям? Он такой лишь по одной причине — чтобы получить больше опыта работы с PowerShell, мы сами установим все зависимости для создания виртуальной машины с помощью Azure. Мы настроим группу ресурсов, виртуальную сеть, учетную запись хранения, общедоступный IP-адрес, сетевой интерфейс

и образ операционной системы. По сути, мы создадим виртуальную машину с нуля. Давайте приступим!

Создание группы ресурсов

В Azure все является *ресурсом* и должно находиться внутри *группы ресурсов*. Первым делом нужно создать группу ресурсов. Для этого мы воспользуемся командой `New-AzResourceGroup`. Этой команде мы передаем имя группы ресурсов и географический регион, в котором она будет создана. В данном примере мы создадим группу ресурсов под именем `PowerShellForSysAdmins-RG` и разместим ее в регионе East US (как показано в листинге 12.5). Все доступные регионы можно вывести с помощью команды `Get-AzLocation`.

Листинг 12.5. Создание группы ресурсов Azure

```
PS> New-AzResourceGroup -Name 'PowerShellForSysAdmins-RG' -Location 'East US'
```

После создания группы ресурсов нужно создать сетевой стек, который будет использовать ваша виртуальная машина.

Создание сетевого стека

Чтобы ваша виртуальная машина могла соединяться с внешним миром и с другими ресурсами Azure, ей необходим сетевой стек: подсеть, виртуальная сеть, общедоступный IP-адрес (необязательно) и виртуальный сетевой адаптер (vNIC), который будет использовать виртуальная машина.

Подсеть

Сперва создадим подсеть. *Подсеть* — это логическая сеть IP-адресов, которые могут обмениваться данными друг с другом без использования маршрутизатора. Подсеть является частью в виртуальной сети. Подсети делят виртуальную сеть на более мелкие.

Чтобы создать конфигурацию подсети, мы используем команду `New-AzVirtualNetworkSubnetConfig` (листинг 12.6). Для этой команды требуется имя и префикс IP-адреса или идентификатор сети.

Листинг 12.6. Создание конфигурации подсети виртуальной сети

```
PS> $newSubnetParams = @{
    'Name' = 'PowerShellForSysAdmins-Subnet'
    'AddressPrefix' = '10.0.1.0/24'
}
PS> $subnet = New-AzVirtualNetworkSubnetConfig @newSubnetParams
```

Зададим подсети имя `PowerShellForSysAdmins-Subnet` и добавим префикс 10.0.1.0/24.

Виртуальная сеть

Теперь, когда вы создали конфигурацию подсети, ее можно использовать для создания виртуальной сети. *Виртуальная сеть* — это ресурс Azure, который позволяет отделять различные ресурсы вроде виртуальных машин от всех прочих. Виртуальную сеть можно рассматривать в том же контексте, что и логическую, которую вы можете реализовать локально на сетевом маршрутизаторе.

Чтобы создать виртуальную сеть, используйте команду `New-AzVirtualNetwork`, как показано в листинге 12.7.

Листинг 12.7. Создание виртуальной сети

```
PS> $newVNetParams = @{
 ❶ 'Name' = 'PowerShellForSysAdmins-vNet'
 ❷ 'ResourceGroupName' = 'PowerShellForSysAdmins-RG'
 ❸ 'Location' = 'East US'
 ❹ 'AddressPrefix' = '10.0.0.0/16'
}
PS> $vNet = New-AzVirtualNetwork @newVNetParams -Subnet $subnet
```

Обратите внимание, что для создания виртуальной сети нужно указать имя сети ❶, группу ресурсов ❷, регион (местоположение) ❸ и адрес частной сети верхнего уровня, частью которой будет ваша подсеть ❹.

Общедоступный IP-адрес

После настройки виртуальной сети вам нужен общедоступный IP-адрес, чтобы вы могли подключить свою виртуальную машину к интернету, а клиенты могли к ней подключаться. Обратите внимание, что технически этот шаг не обязателен, если вы планируете подключать вашу виртуальную машину только к другим ресурсам Azure. Но поскольку ваши планы простираются гораздо дальше, мы сделаем это.

Опять же, вы можете создать общедоступный IP-адрес с помощью одной команды: `New-AzPublicIpAddress`. Вы уже видели множество параметров для этой функции, но стоит обратить внимание на новый параметр `AllocationMethod`. Он задает ресурсу Azure динамический или статический IP-адрес. Как показано в листинге 12.8, мы задали динамический IP-адрес, чтобы не выполнять лишних задач. Нам не нужны постоянно одинаковые IP-адреса, поэтому мы используем динамический.

Листинг 12.8. Создание общедоступного IP-адреса

```
PS> $newPublicIpParams = @{
    'Name' = 'PowerShellForSysAdmins-PubIp'
    'ResourceGroupName' = 'PowerShellForSysAdmins-RG'
    'AllocationMethod' = 'Dynamic' ## Dynamic or Static
    'Location' = 'East US'
}
PS> $publicIp = New-AzPublicIpAddress @newPublicIpParams
```

Наш общедоступный IP-адрес существует, но еще ни с чем не связан. Вам нужно *привязать* его к виртуальному сетевому адаптеру vNIC.

Виртуальный сетевой адаптер (vNIC)

Чтобы создать vNIC, вам нужно выполнить еще одну однострочную команду `New-AzNetworkInterface`. Вам также потребуется ID подсети и ID общедоступного IP-адреса, который вы создали ранее. И подсеть, и общедоступный IP-адрес хранятся как объекты со свойством ID, и вам просто нужно получить доступ к этому свойству, как показано в листинге 12.9.

Листинг 12.9. Создание vNIC в Azure

```
PS> $newVNicParams = @{
    'Name' = 'PowerShellForSysAdmins-vNIC'
    'ResourceGroupName' = 'PowerShellForSysAdmins-RG'
    'Location' = 'East US'
    'SubnetId' = $vNet.Subnets[0].Id
    'PublicIpAddressId' = $publicIp.Id
}
PS> $vNic = New-AzNetworkInterface @newVNicParams
```

Сетевой стек готов! Следующим шагом будет создание учетной записи хранения.

Создание учетной записи хранения

Вашу виртуальную машину нужно где-то хранить. Это «где-то» — *учетная запись хранения*. Создать базовую учетную запись хранения просто — достаточно ввести команду `New-AzStorageAccount`. Как и в случае с предыдущими командами, нужно будет задать имя, группу ресурсов и местоположение. Кроме того, к ним добавился параметр `Type`, который указывает уровень избыточности в учетной записи хранения. Используйте наименее затратный тип учетной записи хранения (*локально избыточный*), задав значение `Standard_LRS`, как показано в листинге 12.10.

Листинг 12.10. Создание учетной записи хранения Azure

```
PS> $newStorageAcctParams = @{
    'Name' = 'powershellforsysadmins'
    'ResourceGroupName' = 'PowerShellForSysAdmins-RG'
    'Type' = 'Standard_LRS'
    'Location' = 'East US'
}
PS> $storageAccount = New-AzStorageAccount @newStorageAcctParams
```

Теперь, когда вам есть где разместить виртуальную машину, пришло время настроить образ операционной системы.

Создание образа операционной системы

Образ операционной системы — это основа виртуального диска для вашей виртуальной машины. Вместо того чтобы устанавливать Windows на виртуальную машину, мы возьмем существующий образ операционной системы и просто включим ее.

Образ операционной системы создается в два этапа: сначала определяются параметры конфигурации ОС, а затем — оффер, или образ ОС для использования. В Azure термин *оффер* означает образ виртуальной машины.

Для настройки параметров конфигурации нужно создать объект конфигурации виртуальной машины. Этот объект определяет имя и размер создаваемой виртуальной машины. Для этого введем команду `New-AzVMConfig`. В листинге 12.11 мы создаем ВМ типа `Standard_A3` (список всех доступных размеров выводится командой `Get-AzVMSize` с указанием региона).

Листинг 12.11. Создание конфигурации виртуальной машины

```
PS> $newConfigParams = @{
    'VMName' = 'PowerShellForSysAdmins-VM'
    'VMSize' = 'Standard_A3'
}
PS> $vmConfig = New-AzVMConfig @newConfigParams
```

После создания конфигурации вы можете передать объект в качестве параметра виртуальной машины для команды `Set-AzVMOperatingSystem`. Эта команда позволяет определять специфичные для операционной системы атрибуты, например имя хоста виртуальной машины, а также включать Центр обновления Windows и другие атрибуты. Мы не будем вдаваться в подробности, но если вы хотите изучить все возможности, то посмотрите справку по команде `Set-AzVMOperatingSystem` с помощью `Get-Help`.

В листинге 12.12 мы создаем объект операционной системы Windows, который будет иметь имя хоста `Automate-VM` (заметьте, что имя хоста должно содержать менее 16 символов). Воспользуйтесь именем пользователя и паролем, которые выдаст команда `Get-Credential`, для создания нового административного пользователя, а также параметром `EnableAutoUpdate` для автоматического применения любых новых обновлений Windows.

Листинг 12.12. Создание образа операционной системы

```
PS> $newVmOsParams = @{
    'Windows' = $true
    'ComputerName' = 'Automate-VM'
    'Credential' = (Get-Credential -Message 'Type the name and password of the
    local administrator account.')
    'EnableAutoUpdate' = $true
    'VM' = $vmConfig
}
PS> $vm = Set-AzVMOperatingSystem @newVmOsParams
```

Теперь вам нужно создать оффер ВМ. Его смысл заключается в том, что Azure позволяет вам выбирать операционную систему на диске виртуальной машины. В этом примере используется образ Windows Server 2012 R2 Datacenter. Этот образ предоставляется Microsoft в готовом виде, поэтому нет необходимости создавать свой собственный.

После этого вы сможете создать образ с помощью команды `Set-AzVMSourceImage`, как показано в листинге 12.13.

Листинг 12.13. Поиск и создание исходного образа виртуальной машины

```
PS> $offer = Get-AzVMImageOffer -Location 'East US'❶ –PublisherName
'MicrosoftWindowsServer'❷ | Where-Object { $_.Offer -eq 'WindowsServer' }❸
PS> $newSourceImageParams = @{
    'PublisherName' = 'MicrosoftWindowsServer'
    'Version' = 'latest'
    'Skus' = '2012-R2-Datacenter'
    'VM' = $vm
    'Offer' = $offer.Offer
}
PS> $vm = Set-AzVMSourceImage @newSourceImageParams
```

В этом листинге мы запрашиваем все предложения в регионе East US ❶ с именем издателя `MicrosoftWindowsServer` ❷. Вы можете использовать команду `Get-AzVMImagePublisher` для вывода списка издателей. Далее мы ограничиваем оффер именем `WindowsServer` ❸. Выделив исходный образ, можно назначить его объекту виртуальной машины. На этом настройка машины завершена.

Чтобы назначить образ объекту виртуальной машины, вам понадобится URI только что созданного диска ОС. Его необходимо передать вместе с объектом виртуальной машины в команду `Set-AzVMOSDisk` (листинг 12.14).

Листинг 12.14. Назначение диска операционной системы виртуальной машине

```
PS> $osDiskName = 'PowerShellForSysAdmins-Disk'
PS> $osDiskUri = '{0}vhds/PowerShellForSysAdmins-VM{1}.vhd' -f $storageAccount
                 .PrimaryEndpoints.Blob.ToString(), $osDiskName
PS> $vm = Set-AzVMOSDisk -Name OSDisk -CreateOption 'fromImage' -VM $vm
-VhdUri $osDiskUri
```

Теперь у вас есть диск с ОС, и он назначен объекту виртуальной машины. Дело движется к завершению!

Закругляемся

Вы *почти* закончили. Осталось лишь подключить созданный вами виртуальный сетевой адаптер и создать настоящую виртуальную машину.

Чтобы подключить vNIC к виртуальной машине, введем команду `Add-AzVmNetworkInterface` и передадим ей ваш объект виртуальной машины вместе с идентификатором vNIC — все это приведено в листинге 12.15.

Листинг 12.15. Подключение vNIC к виртуальной машине

```
PS> $vm = Add-AzVMNetworkInterface -VM $vm -Id $vNic.Id
```

Наконец, мы можем создать виртуальную машину, как показано в листинге 12.16. Вызовите команду `New-AzVm` с объектом `VM`, группой ресурсов и регионом, и вы получите новенькую виртуальную машину прямо с конвейера! Обратите внимание, что с этого момента вам придется платить за услугу.

Листинг 12.16. Создание виртуальной машины Azure

```
PS> New-AzVM -VM $vm -ResourceGroupName 'PowerShellForSysAdmins-RG' -Location
                                        'East US'

RequestId IsSuccessStatusCode StatusCode ReasonPhrase
--------- ------------------- ---------- ------------
                         True         OK OK
```

Ваша новая виртуальная машина в Azure называется `Automate-VM`. Запустите команду `Get-AzVm`, чтобы убедиться, что она действительно появилась. Взгляните на результат в листинге 12.17.

Листинг 12.17. Проверяем существование виртуальной машины Azure

```
PS> Get-AzVm -ResourceGroupName 'PowerShellForSysAdmins-RG' -Name
PowerShellForSysAdmins-VM


ResourceGroupName  : PowerShellForSysAdmins-RG
Id                 : /subscriptions/XXXXXXXXXXXX/resourceGroups
                     /PowerShellForSysAdmins-RG/providers/Microsoft.Compute/
                     virtualMachines/PowerShellForSysAdmins-VM
VmId               : e459fb9e-e3b2-4371-9bdd-42ecc209bc01
Name               : PowerShellForSysAdmins-VM
Type               : Microsoft.Compute/virtualMachines
Location           : eastus
Tags               : {}
DiagnosticsProfile : {BootDiagnostics}
Extensions         : {BGInfo}
HardwareProfile    : {VmSize}
NetworkProfile     : {NetworkInterfaces}
OSProfile          : {ComputerName, AdminUsername, WindowsConfiguration, Secrets}
ProvisioningState  : Succeeded
StorageProfile     : {ImageReference, OsDisk, DataDisks}
```

Если вы видите аналогичный результат, значит, вы успешно создали виртуальную машину Azure!

Автоматизация создания ВМ

Фух! Мы проделали очень много работы, чтобы запустить одну виртуальную машину со всеми необходимыми зависимостями — не хотелось бы повторять все это, если захочется сделать еще одну виртуальную машину. Почему бы нам не создать функцию, которая могла бы сделать все это за нас? В такой функции мы могли бы собрать весь код в виде единого фрагмента, чтобы в будущем использовать его снова и снова.

Если вы считаете себя амбициозным человеком, откройте функцию PowerShell под названием `New-CustomAzVm` в материалах к этой главе. Это отличный пример того, как можно объединить все задачи из этого раздела в единую связную функцию с минимальным объемом входных данных.

Развертывание веб-приложения на Azure

Если вы работаете с Azure, вы должны знать, как развернуть *веб-приложения Azure*. Они позволяют подготавливать к работе веб-сайты и различные веб-службы, работающие на серверах вроде IIS, Apache и других, не беспокоясь

о создании самого веб-сервера. Когда вы узнаете, как развернуть веб-приложение Azure с помощью PowerShell, вы сможете включить этот процесс в более крупные рабочие схемы, в том числе конвейеры разработки, подготовку тестовой среды или лаборатории и многое другое.

Развертывание веб-приложения Azure состоит из двух этапов: сначала мы создаем план службы приложения, а затем само веб-приложение. Веб-приложения Azure являются частью *Azure App Services*, поэтому любой ресурс под этой эгидой должен иметь связанный план службы приложений. Они сообщают веб-приложению тип базовых вычислительных ресурсов, на которых создается программа.

Создание плана службы приложений и веб-приложения

Создать план службы Azure достаточно просто. Как и прежде, вам потребуется всего одна команда. Этой команде требуется имя плана службы приложения, регион исполнения и группа ресурсов. Также может быть включен и необязательный уровень, который определяет тип производительности в рамках веб-приложения.

Как и в предыдущем разделе, вы создаете группу ресурсов с помощью команды `New-AzResource Group -Name 'PowerShellForSysAdmins-App' -Location 'East US'`. После этого вы создаете план службы приложения, а затем помещаете его в эту группу ресурсов.

Ваше веб-приложение под названием `Automate` будет находиться в регионе East US на уровне бесплатных приложений. Весь код для выполнения этих задач приведен в листинге 12.18.

Листинг 12.18. Создание плана службы приложений Azure

```
PS> New-AzAppServicePlan -Name 'Automate' -Location 'East US'
-ResourceGroupName 'PowerShellForSysAdmins-App' -Tier 'Free'
```

После выполнения этой команды у вас будет создан план обслуживания приложений, и вы сможете перейти к созданию самого веб-приложения.

Возможно, вас не удивит тот факт, что создание веб-приложения Azure с помощью PowerShell также выполняется с помощью всего одной команды. Просто запустите `New-AzWebApp` и передайте ей имя и местоположение группы ресурсов, а также план службы приложения, над которым будет располагаться это веб-приложение.

В листинге 12.19 команда `New-AzWebApp` используется для создания веб-приложения `MyApp` внутри группы ресурсов `PowerShellForSysAdmins-App` с помощью плана службы приложений `Automate` (который вы создали ранее). Обратите внимание, что запуск приложения может стоить денег.

Листинг 12.19. Создание веб-приложения Azure

```
PS> New-AzWebApp -ResourceGroupName 'PowerShellForSysAdmins-App' -Name
'AutomateApp' -Location 'East US' -AppServicePlan 'Automate'
```

После запуска команды в выводе окажется множество свойств — это настройки веб-приложения.

Развертывание базы данных SQL Azure

Еще одна распространенная задача Azure — развертывание базы данных Azure SQL. Для этих целей вам нужно сделать три вещи: создать сервер SQL Azure, на котором будет работать база данных, создать саму базу, а затем добавить правило брандмауэра SQL Server для подключения к базе данных.

Как и в предыдущих разделах, вы создаете группу ресурсов для размещения всех ваших новых ресурсов. Запустите команду `New-AzResourceGroup -Name 'PowerShellForSysAdmins-SQL' –Location 'East US'`, а затем создайте SQL-сервер, на котором будет работать база данных.

Создание Azure SQL Server

Для создания сервера SQL Azure требуется еще одна однострочная команда: `New-AzSqlServer`. Вам снова нужно указать имя ресурса, имя сервера и регион, а еще вам понадобятся имя пользователя и пароль администратора SQL на сервере. Работы немного прибавилось. Поскольку вам нужно создать учетные данные для передачи в `New-AzSqlServer`, давайте с этого и начнем. Мы уже знаем, как создать объект `PSCredential`, поэтому не будем вдаваться в подробности.

```
PS> $userName = 'sqladmin'
PS> $plainTextPassword = 's3cretp@SSw0rd!'
PS> $secPassword = ConvertTo-SecureString -String $plainTextPassword -AsPlainText
-Force
PS> $credential = New-Object -TypeName System.Management.Automation.PSCredential
-ArgumentList
$userName,$secPassword
```

Когда у вас появятся учетные данные, останется лишь поместить все параметры в хеш-таблицу и передать их в функцию `New-AzSqlServer`, как показано в листинге 12.20.

Листинг 12.20. Создание сервера SQL Azure

```
PS> $parameters = @{
    ResourceGroupName = 'PowerShellForSysAdmins-SQL'
    ServerName = 'PowerShellForSysAdmins-SQLSrv'
    Location = 'East US'
    SqlAdministratorCredentials = $credential
}
PS> New-AzSqlServer @parameters

ResourceGroupName        : PowerShellForSysAdmins-SQL
ServerName               : powershellsysadmins-sqlsrv
Location                 : eastus
SqlAdministratorLogin    : sqladmin
SqlAdministratorPassword :
ServerVersion            : 12.0
Tags                     :
Identity                 :
FullyQualifiedDomainName : powershellsysadmins-sqlsrv.database.windows.net
ResourceId               : /subscriptions/XXXXXXXXXXXX/resourceGroups
                           /PowerShellForSysAdmins-SQL/providers/Microsoft.Sql
                           /servers/powershellsysadmins-sqlsrv
```

Теперь, когда сервер SQL создан, у вас есть фундамент для базы данных.

Создание базы данных SQL Azure

Чтобы создать базу данных SQL, используйте команду `New-AzSqlDatabase`, как показано в листинге 12.21. Вместе с общим параметром `ResourceGroupName` передайте имя только что созданного сервера и имя базы данных, которую вы хотите создать (в нашем примере это `AutomateSQLDb`).

Листинг 12.21. Создание базы данных Azure SQL

```
PS> New-AzSqlDatabase -ResourceGroupName 'PowerShellForSysAdmins-SQL'
-ServerName 'PowerShellSysAdmins-SQLSrv' -DatabaseName 'AutomateSQLDb'

ResourceGroupName             : PowerShellForSysAdmins-SQL
ServerName                    : PowerShellSysAdmins-SQLSrv
DatabaseName                  : AutomateSQLDb
Location                      : eastus
DatabaseId                    : 79f3b331-7200-499f-9fba-b09e8c424354
Edition                       : Standard
CollationName                 : SQL_Latin1_General_CP1_CI_AS
CatalogCollation              :
```

```
MaxSizeBytes                    : 268435456000
Status                          : Online
CreationDate                    : 9/15/2019 6:48:32 PM
CurrentServiceObjectiveId       : 00000000-0000-0000-0000-000000000000
CurrentServiceObjectiveName     : S0
RequestedServiceObjectiveName : S0
RequestedServiceObjectiveId     :
ElasticPoolName                 :
EarliestRestoreDate             : 9/15/2019 7:18:32 PM
Tags                            :
ResourceId                      : /subscriptions/XXXXXXX/resourceGroups
                                  /PowerShellForSysAdmins-SQL/providers
                                  /Microsoft.Sql/servers/powershellsysadmin-sqlsrv
                                  /databases/AutomateSQLDb
CreateMode                      :
ReadScale                       : Disabled
ZoneRedundant                   : False
Capacity                        : 10
Family                          :
SkuName                         : Standard
LicenseType                     :
```

Теперь у вас есть работающая база данных SQL в Azure. Но если вы попробуете подключиться к ней, у вас ничего не получится. По умолчанию новая база данных SQL Azure защищена от любых внешних подключений. Вам нужно создать правило брандмауэра, чтобы он разрешил соединение с вашей базой данных.

Создание правила брандмауэра SQL Server

Команда для создания правила брандмауэра — `New-AzSqlServerFirewallRule`. Она принимает имена группы ресурсов, созданного ранее сервера, правила брандмауэра, а также начальный и конечный IP-адреса. Последние позволяют указать один IP-адрес или диапазон IP-адресов, которым можно обращаться к вашей БД. Поскольку вы планируете работать только со своего компьютера, ограничим подключения к вашему серверу SQL вашим текущим компьютером. Для этого вам сначала нужно узнать свой IP-адрес. Вы можете легко сделать это с помощью команды PowerShell: `Invoke-RestMethod http://ipinfo.io/json | Select -ExpandProperty ip`. Затем вы можете использовать ваш общедоступный IP-адрес для параметров `StartIPAddress` и `EndIPAddress`. Учтите, что если ваш публичный IP-адрес изменится, вам придется снова выдавать себе новое разрешение.

Также имейте в виду, что имя сервера в листинге 12.22 должно состоять только из строчных букв, дефисов или цифр. Иначе при попытке создать правило брандмауэра вы получите сообщение об ошибке.

Листинг 12.22. Создание правила брандмауэра Azure SQL Server

```
PS> $parameters = @{
    ResourceGroupName = 'PowerShellForSysAdmins-SQL'
    FirewallRuleName = 'PowerShellForSysAdmins-FwRule'
    ServerName = 'powershellsysadmin-sqlsrv'
    StartIpAddress = 'Your Public IP Address'
    EndIpAddress = 'Your Public IP Address'
}
PS> New-AzSqlServerFirewallRule @parameters

ResourceGroupName : PowerShellForSysAdmins-SQL
ServerName        : powershellsys-sqlsrv
StartIpAddress    : 0.0.0.0
EndIpAddress      : 0.0.0.0
FirewallRuleName  : PowerShellForSysAdmins-FwRule
```

Готово! Ваша база данных теперь работает.

Тестирование вашей базы данных SQL

Чтобы протестировать нашу базу данных, давайте создадим небольшую функцию, которая будет использовать метод `Open()` объекта `System.Data.SqlClient.SqlConnection` для установки соединения (см. листинг 12.23).

Листинг 12.23. Тестирование SQL-соединения с базой данных SQL Azure

```
function Test-SqlConnection {
     param(
         [Parameter(Mandatory)]
      ❶  [string]$ServerName,

         [Parameter(Mandatory)]
         [string]$DatabaseName,

         [Parameter(Mandatory)]
      ❷  [pscredential]$Credential
     )

     try {
         $userName = $Credential.UserName
      ❸  $password = $Credential.GetNetworkCredential().Password
      ❹  $connectionString = 'Data Source={0};database={1};User
         ID={2};Password={3}' -f $ServerName,$DatabaseName,$userName,$password
         $sqlConnection = New-Object System.Data.SqlClient.SqlConnection
         $ConnectionString
      ❺  $sqlConnection.Open()
         $true
    } catch {
       if ($_.Exception.Message -match 'cannot open server') {
```

```
            $false
        } else {
            throw $_
        }
    } finally {
     ❻ $sqlConnection.Close()
    }
}
```

Мы используем полное доменное имя созданного ранее SQL-сервера в качестве параметра `ServerName` для этой функции ❶ вместе с именем пользователя и паролем администратора SQL внутри объекта `PSCredential` ❷.

Затем мы разбиваем объект `PSCredential` на имя пользователя и пароль ❸ в формате открытого текста. Создаем строку подключения для соединения с базой данных ❹, вызываем метод `Open()` объекта `SqlConnection`, чтобы попытаться подключиться к базе данных ❺, и, наконец, закрываем соединение с базой данных ❻.

То же самое можно сделать с помощью команды **`Test-SqlConnection -ServerName 'powershellsysadmins-sqlsrv.database.windows.net' -DatabaseName 'AutomateSQLDb' -Credential (Get-Credential)`**. Если удастся подключиться к базе данных, функция вернет `True`; в противном случае вернется `False` (а вам потребуется узнать, почему это произошло).

Чтобы все очистить, достаточно удалить группу ресурсов с помощью команды `Remove-AzResourceGroup -ResourceGroupName 'PowerShellForSysAdmins-SQL'`.

Итоги

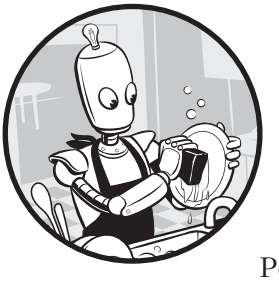
В этой главе вы с головой окунулись в автоматизацию Microsoft Azure с помощью PowerShell. Вы настроили неинтерактивную авторизацию и развернули виртуальную машину, веб-приложение и базу данных SQL. Заметьте: все это было сделано из PowerShell, ведь мы даже не заходили на портал Azure.

Мы не смогли бы сделать это без PowerShell-модуля Az и усердной работы людей, которые его создали. Как и другие облачные модули PowerShell, все эти команды полагаются на различные API, которые вызываются изнутри. Благодаря модулю вам не нужно думать о том, как вызывать методы REST или использовать URL-адреса конечных точек.

В следующей главе вы познакомитесь с использованием PowerShell для автоматизации Amazon Web Services.

13

Работа с Amazon Web Services



В предыдущей главе мы поговорили о работе с Microsoft Azure через PowerShell. Теперь давайте посмотрим, что можно сделать с Amazon Web Services (AWS). В этой главе мы погрузимся в использование PowerShell для работы с AWS. Сперва мы научимся авторизации в AWS с помощью PowerShell, а затем узнаем, как создать экземпляр EC2 с нуля, развернуть приложение Elastic Beanstalk (EBS) и создать базу данных Microsoft SQL Server Amazon Relational Database Service (Amazon RDS).

Как и Azure, сервис AWS имеет огромную силу в мире облачных технологий. Скорее всего, если вы работаете в сфере ИТ, вы так или иначе в какой-то момент столкнетесь с AWS. Как и в случае с Azure, для работы с AWS существует удобный PowerShell-модуль: `AWSPowerShell`.

Модуль `AWSPowerShell` можно установить из PowerShell Gallery аналогично модулю `AzureRm`, вызвав команду `Install-Module AWSPowerShell`.

Когда модуль будет загружен и установлен, мы приступим к работе.

Исходные требования

Предполагаю, что у вас уже есть учетная запись AWS и доступ к учетной записи root. Вы можете зарегистрироваться и получить бесплатную учетную запись

AWS на странице aws.amazon.com/free/. Вам не нужно делать все из-под root, но это понадобится для создания вашего первого пользователя *управления идентификацией и доступом (IAM)*. Вам также нужно будет загрузить и установить модуль `AWSPowerShell`.

Авторизация в AWS

Авторизация в AWS выполняется с помощью службы IAM, которая специализируется на авторизации учетных записей и аудита в AWS. Вам понадобится пользователь IAM с доступом к соответствующим ресурсам. Первым делом необходимо его создать.

При создании учетной записи AWS автоматически создается пользователь root, из-под которого можно создать своего пользователя IAM. *Технически* вы можете задействовать пользователя root для чего угодно в AWS, но я не рекомендую так делать.

Авторизация с пользователя root

Давайте создадим пользователя IAM, с которого мы будем работать в этой главе. Сначала нужно будет как-то его авторизовать. При отсутствии других пользователей IAM это можно сделать только из-под пользователя root. К сожалению, это также означает, что вам придется на время отказаться от PowerShell. Вам нужно будет использовать графический интерфейс консоли управления AWS, чтобы получить и доступ, и секретные ключи пользователя root.

Первым делом нужно войти в свою учетную запись AWS. Перейдите в правый угол экрана и щелкните по раскрывающемуся меню учетной записи, показанному на рис. 13.1.

Щелкните по параметру **My Security Credentials**: при этом появится окошко, предупреждающее о том, что использование ваших учетных данных — не лучшая идея (рис. 13.2). Но сейчас это необходимо для создания пользователя IAM.

Выберите пункт **Continue to Security Credentials**, а затем нажмите кнопку **Access Keys**. Нажав **Create New Access Key**, вы сможете просмотреть идентификатор и секретный ключ доступа к вашей учетной записи. Это позволит вам загрузить файл ключа, в котором есть и то и другое. Если вы еще этого не сделали, загрузите файл и спрячьте его в безопасном месте. Теперь вам нужно

скопировать с этой страницы ключ доступа и секретный ключ и добавить их в свой профиль по умолчанию в сеансе PowerShell.

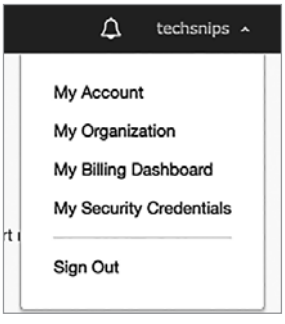


Рис. 13.1. Опция My Security Credentials

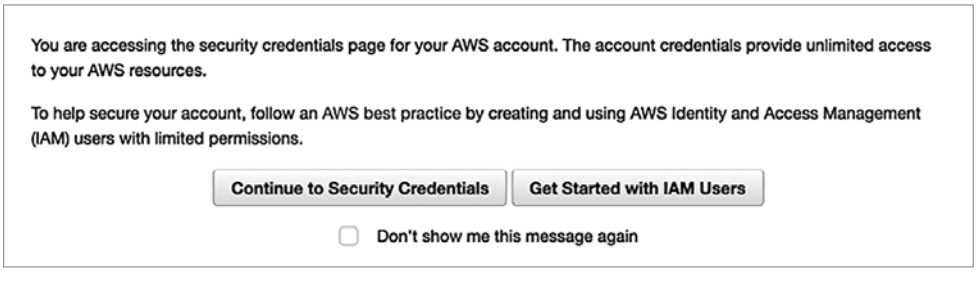


Рис. 13.2. Предупреждение аутентификации

Передайте оба этих ключа команде `Set-AWSCredential` — она сохранит их, и команды в дальнейшем смогут использовать эти данные для создания пользователя IAM (листинг 13.1).

Листинг 13.1. Установка ключей доступа к AWS

```
PS> Set-AWSCredential -AccessKey 'access key' -SecretKey 'secret key'
```

Сделав все это, вы готовы создать пользователя IAM.

Создание пользователя и роли IAM

Теперь, когда вы авторизованы как root-пользователь, можно создать пользователя IAM. В команде `New-IAMUser` укажите имя пользователя IAM, которое вы хотите использовать (в нашем примере — `Automator`). Результат показан в листинге 13.2.

Листинг 13.2. Создание пользователя IAM

```
PS> New-IAMUser -UserName Automator

Arn                 : arn:aws:iam::013223035658:user/Automator
CreateDate          : 9/16/2019 5:01:24 PM
PasswordLastUsed    : 1/1/0001 12:00:00 AM
Path                : /
PermissionsBoundary :
UserId              : AIDAJU2WN5KIFOUMPDSR4
UserName            : Automator
```

Следующим шагом будет предоставление пользователю соответствующего разрешения. Для этого следует назначить пользователю роль, для которой, в свою очередь, будет назначена политика. AWS группирует определенные разрешения в объектах под названием *роли* — они упрощают делегирование разрешений администратором (эта стратегия известна как *управление доступом на основе ролей*, или RBAC — role-based access control). Затем *политика* определяет, к каким разрешениям эта роль имеет доступ.

Вы можете создать роль с помощью команды `New-IAMRole`, но сначала необходимо создать так называемый *документ политики доверительных отношений* — строку текста в формате JSON, определяющую службы, к которым этот пользователь может получить доступ, и уровень, на котором этот доступ дается.

В листинге 13.3 показан пример документа политики доверительных отношений.

Листинг 13.3. Пример документа политики доверительных отношений

```
{
     "Version": "2019-10-17",
     "Statement": [
          {
               "Effect": "Allow",
               "Principal" : { "AWS": "arn:aws:iam::013223035658:user/Automator" },
               "Action": "sts:AssumeRole"
          }
     ]
}
```

Эта строка JSON преобразовывает саму роль (изменяя ее политику доверия), чтобы ваш пользователь `Automator` мог с ней работать. Это дает пользователю разрешение `AssumeRole`, что необходимо для создания роли. Для получения дополнительной информации о создании документа политики доверительных отношений ознакомьтесь с docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_manage_modify.html.

Назначьте эту строку JSON переменной `$json`, а затем передайте ее как значение параметра `AssumeRolePolicyDocument` команды `New-IamRole`, как показано в листинге 13.4.

Листинг 13.4. Создание новой роли IAM

```
PS> $json = '{
>>     "Version": "2019-10-17",
>>      "Statement": [
>>           {
>>                "Effect": "Allow",
>>                "Principal" : { "AWS": "arn:aws:iam::013223035658:user/Automator" },
>>                "Action": "sts:AssumeRole"
>>           }
>>      ]
>> }'
PS> New-IAMRole -AssumeRolePolicyDocument $json -RoleName 'AllAccess'

Path          RoleName                RoleId                   CreateDate
----          --------                ------                   ----------
/             AllAccess               AROAJ2B7YC3HH6M6F2WOM    9/16/2019 6:05:37 PM
```

Теперь, когда роль IAM создана, вам необходимо предоставить ей разрешение на доступ к необходимым ресурсам. Вместо того чтобы тратить сотню-другую страниц на подробное описание ролей и безопасности AWS IAM, давайте сделаем что-нибудь простое и предоставим `Automator` полный доступ ко всему (фактически сделав его могущественным пользователем, как и root).

Учтите, что на практике этого делать *не* следует. Всегда лучше ограничивать доступ тем, что необходимо конкретному пользователю. Подробнее об этом можно почитать в руководстве AWS IAM Best Practices (docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html). Но пока давайте забудем об этом и назначим этому пользователю политику `AdministratorAccess` с помощью команды `Register-IAMUserPolicy`. Необходимо узнать для этой политики имя ресурса Amazon (ARN). Воспользуйтесь командой `Get-IAMPolicies` и добавьте фильтр по имени политики. Затем сохраните это имя в переменной и передайте ее в `Register-IAMUserPolicy` (листинг 13.5).

Листинг 13.5. Прикрепление политики к пользователю

```
PS> $policyArn = (Get-IAMPolicies | where {$_.PolicyName -eq
'AdministratorAccess'}).Arn
PS> Register-IAMUserPolicy -PolicyArn $policyArn -UserName Automator
```

Последнее, что вам нужно сделать, — это сгенерировать ключ доступа, который позволит вам аутентифицировать своего пользователя. Это можно сделать с помощью команды `New-IAMAcessKey`, как показано в листинге 13.6.

Листинг 13.6. Создание ключа доступа IAM

```
PS> $key = New-IAMAccessKey -UserName Automator
PS> $key

AccessKeyId     : XXXXXXXX
CreateDate      : 9/16/2019 6:17:40 PM
SecretAccessKey : XXXXXXXXX
Status          : Active
UserName        : Automator
```

Новый пользователь IAM настроен. Давайте теперь его авторизируем.

Авторизация вашего пользователя IAM

В предыдущем разделе мы авторизовывались под пользователем root, но это была временная мера. Теперь вам нужно авторизовать своего пользователя IAM, чтобы проделать работу по-настоящему! Снова воспользуемся командой `Set-AWSCredential` для добавления в профиль новых ключей доступа и секретных ключей. Однако мы немного изменим эту команду с помощью параметра `StoreAs`, как показано в листинге 13.7. Поскольку мы будем использовать этого пользователя IAM в течение всего оставшегося сеанса, мы сохраним ключ доступа и секретный ключ в профиле AWS, чтобы не приходилось запускать эту команду в каждом сеансе.

Листинг 13.7. Установка ключей доступа к AWS по умолчанию

```
PS> Set-AWSCredential -AccessKey $key.AccessKeyId -SecretKey
$key.SecretAccessKey -StoreAs 'Default'
```

Последняя команда, которую нужно запустить, — `Initialize-AWSDefaultConfiguration -Region 'название региона'`, после чего вам не нужно будет каждый раз указывать регион при вызове команды. С помощью команды `Get-AWSRegion` можно вывести список всех регионов и выбрать ближайший из них.

Готово! Теперь у вас есть сеанс с аутентификацией в AWS, и вы можете перейти к работе с этими сервисами. Чтобы проверить это, запустите команду `Get-AWSCredentials` с параметром `ListProfileDetail` и найдите все сохраненные учетные данные. Если все в порядке, вы увидите профиль по умолчанию:

```
PS> Get-AWSCredentials -ListProfileDetail
ProfileName StoreTypeName         ProfileLocation
----------- -------------         ---------------
Default     NetSDKCredentialsFile
```

Создание экземпляра AWS EC2

В главе 12 мы создали виртуальную машину Azure. Сейчас мы сделаем нечто подобное, но с *экземпляром* AWS EC2. Он научит вас тому же, что и виртуальная машина Azure. Само создание виртуальных машин — чрезвычайно распространенное явление, независимо от того, используете ли вы Azure или AWS. Однако при создании виртуальной машины в AWS подход к подготовке будет несколько отличным от Azure. Лежащие в основе API в этих случаях разные, поэтому и выполняемые вами команды тоже будут различаться, но в целом мы выполняем ту же задачу — создаем виртуальную машину. Ситуация усложняется тем, что у AWS есть своя терминология! Я попробовал расписать всё через те же шаги, которые мы выполняли для создания виртуальной машины в предыдущей главе, но, конечно же, из-за архитектурных и синтаксических различий между Azure и AWS возникнут заметные различия.

К счастью, как и в случае с Azure, у нас есть модуль `AWSPowerShell`, который упрощает процедуру написания сценария. Как и в предыдущей главе, мы сделаем все с нуля: настроим все необходимые зависимости, а затем создадим экземпляр EC2.

Виртуальное частное облако

Первое, что нам потребуется — это сеть. Вы можете использовать существующую сеть или создать собственную. Поскольку эта книга главным образом учит практическим навыкам, мы создадим свою сеть с нуля. В Azure вы сделали это с помощью виртуальной сети, а в AWS вы будете работать с *виртуальными частными облаками* (VPC), которые по своей структуре позволяют виртуальной машине подключаться к остальной части облака. Чтобы воспроизвести те же настройки, что были у нашей виртуальной сети Azure, мы создадим VPC с одной подсетью, настроенной на самом простом уровне. Поскольку у нас существует такой широкий спектр вариантов конфигурации, я решил, что лучше всего как можно точнее отразить конфигурацию сети Azure.

Перед началом нам нужно знать подсеть, которую мы хотим создать. В качестве примера сети возьмем 10.10.0.0/24. Сохраним эту информацию и переменную, затем воспользуемся командой `New-EC2Vpc`, как показано в листинге 13.8.

После создания VPC вам необходимо вручную включить поддержку DNS (в Azure это автоматически сделали за нас). Включение поддержки DNS позволяет перенаправить подключенные к этому VPC серверы на внутренний DNS-сервер Amazon. Точно так же вам нужно вручную указать общедоступное

Листинг 13.8. Создание AWS VPC

```
PS> $network = '10.0.0.0/16'
PS> $vpc = New-EC2Vpc -CidrBlock $network
PS> $vpc

CidrBlock                   : 10.0.0.0/24
CidrBlockAssociationSet     : {vpc-cidr-assoc-03f1edbc052e8c207}
DhcpOptionsId               : dopt-3c9c3047
InstanceTenancy             : default
Ipv6CidrBlockAssociationSet : {}
IsDefault                   : False
State                       : pending
Tags                        : {}
VpcId                       : vpc-03e8c773094d52eb3
```

имя хоста (еще одна вещь, которую в Azure сделали без нашего участия). Для этого вам необходимо включить DNS-имена хостов. Сделаем и то и другое, используя код из листинга 13.9.

Листинг 13.9. Включение поддержки VPC DNS и имен хостов

```
PS> Edit-EC2VpcAttribute -VpcId $vpc.VpcId -EnableDnsSupport $true
PS> Edit-EC2VpcAttribute -VpcId $vpc.VpcId -EnableDnsHostnames $true
```

Обратите внимание, что в обоих случаях мы используем команду `Edit-EC2VpcAttribute`. Как следует из названия, эта команда позволяет вам редактировать несколько атрибутов вашего EC2 VPC.

Интернет-шлюз

Следующим шагом будет создание интернет-шлюза. Это позволяет вашему экземпляру EC2 направлять трафик в интернет и обратно. Опять же, это придется сделать вручную, используя команду `New-EC2InternetGateay` (листинг 13.10).

Листинг 13.10. Создание интернет-шлюза

```
PS> $gw = New-EC2InternetGateway
PS> $gw

Attachments InternetGatewayId     Tags
----------- -----------------     ----
{}          igw-05ca5aaa3459119b1 {}
```

После создания шлюза следует подключить его к VPC с помощью команды `Add-EC2InternetGateway`, как показано в листинге 13.11.

Листинг 13.11. Присоединение VPC к интернет-шлюзу

```
PS> Add-EC2InternetGateway -InternetGatewayId $gw.InternetGatewayId
    -VpcId $vpc.VpcId
```

С VPC разобрались. Теперь давайте сделаем следующий шаг и добавим маршрут к вашей сети.

Маршруты

После создания шлюза нужно создать таблицу маршрутов и сам маршрут, чтобы экземпляры EC2 на вашем VPC могли выходить в интернет. *Маршрут* — это путь, по которому сетевой трафик попадает в пункт назначения. В свою очередь, *таблица маршрутов* — это, собственно, таблица этих маршрутов. Ваш маршрут должен быть занесен в таблицу, поэтому давайте сначала создадим ее. Используйте команду `New-EC2RouteTable`, чтобы передать свой идентификатор VPC (листинг 13.12).

Листинг 13.12. Создание таблицы маршрутов

```
PS> $rt = New-EC2RouteTable -VpcId $vpc.VpcId
PS> $rt

Associations    : {}
PropagatingVgws : {}
Routes          : {}
RouteTableId    : rtb-09786c17af32005d8
Tags            : {}
VpcId           : vpc-03e8c773094d52eb3
```

Внутри таблицы маршрутов мы создаем маршрут, который указывает на только что созданный шлюз. Мы создаем *маршрут по умолчанию* или *шлюз по умолчанию*. Это значит, что маршрут будет использовать исходящий сетевой трафик, если не определен более конкретный маршрут. Весь трафик (0.0.0.0/0) направляется через интернет-шлюз.

Воспользуйтесь командой `New-EC2Route`, которая в случае успеха вернет `True`, как показано в листинге 13.13.

Листинг 13.13. Создание маршрута

```
PS> New-EC2Route -RouteTableId $rt.RouteTableId -GatewayId
$gw.InternetGatewayId -DestinationCidrBlock '0.0.0.0/0'

True
```

Как мы видим, маршрут успешно создан!

Подсеть

Теперь нам нужно создать подсеть внутри нашего более крупного VPC и связать ее с таблицей маршрутизации. Помните, что подсеть определяет логическую сеть, частью которой будет сетевой адаптер вашего экземпляра EC2. Чтобы создать ее, применим команду `New-EC2Subnet`, а затем — команду `Register-EC2RouteTable`, чтобы зарегистрировать подсеть в таблице маршрутизации, которую вы создали ранее. Однако сперва нужно определить *зону доступности* для этой подсети (в этой зоне центры обработки данных AWS будут размещать вашу подсеть). Если вы не знаете, какая вам нужна зона доступности, воспользуйтесь командой `Get-EC2AvailabilityZone`, чтобы посмотреть все доступные варианты. В листинге 13.14 показано, что при этом должно произойти.

Листинг 13.14. Список зон доступности EC2

```
PS> Get-EC2AvailabilityZone

Messages RegionName State     ZoneName
-------- ---------- -----     --------
{}       us-east-1 available us-east-1a
{}       us-east-1 available us-east-1b
{}       us-east-1 available us-east-1c
{}       us-east-1 available us-east-1d
{}       us-east-1 available us-east-1e
{}       us-east-1 available us-east-1f
```

Если вам все равно, давайте воспользуемся зоной доступности `us-east-1d`. В листинге 13.15 показан код для создания подсети с помощью команды `New-EC2Subnet`, которая принимает созданный ранее идентификатор VPC, блок CIDR (подсеть), а также найденную вами зону доступности и код для регистрации таблицы (с помощью команды `Register-EC2RouteTable`).

Листинг 13.15. Создание и регистрация подсети

```
PS> $sn = New-EC2Subnet -VpcId $vpc.VpcId -CidrBlock '10.0.1.0/24'
-AvailabilityZone 'us-east-1d'
PS> Register-EC2RouteTable -RouteTableId $rt.RouteTableId -SubnetId $sn.SubnetId
rtbassoc-06a8b5154bc8f2d98
```

Теперь, когда подсеть создана и зарегистрирована, у вас есть готовый сетевой стек!

Назначение образа AMI экземпляру EC2

После создания сетевого стека вам необходимо назначить образ машины Amazon (AMI — Amazon Machine Image) вашей виртуальной машине. *AMI* — это «снимок»

диска, который используется в качестве шаблона. Он позволяет избежать установки операционной системы на экземпляры EC2 с нуля. Вам нужно найти существующий AMI, который соответствует вашим потребностям. Наш AMI должен поддерживать экземпляр Windows Server 2016, поэтому сначала следует найти его имя. Просмотрите все доступные экземпляры с помощью команды `Get-EC2ImageByName`, после чего найдите образ под именем `WINDOWS_2016_BASE`.

Теперь, когда имя образа известно, снова выполните команду `Get-EC2ImageByName` и на этот раз укажите тот образ, который вы хотите использовать. В результате команда вернет нужный объект образа, как показано в листинге 13.16.

Листинг 13.16. Находим AMI

```
PS> $ami = Get-EC2ImageByName -Name 'WINDOWS_2016_BASE'
PS> $ami

Architecture        : x86_64
BlockDeviceMappings : {/dev/sda1, xvdca, xvdcb, xvdcc...}
CreationDate        : 2019-08-15T02:27:20.000Z
Description         : Microsoft Windows Server 2016...
EnaSupport          : True
Hypervisor          : xen
ImageId             : ami-0b7b74ba8473ec232
ImageLocation       : amazon/Windows_Server-2016-English-Full-Base-2019.08.15
ImageOwnerAlias     : amazon
ImageType           : machine
KernelId            :
Name                : Windows_Server-2016-English-Full-Base-2019.08.15
OwnerId             : 801119661308
Platform            : Windows
ProductCodes        : {}
Public              : True
RamdiskId           :
RootDeviceName      : /dev/sda1
RootDeviceType      : ebs
SriovNetSupport     : simple
State               : available
StateReason         :
Tags                : {}
VirtualizationType  : hvm
```

Ваш образ сохранен и готов к работе. Теперь вы, наконец, можете создать свой экземпляр EC2. Нужен всего лишь тип экземпляра. К сожалению, вы не можете посмотреть их список с помощью командлета PowerShell, но его можно найти по ссылке aws.amazon.com/ec2/instance-types/. Воспользуемся бесплатным вариантом — t2.micro. Загрузите свои параметры — идентификатор образа, статус необходимости связать его с общедоступным IP-адресом, тип экземпляра и ID подсети — и запустите команду `New-EC2Instance` (листинг 13.17).

Листинг 13.17. Создание экземпляра EC2

```
PS> $params = @{
>>      ImageId = $ami.ImageId
>>      AssociatePublicIp = $false
>>      InstanceType = 't2.micro'
>>      SubnetId = $sn.SubnetId
}
PS> New-EC2Instance @params

GroupNames    : {}
Groups        : {}
Instances     : {}
OwnerId       : 013223035658
RequesterId   :
ReservationId : r-05aa0d9b0fdf2df4f
```

Готово! В вашей консоли управления появится новенький экземпляр EC2 AWS. Вы также можете использовать команду `Get-EC2Instance` для возврата только что созданного экземпляра.

Закругляемся

Мы написали весь код для создания экземпляра EC2, но он все еще довольно неказист. Давайте упростим его. Скорее всего, нам часто придется создавать экземпляр EC2, поэтому мы создадим настраиваемую функцию, чтобы не повторять каждый раз действия по одному. На высоком уровне эта функция работает аналогично созданной в главе 12. Я не буду вдаваться в подробности работы этой функции — сам сценарий можно найти в прилагаемых к книге материалах. Однако я настоятельно рекомендую вам попытаться создать функцию самостоятельно.

При запуске сценария со всеми существующими зависимостями кроме непосредственно экземпляра EC2 вы получите вывод, показанный в листинге 13.18, если запускаете его с параметром `Verbose`.

Листинг 13.18. Запуск пользовательской функции создания экземпляра EC2

```
PS> $parameters = @{
>>      VpcCidrBlock = '10.0.0.0/16'
>>      EnableDnsSupport = $true
>>      SubnetCidrBlock = '10.0.1.0/24'
>>      OperatingSystem = 'Windows Server 2016'
>>      SubnetAvailabilityZone = 'us-east-1d'
>>      InstanceType = 't2.micro'
>>      Verbose = $true
}
```

```
PS> New-CustomEC2Instance @parameters

VERBOSE: Invoking Amazon Elastic Compute Cloud operation 'DescribeVpcs' in region
         'us-east-1'
VERBOSE: A VPC with the CIDR block [10.0.0.0/16] has already been created.
VERBOSE: Enabling DNS support on VPC ID [vpc-03ba701f5633fcfac]...
VERBOSE: Invoking Amazon EC2 operation 'ModifyVpcAttribute' in region 'us-east-1'
VERBOSE: Invoking Amazon EC2 operation 'ModifyVpcAttribute' in region 'us-east-1'
VERBOSE: Invoking Amazon Elastic Compute Cloud operation 'DescribeInternetGateways'
         in region 'us-east-1'
VERBOSE: An internet gateway is already attached to VPC ID [vpc-03ba701f5633fcfac].
VERBOSE: Invoking Amazon Elastic Compute Cloud operation 'DescribeRouteTables'
         in region 'us-east-1'
VERBOSE: Route table already exists for VPC ID [vpc-03ba701f5633fcfac].
VERBOSE: A default route has already been created for route table ID
         [rtb-0b4aa3a0e1801311f rtb-0aed41cac6175a94d].
VERBOSE: Invoking Amazon Elastic Compute Cloud operation 'DescribeSubnets'
         in region 'us-east-1' VERBOSE: A subnet has already been created and
         registered with VPC ID [vpc-03ba701f5633fcfac].
VERBOSE: Invoking Amazon EC2 operation 'DescribeImages' in region 'us-east-1'
VERBOSE: Creating EC2 instance...
VERBOSE: Invoking Amazon EC2 operation 'RunInstances' in region 'us-east-1'

GroupNames    : {}
Groups        : {}
Instances     : {}
OwnerId       : 013223035658
RequesterId   :
ReservationId : r-0bc2437cfbde8e92a
```

Теперь у вас есть все нужные инструменты для автоматизации создания экземпляров EC2 в AWS!

Развертывание приложения Elastic Beanstalk

Как и в службе веб-приложений Microsoft Azure, у AWS есть аналогичная служба веб-приложений. *Elastic Beanstalk (EB)* позволяет загружать веб-пакеты для размещения в инфраструктуре AWS. В этом разделе мы рассмотрим, что требуется для создания EB-приложения и последующего развертывания в нем пакета. Этот процесс состоит из пяти шагов:

1. Создаем приложение.
2. Создаем среду.
3. Загружаем пакет для приложения.
4. Создаем новую версию.
5. Разворачиваем новую версию в среде.

Создание приложения

Чтобы создать новое приложение, воспользуемся командой `New-EBApplication`, которая задает его имя. Назовем его `AutomateWorkflow`. Запустите команду — в результате получится вывод из листинга 13.19.

Листинг 13.19. Создание нового приложения Elastic Beanstalk

```
PS> $ebApp = New-EBApplication -ApplicationName 'AutomateWorkflow'
PS> $ebSApp

ApplicationName         : AutomateWorkflow
ConfigurationTemplates  : {}
DateCreated             : 9/19/2019 11:43:56 AM
DateUpdated             : 9/19/2019 11:43:56 AM
Description             :
ResourceLifecycleConfig : Amazon.ElasticBeanstalk.Model
                            .ApplicationResourceLifecycleConfig
Versions                : {}
```

Следующим шагом мы создадим *среду* — инфраструктуру, в которой будет размещено приложение. Для создания новой среды воспользуемся командой `New-EBEnvironment`. К сожалению, создать среду не так просто, как создать приложение. Некоторые параметры вроде имен приложения и среды мы задаем самостоятельно, но помимо этого необходимо иметь значения параметров `SolutionStackName`, `Tier_Type` и `Tier_Name`. Давайте поговорим о них подробнее.

Параметр `SolutionStackName` позволяет задать операционную систему и версию IIS, в которой должно работать ваше приложение. Чтобы вывести список доступных стеков решений, запустите команду `Get-EBAvailableSolutionStackList` и проверьте свойство `SolutionStackDetails`, как показано в листинге 13.20.

Листинг 13.20. Поиск доступных стеков решений

```
PS> (Get-EBAvailableSolutionStackList).SolutionStackDetails

PermittedFileTypes SolutionStackName
------------------ -----------------
{zip}              64bit Windows Server Core 2016 v1.2.0 running IIS 10.0
{zip}              64bit Windows Server 2016 v1.2.0 running IIS 10.0
{zip}              64bit Windows Server Core 2012 R2 v1.2.0 running IIS 8.5
{zip}              64bit Windows Server 2012 R2 v1.2.0 running IIS 8.5
{zip}              64bit Windows Server 2012 v1.2.0 running IIS 8
{zip}              64bit Windows Server 2008 R2 v1.2.0 running IIS 7.5
{zip}              64bit Amazon Linux 2018.03 v2.12.2 runni...
{jar, zip}         64bit Amazon Linux 2018.03 v2.7.4 running Java 8
{jar, zip}         64bit Amazon Linux 2018.03 v2.7.4 running Java 7
{zip}              64bit Amazon Linux 2018.03 v4.5.3 running Node.js
{zip}              64bit Amazon Linux 2015.09 v2.0.8 running Node.js
```

```
{zip}              64bit Amazon Linux 2015.03 v1.4.6 running Node.js
{zip}              64bit Amazon Linux 2014.03 v1.1.0 running Node.js
{zip}              32bit Amazon Linux 2014.03 v1.1.0 running Node.js
{zip}              64bit Amazon Linux 2018.03 v2.8.1 running PHP 5.4
--пропуск--
```

Как видите, вариантов много. В этом примере мы выберем 64-разрядную версию Windows Server Core 2012 R2 с IIS 8.5.

Теперь давайте рассмотрим параметр `Tier_Type`. Он задает тип среды, в которой будет работать ваш веб-сервис. Если вы будете использовать эту среду для размещения веб-сайта, вам нужен тип `Standard`.

И наконец, для параметра `Tier_Name` существуют варианты `WebServer` и `Worker`. Для размещения веб-сайта вам нужен `WebServer`, а для API потребуется `Worker`.

Теперь, когда все параметры определены, запустим команду `New-EBEnvironment`. В листинге 13.21 показана эта команда и результат.

Листинг 13.21. Создание приложения Elastic Beanstalk

```
PS> $parameters = @{
>>      ApplicationName = 'AutomateWorkflow'
>>      EnvironmentName = 'Testing'
>>      SolutionStackName = '64bit Windows Server Core 2012 R2 running IIS 8.5'
>>      Tier_Type = 'Standard'
>>      Tier_Name = 'WebServer'
}
PS> New-EBEnvironment @parameters

AbortableOperationInProgress : False
ApplicationName              : AutomateWorkflow
CNAME                        :
DateCreated                  : 9/19/2019 12:19:36 PM
DateUpdated                  : 9/19/2019 12:19:36 PM
Description                  :
EndpointURL                  :
EnvironmentArn               : arn:aws:elasticbeanstalk:...
EnvironmentId                : e-wkba2k4kcf
EnvironmentLinks             : {}
EnvironmentName              : Testing
Health                       : Grey
HealthStatus                 :
PlatformArn                  : arn:aws:elasticbeanstalk...
Resources                    :
SolutionStackName            : 64bit Windows Server Core 2012 R2 running IIS 8.5
Status                       : Launching
TemplateName                 :
Tier                         : Amazon.ElasticBeanstalk.Model.EnvironmentTier
VersionLabel                 :
```

Вы заметите, что в свойстве `Status` написано `Launching`. Это означает, что приложение еще не доступно, и вам придется немного подождать, пока не появится среда. Вы можете периодически проверять состояние приложения, выполняя команду `Get-EB Environment -ApplicationName 'AutomateWorkflow' -EnvironmentName 'Testing'`. Среда может запускаться в течение нескольких минут.

Когда свойство `Status` изменится на `Ready`, это будет означать, что среда готова и можно разворачивать пакет на сайте.

Развертывание пакета

Давайте выполним развертывание. Пакет, который вы развернете, будет содержать различные файлы для размещения на вашем веб-сайте. Вы можете поместить туда все, что захотите, — для наших целей это не имеет значения. Достаточно лишь, чтобы пакет находился в ZIP-файле. Используйте команду `Compress-Archive` для архивирования файлов, которые нужно развернуть:

```
PS> Compress-Archive -Path 'C:\MyPackageFolder\*' -DestinationPath 'C:\package.zip'
```

Когда ваш пакет будет собран и заархивирован, нужно будет положить его в то место, где приложение могло бы его найти. Тут есть различные варианты, но в нашем примере мы поместим его в корзину Amazon S3. Это довольнотаки распространенный способ хранения данных в AWS. Но чтобы поместить пакет в корзину Amazon S3, вам сначала понадобится она сама! Создадим ее в PowerShell. Запустим команду `New-S3Bucket -BucketName 'automateworkflow'`.

Когда ваша корзина будет готова к наполнению, загрузите ZIP-файл с помощью команды `Write-S3Object`, как в листинге 13.22.

Листинг 13.22. Загрузка пакета в S3

```
PS> Write-S3Object -BucketName 'automateworkflow' -File 'C:\package.zip'
```

Теперь вам нужно указать приложению на только что созданный ключ S3 и задать метку версии. Метка версии может быть любой, но обычно для нее используют уникальный номер, который зависит от времени. В нашем случае возьмем количество тиков, которые отражают дату и время. Получив метку версии, запустите `New-EBApplicationVersion` с еще несколькими параметрами, как показано в листинге 13.23.

Листинг 13.23. Создание новой версии приложения

```
PS> $verLabel = [System.DateTime]::Now.Ticks.ToString()
PS> $newVerParams = @{
```

```
>>       ApplicationName       = 'AutomateWorkflow'
>>       VersionLabel          = $verLabel
>>       SourceBundle_S3Bucket = 'automateworkflow'
>>       SourceBundle_S3Key    = 'package.zip'
}
PS> New-EBApplicationVersion @newVerParams

ApplicationName        : AutomateWorkflow
BuildArn               :
DateCreated            : 9/19/2019 12:35:21 PM
DateUpdated            : 9/19/2019 12:35:21 PM
Description            :
SourceBuildInformation :
SourceBundle           : Amazon.ElasticBeanstalk.Model.S3Location
Status                 : Unprocessed
VersionLabel           : 636729573206374337
```

Ваша версия приложения создана! Пришло время развернуть ее в среде. Сделаем это с помощью команды `Update-EBEnvironment`, как показано в листинге 13.24.

Листинг 13.24. Развертывание приложения в среде EB

```
PS> Update-EBEnvironment -ApplicationName 'AutomateWorkflow' -EnvironmentName
'Testing' -VersionLabel $verLabel -Force

AbortableOperationInProgress : True
ApplicationName              : AutomateWorkflow
CNAME                        : Testing.3u2ukxj2ux.us-ea...
DateCreated                  : 9/19/2019 12:19:36 PM
DateUpdated                  : 9/19/2019 12:37:04 PM
Description                  :
EndpointURL                  : awseb-e-w-AWSEBL...
EnvironmentArn               : arn:aws:elasticbeanstalk...
EnvironmentId                : e-wkba2k4kcf
EnvironmentLinks             : {}
EnvironmentName              : Testing
Health                       : Grey
HealthStatus                 :
PlatformArn                  : arn:aws:elasticbeanstalk:...
Resources                    :
SolutionStackName            : 64bit Windows Server Core 2012 R2 running IIS 8.5
Status                       : ❶Updating
TemplateName                 :
Tier                         : Amazon.ElasticBeanstalk.Model.EnvironmentTier
VersionLabel                 : 636729573206374337
```

Свойство `Status` изменилось с `Ready` на `Updating` ❶. Опять же, придется немного подождать до тех пор, пока статус снова не станет `Ready`, как в листинге 13.25.

Листинг 13.25. Подтверждение готовности приложения

```
PS> Get-EBEnvironment -ApplicationName 'AutomateWorkflow'
-EnvironmentName 'Testing'

AbortableOperationInProgress : False
ApplicationName              : AutomateWorkflow
CNAME                        : Testing.3u2ukxj2ux.us-e...
DateCreated                  : 9/19/2019 12:19:36 PM
DateUpdated                  : 9/19/2019 12:38:53 PM
Description                  :
EndpointURL                  : awseb-e-w-AWSEBL...
EnvironmentArn               : arn:aws:elasticbeanstalk...
EnvironmentId                : e-wkba2k4kcf
EnvironmentLinks             : {}
EnvironmentName              : Testing
Health                       : Green
HealthStatus                 :
PlatformArn                  : arn:aws:elasticbeanstalk:...
Resources                    :
SolutionStackName            : 64bit Windows Server Core 2012 R2 running IIS 8.5
Status                       : ❶Ready
TemplateName                 :
Tier                         : Amazon.ElasticBeanstalk.Model.EnvironmentTier
VersionLabel                 :
```

Статус снова стал `Ready` ❶. Выглядит отлично!

Создание базы данных SQL Server в AWS

Вам, как администратору AWS, может потребоваться работа с различными реляционными базами данных. В AWS есть служба Amazon Relational Database Service (Amazon RDS), которая позволяет администраторам легко задавать несколько типов баз данных. Существует несколько вариантов, но мы пока остановимся на SQL.

В этом разделе мы создадим пустую базу данных Microsoft SQL Server в RDS. Основной командой будет `New-RDSDBInstance`. Как и `New-AzureRm SqlDatabase`, команда `New-RDSDBInstance` имеет гораздо больше параметров, чем я смогу описать в этом разделе. Если вам интересно побольше узнать о способах подготовки экземпляров RDS, можете ознакомиться со справкой по команде `New-RDSDBInstance`.

Для наших целей вам понадобится следующая информация:

- Название экземпляра.
- Движок базы данных (SQL Server, MariaDB, MySQL и т. д.).

- Класс экземпляра, определяющий тип ресурсов, на которых работает SQL Server.
- Мастер-логин и пароль.
- Размер базы данных (в ГБ).

С некоторыми пунктами все понятно: название, имя пользователя, пароль и размер определить легко. С остальными нам придется разобраться.

Начнем с версии движка. Список всех доступных движков и их версий можно получить с помощью команды `Get-RDSDBEngineVersion`. Если запустить ее без параметров, она вернет много информации — слишком много. И не все из этого вам потребуется. Можно использовать команду `Group-Object`, чтобы сгруппировать все объекты по движку: вы получите список всех его версий, сгруппированных по названию. Как видно из листинга 13.26, вы получаете более управляемый вывод, где видны все доступные механизмы, которые можно использовать.

Листинг 13.26. Исследование версий движка RDS DB

```
PS> Get-RDSDBEngineVersion | Group-Object -Property Engine

Count Name                     Group
----- ----                     -----
    1 aurora-mysql             {Amazon.RDS.Model.DBEngineVersion}
    1 aurora-mysql-pq          {Amazon.RDS.Model.DBEngineVersion}
    1 neptune                  {Amazon.RDS.Model.DBEngineVersion}
--пропуск--
   16 sqlserver-ee             {Amazon.RDS.Model.DBEngineVersion,
                               Amazon.RDS.Model.DBEngineVersion,
                               Amazon.RDS.Model.DBEngineVersion,
                               Amazon.RDS.Mo...

   17 sqlserver-ex             {Amazon.RDS.Model.DBEngineVersion,
                               Amazon.RDS.Model.DBEngineVersion,
                               Amazon.RDS.Model.DBEngineVersion,
                               Amazon.RDS.Mo...

   17 sqlserver-se             {Amazon.RDS.Model.DBEngineVersion,
                               Amazon.RDS.Model.DBEngineVersion,
                               Amazon.RDS.Model.DBEngineVersion,
                               Amazon.RDS.Mo...

   17 sqlserver-web            {Amazon.RDS.Model.DBEngineVersion,
                               Amazon.RDS.Model.DBEngineVersion,
                               Amazon.RDS.Model.DBEngineVersion,
                               Amazon.RDS.Mo...
--пропуск--
```

У вас есть четыре записи `sqlserver`, соответствующие SQL Server Express, Web, Standard Edition и Enterprise Edition. Поскольку это всего лишь пример, выберем SQL Server Express. Это простой и, самое главное, бесплатный движок базы данных, что позволяет нам экспериментировать с ним без каких-либо ограничений. Выберите ядро SQL Server Express с помощью **`sqlserver-ex`**.

После выбора движка нужно указать версию. По умолчанию команда `New-RDSDBInstance` дает последнюю версию (которую вы будете использовать), но с помощью параметра `EngineVersion` вы можете указать и другую. Чтобы просмотреть все доступные версии, запустите `Get-RDSDBEngineVersion`, затем ограничьте поиск до `sqlserver-ex` и возвращайте только версии движка (листинг 13.27).

Листинг 13.27. Поиск версий ядра SQL Server Express

```
PS> Get-RDSDBEngineVersion -Engine 'sqlserver-ex' |
Format-Table -Property EngineVersion

EngineVersion
-------------
10.50.6000.34.v1
10.50.6529.0.v1
10.50.6560.0.v1
11.00.5058.0.v1
11.00.6020.0.v1
11.00.6594.0.v1
11.00.7462.6.v1
12.00.4422.0.v1
12.00.5000.0.v1
12.00.5546.0.v1
12.00.5571.0.v1
13.00.2164.0.v1
13.00.4422.0.v1
13.00.4451.0.v1
13.00.4466.4.v1
14.00.1000.169.v1
14.00.3015.40.v1
```

Следующее значение параметра, которое необходимо указать для `New-RDSDBInstance`, — это класс экземпляра. Класс экземпляра определяет работу базовой инфраструктуры (памяти, ЦП и т. д.), на которой будет размещена база данных. К сожалению, в PowerShell нет команды, которая позволяла бы вывести все доступные параметры класса экземпляра, но вы можете почитать об этом по ссылке docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.DBInstanceClass.html.

При выборе класса экземпляра важно убедиться, что он поддерживается выбранным вами движком. Мы будем использовать класс `db2.t2.micro` для создания базы данных RDS, но многие другие параметры в этом случае работать не будут. Для получения полной информации о том, какие классы экземпляров поддерживаются в разных БД RDS, ознакомьтесь с FAQ о AWS RDS (aws.amazon.com/rds/faqs/).

Если вы выберете класс экземпляра, который не поддерживается используемым движком, вы получите сообщение об ошибке, как показано в листинге 13.28.

Листинг 13.28. Ошибка при указании недопустимой конфигурации экземпляра

```
New-RDSDBInstance : RDS does not support creating a DB instance with the following
combination: DBInstanceClass=db.t1.micro, Engine=sqlserver-ex,
EngineVersion=14.00.3015.40.v1, LicenseModel=license-included. For supported
combinations of instance class and database engine version, see the documentation.
```

Когда вы выберете поддерживаемый класс экземпляра, вам необходимо будет выбрать имя пользователя и пароль. Обратите внимание, что AWS не принимает никаких старых паролей: в вашем пароле не может быть косой черты, знака @, запятой или пробела, иначе вы получите сообщение об ошибке, как показано в листинге 13.29.

Листинг 13.29. Указание недопустимого пароля с помощью New-RDSDBInstance

```
New-RDSDBInstance : The parameter MasterUserPassword is not a valid password.
Only printable ASCII characters besides '/', '@', '"', ' ' may be used.
```

Теперь у вас есть все параметры, которые нужны для запуска команды `New-RDSDBInstance`! Ожидаемый результат показан в листинге 13.30.

Листинг 13.30. Подготовка нового экземпляра базы данных RDS

```
PS> $parameters = @{
>>      DBInstanceIdentifier = 'Automating'
>>      Engine = 'sqlserver-ex'
>>      DBInstanceClass = 'db.t2.micro'
>>      MasterUsername = 'sa'
>>      MasterUserPassword = 'password'
>>      AllocatedStorage = 20
}
PS> New-RDSDBInstance @parameters

AllocatedStorage        : 20
AutoMinorVersionUpgrade : True
AvailabilityZone        :
BackupRetentionPeriod   : 1
```

```
CACertificateIdentifier : rds-ca-2015
CharacterSetName        :
CopyTagsToSnapshot      : False
--пропуск--
```

Поздравляю! У вашего AWS теперь есть готовая база данных RDS.

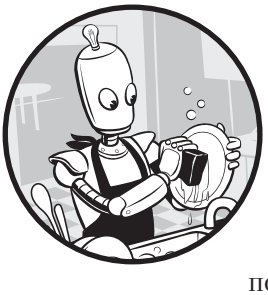
Итоги

В этой главе мы рассмотрели основы использования AWS в PowerShell. Мы рассмотрели его аутентификацию, а затем выполнили несколько типовых задач — создание экземпляров EC2, развертывание веб-приложений Elastic Beanstalk и подготовку базы данных Amazon RDS SQL.

В результате освоения последних двух глав вы должны хорошо понимать, как использовать PowerShell для работы с облаком. Конечно, вам еще многое предстоит узнать — больше, чем я мог бы охватить, но пока мы перейдем к следующей части этой книги: созданию собственного полнофункционального PowerShell-модуля.

14

Создание сценария инвентаризации сервера



Ранее в этой книге мы в основном занимались изучением PowerShell как языка, знакомились с его синтаксисом и командами. Однако PowerShell — это не только язык, но еще и инструмент. И теперь, когда вы овладели всеми тонкостями PowerShell, пора заняться по-настоящему интересными вещами!

Истинная мощь PowerShell заключается в его способности создавать инструменты. В этом контексте под *инструментом* понимается сценарий, модуль, функция PowerShell или что-то, что помогает выполнить некую задачу управления. Независимо от типа вашей задачи, ее можно автоматизировать с помощью PowerShell.

В этой главе я покажу вам, как с помощью PowerShell собирать данные для принятия более взвешенных решений. Если быть точнее, мы создадим проект инвентаризации серверов. Вы научитесь создавать сценарии с параметрами, вводить в них имена серверов и собирать различную информацию вроде спецификации операционной системы и информации об оборудовании, включая размер хранилища, свободное место, память и многое другое.

Исходные требования

Для работы с задачами этой главы вам потребуется подключенный к домену компьютер с Windows, разрешение на чтение для компьютерных объектов

Active Directory, OU (организационная единица — *organizational unit*) Active Directory учетных записей компьютеров и программный пакет Remote Server Administration Toolkit (RSAT), который вы можете скачать по ссылке www.microsoft.com/en-us/download/details.aspx?id=45520.

Создание сценария проекта

Поскольку в этой главе вы будете создавать сценарии, а не просто выполнять код в консоли, вам сперва нужно создать новый сценарий PowerShell. Назовите его Get-ServerInformation.ps1. Я поместил его на диск C:\. Мы будем добавлять код в этот сценарий на протяжении всей главы.

Определение окончательного результата

Перед тем как приступить к написанию кода, продумаем «изнанку» ожидаемого результата. Подобный набросок — отличный способ оценить прогресс, особенно при создании больших скриптов.

Результат выполнения рассматриваемого сценария инвентаризации сервера должен выглядеть следующим образом:

```
ServerName  IPAddress  OperatingSystem  AvailableDriveSpace (GB)
MYSERVER    x.x.x.x    Windows....      10

Memory (GB)   UserProfilesSize (MB)   StoppedServices
4             50.4                    service1,service2,service3
```

Теперь, когда вы знаете, как все должно выглядеть, давайте воплотим это в жизнь.

Обнаружение и ввод сценария

Первым делом надо решить, как настроить ваш сценарий на запрос нужной информации. Ее мы будем собирать с нескольких серверов. Как указано в разделе «Исходные требования», для определения имен своих серверов понадобится Active Directory.

Конечно, вы также можете брать имена серверов из текстовых файлов, из массива имен серверов, хранящихся в сценарии PowerShell, из реестра, из

репозитория Windows Management Instrumentation (WMI), из баз данных — все это подойдет. Покуда ваш сценарий каким-либо образом выдает массив строк с именами серверов, все идет хорошо. Однако для нашего проекта мы будем использовать серверы из Active Directory.

В этом примере все серверы находятся в одном подразделении. Если вы обнаружите, что это не так, то ничего страшного: вам просто нужно будет перебрать ваши OU и считать в каждом объекты компьютеров. Наша задача сейчас — прочитать все объекты компьютеров в OU. В этой среде все серверы находятся в OU Servers. Ваш домен называется `powerlab.local`. Чтобы получить компьютер из AD, используйте команду `Get-ADComputer`, как показано в листинге 14.1. Эта команда должна вернуть все компьютерные объекты AD для интересующих вас серверов.

Листинг 14.1. Использование Get-AdComputer для возврата данных сервера

```
PS> $serversOuPath = 'OU=Servers,DC=powerlab,DC=local'
PS> $servers = Get-ADComputer -SearchBase $serversOuPath -Filter *
PS> $servers

DistinguishedName : CN=SQLSRV1,OU=Servers,DC=Powerlab,DC=local
DNSHostName       : SQLSRV1.Powerlab.local
Enabled           : True
Name              : SQLSRV1
ObjectClass       : computer
ObjectGUID        : c288d6c1-56d4-4405-ab03-80142ac04b40
SamAccountName    : SQLSRV1$
SID               : S-1-5-21-763434571-1107771424-1976677938-1105
UserPrincipalName :
DistinguishedName : CN=WEBSRV1,OU=Servers,DC=Powerlab,DC=local
DNSHostName       : WEBSRV1.Powerlab.local
Enabled           : True
Name              : WEBSRV1
ObjectClass       : computer
ObjectGUID        : 3bd2da11-4abb-4eb6-9c71-7f2c58594a98
SamAccountName    : WEBSRV1$
SID               : S-1-5-21-763434571-1107771424-1976677938-1106
UserPrincipalName :
```

Обратите внимание, что вместо прямой установки аргумента параметра `SearchBase` мы самостоятельно определяем переменную. Согласен, к этому надо привыкнуть. Фактически, когда у вас есть такая конкретная конфигурация, хорошо бы поместить ее в переменную, потому что она может снова вам понадобиться. Вывод команды `Get-ADComputer` также возвращаем в переменную. Поскольку вы будете работать с этими серверами позже, вам нужно держать имена под рукой.

Команда `Get-ADComputer` возвращает все объекты AD, но нам нужны только имена серверов. Вы можете ограничить поиск, используя фильтр `Select-Object` и возвращая только свойство `Name`:

```
PS> $servers = Get-ADComputer -SearchBase $serversOuPath -Filter * |
Select-Object -ExpandProperty Name
PS> $servers
SQLSRV1
WEBSRV1
```

Теперь, когда у вас есть базовое представление о том, как запрашивать отдельный сервер, давайте посмотрим, как запросить сразу все.

Запрос всех серверов

Создадим цикл, который будет единожды опрашивать каждый сервер в вашем массиве.

Не стоит думать, что ваш код сразу будет работать правильно (обычно это не так). Мне, к примеру, нравится писать его медленно и постепенно, тестировать каждую деталь. В нашем случае мы не будем пытаться сделать все за один раз, а воспользуемся командой `Write-Host`, чтобы убедиться, что сценарий возвращает ожидаемые имена серверов:

```
foreach ($server in $servers) {
    Write-Host $server
}
```

Сейчас у вас должен быть сценарий Get-ServerInformation.ps1, который выглядит как в листинге 14.2.

Листинг 14.2. Ваш нынешний сценарий

```
$serversOuPath = 'OU=Servers,DC=powerlab,DC=local'
$servers = Get-ADComputer -SearchBase $serversOuPath -Filter * | Select-Object
                                                              -ExpandProperty Name
foreach ($server in $servers) {
    Write-Host $server
}
```

После запуска сценария вы получите несколько имен серверов. Конкретный результат будет зависеть от того, какие серверы вы использовали:

```
PS> C:\Get-ServerInformation.ps1
SQLSRV1
WEBSRV1
```

Отлично! У нас настроен цикл, который выполняет итерацию по именам серверов в нашем массиве. Первая задача выполнена.

Думаем наперед: объединение различных типов информации

Один из ключей к успеху в работе с PowerShell — хорошее планирование и организация. Сюда же входит и понимание того, чего можно ожидать. Для многих новичков, у которых нет большого опыта работы с выводами PowerShell, может возникнуть проблема. Они знают, чего хотят (по крайней мере, я на это надеюсь), но не понимают, что может произойти в процессе. В результате их сценарии бегают туда-сюда между источниками, получая данные то из одного, то из другого, затем снова из первого, а затем из третьего, соединяют все вместе и т. д. Есть более простые и рабочие способы, и я оказал бы вам медвежью услугу, если бы не объяснил все более подробно.

Глядя на вывод в листинге 14.1, вы можете заметить, что для получения информации из различных источников (WMI, файловая система, службы Windows) требуется несколько разных команд. Каждый источник будет возвращать свои типы объектов, и в результате, если вы попытаетесь их бездумно комбинировать, получится хаос.

Немного забегая вперед, давайте взглянем на то, как должен выглядеть результат, если вы вдруг попытаетесь получить служебные имя и память без какого-либо форматирования и обработки выходных данных. Получится что-то вроде этого:

```
Status  Name         DisplayName
------  ----         -----------
Running wuauserv     Windows Update

__GENUS            : 2
__CLASS            : Win32_PhysicalMemory
__SUPERCLASS       : CIM_PhysicalMemory
__DYNASTY          : CIM_ManagedSystemElement
__RELPATH          : Win32_PhysicalMemory.Tag="Physical Memory 0"
__PROPERTY_COUNT   : 30
__DERIVATION       : {CIM_PhysicalMemory, CIM_Chip, CIM_PhysicalComponent,
                     CIM_PhysicalElement...}
__SERVER           : DC
__NAMESPACE        : root\cimv2
__PATH             : \\DC\root\cimv2:Win32_PhysicalMemory.Tag="Physical Memory 0"
```

Здесь вы одновременно запрашиваете службу и пытаетесь получить данные с сервера. Объекты разные, их атрибуты — тоже, и если вы объедините весь вывод и сбросите его, это будет выглядеть ужасно.

Давайте посмотрим, как избежать такого развития событий. Поскольку вы будете комбинировать разные виды вывода, а результат должен соответствовать заданным характеристикам, придется создать собственный тип вывода. Не волнуйтесь, это не так сложно, как вы думаете. В главе 2 вы узнали, как создавать типы `PSCustomObject`. Эти универсальные объекты в PowerShell позволяют добавлять собственные свойства — идеально для нашей задачи.

Вы знаете заголовки нужного вам вывода (я уверен, вам уже известно, что эти «заголовки» всегда будут атрибутами объекта). Давайте создадим настраиваемый объект с нужными вам атрибутами. По понятным причинам я назвал этот объект `$output`, так как мы вернем его после заполнения:

```
$output = [pscustomobject]@{
    'ServerName'                = $null
    'IPAddress'                 = $null
    'OperatingSystem'           = $null
    'AvailableDriveSpace (GB)'  = $null
    'Memory (GB)'               = $null
    'UserProfilesSize (MB)'     = $null
    'StoppedServices'           = $null
}
```

Наверное, вы заметили, что ключи хеш-таблицы заключены в одинарные кавычки. Это необязательно, если в ключе нет пробелов. Однако, поскольку в некоторых именах ключей я использую пробелы, для себя я решил везде использовать одинарные кавычки. Обычно рекомендуется не добавлять пробелы в именах атрибутов объекта, а пользоваться вместо этого настраиваемым форматированием, но это выходит за рамки темы этой книги. Дополнительные сведения о настраиваемом форматировании приведены в разделе справки about_Format.ps1xml.

Если вы скопируете объект в консоль и вернете его с помощью командлета форматирования `Format-Table`, то увидите нужные заголовки:

```
PS> $output | Format-Table -AutoSize

ServerName IPAddress OperatingSystem AvailableDriveSpace (GB) Memory (GB)
UserProfilesSize (MB) StoppedServices
```

Команда `Format-Table` — одна из немногих команд форматирования в PowerShell, которые используются последними в конвейере. Они преобразуют текущий

вывод и отображают его заданным образом. В нашем случае вы просите PowerShell преобразовать вывод вашего объекта в формат таблицы и автоматически изменить размер строк под ширину консоли.

Как только вы определите нужный настраиваемый объект вывода, вы сможете вернуться к своему циклу и убедиться, что данные всех серверов возвращаются в этом формате. Поскольку вы уже знаете имя сервера, вы можете сразу установить это свойство, как показано в листинге 14.3.

Листинг 14.3. Помещение объекта вывода в цикл и установка имени сервера

```
$serversOuPath = 'OU=Servers,DC=powerlab,DC=local'
$servers = Get-ADComputer -SearchBase $serversOuPath -Filter * | Select-Object
-ExpandProperty Name
foreach ($server in $servers) {
    $output = @{
        'ServerName'                = $server
        'IPAddress'                 = $null
        'OperatingSystem'           = $null
        'AvailableDriveSpace (GB)'  = $null
        'Memory (GB)'               = $null
        'UserProfilesSize (MB)'     = $null
        'StoppedServices'           = $null
    }
    [pscustomobject]$output
}
```

Обратите внимание, что вы создали объект `output` в виде хеш-таблицы и преобразовали его в объект `PSCustom` только после заполнения его данными. Это связано с тем, что значения атрибутов проще хранить в хеш-таблице, чем в `PSCustomObject`. То есть мы думаем о формате `output` только при выводе, а при вводе источников информации они все будут одного типа.

Можно вывести все имена атрибутов `PSCustomObject`, а также имена серверов, которые вы запрашиваете, с помощью этого кода:

```
PS> C:\Get-ServerInformation.ps1 | Format-Table -AutoSize

ServerName UserProfilesSize (MB) AvailableDriveSpace (GB) OperatingSystem
---------- -------------------- ----------------------- --------------
SQLSRV1
WEBSRV1

StoppedServices IPAddress Memory (GB)
--------------- --------- -----------
```

Как видите, данные у вас есть. Может показаться, что это не так уж много, но вы на правильном пути!

Запрос файлов на удаленном расположении

Теперь, когда понятно, как мы хотим хранить данные, нам остается только получить их. Это значит, что нужно достать необходимую информацию с каждого сервера и вернуть нужные вам атрибуты. Начнем со значения `UserProfileSize` (МБ). Давайте узнаем, сколько места занимают профили в папке `C:\Users` на каждом сервере.

Мы будем использовать цикл, но сначала нужно выяснить, как решить эту задачу для одного сервера. Поскольку путь к папке `C:\Users` известен, давайте сначала посмотрим, сможете ли вы запросить все файлы во всех папках профиля пользователя сервера.

Когда вы запускаете команду `Get-ChildItem -Path \\WEBSRV1\c$\Users -Recurse -File`, вы получаете доступ к этому файловому ресурсу. Обратите внимание, что она возвращает все файлы и папки из всех профилей пользователей, но не выводит данные о размере. Давайте направим вывод в `Select-Object` и вернем все атрибуты:

```
PS> Get-ChildItem -Path \\WEBSRV1\c$\Users -Recurse -File | Select-Object -Property *

PSPath           : Microsoft.PowerShell.Core\FileSystem::\WEBSRV1\c$\Users\Adam
                   \file.log
PSParentPath     : Microsoft.PowerShell.Core\FileSystem::\\WEBSRV1\c$\Users\Adam
PSChildName      : file.log
PSProvider       : Microsoft.PowerShell.Core\FileSystem
PSIsContainer    : False
Mode             : -a----
VersionInfo      : File:             \\WEBSRV1\c$\Users\Adam\file.log
                   InternalName:
                   OriginalFilename:
                   FileVersion:
                   FileDescription:
                   Product:
                   ProductVersion:
                   Debug:          False
                   Patched:        False
                   PreRelease:     False
                   PrivateBuild:   False
                   SpecialBuild:   False
                   Language:

BaseName         : file
Target           :
LinkType         :
Name             : file.log
Length           : 8926
DirectoryName    : \\WEBSRV1\c$\Users\Adam
--пропуск--
```

Атрибут Length показывает размер файла в байтах. Зная это, нам остается выяснить, как сложить значения длины каждого файла в папке C:\Users на сервере. PowerShell позволяет делать это с помощью командлета Measure-Object. Он принимает входные данные из конвейера и автоматически складывает значения определенного атрибута:

```
PS> Get-ChildItem -Path '\\WEBSRV1\c$\Users\' -File | Measure-Object -Property
Length -Sum

Count    : 15
Average  :
Sum      : 600554
Maximum  :
Minimum  :
Property : Length
```

Теперь у нас есть атрибут (Sum), который отображает в выводе размер общего профиля пользователя. Сейчас достаточно просто вставить код в цикл и установить соответствующий атрибут в вашей хеш-таблице $output. Поскольку вам нужен только атрибут Sum объекта, который вернет Measure-Object, нужно заключить команду в круглые скобки и указать атрибут Sum, как показано в листинге 14.4.

Листинг 14.4. Обновление сценария для хранения UserProfilesSize

```
Get-ServerInformation.ps1
-------------------
$serversOuPath = 'OU=Servers,DC=powerlab,DC=local'
$servers = Get-ADComputer -SearchBase $serversOuPath -Filter * | Select-Object
-ExpandProperty Name
foreach ($server in $servers) {
    $output = @{
        'ServerName'                  = $null
        'IPAddress'                   = $null
        'OperatingSystem'             = $null
        'AvailableDriveSpace (GB)'    = $null
        'Memory (GB)'                 = $null
        'UserProfilesSize (MB)'       = $null
        'StoppedServices'             = $null
    }
    $output.ServerName = $server
    $output.'UserProfilesSize (MB)' = (Get-ChildItem -Path "\\$server\c$\Users
    \" -File |
    Measure-Object -Property Length -Sum).Sum
    [pscustomobject]$output
}
```

Если запустить сценарий, получится следующее:

```
PS> C:\Get-ServerInformation.ps1 | Format-Table -AutoSize

ServerName  UserProfilesSize (MB)  AvailableDriveSpace (GB)  OperatingSystem
----------  --------------------  ------------------------  ---------------
SQLSRV1                   636245
WEBSRV1                   600554

StoppedServices  IPAddress  Memory (GB)
---------------  ---------  -----------
```

Теперь вам известен общий размер профилей пользователей, но он выражен не в мегабайтах. Вы вычислили сумму `Length`, а она указана в байтах. PowerShell упрощает такие преобразования: если вы просто разделите число на 1 МБ, то получите нужный результат. Результат выводится в виде дробных чисел. Вы можете преобразовать его в целое значение, «округлив» число до целых мегабайтов:

```
$userProfileSize = (Get-ChildItem -Path "\\$server\c$\Users\" -File |
Measure-Object -Property Length -Sum).Sum
$output.'UserProfilesSize (MB)' = [int]($userProfileSize / 1MB)
```

Запрос инструментария управления Windows

Вам нужно заполнить еще пять значений. Для четырех из них вы будете использовать встроенный инструмент Microsoft под названием *Windows Management Instrumentation (WMI)*. Инструмент WMI, основанный на отраслевой стандартной модели Common Information Model (CIM), представляет собой репозиторий, содержащий информацию о тысячах атрибутов, относящихся к операционной системе и оборудованию, на котором она работает. Информация разделена на различные пространства имен, классов и атрибутов. Если вы ищете информацию о компьютере, вы будете часто использовать WMI.

Для нашего сценария нужно получить информацию о пространстве на жестком диске, версии операционной системы, IP-адресе сервера и объеме памяти, которую он содержит.

В PowerShell есть две команды для запроса WMI: `Get-WmiObject` и `Get-CimInstance`. Команда `Get-WmiObject` более старая и не такая гибкая, как `Get-CimInstance` (если вам интересны технические подробности, то это связано

с тем, что `Get-WmiObject` для подключения к удаленным компьютерам использует только DCOM, а `Get-CimInstance` по умолчанию использует WSMAN, но также может использовать DCOM). Сейчас Microsoft, похоже, всеми силами склоняет нас к использованию команды `Get-CimInstance`. Подробное сравнение CIM и WMI можно найти в этой статье: blogs.technet.microsoft.com/heyscriptingguy/2016/02/08/should-i-use-cim-or-wmi-with-windows-powershell/.

Самая сложная часть запроса WMI — выяснить, где скрыта нужная информация. Обычно это выясняется самостоятельно (и я рекомендую вам попытаться сделать это), но для экономии времени позвольте предложить вам список ответов для этого сценария: информация о хранилище находится в `Win32_LogicalDisk`, информация о работе системы находится в `Win32_OperatingSystem`, все службы Windows представлены в `Win32_Service`, любая информация о сетевом адаптере находится в `Win32_NetworkAdapterConfiguration`, а информация о памяти находится в `Win32_PhysicalMemory`.

Теперь давайте посмотрим, как использовать `Get-CimInstance` для запроса этих классов WMI и поиска нужных вам атрибутов.

Свободное место на диске

Начнем с доступного места на жестком диске. Оно хранится в `Win32_LogicalDisk`. Как и в случае с `UserProfilesSize`, мы начнем с одного сервера, а затем обобщим все в цикле. В этом случае нам повезло — не придется использовать `Select-Object` и копаться во всех атрибутах `FreeSpace`:

```
PS> Get-CimInstance -ComputerName sqlsrv1 -ClassName Win32_LogicalDisk

DeviceID DriveType ProviderName VolumeName Size        FreeSpace   PSComputerName
-------- --------- ------------ ---------- ----        ---------   --------------
C:       3                                 42708496384 34145906688 sqlsrv1
```

Зная, что команда `Get-CimInstance` возвращает объект, вы можете просто получить доступ к необходимому атрибуту и узнать количество свободного места:

```
PS> (Get-CimInstance -ComputerName sqlsrv1 -ClassName Win32_LogicalDisk).FreeSpace
34145906688
```

У вас есть значение, но, как и в прошлый раз, оно выражено в байтах (это типично для WMI). Вы можете проделать тот же трюк с преобразованием, за

исключением того, что теперь вам нужны гигабайты, поэтому делить будем на 1 ГБ. Разделив свойство `FreeSpace` на 1 ГБ, вы получите примерно такой результат:

```
PS> C:\Get-ServerInformation.ps1 | Format-Table -AutoSize

ServerName  UserProfilesSize (MB)  AvailableDriveSpace (GB)  OperatingSystem
----------  --------------------  ------------------------  ---------------
SQLSRV1                   636245             31.800853729248
WEBSRV1                   603942            34.5973815917969

StoppedServices  IPAddress  Memory (GB)
---------------  ---------  -----------
```

Вам не нужна точность в 12 знаков, поэтому можно немного округлить результат, используя метод `Round()` в классе `[Math]`, чтобы результат выглядел приятнее:

```
$output.'AvailableDriveSpace (GB)' = [Math]::Round(((Get-CimInstance -ComputerName
$server -ClassName Win32_LogicalDisk).FreeSpace / 1GB),1)

ServerName  UserProfilesSize (MB)  AvailableDriveSpace (GB)  OperatingSystem
----------  --------------------  ------------------------  ---------------
SQLSRV1                   636245                      31.8
WEBSRV1                   603942                      34.6

StoppedServices  IPAddress  Memory (GB)
---------------  ---------  -----------
```

Теперь значения будет намного легче читать. Три готово, осталось четыре.

Информация об операционной системе

Общая схема уже должна быть понятна: мы опрашиваем каждый сервер, находим интересующий нас атрибут и добавляем запрос в цикл `foreach`.

С этого момента мы просто будем добавлять код в имеющийся цикл `foreach`. Процесс сужения класса, его атрибутов или значений атрибутов одинаков для любого значения, которое вы будете запрашивать из WMI. Работаем по той же общей схеме:

```
$output.'PropertyName' = (Get-CimInstance -ComputerName ServerName
-ClassName WMIClassName).WMIClassPropertyName
```

Добавим следующее значение и получим сценарий из листинга 14.5.

Листинг 14.5. Ваш сценарий обновился и теперь содержит запрос к OperatingSystem

```
Get-ServerInformation.ps1
-------------------
$serversOuPath = 'OU=Servers,DC=powerlab,DC=local'
$servers = Get-ADComputer -SearchBase $serversOuPath -Filter * |
Select-Object -ExpandProperty Name
foreach ($server in $servers) {
    $output = @{
        'ServerName'                  = $null
        'IPAddress'                   = $null
        'OperatingSystem'             = $null
        'AvailableDriveSpace (GB)'    = $null
        'Memory (GB)'                 = $null
        'UserProfilesSize (MB)'       = $null
        'StoppedServices'             = $null
    }
    $output.ServerName = $server
    $output.'UserProfilesSize (MB)' = (Get-ChildItem -Path "\\$server\c$\
    Users\" -File | Measure-Object -Property Length -Sum).Sum / 1MB
    $output.'AvailableDriveSpace (GB)' = [Math]::Round(((Get-CimInstance
    -ComputerName $server -ClassName Win32_LogicalDisk).FreeSpace / 1GB),1)
    $output.'OperatingSystem' = (Get-CimInstance -ComputerName $server
    -ClassName Win32_OperatingSystem).Caption
    [pscustomobject]$output
}
```

Запустим сценарий:

```
PS> C:\Get-ServerInformation.ps1 | Format-Table -AutoSize

ServerName UserProfilesSize (MB) AvailableDriveSpace (GB)
---------- --------------------- -----------------------
SQLSRV1                  636245         31.8005790710449
WEBSRV1                  603942         34.5973815917969

OperatingSystem                            StoppedServices IPAddress Memory (GB)
---------------                            --------------- --------- -----------
Microsoft Windows Server 2016 Standard
Microsoft Windows Server 2012 R2 Standard
```

Вы получили полезную информацию об ОС. Движемся дальше. Выясним, как запросить информацию о памяти.

Память

Для работы с информацией о памяти будем использовать класс `Win32_PhysicalMemory`. И снова тестирование вашего запроса на одном сервере

позволяет получить необходимую информацию. В этом случае информация о памяти хранится в атрибуте `Capacity`:

```
PS> Get-CimInstance -ComputerName sqlsrv1 -ClassName Win32_PhysicalMemory

Caption              : Physical Memory
Description          : Physical Memory
InstallDate          :
Name                 : Physical Memory
Status               :
CreationClassName    : Win32_PhysicalMemory
Manufacturer         : Microsoft Corporation
Model                :
OtherIdentifyingInfo :
--пропуск--
Capacity             : 2147483648
--пропуск--
```

Каждый экземпляр в `Win32_PhysicalMemory` представляет собой *банк* оперативной памяти. Можно рассматривать банк как физический накопитель оперативной памяти на сервере. Так уж вышло, что у моего сервера `SQLSRV1` есть только один банк памяти. Однако вы, несомненно, найдете серверы, где их гораздо больше.

Поскольку нам нужно знать общий объем памяти на сервере, придется выполнить ту же процедуру, что и для получения размера профиля. А именно — сложить значения емкости для всех экземпляров. К счастью, командлет `Measure-Object` работает с любым количеством типов объектов. До тех пор пока атрибуты выражены числом, их можно складывать.

Опять же, поскольку емкость была представлена в байтах, преобразуем ее в другие единицы:

```
PS> (Get-CimInstance -ComputerName sqlsrv1 -ClassName Win32_PhysicalMemory |
Measure-Object -Property Capacity -Sum).Sum /1GB
2
```

Как видно из листинга 14.6, ваш сценарий все растет!

Листинг 14.6. Сценарий с запросом количества памяти

```
Get-ServerInformation.ps1
-------------------
$serversOuPath = 'OU=Servers,DC=powerlab,DC=local'
$servers = Get-ADComputer -SearchBase $serversOuPath -Filter * | Select-Object
-ExpandProperty Name
foreach ($server in $servers) {
```

```
    $output = @{
        'ServerName'                = $null
        'IPAddress'                 = $null
        'OperatingSystem'           = $null
        'AvailableDriveSpace (GB)'  = $null
        'Memory (GB)'               = $null
        'UserProfilesSize (MB)'     = $null
        'StoppedServices'           = $null
    }
    $output.ServerName = $server
    $output.'UserProfilesSize (MB)' = (Get-ChildItem -Path "\\$server\c$\
    Users\" -File | Measure-Object -Property Length -Sum).Sum / 1MB
    $output.'AvailableDriveSpace (GB)' = [Math]::Round(((Get-CimInstance
    -ComputerName $server -ClassName Win32_LogicalDisk).FreeSpace / 1GB),1)
    $output.'OperatingSystem' = (Get-CimInstance -ComputerName $server
    -ClassName Win32_OperatingSystem).Caption
    $output.'Memory (GB)' = (Get-CimInstance -ComputerName $server -ClassName
    Win32_PhysicalMemory | Measure-Object -Property Capacity -Sum).Sum /1GB
    [pscustomobject]$output
}
```

Давайте посмотрим на результат:

```
PS> C:\Get-ServerInformation.ps1 | Format-Table -AutoSize

ServerName UserProfilesSize (MB) AvailableDriveSpace (GB)
---------- --------------------- ------------------------
SQLSRV1                  636245                     31.8
WEBSRV1                  603942                     34.6

OperatingSystem                          StoppedServices IPAddress Memory (GB)
---------------                          --------------- --------- -----------
Microsoft Windows Server 2016 Standard                                       2
Microsoft Windows Server 2012 R2 Standard                                    2
```

Теперь нам осталось заполнить всего два поля!

Информация о сети

Последнее, что нам нужно из WMI — это IP-адрес, который вы получите из класса `Win32_NetworkAdapterConfiguration`. Я отложил задачу поиска IP-адреса напоследок, потому что, в отличие от других данных, поиск IP-адреса организуется чуть сложнее, чем просто «взять значение и вставить его в таблицу». Вам нужно будет выполнить фильтрацию, чтобы сузить область поиска.

Давайте сначала посмотрим, как выглядит результат при использовании того же метода, что и прежде:

```
PS> Get-CimInstance -ComputerName SQLSRV1 -ClassName Win32_
NetworkAdapterConfiguration

ServiceName    DHCPEnabled    Index    Description    PSComputerName
-----------    -----------    -----    -----------    --------------
kdnic          True           0        Microsoft...   SQLSRV1
netvsc         False          1        Microsoft...   SQLSRV1
tunnel         False          2        Microsoft...   SQLSRV1
```

Видно, что в выходных данных по умолчанию не отображается IP-адрес, хотя когда нас это останавливало? Сложности добавляет то, что данная команда не возвращает ни одного экземпляра. На этом сервере есть три сетевых адаптера. Как выбрать тот, чей IP-адрес нам нужен?

Для начала надо просмотреть все атрибуты с помощью команды `Select-Object`. Используя команду `Get-CimInstance -ComputerName SQLSRV1 -ClassName Win32_NetworkAdapter Configuration| Select-Object -Property *`, можно прокручивать входные данные. В зависимости от установленных на сервере сетевых адаптеров можно увидеть поля вообще без атрибута `IPAddress`. Это обычное явление, поскольку сетевые адаптеры не имеют IP-адресов. Однако если вы все же найдете тот, у которого есть IP-адрес, он должен выглядеть примерно как в коде, приведенном ниже, где есть атрибут `IPAddress` ❶ типа IPv4 со значением 192.168.0.40 и несколькими адресами IPv6:

```
DHCPLeaseExpires             :
Index                        : 1
Description                  : Microsoft Hyper-V Network Adapter
DHCPEnabled                  : False
DHCPLeaseObtained            :
DHCPServer                   :
DNSDomain                    : Powerlab.local
DNSDomainSuffixSearchOrder   : {Powerlab.local}
DNSEnabledForWINSResolution  : False
DNSHostName                  : SQLSRV1
DNSServerSearchOrder         : {192.168.0.100}
DomainDNSRegistrationEnabled : True
FullDNSRegistrationEnabled   : True
❶ IPAddress                    : {192.168.0.40...
IPConnectionMetric           : 20
IPEnabled                    : True
IPFilterSecurityEnabled      : False
--пропуск--
```

Сценарий должен динамически подстраиваться под множество конфигураций сетевого адаптера. Важно, чтобы он мог обрабатывать другие типы сетевых адаптеров помимо `Microsoft Hyper-V Network Adapter`, с которым вы здесь

работаете. Нужно лишь выбрать стандартный критерий для фильтрации, который мог бы применяться ко всем серверам.

Выберем `IPEnabled`. Если для этого атрибута установлено значение `True`, к этому сетевому адаптеру привязан протокол TCP/IP, что является предварительным условием для наличия IP-адреса. Если вы отфильтруете вывод так, чтобы выводились только данные с атрибутами `IPEnabled = True`, то найдете искомый адаптер.

При фильтрации экземпляров WMI всегда лучше использовать параметр `Filter` в команде `Get-CimInstance`. В сообществе PowerShell есть негласное правило: *фильтруй слева*. Это означает, что по возможности фильтруйте вывод как можно левее, то есть как можно раньше, чтобы не пропускать ненужные объекты через конвейер. Не используйте `Where-Object` без необходимости. Когда конвейер не забит ненужными объектами, производительность сценария намного выше.

Параметр `Filter` в `Get-CimInstance` использует *язык запросов Windows (Windows Query Language, WQL)*, который представляет собой небольшое подмножество *языка структурированных запросов (Structured Query Language, SQL)*. Этот параметр понимает тот же синтаксис предложения `WHERE`, что и WQL. Возьмем следующий пример: если в WQL нужно получить все экземпляры класса `Win32_NetworkAdapterConfiguration` с атрибутом `IPEnabled`, равным `True`, можно использовать запрос `SELECT * FROM Win32_NetworkAdapterConfiguration WHERE IPEnabled = 'True'`. Поскольку имя класса уже фигурирует в качестве параметра `ClassName` в `Get-CimInstance`, вам остается указать `IPEnabled = 'True'` для фильтра:

```
Get-CimInstance -ComputerName SQLSRV1 -ClassName Win32_NetworkAdapterConfiguration
-Filter "IPEnabled = 'True'" | Select-Object -Property *
```

В этом случае код вернет только сетевые адаптеры с `IPEnabled` (то есть те, у которых есть IP-адрес).

Теперь, когда у вас есть единственный экземпляр WMI и вы знаете искомый атрибут (`IPAddress`), давайте посмотрим, как будет выглядеть запрос сервера. Мы будем использовать тот же синтаксис `object.property`, что и раньше:

```
PS> (Get-CimInstance -ComputerName SQLSRV1 -ClassName Win32_
NetworkAdapterConfiguration
-Filter "IPEnabled = 'True'").IPAddress

192.168.0.40
fe80::e4e1:c511:e38b:4f05
2607:fcc8:acd9:1f00:e4e1:c511:e38b:4f05
```

Ой! Кажется, мы получили адреса IPv4 и IPv6. Придется использовать больше фильтров. Поскольку WQL не может фильтровать глубже значения атрибута, необходимо парсить адрес IPv4.

После небольшого исследования вы увидите, что все адреса заключены в фигурные скобки, разделенные запятыми:

```
IPAddress : {192.168.0.40, fe80::e4e1:c511:e38b:4f05, 2607:fcc8:acd9:1f00:e4e1:c511
:e38b:4f05}
```

Это хороший признак того, что атрибут хранится не как одна большая строка, а как массив. Чтобы убедиться в этом, можно попробовать использовать индекс и уточнить возможность получения только адреса IPv4:

```
PS> (Get-CimInstance -ComputerName SQLSRV1 -ClassName Win32_
NetworkAdapterConfiguration -Filter "IPEnabled = 'True'").IPAddress[0]

192.168.0.40
```

Вам повезло! Атрибут `IPAddress` представляет собой массив. Мы получили нужное значение, и теперь можно добавить весь новый код в сценарий, как показано в листинге 14.7.

Листинг 14.7. Обновленный код, который теперь обрабатывает атрибут IPAddress

```
Get-ServerInformation.ps1
-------------------
$serversOuPath = 'OU=Servers,DC=powerlab,DC=local'
$servers = Get-ADComputer -SearchBase $serversOuPath -Filter * |
Select-Object -ExpandProperty Name
foreach ($server in $servers) {
    $output = @{
        'ServerName'                = $null
        'IPAddress'                 = $null
        'OperatingSystem'           = $null
        'AvailableDriveSpace (GB)'  = $null
        'Memory (GB)'               = $null
        'UserProfilesSize (MB)'     = $null
        'StoppedServices'           = $null
    }
    $output.ServerName = $server
    $output.'UserProfilesSize (MB)' = (Get-ChildItem -Path "\\$server\c$\
    Users\" -File | Measure-Object -Property Length -Sum).Sum / 1MB
    $output.'AvailableDriveSpace (GB)' = [Math]::Round(((Get-CimInstance
    -ComputerName $server -ClassName Win32_LogicalDisk).FreeSpace / 1GB),1)
    $output.'OperatingSystem' = (Get-CimInstance -ComputerName $server
    -ClassName Win32_OperatingSystem).Caption
    $output.'Memory (GB)' = (Get-CimInstance -ComputerName $server -ClassName
```

```
    Win32_PhysicalMemory | Measure-Object -Property Capacity -Sum).Sum /1GB
    $output.'IPAddress' = (Get-CimInstance -ComputerName $server -ClassName
    Win32_NetworkAdapterConfiguration -Filter "IPEnabled = 'True'").IPAddress[0]
    [pscustomobject]$output
}
```

Запустим его:

```
PS> C:\Get-ServerInformation.ps1 | Format-Table -AutoSize

ServerName UserProfilesSize (MB) AvailableDriveSpace (GB)
---------- --------------------- ------------------------
SQLSRV1                   636245                     31.8
WEBSRV1                   603942                     34.6

OperatingSystem                            StoppedServices  IPAddress Memory (GB)
---------------                            ---------------  --------- -----------
Microsoft Windows Server 2016 Standard                      192.168.0.40        2
Microsoft Windows Server 2012 R2 Standard                   192.168.0.70        2
```

Теперь, когда у нас есть вся необходимая информация из WMI, осталось сделать лишь одну вещь.

Службы Windows

Последний фрагмент необходимых нам данных — это список остановленных на сервере служб. Следуя нашему базовому принципу, сначала опробуем сценарий на одном сервере. Для этого воспользуемся командой `Get-Service`, которая вернет все используемые на сервере службы. Затем передадим ее вывод команде `Where-Object`, которая будет фильтровать только остановленные службы. Команда будет выглядеть так: `Get-Service -ComputerName sqlsrv1 | Where-Object {$_.Status -eq 'Stopped'}`.

Эта команда возвращает целые объекты со всеми их атрибутами. Но нам нужны только имена служб. Тут нам поможет тот же метод, который вы уже использовали, — обращение к имени атрибута.

```
PS> (Get-Service -ComputerName sqlsrv1 | Where-Object { $_.Status -eq
                'Stopped' }).DisplayName
Application Identity
Application Management
AppX Deployment Service (AppXSVC)
--пропуск--
```

Добавив это в наш сценарий, получаем листинг 14.8.

Листинг 14.8. Обновление и использование вашего сценария для выдачи остановленных служб

```
Get-ServerInformation.ps1
-------------------
$serversOuPath = 'OU=Servers,DC=powerlab,DC=local'
$servers = Get-ADComputer -SearchBase $serversOuPath -Filter * |
Select-Object -ExpandProperty Name
foreach ($server in $servers) {
    $output = @{
        'ServerName'                    = $null
        'IPAddress'                      = $null
        'OperatingSystem'               = $null
        'AvailableDriveSpace (GB)'     = $null
        'Memory (GB)'                    = $null
        'UserProfilesSize (MB)'         = $null
        'StoppedServices'                = $null
    }
    $output.ServerName = $server
    $output.'UserProfilesSize (MB)' = (Get-ChildItem -Path "\\$server\c$\
    Users\" -File | Measure-Object -Property Length -Sum).Sum / 1MB
    $output.'AvailableDriveSpace (GB)' = [Math]::Round(((Get-CimInstance
    -ComputerName $server -ClassName Win32_LogicalDisk).FreeSpace / 1GB),1)
    $output.'OperatingSystem' = (Get-CimInstance -ComputerName $server
    -ClassName Win32_OperatingSystem).Caption
    $output.'Memory (GB)' = (Get-CimInstance -ComputerName $server -ClassName
    Win32_PhysicalMemory | Measure-Object -Property Capacity -Sum).Sum /1GB
    $output.'IPAddress' = (Get-CimInstance -ComputerName $server -ClassName
    Win32_NetworkAdapterConfiguration -Filter "IPEnabled = 'True'").IPAddress[0]
    $output.StoppedServices = (Get-Service -ComputerName $server |
    Where-Object { $_.Status -eq 'Stopped' }).DisplayName
     [pscustomobject]$output
}
```

Выполните следующий код, чтобы проверить свой сценарий:

```
PS> C:\Get-ServerInformation.ps1 | Format-Table -AutoSize

ServerName UserProfilesSize (MB) AvailableDriveSpace (GB)
---------- --------------------- -----------------------
SQLSRV1                  636245                    31.8
WEBSRV1                  603942                    34.6

OperatingSystem                              StoppedServices
---------------                              ---------------
Microsoft Windows Server 2016 Standard       {Application Identity,
                                             Application Management,
                                             AppX Deployment Servi...
Microsoft Windows Server 2012 R2 Standard    {Application Experience,
                                             Application Management,
                                             Background Intellig...
```

С остановленными службами все нормально, но куда делись остальные атрибуты? Сейчас в окне консоли не осталось места. Удаление ссылки `Format-Table` позволяет увидеть все значения:

```
PS> C:\Get-ServerInformation.ps1 | Format-Table -AutoSize

ServerName               : SQLSRV1
UserProfilesSize (MB)    : 636245
AvailableDriveSpace (GB) : 31.8
OperatingSystem          : Microsoft Windows Server 2016 Standard
StoppedServices          : {Application Identity, Applic...
IPAddress                : 192.168.0.40
Memory (GB)              : 2

ServerName               : WEBSRV1
UserProfilesSize (MB)    : 603942
AvailableDriveSpace (GB) : 34.6
OperatingSystem          : Microsoft Windows Server 2012 R2 Standard
StoppedServices          : {Application Experience, Application Management,
                           Background Intelligent Transfer Service, Computer
                           Browser...}
IPAddress                : 192.168.0.70
Memory (GB)              : 2
```

Выглядит неплохо!

Очистка и оптимизация скрипта

Пока что не будем праздновать победу и бежать творить подвиги, а немного поразмышляем. Написание кода — итеративный процесс. Бывает так, что даже достигнув цели, вы все равно получаете плохой код, ведь хорошая программа — это нечто большее, чем просто выполнение поставленной задачи. Сценарий уже делает то, что нужно, но можно было бы сделать лучше. Как?

Вспомните метод DRY: *не повторяйтесь*. В этом сценарии у вас много повторов. Например, там есть множество ссылок на `Get-CimInstance`, где снова и снова используются одни и те же параметры. Также для одного и того же сервера делается много вызовов WMI. Похоже, можно было бы повысить эффективность кода в этих местах.

Прежде всего командлеты CIM имеют параметр `CimSession`. Этот параметр позволяет создать один сеанс CIM, а затем использовать его повторно. Вместо того чтобы создавать временный сеанс, использовать его и разрывать, можно создать один сеанс, использовать его как угодно, а затем разорвать в конце

работы, как показано в листинге 14.9. Эта концепция аналогична параметру `Session` команды `Invoke-Command`, который мы рассмотрели в главе 8.

Листинг 14.9. Обновление кода для создания и повторного использования одного сеанса

```
Get-ServerInformation.ps1
-------------------
$serversOuPath = 'OU=Servers,DC=powerlab,DC=local'
$servers = Get-ADComputer -SearchBase $serversOuPath -Filter * |
Select-Object -ExpandProperty Name
foreach ($server in $servers) {
    $output = @{
        'ServerName'                    = $null
        'IPAddress'                     = $null
        'OperatingSystem'               = $null
        'AvailableDriveSpace (GB)'      = $null
        'Memory (GB)'                   = $null
        'UserProfilesSize (MB)'         = $null
        'StoppedServices'               = $null
    }
    $cimSession = New-CimSession -ComputerName $server
    $output.ServerName = $server
    $output.'UserProfilesSize (MB)' = (Get-ChildItem -Path "\\$server\c$\
    Users\" -File | Measure-Object -Property Length -Sum).Sum
    $output.'AvailableDriveSpace (GB)' = [Math]::Round(((Get-CimInstance
    -CimSession $cimSession -ClassName Win32_LogicalDisk).FreeSpace / 1GB),1)
    $output.'OperatingSystem' = (Get-CimInstance -CimSession $cimSession
    -ClassName Win32_OperatingSystem).Caption
    $output.'Memory (GB)' = (Get-CimInstance -CimSession $cimSession
    -ClassName Win32_PhysicalMemory | Measure-Object -Property Capacity -Sum)
    .Sum /1GB
    $output.'IPAddress' = (Get-CimInstance -CimSession $cimSession -ClassName
    Win32_NetworkAdapterConfiguration -Filter "IPEnabled = 'True'").IPAddress[0]
    $output.StoppedServices = (Get-Service -ComputerName $server |
    Where-Object { $_.Status -eq 'Stopped' }).DisplayName
    Remove-CimSession -CimSession $cimSession
    [pscustomobject]$output
}
```

Теперь мы будем использовать только один сеанс CIM. Однако вы по-прежнему часто ссылаетесь на него в параметрах разных команд. Чтобы еще лучше оптимизировать код, можно создать хеш-таблицу и задать в качестве ключа имя `CIMSession`, а в качестве значения — сам сеанс CIM. Если у вас есть общий набор параметров, сохраненный в виде хеш-таблицы, можно повторно использовать его во всех ссылках `Get-CimInstance`.

Этот метод известен как *сплаттинг*, и для его реализации можно указать только что созданную хеш-таблицу при вызове каждой из ссылок `Get-CimInstance` через символ `@`, за которым следует имя хеш-таблицы, как показано в листинге 14.10.

Листинг 14.10. Создание многоразового параметра CIMSession

```
Get-ServerInformation.ps1
-------------------
$serversOuPath = 'OU=Servers,DC=powerlab,DC=local'
$servers = Get-ADComputer -SearchBase $serversOuPath -Filter * |
Select-Object -ExpandProperty Name
foreach ($server in $servers) {
    $output = @{
        'ServerName'                    = $null
        'IPAddress'                     = $null
        'OperatingSystem'               = $null
        'AvailableDriveSpace (GB)'      = $null
        'Memory (GB)'                   = $null
        'UserProfilesSize (MB)'         = $null
        'StoppedServices'               = $null
    }
    $getCimInstParams = @{
        CimSession = New-CimSession -ComputerName $server
    }
    $output.ServerName = $server
    $output.'UserProfilesSize (MB)' = (Get-ChildItem -Path "\\$server\c$\
    Users\" -File | Measure-Object -Property Length -Sum).Sum
    $output.'AvailableDriveSpace (GB)' = [Math]::Round(((Get-CimInstance
    @getCimInstParams -ClassName Win32_LogicalDisk).FreeSpace / 1GB),1)
    $output.'OperatingSystem' = (Get-CimInstance @getCimInstParams -ClassName
    Win32_OperatingSystem).Caption
    $output.'Memory (GB)' = (Get-CimInstance @getCimInstParams -ClassName
    Win32_PhysicalMemory | Measure-Object -Property Capacity -Sum).Sum /1GB
    $output.'IPAddress' = (Get-CimInstance @getCimInstParams -ClassName
    Win32_NetworkAdapterConfiguration -Filter "IPEnabled = 'True'").IPAddress[0]
    $output.StoppedServices = (Get-Service -ComputerName $server |
    Where-Object { $_.Status -eq 'Stopped' }).DisplayName
    Remove-CimSession -CimSession $cimSession
    [pscustomobject]$output
}
```

Вы наверняка уже привыкли передавать командам параметры в формате *тире<имя параметра> <значение параметра>*. Этот метод, безусловно, работает, но не является самым эффективным. Вместо него можно создавать хеш-таблицу, а затем просто передавать ее каждой команде, которой нужны эти параметры.

Теперь мы полностью исключили переменную `$cimSession`.

Итоги

В этой главе мы обобщили знания из всех предыдущих глав и применили их на практике. Сценарий, запрашивающий информацию, — это одна из первых вещей, которые я обычно рекомендую создавать. Такие сценарии хорошо учат работать с PowerShell. К тому же у вас мало шансов где-то напортачить!

Шаг за шагом мы продвигались по этой главе, переходя от цели к решению, а затем — к еще лучшему решению. Работая с PowerShell, вы будете повторять этот процесс снова и снова. Определите свою цель, начните с малого, наметьте структуру (в нашем случае цикл `foreach`) и начните добавлять код по частям, преодолевая по одной проблеме за раз, пока все не сложится воедино.

Когда вы завершите свой сценарий, имейте в виду, что по-настоящему вы закончите, лишь когда просмотрите свой код: проверьте, как можно сделать его более эффективным, как использовать меньше ресурсов или как заставить его работать быстрее. С опытом этот процесс будет даваться легче. Вы будете все лучше и лучше понимать суть работы, пока оптимизация не станет вашим вторым «я». Когда вы закончите оптимизацию, расслабьтесь, насладитесь успехом и приготовьтесь начать следующий проект!

ЧАСТЬ III

СОЗДАЕМ СВОЙ МОДУЛЬ

К этому моменту вы уже должны иметь твердое представление о том, что делает PowerShell тем, чем он является. Мы рассмотрели синтаксис языка, а также несколько модулей, которые можно использовать в своей повседневной работе по автоматизации. До предыдущей главы мы делали вещи только по частям: немного синтаксиса здесь, немного там, но ничего серьезного. В главе 14 мы придумали проект по инвентаризации серверов и впервые попробовали поработать над большим проектом PowerShell. В части III мы собираемся пойти еще дальше — создать собственный PowerShell-модуль.

PowerLab

PowerLab — это PowerShell-модуль, содержащий функции, необходимые для настройки серверов Windows с нуля. Мы построим PowerLab кирпичик за кирпичиком, но если вам не терпится увидеть окончательный результат, вы можете найти его в этом репозитории GitHub: github.com/adbertram/PowerLab.

Процесс подготовки сервера Windows с нуля будет выглядеть примерно так:

- Создание виртуальной машины.
- Установка операционной системы Windows.
- Установка службы сервера (Active Directory, SQL Server или IIS).

Это означает, что модуль PowerLab понадобится для выполнения пяти задач:

- Создание виртуальной машины Hyper-V.
- Установка сервера Windows.

- Создание Active Directory.
- Подготовка серверов SQL.
- Подготовка веб-серверов IIS.

Для выполнения этих задач вы будете использовать три основные команды:

- `New-PowerLabActiveDirectoryForest`.
- `New-PowerLabSqlServer`.
- `New-PowerLabWebServer`.

На самом деле понадобится больше трех команд. Мы создадим каждую из них плюс еще несколько вспомогательных, которые будут отвечать за внутренние функции, включая создание виртуальной машины и установку операционной системы. Все это мы рассмотрим в следующих главах.

Исходные требования

Для создания PowerLab вам понадобится несколько вещей:

- Клиентский компьютер с Windows 10 Professional в рабочей группе. Компьютер с Windows 10, присоединенный к домену, может подойти, но эта возможность не проверялась.
- Узел Hyper-V в рабочей группе под управлением Windows Server 2012 R2 (как минимум), находящийся в одной сети с клиентом. Узел также может быть присоединен к домену, но это опять же не проверялось.
- ISO-файлы с Windows Server 2016, расположенные на вашем хосте Hyper-V. Использование Windows Server 2019 не тестировалось. Вы можете скачать ознакомительные версии Windows Server с сайта www.microsoft.com/en-us/evalcenter/evaluate-windows-server-2016?filetype=ISO.
- Инструменты удаленного администрирования сервера (RSAT) на клиентском компьютере (их можно скачать с www.microsoft.com/en-us/download/details.aspx?id=45520).
- Последняя версия модуля Pester PowerShell на вашем клиентском компьютере.

Вам также необходимо войти в систему под видом члена группы локальных администраторов на клиентском компьютере и настроить неограниченную политику выполнения PowerShell. (Вы можете запустить команду `Set-ExecutionPolicy Unrestricted`, чтобы изменить политику выполнения, но

я рекомендую после завершения настройки изменить ее обратно на `AllSigned` или `RemoteSigned`.)

Настройка PowerLab

Предоставляя потребителям нечто вроде PowerLab, нужно сделать настройку модуля максимально простой. Один из способов сделать это — дать им сценарий, который выполняет установку и настройку вашего модуля с минимальным участием пользователя.

Я уже написал такой сценарий для PowerLab. Его можно найти в репозитории PowerLab GitHub: raw.githubusercontent.com/adbertram/PowerLab/master/Install-PowerLab.ps1. По этой ссылке вы найдете сырой исходный код сценария. Вы могли бы скопировать и вставить его в новый текстовый файл и сохранить под именем Install-PowerLab.ps1, но мы же изучаем PowerShell, поэтому давайте попробуем самостоятельно выполнить следующую команду:

```
PS> Invoke-WebRequest -Uri 'http://bit.ly/powerlabinstaller' -OutFile 'C:\Install-
PowerLab.ps1'
```

Внимание: когда вы запустите сценарий, вам придется ответить на некоторые вопросы. Вам понадобится имя хоста Hyper-V, IP-адрес хоста Hyper-V, имя пользователя и пароль локального администратора для хоста Hyper-V и ключи продукта (если не используется пробная версия Windows Server) для каждой операционной системы, которую вы хотите установить.

Когда вся эта информация будет у вас под рукой, запустите сценарий установки с помощью следующей команды:

```
PS> C:\Install-PowerLab.ps1

Name of your HYPERV host: HYPERVSRV
IP address of your HYPERV host: 192.168.0.200
Enabling PS remoting on local computer...
Adding server to trusted computers...
PS remoting is already enabled on [HYPERVSRV]
Setting firewall rules on Hyper-V host...
Adding the ANONYMOUS LOGON user to the local machine and host server
Distributed COM Users group for Hyper-V manager
Enabling applicable firewall rules on local machine...
Adding saved credential on local computer for Hyper-V host...
Ensure all values in the PowerLab configuration file are valid and close the
ISE when complete.
Enabling the Microsoft-Hyper-V-Tools-All features...
Lab setup is now complete.
```

Если вы хотите узнать, что делает этот сценарий, вы всегда можете скачать его из прилагаемых к книге материалов и изучить. Не удивляйтесь, если вы мало что поймете, так как этот сценарий лишь создает для нас одинаковую инфраструктуру, а не показывает, что и как работает. Он предназначен для того, чтобы вы могли повторять за мной.

Демокод

Весь код, который вы напишете в следующих главах, можно найти по ссылке github.com/adbertram/PowerShellForSysadmins/tree/master/Part%20III. Помимо кода PowerLab там есть нужные файлы данных и сценарии Pester для тестирования модуля, а также проверки соответствия всем требованиям вашей среды. Перед началом каждой главы я настоятельно рекомендую использовать команду `Invoke-Pester` для запуска Pester-сценария Prerequisites.Tests.ps1, который находится в файлах каждой главы. Это избавит вас от головной боли с разбором багов по пути.

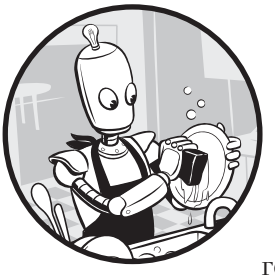
Итоги

Сейчас у вас должно быть все нужное для создания PowerLab. В следующих главах мы рассмотрим множество вопросов и затронем много функций PowerShell, поэтому не удивляйтесь, если вдруг столкнетесь с чем-то неизвестным. Существует множество онлайн-ресурсов, которые могут помочь вам разобраться с тернистым синтаксисом, и если вы чего-то не понимаете, то всегда можете связаться со мной в твиттере (@adbertram) или попросить помощи в интернете.

Итак, приступим!

15

Создание виртуальной среды



PowerLab — это наш заключительный масштабный проект, в котором будут использоваться все изученные понятия.

В этом проекте мы полностью автоматизируем подготовку виртуальных машин (ВМ) Hyper-V, вплоть до установки и настройки служб, включая SQL и IIS. Представьте себе — вы сможете запустить одну команду вроде `New-PowerLabSqlServer`, `New-PowerLabIISServer` или даже `New-PowerLab` и через несколько минут получить полностью сконфигурированный компьютер (или машину). Именно это мы и сделаем ближе к концу книги.

Цель проекта PowerLab — взять на себя все повторяющиеся, отнимающие много времени задачи, которые нужны для создания тестовой среды или лаборатории. Когда вы закончите, в сухом остатке у вас будет несколько команд для создания целого леса Active Directory из ничего, кроме хоста Hyper-V и нескольких файлов ISO.

Я намеренно не рассмотрел в частях I и II *все*, что входит в PowerLab. Вместо этого я призываю вас обратить внимание на непонятные вам вещи и самостоятельно найти решение. Ведь в программировании всегда существует много способов выполнить одну и ту же задачу. Если возникнут трудности, напишите мне в твиттер по адресу @adbertram.

Создав проект такого масштаба, вы не только охватите сотни аспектов PowerShell, но также увидите, насколько мощным может быть язык сценариев,

и, в конце концов, получите полезную утилиту, значительно экономящую время.

В этой главе мы начнем работу над PowerLab с создания его базового модуля. Затем мы добавим возможность автоматизировать создание виртуального коммутатора, виртуальной машины и виртуального жесткого диска (VHD).

Исходные требования для модуля PowerLab

Чтобы выполнить все примеры кода, с которыми вы будете работать в части III, вам нужно выполнить несколько предварительных условий. В каждой главе этой части есть раздел «Исходные требования». Это сделано для того, чтобы вы всегда знали, чего можно ожидать. Для проекта из этой главы вам понадобится хост Hyper-V со следующей конфигурацией:

- Сетевой адаптер.
- IP: 10.0.0.5 (необязательно, но для точного следования примерам вам понадобится именно этот IP).
- Маска подсети: 255.255.255.0.
- Рабочая группа.
- Не менее 100 ГБ свободного места на диске.
- Windows Server 2016 с полным графическим интерфейсом.

Чтобы создать сервер Hyper-V, необходимо установить роль Hyper-V на сервере Windows, который вы собираетесь использовать. Вы можете ускорить процесс установки, если загрузите и запустите Hyper-V-сценарий Setup.ps1 из материалов книги, расположенный по ссылке github.com/adbertram/PowerShellForSysadmins/. Сценарий настроит Hyper-V и создаст несколько нужных папок.

ПРИМЕЧАНИЕ

Если вы планируете следовать за текстом слово в слово, запустите сценарий исходных требований Pester из соответствующей главы (Prerequisites.Tests.ps1), чтобы убедиться в том, что ваш сервер Hyper-V настроен правильно. Тестирование подтвердит, что ваша лабораторная среда настроена точно так же, как моя. Запустите команду `Invoke-Pester`, передав ей сценарий с исходными требованиями, как показано в листинге 15.1. В оставшейся части книги весь код будет выполняться на самом хосте Hyper-V.

Листинг 15.1. Выполнение предварительных проверок Pester для работы Hyper-V

```
PS> Invoke-Pester -Path 'C:\PowerShellForSysadmins\Part III
                         \Automating Hyper-V\Prerequisites.Tests.ps1'

Describing Automating Hyper-V Chapter Prerequisites
 [+] Hyper-V host server should have the Hyper-V Windows feature installed 2.23s
 [+] Hyper-V host server is Windows Server 2016 147ms
 [+] Hyper-V host server should have at least 100GB of available storage 96ms
 [+] has a PowerLab folder at the root of C 130ms
 [+] has a PowerLab\VMs folder at the root of C 41ms
 [+] has a PowerLab\VHDs folder at the root of C 47ms
Tests completed in 2.69s
Passed: 5 Failed: 0 Skipped: 0 Pending: 0 Inconclusive: 0
```

Если вы успешно настроили среду, выходные данные должны подтвердить пять проходов. Убедитесь, что ваша среда настроена, и можете приступать к работе над проектом!

Создание модуля

Поскольку нам заранее известно, что придется автоматизировать много связанных задач, то необходимо создать PowerShell-модуль. Как вы узнали из главы 7, модуль PowerShell — отличный способ объединить множество похожих функций в одном месте. PowerLab будет построен именно по этому принципу. Нам незачем хвататься сразу за всё, поэтому начнем с малого — добавим функцию, протестируем ее и т. д.

Создание пустого модуля

Сперва нужно создать пустой модуль. Для этого подключимся с удаленного рабочего стола к вашему будущему хосту Hyper-V и войдем в систему как локальный администратор или используем любую учетную запись в группе локальных администраторов. Мы создадим этот модуль непосредственно на хосте Hyper-V, чтобы упростить создание и администрирование виртуальных машин. Это означает, что мы будем использовать сеанс RDP для подключения к сеансу консоли хоста Hyper-V. Затем мы создадим папку модуля, сам модуль (файл .psm1) и не являющийся обязательным манифест (файл .psd1).

Поскольку вы вошли в систему под учетной записью локального администратора и можете однажды предоставить ваш модуль PowerLab другим пользователям, создайте его в папке All Users, расположенной в C:\Files. Это позволит

вам получить доступ к модулю, если вы вошли в систему как любой административный пользователь на хосте.

Затем откройте консоль PowerShell и выполните запуск от имени администратора.

После этого создайте папку модуля PowerLab с помощью следующего кода:

```
PS> New-Item -Path C:\Program Files\WindowsPowerShell\Modules
                    \PowerLab -ItemType Directory
```

Наконец, создайте пустой текстовый файл с именем PowerLab.psm1. Для этого нужна команда `New-Item`:

```
PS> New-Item -Path 'C:\Program Files\WindowsPowerShell\Modules\PowerLab
                     \PowerLab.psm1'
```

Создание манифеста модуля

Теперь мы создадим манифест модуля. Для этого есть удобная команда `New-ModuleManifest`. Она создает шаблон манифеста, который затем можно открыть в текстовом редакторе и при необходимости настроить после сборки кода. Ниже приведены параметры, которые я использовал для создания манифеста шаблона:

```
PS> New-ModuleManifest -Path 'C:\Program Files\WindowsPowerShell\Modules\PowerLab\
PowerLab.psd1'
-Author 'Adam Bertram'
-CompanyName 'Adam the Automator, LLC'
-RootModule 'PowerLab.psm1'

-Description 'This module automates all tasks to provision entire environments of a
domain controller, SQL server and IIS web server from scratch.'
```

Вы можете изменять значения параметров под свои нужды.

Использование встроенных префиксов для имен функций

Имя функции не обязательно должно быть конкретным. Однако когда вы создаете модуль, который представляет собой группу связанных функций, всегда рекомендуется ставить перед существительной частью функции один и тот же префикс. Например, ваш проект называется PowerLab; в нем мы создаем

функции, относящиеся к одной теме. Чтобы отличать функции в PowerLab от функций в других потенциальных модулях, можно добавить имя модуля непосредственно перед существительным. Это означает, что существительные большинства функций будут начинаться со слова *PowerLab*.

Однако не все функции будут начинаться с имени модуля. К таковым, например, относятся вспомогательные функции, которые выполняют внутреннюю работу и не вызываются пользователем.

Если вы хотите добавить один и тот же префикс к существительным всех функций без необходимости явно определять его в имени, то можете воспользоваться опцией манифеста модуля `DefaultCommandPrefix`. Она принудительно добавляет к существительному определенную строку. Например, если вы определяете в манифесте ключ `DefaultCommandPrefix` и создаете функцию внутри модуля `New-Switch`, то при импортировании модуля функция будет доступна под именем `New-PowerLabSwitch` вместо `New-Switch`:

```
# Default prefix for commands exported from this modul...
# DefaultCommandPrefix = ''
```

Я предпочитаю *не* полагаться на этот подход, так как в этом случае у *всех* существительных в именах функций модуля появляется приписка в виде строки.

Импорт нового модуля

Теперь, когда мы создали манифест, надо посмотреть, успешно ли он импортируется. Поскольку мы еще не написали никаких функций, модуль не сможет ничего выполнить, но нужно проверить, видит ли его PowerShell в принципе. Если вы видите следующий результат, значит, все в порядке.

```
PS> Get-Module -Name PowerLab –ListAvailable

    Directory: C:\Program Files\WindowsPowerShell\Modules

ModuleType Version   Name                                ExportedCommands
---------- -------   ----                                ----------------
Script     1.0       PowerLab
```

Если модуль PowerLab не отображается в выводе, вернитесь к предыдущему этапу. Кроме того, убедитесь, что в папке C:\Program Files\WindowsPowerShell\Modules появилась папка PowerLab с файлами PowerLab.psm1 и PowerLab.psd1.

Автоматизация подготовки виртуальной среды

Теперь, когда мы создали базовую структуру модуля, пришло время сделать его функциональным. Поскольку задача создания сервера, такого как SQL или IIS, состоит из нескольких зависящих друг от друга этапов, мы сперва займемся автоматизацией создания виртуального коммутатора, виртуальной машины и виртуального диска. Затем мы автоматизируем развертывание операционной системы на этих виртуальных машинах и, наконец, установим на этих виртуальных машинах SQL Server и IIS.

Виртуальные коммутаторы

Прежде чем начать процесс автоматизации создания виртуальных машин, необходимо убедиться, что на хосте Hyper-V настроен *виртуальный коммутатор*: он позволяет виртуальным машинам связываться с клиентскими и с другими виртуальными машинами, созданными на хосте.

Создание виртуального коммутатора вручную

Наш виртуальный коммутатор будет *внешним,* с названием `PowerLab`. Скорее всего, коммутатора с таким именем на узле Hyper-V еще нет, но лучше на всякий случай убедиться в этом. Вы никогда не пожалеете о лишней проверке.

Чтобы найти все коммутаторы, настроенные на вашем хосте Hyper-V, используйте команду `Get-VmSwitch`. Убедившись в их отсутствии, создайте новый виртуальный коммутатор с помощью команды `New-VmSwitch`, указав имя (`PowerLab`) и тип коммутатора:

```
PS> New-VMSwitch -Name PowerLab -SwitchType External
```

Поскольку нам нужно, чтобы виртуальные машины могли взаимодействовать с хостами за пределами Hyper-V, передадим параметру `SwitchType` значение `External`. Любой, с кем вы поделитесь этим проектом, также должен будет создать внешний коммутатор.

После создания коммутатора пришло время добавить первую функцию модуля PowerLab.

Автоматизация создания коммутатора виртуальных машин

Первая функция PowerLab, которую мы назовем `New-PowerLabSwitch`, создаст коммутатор Hyper-V. Функция довольно простая: фактически, без нее вам

пришлось бы выполнить в командной строке лишь одну команду — `New-VmSwitch`. Но если вы заключите эту команду Hyper-V в пользовательскую функцию, то сможете выполнять дополнительные задачи, например, добавлять в коммутатор любую конфигурацию по умолчанию.

Я большой поклонник *идемпотентности* — этот мудреный термин означает, что «независимо от состояния, в котором выполняется команда, она постоянно выполняет одну и ту же задачу». В нашем примере, если задача создания коммутатора не является идемпотентной, запуск команды `New-VmSwitch` приведет к ошибке, если коммутатор уже существует.

Чтобы избавиться от необходимости вручную проверять его наличие перед созданием, можно использовать команду `Get-VmSwitch`. Затем, если (и только если) коммутатора не существует, стоит создать новый. Это позволяет запускать команду `New-PowerLabSwitch` в любой среде без ошибки, независимо от состояния хоста Hyper-V.

Откройте файл C:\Program Files\WindowsPowerShell\Modules\PowerLab\PowerLab.psm1 и создайте функцию `New-PowerLabSwitch`, как показано в листинге 15.2.

Листинг 15.2. Функция New-PowerLabSwitch модуля PowerLab

```
function New-PowerLabSwitch {
    param(
        [Parameter()]
        [string]$SwitchName = 'PowerLab',

        [Parameter()]
        [string]$SwitchType = 'External'
    )

    if (-not (Get-VmSwitch -Name $SwitchName -SwitchType $SwitchType -ErrorAction
    SilentlyContinue)) { ❶
        $null = New-VMSwitch -Name $SwitchName -SwitchType $SwitchType ❷
    } else {
        Write-Verbose -Message "The switch [$($SwitchName)] has already been
                                                                    created." ❸
    }
}
```

Эта функция сначала проверяет наличие коммутатора ❶. Если он не существует, функция создает его ❷. Если коммутатор уже создан, функция просто вернет сообщение в консоль ❸.

Сохраните модуль, а затем принудительно импортируйте его снова, используя команду `Import-Module -Name PowerLab -Force`.

Поскольку ранее мы уже импортировали модуль, PowerShell не загрузил в сеанс никаких функций. Когда вы добавляете в модуль новые функции, его нужно снова импортировать. Если модуль уже был импортирован, вам понадобится параметр `Force` для принудительного импорта. В противном случае PowerShell увидит, что модуль уже импортирован, и не станет выполнять команду.

После импорта модуля у вас появится функция `New-PowerLabSwitch`. Выполните эту команду:

```
PS> New-PowerLabSwitch –Verbose
VERBOSE: The switch [PowerLab] has already been created.
```

Обратите внимание, что выводится не текст ошибки, а полезное сообщение о том, что коммутатор уже создан. Это связано с тем, что мы передали функции необязательный параметр `Verbose`. Для параметров `SwitchName` и `SwitchType` были выбраны значения по умолчанию, поскольку они, как правило, идентичны.

Создание виртуальных машин

Теперь, когда мы настроили виртуальный коммутатор, пришло время создать виртуальную машину. Для примера создадим ВМ второго поколения с именем LABDC и 2 ГБ памяти. Машина будет подключена к вашему виртуальному коммутатору в папке C:\PowerLab\VMs на узле Hyper-V. Я выбрал имя LABDC, потому что в конечном счете мы получим контроллер домена Active Directory. Эта виртуальная машина станет контроллером домена, который мы будем использовать для готовой лаборатории.

Сначала посмотрим на уже имеющиеся ВМ и убедимся, что машин с таким именем нет. Поскольку вы уже знаете имя виртуальной машины, которую хотите создать, передайте это значение параметру `Name Get-Vm`:

```
PS> Get-Vm -Name LABDC
Get-Vm : A parameter is invalid. Hyper-V was unable to find a virtual machine with
name LABDC.
At line:1 char:1
+ Get-Vm -Name LABDC
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (LABDC:String) [Get-VM],
                              VirtualizationInvalidArgumentException
    + FullyQualifiedErrorId : InvalidParameter,Microsoft.HyperV.PowerShell.
Commands.GetVMCommand
```

Команда `Get-Vm` вернет ошибку, если не сможет найти виртуальную машину с указанным именем. Поскольку мы просто проверяем его наличие и результат

для нас не слишком важен, используем параметр `ErrorAction` со значением `SilentlyContinue`, чтобы команда ничего не возвращала при отсутствии ВМ. Для простоты обойдемся без блока `try/catch`.

Этот метод работает только в том случае, если команда возвращает незавершающую ошибку. При возвращении завершающей ошибки вам придется вернуть все объекты и отфильтровать их с помощью `Where-Object` либо заключить команду в блок `try/catch`.

Создание виртуальной машины вручную

Если виртуальной машины не существует, ее следует создать. Для этого нужно запустить команду `Get-Vm` и передать ей значения, которые вы определили в начале этого раздела.

```
PS> New-VM -Name 'LABDC' -Path 'C:\PowerLab\VMs'
-MemoryStartupBytes 2GB -Switch 'PowerLab' -Generation 2

Name  State CPUUsage(%) MemoryAssigned(M) Uptime   Status             Version
----  ----- ----------- ----------------- ------   ------             -------
LABDC Off   0           0                 00:00:00 Operating normally 8.0
```

Сейчас у вас уже должна быть виртуальная машина, но лучше перепроверить это, снова запустив команду `Get-Vm`.

Автоматизация создания ВМ

Чтобы автоматизировать создание простой виртуальной машины, вам снова нужно добавить еще одну функцию. Схема та же, что и при создании нового виртуального коммутатора: мы создаем идемпотентную функцию, которая выполняет задачу независимо от состояния хоста Hyper-V.

Введите функцию `New-PowerLabVm` в свой модуль PowerLab.psm1, как показано в листинге 15.3.

Листинг 15.3. Функция New-PowerLabVm в модуле PowerLab

```
function New-PowerLabVm {
    param(
        [Parameter(Mandatory)]
        [string]$Name,

        [Parameter()]
        [string]$Path = 'C:\PowerLab\VMs',

        [Parameter()]
```

```
        [string]$Memory = 4GB,

        [Parameter()]
        [string]$Switch = 'PowerLab',

        [Parameter()]
        [ValidateRange(1, 2)]
        [int]$Generation = 2
    )

❶  if (-not (Get-Vm -Name $Name -ErrorAction SilentlyContinue)) {
  ❷     $null = New-VM -Name $Name -Path $Path -MemoryStartupBytes $Memory
        -Switch $Switch -Generation $Generation
    } else {
  ❸      Write-Verbose -Message "The VM [$($Name)] has already been created."
    }
}
```

Эта функция проверяет существование ВМ ❶. Если ее нет, то машина будет создана ❷. Если ВМ уже существует, сообщение об этом выведется на консоль ❸.

Сохраните сценарий PowerLab.psm1 и выполните новую функцию с помощью следующей команды:

```
PS> New-PowerLabVm -Name 'LABDC' –Verbose
VERBOSE: The VM [LABDC] has already been created.
```

Опять же, с помощью этой команды можно создать виртуальную машину с указанными значениями параметров, независимо от того, существует ли эта ВМ (после того, как вы снова принудительно импортируете модуль) или нет.

Виртуальные жесткие диски

Теперь у вас есть подключенная к коммутатору виртуальная машина, но без связанного с ней хранилища от нее мало толку. Для решения этой проблемы необходимо создать локальный виртуальный жесткий диск (VHD) и подключить его к виртуальной машине.

ПРИМЕЧАНИЕ

В главе 16 мы возьмем готовый сценарий, который преобразует файл ISO в VHD. Следовательно, сам VHD создавать не придется. Но если вы не планируете автоматизировать развертывание операционной системы или вам нужно автоматизировать создание VHD как часть другого сценария, я все же рекомендую вам освоить этот раздел.

Создание VHD вручную

Чтобы создать VHD-файл, вам понадобится всего одна команда — `New-Vhd`. В этом разделе мы создадим VHD, размер которого можно будет увеличить до 50 ГБ, а для экономии места зададим динамический размер VHD.

Сначала на хосте нужно создать папку Hyper-V в `C:\PowerLab\VHDs`, где будет находиться VHD. Обязательно назовите свой виртуальный жесткий диск так же, как и виртуальную машину, к которой вы собираетесь его подключить. В дальнейшем это упростит задачу.

Создайте VHD с помощью команды `New-Vhd`:

```
PS> New-Vhd ❶-Path 'C:\PowerLab\VHDs\MYVM.vhdx' ❷-SizeBytes 50GB ❸-Dynamic

ComputerName            : HYPERVSRV
Path                    : C:\PowerLab\VHDs\LABDC.vhdx
VhdFormat               : VHDX
VhdType                 : Dynamic
FileSize                : 4194304
Size                    : 53687091200
MinimumSize             :
LogicalSectorSize       : 512
PhysicalSectorSize      : 4096
BlockSize               : 33554432
ParentPath              :
DiskIdentifier          : 3FB5153D-055D-463D-89F3-BB733B9E69BC
FragmentationPercentage : 0
Alignment               : 1
Attached                : False
DiskNumber              :
Number                  :
```

Передаем путь ❶ и размер VHD ❷ команде `New-Vhd` и указываем, что последний будет динамическим ❸.

С помощью команды `Test-Path` убедимся, что мы успешно создали VHD. Если `Test-Path` возвращает `True`, значит, все получилось:

```
PS> Test-Path -Path 'C:\PowerLab\VHDs\MYVM.vhdx'
True
```

Теперь вам нужно подключить виртуальный жесткий диск к вашей ВМ. С этим поможет команда `Add-VMHardDiskDrive`. Но поскольку мы не собираемся подключать VHD к LABDC (мы займемся автоматизацией развертывания в главе 16), создадим еще одну виртуальную машину под названием MYVM. Ее мы и подключим к виртуальному жесткому диску:

```
PS> New-PowerLabVm -Name 'MYVM'
PS> ❶Get-VM -Name MYVM | Add-VMHardDiskDrive -Path 'C:\PowerLab\VHDs\MYVM.vhdx'
PS> ❷Get-VM -Name MYVM | Get-VMHardDiskDrive

VMName ControllerType ControllerNumber ControllerLocation
------ -------------- ---------------- ------------------
MYVM   SCSI           0                0

DiskNumber Path
---------- ----
           C:\PowerLab\VHDs\LABDC.vhdx
```

Команда `Add-VMHardDiskDrive` через конвейер принимает тип объекта, который возвращает команда `Get-VM`, поэтому передать виртуальную машину можно напрямую из `Get-VM` в `Add-VMHardDiskDrive`, указав путь к виртуальному жесткому диску на хосте Hyper-V ❶.

Сразу после этого используйте команду `Get-VMHardDiskDrive`, чтобы убедиться в успешности создания VHDX ❷.

Автоматизация создания VHD

Для автоматизации создания виртуального жесткого диска и его подключения к ВМ можно добавить в модуль еще одну функцию. При создании сценариев или функций важно предусмотреть возможность различных конфигураций.

В листинге 15.4 определена функция `New-PowerLabVhd`, которая создает VHD и подключает к нему виртуальную машину.

Листинг 15.4. Функция New-PowerLabVhd в модуле PowerLab

```
function New-PowerLabVhd {
    param
    (
        [Parameter(Mandatory)]
        [string]$Name,

        [Parameter()]
        [string]$AttachToVm,

        [Parameter()]
        [ValidateRange(512MB, 1TB)]
        [int64]$Size = 50GB,

        [Parameter()]
        [ValidateSet('Dynamic', 'Fixed')]
        [string]$Sizing = 'Dynamic',
```

```
        [Parameter()]
        [string]$Path = 'C:\PowerLab\VHDs'
    )

    $vhdxFileName = "$Name.vhdx"
    $vhdxFilePath = Join-Path -Path $Path -ChildPath "$Name.vhdx"

    ### Ensure we don't try to create a VHD when there's already one there
    if (-not (Test-Path -Path $vhdxFilePath -PathType Leaf)) { ❶
        $params = @{
            SizeBytes = $Size
            Path = $vhdxFilePath
        }
        if ($Sizing -eq 'Dynamic') { ❷
            $params.Dynamic = $true
        } elseif ($Sizing -eq 'Fixed') {
            $params.Fixed = $true
        }

        New-VHD @params
        Write-Verbose -Message "Created new VHD at path [$($vhdxFilePath)]"
    }

    if ($PSBoundParameters.ContainsKey('AttachToVm')) {
        if (-not ($vm = Get-VM -Name $AttachToVm -ErrorAction SilentlyContinue)) { ❸
            Write-Warning -Message "The VM [$($AttachToVm)] does not exist. Unable
                                                                to attach VHD."
        } elseif (-not ($vm | Get-VMHardDiskDrive | Where-Object { $_.Path -eq
                                                        $vhdxFilePath })) { ❹
            $vm | Add-VMHardDiskDrive -Path $vhdxFilePath
            Write-Verbose -Message "Attached VHDX [$($vhdxFilePath)] to VM
                                                    [$($AttachToVM)]."
        } else { ❺
            Write-Verbose -Message "VHDX [$($vhdxFilePath)] already attached to VM
                                                                [$($AttachToVM)]."
        }
    }
}
```

Эта функция поддерживает как динамический, так и фиксированный размер ❷, а также учитывает четыре возможных варианта развития событий:

- VHD уже существует ❶.
- Виртуальная машина для подключения VHD не существует ❸.
- Виртуальная машина для подключения VHD создана, но он еще не подключен ❹.
- Виртуальная машина для подключения VHD существует, и он уже подключен ❺.

Дизайн функции — это вообще другая тема. Для создания сценария или функции, которая учитывала бы множество разных ситуаций, требуются годы практики. Это искусство, которое нельзя по-настоящему довести до совершенства. Однако если вы сможете заранее учесть как можно больше разных проблем и продумать способы их решения, ваша функция будет намного лучше. И все же не стоит тратить уйму времени на функцию или сценарий, проверяя каждую мелочь! Это ведь просто код. Вы всегда можете поправить что-то позже.

Выполнение функции New-PowerLabVhd

Вы можете выполнять этот код в различных состояниях, равно как и учитывать каждое из них. Давайте протестируем различные состояния, чтобы убедиться в работоспособности сценария автоматизации в каждой ситуации:

```
PS> New-PowerLabVhd -Name MYVM -Verbose -AttachToVm MYVM

VERBOSE: VHDX [C:\PowerLab\VHDs\MYVM.vhdx] already attached to VM [MYVM].

PS> Get-VM -Name MYVM | Get-VMHardDiskDrive | Remove-VMHardDiskDrive
PS> New-PowerLabVhd -Name MYVM -Verbose -AttachToVm MYVM

VERBOSE: Attached VHDX [C:\PowerLab\VHDs\MYVM.vhdx] to VM [MYVM].
PS> New-PowerLabVhd -Name MYVM -Verbose -AttachToVm NOEXIST

WARNING: The VM [NOEXIST] does not exist. Unable to attach VHD.
```

Формально, это не совсем тестирование. Мы просто заставляем функцию обработать все возможные варианты.

Тестирование новых функций с помощью Pester

Мы научились автоматизировать создание виртуальной машины Hyper-V. Важный момент: вы всегда должны строить тесты Pester для всего, что создаете. Так вы сможете убедиться, что все работает как задумано, а также отслеживать процесс автоматизации. Мы будем использовать тесты Pester для проверки всей работы, которую вы проделаете с оставшейся частью книги. Сами тесты Pester можно найти в прилагаемых к книге материалах: github.com/adbertram/PowerShellForSysadmins/.

В этой главе мы выполнили четыре задачи:

- создали виртуальный коммутатор;
- создали виртуальную машину;

- создали VHDX;
- подключили VHDX к виртуальной машине.

Я разделил тесты Pester на четыре группы, соответствующие этим достижениям. Подобный шаг помогает все упорядочить.

Давайте протестируем код, который мы написали в этой главе. Чтобы запустить тестовый сценарий, убедитесь, что вы скачали сценарий Automating-Hyper-V.Tests.ps1 из материалов книги. В приведенном ниже коде сценарий теста расположен в папке C:\, но в вашем случае путь может отличаться.

```
PS> Invoke-Pester 'C:\Automating-Hyper-V.Tests.ps1'
Describing Automating Hyper-V Chapter Demo Work
   Context Virtual Switch
    [+] created a virtual switch called PowerLab 195ms
   Context Virtual Machine
    [+] created a virtual machine called LABDC 62ms
   Context Virtual Hard Disk
    [+] created a VHDX called MYVM at C:\PowerLab\VHDs 231ms
    [+] attached the MYVM VHDX to the MYVM VM 194ms
Tests completed in 683ms
Passed: 4 Failed: 0 Skipped: 0 Pending: 0 Inconclusive: 0
```

Все четыре теста пройдены, так что мы можем переходить к следующей главе.

Итоги

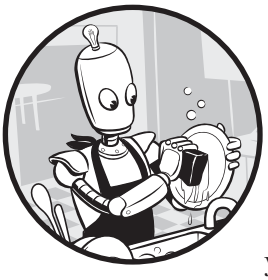
Мы создали основу для нашего первого крупного проекта автоматизации в PowerShell! Надеюсь, вы уже оценили, сколько времени вы можете сэкономить, автоматизируя с помощью PowerShell! Использовав бесплатный PowerShell-модуль от Microsoft, вы смогли всего лишь с парой команд создать виртуальный коммутатор, ВМ и VHD. В Microsoft нам предоставили команды, но строить логику вокруг них должны уже мы.

Теперь вы, пожалуй, поняли, что можно на лету создавать рабочие сценарии, но, если заранее подумать и подключить условную логику, они будут учитывать большее количество ситуаций.

В следующей главе мы возьмем только что созданную виртуальную машину и автоматизируем развертывание операционной системы на ней с помощью файла ISO и кое-чего еще.

16

Установка операционной системы



В предыдущей главе вы настроили и подготовили к работе модуль PowerLab. Теперь мы сделаем следующий шаг в нашем путешествии и научимся автоматизировать установку операционной системы. Поскольку у нас уже есть виртуальная машина и подключенный к ней VHD, остается установить на него Windows. Для этого возьмем ISO-файл Windows Server, сценарий Convert-WindowsImage.ps1 и много-много кода, чтобы добиться полностью автоматизированного развертывания Windows!

Исходные требования

Я надеюсь, что вы четко выполняли инструкции из предыдущей главы. В этой главе вам понадобится еще несколько вещей. Во-первых, поскольку мы хотим развернуть операционную систему, нам понадобится ISO-образ Windows Server 2016. Бесплатную пробную версию можно скачать по ссылке http://bit.ly/2r5TPRP после входа в систему с бесплатной учетной записью Microsoft.

Также, исходя из предыдущей главы, на вашем сервере Hyper-V должна быть папка C:\PowerLab. Теперь необходимо создать подпапку для образов ISO, C:\PowerLab\ISOs, и расположить в ней ISO-образ Windows Server 2016. На момент написания этой статьи ISO-файл назывался en_windows_server_2016_x64_dvd_9718492.iso. Мы будем использовать этот путь к файлу в своих сценариях,

поэтому, если ваш путь отличается, обязательно обновите код сценария соответствующим образом.

Также в папке модуля PowerLab должен быть PowerShell-сценарий Convert-WindowsImage.ps1. Если вы скачали прилагаемые к книге материалы, этот сценарий будет среди файлов к этой главе.

И еще несколько вещей напоследок: я рассчитываю, что у вас на сервере Hyper-V создана виртуальная машина LABDC. К ней мы привяжем вновь созданный виртуальный диск.

Наконец, в папке модуля PowerLab нужно будет разместить простой файл ответов XML (который также есть в материалах к этой главе) под названием unattend.xml.

Запустите, как и прежде, связанный с главой сценарий Prerequisites.Tests.ps1, чтобы убедиться в выполнении всех требований.

Развертывание ОС

Когда дело касается автоматизации развертывания ОС, работать придется с тремя основными компонентами:

- ISO-файл, содержащий саму ОС.
- Файл ответов, содержащий все данные, которые обычно вводятся во время установки вручную.
- Сценарий Microsoft PowerShell, который преобразует файл ISO в VHDX.

Ваша задача — объединить все эти компоненты. Большая часть сложной работы выполняется файлом ответов и сценарием преобразования ISO. А нам нужно создать небольшой сценарий, который будет запускать преобразование с соответствующими параметрами и подключать вновь созданный виртуальный жесткий диск к нужной виртуальной машине.

Этот сценарий в прилагаемых материалах называется Install-LABDCOperatingSystem.ps1.

Создание VHDX

У виртуальной машины LABDC будет динамическая таблица разделов GUID (*GUID Partition Table, GPT*) с дисковыми разделами VHDX объемом 40 ГБ,

работающая под управлением Windows Server 2016 Standard Core. Эта информация понадобится сценарию преобразования. Нам также потребуется знать путь к исходному ISO и к файлу ответов автоматической установки.

Сначала определим пути к файлам ISO и предварительно заполненным ответам:

```
$isoFilePath = 'C:\PowerLab\ISOs\en_windows_server_2016_x64_dvd_9718492.iso'
$answerFilePath = 'C:\PowerShellForSysAdmins\PartII\Automating Operating System
Installs\LABDC.xml'
```

Затем мы создадим параметры для сценария преобразования. Используя технику разбиения PowerShell, можно создать единую хеш-таблицу и определить все эти параметры как один. Использовать этот метод определения и передачи параметров командам намного удобнее, чем вводить их по одному:

```
$convertParams = @{
    SourcePath        = $isoFilePath
    SizeBytes         = 40GB
    Edition           = 'ServerStandardCore'
    VHDFormat         = 'VHDX'
    VHDPath           = 'C:\PowerLab\VHDs\LABDC.vhdx'
    VHDType           = 'Dynamic'
    VHDPartitionStyle = 'GPT'
    UnattendPath      = $answerFilePath
}
```

После определения всех параметров для сценария преобразования перейдем к исходному сценарию Convert-WindowsImage.ps1. Нам не нужно вызывать его напрямую, потому что он содержит функцию `Convert-WindowsImage`. Если бы вы просто выполнили сценарий Convert-WindowsImage.ps1, то ничего бы не произошло, потому что он просто загружал бы функцию внутри сценария.

Воспользуемся *dot sourcing* — способом предварительной загрузки функции в память для дальнейшего использования. В этом случае система загружает все функции, определенные в сценарии, но не выполняет их. Выполним загрузку на примере Convert-WindowsImage.pst1:

```
. "$PSScriptRoot\Convert-WindowsImage.ps1"
```

Взгляните на этот код. У нас появилась новая переменная — `$PSScriptRoot`. Это автоматическая переменная, в которой содержится путь к папке со сценарием. В этом примере, поскольку сценарий Convert-WindowsImage.ps1 находится в той же папке, что и модуль PowerLab, мы будем ссылаться на него в модуле PowerLab.

После загрузки сценария преобразования в сеанс можно вызывать функции, которые находились внутри него, включая `Convert-WindowsImage`. Эта функция будет делать за вас всю грязную работу: откроет файл ISO, соответствующим образом отформатирует новый виртуальный диск, установит загрузочный том, вставит предоставленный вами файл ответов и в итоге выдаст файл VHDX с готовой к загрузке копией Windows!

```
Convert-WindowsImage @convertParams

Windows(R) Image to Virtual Hard Disk Converter for Windows(R) 10
Copyright (C) Microsoft Corporation. All rights reserved.
Version 10.0.9000.0.amd64fre.fbl_core1_hyp_dev(mikekol).141224-3000 Beta

INFO   : Opening ISO en_windows_server_2016_x64_dvd_9718492.iso...
INFO   : Looking for E:\sources\install.wim...
INFO   : Image 1 selected (ServerStandardCore)...
INFO   : Creating sparse disk...
INFO   : Attaching VHDX...
INFO   : Disk initialized with GPT...
INFO   : Disk partitioned
INFO   : System Partition created
INFO   : Boot Partition created
INFO   : System Volume formatted (with DiskPart)...
INFO   : Boot Volume formatted (with Format-Volume)...
INFO   : Access path (F:\) has been assigned to the System Volume...
INFO   : Access path (G:\) has been assigned to the Boot Volume...
INFO   : Applying image to VHDX. This could take a while...
INFO   : Applying unattend file (LABDC.xml)...
INFO   : Signing disk...
INFO   : Image applied. Making image bootable...
INFO   : Drive is bootable. Cleaning up...

INFO   : Closing VHDX...
INFO   : Closing Windows image...
INFO   : Closing ISO...

INFO   : Done.
```

Использование готовых сценариев вроде Convert-WindowsImage.ps1 — отличный способ ускорить разработку. Они значительно экономят время, и к тому же их работе можно доверять. Если вам хочется знать, что конкретно делает этот сценарий, не стесняйтесь его открыть. Он делает уйму всего, и лично я благодарен возможности иметь подобный ресурс для автоматизации установки операционных систем.

Подключение виртуальной машины

Когда сценарий преобразования будет завершен, файл LABDC.vhdx будет расположен в папке C:\PowerLab\VHDs и готов к запуску. Но мы еще не закончили.

Сейчас диск не подключен к виртуальной машине. Вам нужно подключить его к существующей ВМ (а именно к ранее созданной машине LABDC). Как и в предыдущей главе, для подключения используем функцию `Add-VmHardDiskDrive`:

```
$vm = Get-Vm -Name 'LABDC'
Add-VMHardDiskDrive -VMName 'LABDC' -Path 'C:\PowerLab\VHDs\LABDC.vhdx'
```

Вам нужно загрузиться с этого диска, поэтому давайте убедимся, что приоритет загрузки указан правильно. Чтобы посмотреть порядок загрузки, выполните команду `Get-VMFirmware` и проверьте атрибут `BootOrder`:

```
$bootOrder = (Get-VMFirmware -VMName 'LABDC').Bootorder
```

Обратите внимание, что в порядке загрузки первоочередной является загрузка по сети. Это не совсем то, что нам нужно. Мы хотим, чтобы виртуальная машина загружалась с только что созданного диска.

```
$bootOrder.BootType

BootType
------
Network
```

Чтобы установить только что созданный VHDX в качестве первого загрузочного устройства, используйте команду `Set-VMFirmware` и параметр `FirstBootDevice`:

```
$vm | Set-VMFirmware -FirstBootDevice $vm.HardDrives[0]
```

К этому моменту у вас должна быть готовая виртуальная машина LABDC с подключенным виртуальным диском, с которого будет загружаться Windows. Запустите ВМ с помощью команды **`Start-VM -Name LABDC`** и убедитесь в выполнении загрузки. Если это так, то все готово!

Автоматизация развертывания ОС

Итак, вы успешно создали виртуальную машину LABDC, которая загружается на Windows. Важно понимать, что сценарий, который вы использовали, был специально разработан для вашей виртуальной машины. В настоящем мире такая роскошь бывает редко. Хороший сценарий — это тот, который можно повторно использовать без изменений в коде. Ваш сценарий должен работать с набором постоянно меняющихся значений параметров.

Давайте посмотрим на функцию `Install-PowerLabOperatingSystem` в модуле PowerLab из материалов к этой главе. Эта функция представляет собой хороший пример того, как превратить знакомый нам Install-LABDCOperatingSystem.ps1 в сценарий для развертывания операционных систем на разных виртуальных дисках с помощью простого изменения значений параметров.

Я не буду описывать здесь весь сценарий, поскольку мы уже рассмотрели большую часть его функций в предыдущем разделе, но стоит отметить пару аспектов. Во-первых, обратите внимание на то, что мы используем больше переменных: они позволяют сделать ваш сценарий более гибким благодаря использованию плейсхолдеров для значений вместо жесткого кодирования.

Также обратим внимание на условную логику. Взгляните на код из листинга 16.1: оператор `switch` находит путь к ISO-файлу по имени операционной системы. В предыдущем сценарии он нам не пригодился, ведь все уже было прописано.

Поскольку у функции `Install-PowerLabOperatingSystem` есть параметр `OperatingSystem`, она позволяет выполнить установку различных операционных систем. Вам просто нужно учесть все варианты. Отличный способ сделать это — использовать оператор `switch`, который позволяет легко добавить еще одно условие.

Листинг 16.1. Использование PowerShell-переключателя switch

```
switch ($OperatingSystem) {
    'Server 2016' {
        $isoFilePath = "$IsoBaseFolderPath\en_windows_server_2016_x64_dvd_
                                                                9718492.iso"
    }
    default {
        throw "Unrecognized input: [$_]"
    }
}
```

Заметно, как мы переместили жестко закодированные значения в параметры. Нельзя переоценить важность этого действия: именно параметры являются ключом к созданию универсальных и многоразовых сценариев. Избегайте жесткого кодирования при всякой возможности и всегда следите за значениями, которые вам придется менять во время выполнения (и используйте для них параметр!). Но вот вопрос: что, если нам нужно менять значения лишь изредка? Далее вы увидите, что у нескольких параметров существуют значения по умолчанию. Это позволяет вам статически устанавливать «типовые» значения, а затем при необходимости менять их.

```
param
(
    [Parameter(Mandatory)]
    [string]$VmName,

    [Parameter()]
    [string]$OperatingSystem = 'Server 2016',

    [Parameter()]
    [ValidateSet('ServerStandardCore')]
    [string]$OperatingSystemEdition = 'ServerStandardCore',

    [Parameter()]
    [string]$DiskSize = 40GB,

    [Parameter()]
    [string]$VhdFormat = 'VHDX',

    [Parameter()]
    [string]$VhdType = 'Dynamic',

    [Parameter()]
    [string]$VhdPartitionStyle = 'GPT',

    [Parameter()]
    [string]$VhdBaseFolderPath = 'C:\PowerLab\VHDs',

    [Parameter()]
    [string]$IsoBaseFolderPath = 'C:\PowerLab\ISOs',

    [Parameter()]
    [string]$VhdPath
)
```

С помощью функции `Install-PowerLabOperatingSystem` можно превратить все это в одну строку, которая поддерживает десятки конфигураций. Теперь у вас есть единая связная единица кода, которую вы можете вызывать различными способами, не меняя ни одной строчки сценария!

Хранение зашифрованных учетных данных на диске

Наш проект скоро будет готов, но, прежде чем продолжить, стоит немного отвлечься. Дело в том, что с помощью PowerShell нам надо делать нечто, требующее учетных данных. При написании сценария достаточно часто приходится хранить конфиденциальную информацию (например, комбинацию имени пользователя и пароля) в виде открытого текста внутри самого сценария.

Нередко считается, что раз это происходит в тестовой среде, в этом нет ничего страшного, — однако это создает опасный прецедент. Важно помнить о мерах безопасности даже во время тестирования и взять эту привычку прежде, чем переходить в производство.

Простой способ скрыть пароли в вашем сценарии — зашифровать их в файле; при необходимости ваш сценарий сможет ими воспользоваться. К счастью, для этого в PowerShell есть инструмент — Windows Data Protection API. Он используется командой `Get-Credential`, которая возвращает объект `PSCredential`.

Команда `Get-Credential` создает зашифрованную форму пароля, известную как *защищенная строка*. Весь объект учетных данных в этом формате может быть сохранен на диск с помощью команды `Export-CliXml`, и наоборот, вы можете прочитать объект `PSCredential` с помощью `Import-CliXml`. Эти команды позволяют легко управлять системой паролей.

При обработке учетных данных в PowerShell мы хотим хранить объекты `PSCredential` — типы объектов, которые принимает большинство параметров `Credential`. В предыдущих главах вы либо вводили имя пользователя и пароль в интерактивном режиме, либо сохраняли их в виде открытого текста. Но теперь, раз уж речь зашла о шифровании, давайте сделаем все как положено и добавим защиту учетных данных.

Для сохранения объекта `PSCredential` на диск в зашифрованном формате используется команда `Export-CliXml`. С помощью команды `Get-Credential` можно создать запрос имени пользователя и пароля, а затем передать результат `Export-CliXml`. Эта команда принимает путь для сохранения XML-файла, как показано в листинге 16.2.

Листинг 16.2. Экспорт учетных данных в файл

```
Get-Credential | Export-CliXml -Path C:\DomainCredential.xml
```

Если вы откроете XML-файл, он будет выглядеть примерно так:

```
<TN RefId="0">
  <T>System.Management.Automation.PSCredential</T>
  <T>System.Object</T>
  </TN>
  <ToString>System.Management.Automation.PSCredential</ToString>
  <Props>
  <S N="UserName">userhere</S>
  <SS N="Password">ENCRYPTEDTEXTHERE</SS>
  </Props>
  </Obj>
</Objs>
```

Теперь, когда мы сохранили учетные данные на диск, давайте вернем их в PowerShell. Используйте команду `Import-CliXml`, чтобы интерпретировать XML-файл и создать объект `PSCredential`:

```
$cred = Import-Clixml -Path C:\DomainCredential.xml
$cred | Get-Member


   TypeName: System.Management.Automation.PSCredential

Name                  MemberType Definition
----                  ---------- ----------
Equals                Method     bool Equals(System.Object obj)
GetHashCode           Method     int GetHashCode()
GetNetworkCredential Method      System.Net.NetworkCredential
                                 GetNetworkCredential()
GetObjectData         Method     void GetObjectData(System.Runtime...
GetType               Method     type GetType()
ToString              Method     string ToString()
Password              Property   securestring Password {get;}
UserName              Property   string UserName {get;}
```

Код написан таким образом, что вам остается лишь передать переменную `$cred` любому параметру `Credential`. Теперь программа будет работать так же, как если бы вы просто интерактивно вводили логин и пароль. Этот метод весьма прост, но обычно им не пользуются в производственной среде: пользователь, зашифровавший текст, также должен и расшифровывать его (а шифрование должно работать по-другому!). Это требование вообще-то ограничивает возможности масштабирования. Но вместе с тем, в тестовой среде все отлично работает!

PowerShell Direct

Давайте вернемся к нашему проекту. Обычно в PowerShell при запуске команды на удаленных компьютерах приходится использовать удаленное управление PowerShell. Для этого необходимо наличие сетевого подключения между локальным и удаленным хостами. Было бы здорово иметь способ упростить эту настройку и не думать о подключении к сети, не правда ли? Ну, вообще-то, это возможно!

Поскольку вы запускаете весь свой процесс автоматизации на узле Windows Server 2016 Hyper-V, в вашем распоряжении есть одна полезная функция — *PowerShell Direct*. Эта новая функция PowerShell позволяет запускать команды *без подключения к сети* на любых виртуальных машинах, расположенных

на сервере Hyper-V. Нет необходимости заранее настраивать сетевые адаптеры на ВМ (хотя вы уже сделали это с помощью XML-файла в процессе установки).

Вместо использования полного сетевого стека мы попробуем ради удобства воспользоваться PowerShell Direct. В противном случае, поскольку вы находитесь в среде рабочей группы, вам пришлось бы настроить в ней удаленное управление PowerShell, а это непростая задача (см. руководство: http://bit.ly/2D3deUX).

Чтобы инициировать команду на удаленном компьютере через удаленное управление PowerShell, используйте команду `Invoke-Command` с параметрами `ComputerName` и `ScriptBlock`:

```
Invoke-Command -ComputerName LABDC -ScriptBlock { hostname }
```

Однако при использовании PowerShell Direct уже знакомый параметр `ComputerName` превращается в `VMName`, а также добавляется параметр `Credential`. Через PowerShell Direct будет выполняться та же команда, но только с самого хоста Hyper-V. Чтобы упростить задачу, сохраним объект `PSCredential` на диске, чтобы вам не приходилось постоянно запрашивать учетные данные.

В этом примере возьмем имя пользователя `powerlabuser` и пароль `P@$$w0rd12`:

```
Get-Credential | Export-CliXml -Path C:\PowerLab\VMCredential.xml
```

Мы сохранили учетные данные на диск, и теперь их нужно расшифровать и передать в `Invoke-Command`. Давайте считаем учетные данные из VMCredential.xml, а затем воспользуемся ими для выполнения кода на виртуальной машине LABDC:

```
$cred = Import-CliXml -Path C:\PowerLab\VMCredential.xml
Invoke-Command -VMName LABDC -ScriptBlock { hostname } -Credential $cred
```

Вообще, под капотом у PowerShell Direct скрывается гораздо больше, но я не буду здесь вдаваться в подробности. Чтобы получить полное представление о работе PowerShell Direct, рекомендую заглянуть в блог Microsoft, посвященный этой функции (https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/user-guide/powershell-direct).

Тестирование с помощью Pester

Настало время для самой важной части главы: давайте объединим все это и добавим тесты Pester! Работать будем по схеме предыдущей главы, но здесь я хочу

обратить ваше внимание на особенность некоторых тестов. В тестах Pester вы будете использовать блоки `BeforeAll` и `AfterAll` (листинг 16.3).

Как следует из названия, код из блока `BeforeAll` выполняется перед всеми тестами, а из `AfterAll` — после. Эти блоки нам нужны для многоразового подключения к серверу LABDC через PowerShell Direct. Удаленное управление PowerShell и PowerShell Direct поддерживают концепцию сеанса, о которой мы говорили в первой части книги (глава 8). Вместо использования `Invoke-Command` для сборки и разрыва нескольких сеансов лучше заранее определить один сеанс и использовать его многократно.

Листинг 16.3. Tests.ps1 — блоки BeforeAll и AfterAll

```
BeforeAll {
    $cred = Import-CliXml -Path C:\PowerLab\VMCredential.xml
    $session = New-PSSession -VMName 'LABDC' -Credential $cred
}

AfterAll {
    $session | Remove-PSSession
}
```

Заметно, что сохраненные учетные данные расшифровываются внутри блока `BeforeAll`. После создания учетных данных вы передаете их, а также имя виртуальной машины, команде `New-PSSession`. Это та же команда, что и в главе 8, но здесь мы в качестве параметра используем параметр `VMName` вместо `ComputerName`.

В результате будет создан единый удаленный сеанс, который можно использовать повторно во время тестирования. После завершения всех тестов Pester заглянет в блок `AfterAll` и удалит сеанс. Такой подход намного эффективнее, чем многократное создание сеанса, особенно если у вас есть десятки или сотни тестов, для которых требуется удаленный запуск кода.

Остальная часть сценария в материалах к этой главе достаточно проста и следует этому шаблону. Как видите, все тесты Pester оказались положительными, а это значит, что вы на правильном пути!

```
PS> Invoke-Pester 'C:\PowerShellForSysadmins\Part II\Automating Operating
System Installs\Automating Operating System Installs.Tests.ps1'
Describing Automating Operating System Installs
   Context Virtual Disk
    [+] created a VHDX called LABDC in the expected location 305ms
    [+] attached the virtual disk to the expected VM 164ms
    [+] creates the expected VHDX format 79ms
    [+] creates the expected VHDX partition style 373ms
    [+] creates the expected VHDX type 114ms
```

```
    [+] creates the VHDDX of the expected size 104ms
   Context Operating System
    [+] sets the expected IP defined in the unattend XML file 1.07s
    [+] deploys the expected Windows version 65ms
Tests completed in 2.28s
Passed: 8 Failed: 0 Skipped: 0 Pending: 0 Inconclusive: 0
```

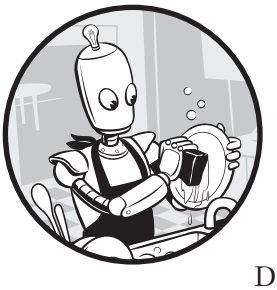
Итоги

В этой главе мы еще немного углубились в наш проект. Мы использовали виртуальную машину из предыдущей главы и развернули на ней операционную систему вручную и автоматически. На этом этапе у нас готова полностью функционирующая виртуальная машина Windows.

В следующей главе мы настроим Active Directory (AD) на своей виртуальной машине LABDC. В результате будет создан новый лес и домен AD, к которому под конец сеанса будет присоединено еще больше серверов.

17

Развертывание Active Directory



В этой главе мы воспользуемся информацией из нескольких последних глав части II и начнем разворачивать службы поверх виртуальных машин. Поскольку многим сервисам необходимо наличие Active Directory, сначала необходимо развернуть лес и домен Active Directory. Лес и домен AD будут выполнять задачи аутентификации и авторизации в оставшихся главах.

Я предполагаю, что вы прочитали предыдущие главы и настроили виртуальную машину LABDC, потому что именно ее мы и будем использовать для полной автоматизации леса Active Directory и заполнения тестовыми пользователями и группами.

Исходные требования

Нам понадобятся результаты нашей работы из главы 16, так что я надеюсь, что у вас уже настроена виртуальная машина LABDC, выполнена сборка с автоматизацией XML и загружен Windows Server 2016. Если это так, то все в порядке. Если нет, то в этой главе есть примеры автоматизации Active Directory. Но предупреждаю, что в таком случае вы не сможете повторить инструкции точь-в-точь.

Как всегда, запустите предварительный тест Pester, чтобы убедиться в выполнении всех исходных требований для этой главы.

Создание леса Active Directory

Хорошая новость заключается в том, что создать лес AD с помощью PowerShell довольно просто. Нам нужны всего две команды — `Install-WindowsFeature` и `Install-ADDSForest`. С их помощью вы можете построить лес и домен и подготовить сервер Windows на роль контроллера домена.

Поскольку мы будем использовать лес в лабораторной среде, нам понадобятся организационные единицы, пользователи и группы. Нахождение в лабораторной среде означает, что у вас нет производственных объектов для работы. Гораздо проще создать множество объектов, имитирующих производственные, чем пытаться синхронизировать реальные объекты с лабораторной средой.

Создаем лес

Первое, что вам нужно сделать при построении нового леса AD — это создать *контроллер домена*, краеугольный камень Active Directory. Чтобы создать работающую среду AD, нужно завести хотя бы один контроллер домена.

Поскольку у нас лабораторная среда, нам нужен всего один контроллер домена. В реальной ситуации вам понадобятся как минимум два из них (про запас). В нашей ситуации в лабораторной среде нет данных — их нужно придумывать с нуля, — поэтому нам хватит одного контроллера. Перед началом необходимо установить Windows-компонент `AD-Domain-Services` на вашем сервере LABDC с помощью команды `Install-WindowsFeature`:

```
PS> $cred = Import-CliXml -Path C:\Files.xml
PS> Invoke-Command -VMName 'LABDC' -Credential $cred -ScriptBlock
{ Install-windowsfeature -Name AD-Domain-Services }
PSComputerName : LABDC RunspaceId : 33d41d5e-50f3-475e-a624-4cc407858715
Success : True RestartNeeded : No FeatureResult : {Active Directory Domain
Services, Remote Server Administration Tools, Active Directory module for
Windows PowerShell, AD DS and AD LDS Tools...} ExitCode : Success ```
```

После предоставления учетных данных для подключения к серверу используем команду `Invoke-Command` для удаленного запуска команд `Install-WindowsFeature` на удаленном сервере.

После установки компонента с помощью команды `Install-ADDSForest` можно создать лес. Эта команда является частью PowerShell-модуля `ActiveDirectory`, который вы уже установили на LABDC.

Команда `Install-ADDSForest` — единственная, которая вам нужна для создания леса. У нее еще есть параметры, которые мы зададим с помощью кода, но

обычно их вводят с помощью графического интерфейса. Наш лес будет называться `powerlab.local`. Поскольку контроллером домена является Windows Server 2016, следует установить значение `WinThreshold` для режимов домена и леса. Все доступные значения параметров `DomainMode` и `ForestMode` можно посмотреть на странице документации Microsoft *Install-ADDSForest* (http://bit.ly/2rrgUi6).

Сохранение защищенных строк на диск

В главе 16, когда вам потребовались учетные данные, вы сохраняли их в объекты `PSCredential`, а затем повторно использовали их в своих командах. Но на этот раз нам не нужен объект `PSCredential`: нам хватит одной зашифрованной строки.

В этом разделе нам понадобится передать команде пароль администратора безопасного режима. Как и в случае с любой другой конфиденциальной информацией, необходимо использовать шифрование. Для сохранения и извлечения объектов PowerShell из файловой системы вы будете использовать команды `Export-CliXml` и `Import-CliXml`. Однако здесь вместо вызова `Get-Credential` мы создадим безопасную строку с помощью команды `ConvertTo-SecureString`, а затем сохраним этот объект в файл.

Чтобы сохранить зашифрованный пароль в файл, передайте его в виде обычного текста в `ConvertTo-SecureString`, а затем экспортируйте этот защищенный строковый объект в `Export-CliXml`, создав файл, на который можно ссылаться позже:

```
PS> 'P@$$w0rd12' | ConvertTo-SecureString -Force -AsPlainText
| Export-Clixml -Path C:\PowerLab\SafeModeAdministratorPassword.xml
```

Итак, после сохранения пароля администратора безопасного режима на диск его можно считать с помощью `Import-CliXml` и передать все остальные параметры, которые необходимо запустить с `Install-ADDSForest`. Мы сделаем это с помощью следующего кода:

```
PS> $safeModePw = Import-CliXml -Path C:\PowerLab\
SafeModeAdministratorPassword.xml
PS> $cred = Import-CliXml -Path C:\PowerLab\VMCredential.xml
PS> $forestParams                 = @{
>>> DomainName                    = 'powerlab.local' ❶
>>> DomainMode                    = 'WinThreshold' ❷
>>> ForestMode                    = 'WinThreshold'
>>> Confirm                       = $false ❸
```

```
>>> SafeModeAdministratorPassword = $safeModePw ❹
>>> WarningAction                 = 'Ignore ❺
>>>}
PS> Invoke-Command -VMName 'LABDC' -Credential $cred -ScriptBlock { $null =
Install-ADDSForest @using:forestParams }
```

Здесь мы создаем лес и домен под названием powerlab.local ❶, работающий на функциональном уровне Windows Server 2016 (`WinThreshold`) ❷. Затем мы обходим все подтверждения ❸, передаем пароль администратора безопасного режима ❹ и игнорируем возникающие нерелевантные сообщения с предупреждениями ❺.

Автоматизация создания леса

Теперь, когда мы сделали все вручную, создадим в модуле PowerLab функцию, которая будет автоматически обрабатывать создание леса AD. Позже ее можно будет использовать и в других средах.

В модуле PowerLab, включенном в материалы этой главы, вы увидите функцию `New-PowerLabActiveDirectoryForest`, которая приведена в листинге 17.1.

Листинг 17.1. Функция New-PowerLabActiveDirectoryForest

```
function New-PowerLabActiveDirectoryForest {
    param(
        [Parameter(Mandatory)]
        [pscredential]$Credential,

        [Parameter(Mandatory)]
        [string]$SafeModePassword,

        [Parameter()]
        [string]$VMName = 'LABDC',

        [Parameter()]
        [string]$DomainName = 'powerlab.local',

        [Parameter()]
        [string]$DomainMode = 'WinThreshold',

        [Parameter()]
        [string]$ForestMode = 'WinThreshold'
    )

    Invoke-Command -VMName $VMName -Credential $Credential -ScriptBlock {

        Install-windowsfeature -Name AD-Domain-Services

        $forestParams = @{
            DomainName                    = $using:DomainName
```

```
            DomainMode                    = $using:DomainMode
            ForestMode                    = $using:ForestMode
            Confirm                       = $false
            SafeModeAdministratorPassword = (ConvertTo-SecureString
                                             -AsPlainText -String $using:
                                             SafeModePassword -Force)
            WarningAction                 = 'Ignore'
        }
        $null = Install-ADDSForest @forestParams
    }
}
```

Как и в предыдущей главе, здесь мы определяем несколько параметров, которые будем использовать для передачи в команду `Install-ADDSForest` модуля `ActiveDirectory`. Обратите внимание, что эти два параметра учетных данных и пароля являются обязательными (`Mandatory`). Как следует из названия, пользователь должен задать эти параметры самостоятельно (прочие параметры имеют значения по умолчанию, поэтому пользователю не обязательно передавать их). С помощью этой функции вы можете прочитать сохраненный пароль администратора и учетные данные, а затем передать их в функцию:

```
PS> $safeModePw = Import-CliXml -Path C:\PowerLab\SafeModeAdministratorPassword.xml
PS> $cred = Import-CliXml -Path C:\PowerLab\VMCredential.xml
PS> New-PowerLabActiveDirectoryForest -Credential $cred
                                      -SafeModePassword $safeModePw
```

После запуска этого кода у вас будет полностью рабочий лес AD! И все же, давайте найдем способ подтвердить, что все работает как надо. Как вариант, можно запросить все учетные записи пользователей в домене по умолчанию. Однако для этого необходимо создать другой объект `PSCredential` на диске. Поскольку LABDC теперь является контроллером домена, вместо учетной записи локального пользователя потребуется учетная запись пользователя домена. Мы создадим и сохраним данные с именем пользователя из `powerlab.local\administrator` и паролем из `P@$$w0rd12` в файле C:\PowerLab\DomainCredential.xml. Помните, что вам нужно сделать это только один раз. Затем вы сможете использовать новые учетные данные домена для подключения к LABDC:

```
PS> Get-Credential | Export-CliXml -Path C:\PowerLab\DomainCredential.xml
```

После создания учетных данных домена добавим в наш модуль PowerLab еще одну функцию под названием `Test-PowerLabActiveDirectoryForest`. Сейчас эта функция просто опрашивает всех пользователей в домене, но поскольку эта задача организована в функцию, вы можете настроить этот тест на свое усмотрение:

```
function Test-PowerLabActiveDirectoryForest {
    param(
        [Parameter(Mandatory)]
        [pscredential]$Credential,

        [Parameter()]
        [string]$VMName = 'LABDC'
    )

    Invoke-Command -Credential $Credential -ScriptBlock {Get-AdUser -Filter * }
}
```

Попробуйте выполнить функцию `Test-PowerLabActiveDirectoryForest`, используя учетные данные домена и имя виртуальной машины LABDC. Если в выводе будет несколько учетных записей пользователей, то поздравляю — все работает! Теперь вы успешно настроили контроллер домена и сохранили учетные данные для подключения к виртуальным машинам в рабочей группе (и к любым виртуальным машинам, подключенным к домену, на будущее).

Заполнение домена

В предыдущем разделе мы настроили контроллер домена в PowerLab. Давайте теперь создадим несколько тестовых объектов. Поскольку это тестовая среда, нам нужно самостоятельно создать различные объекты (подразделения, пользователи, группы и т. д.) для проверки базы данных. Можно запустить отдельную команду для создания каждого отдельного объекта, но это будет непрактично. Гораздо удобнее было бы определить все это в одном файле, прочитать каждый объект и создать их все за один раз.

Работа с таблицей объектов

В качестве исходного файла будем использовать электронную таблицу Excel для определения входных данных — она есть в прилагаемых к этой главе материалах. В таблице есть два листа: Users (рис. 17.1) и Groups (рис. 17.2).

Каждая строка в этих листах соответствует пользователю или группе, которую необходимо создать, а также содержит информацию для передачи в PowerShell. Как было сказано в главе 10, встроенная оболочка PowerShell не может обрабатывать электронные таблицы Excel без вашего участия. Однако с помощью готового модуля мы сделаем все проще. С модулем `ImportExcel` можно считывать электронные таблицы Excel столь же легко, как и CSV-файлы. Загрузите его из PowerShell Gallery с помощью команды **`Install-Module -Name ImportExcel`**. После нескольких запросов безопасности модуль готов к использованию.

	A	B	C	D	E
1	OUName	UserName	FirstName	LastName	MemberOf
2	PowerLab Users	jjones	Joe	Jones	Accounting
3	PowerLab Users	abertram	Adam	Bertram	Accounting
4	PowerLab Users	jhicks	Jeff	Hicks	Accounting
5	PowerLab Users	dtrump	Donald	Trump	Human Resources
6	PowerLab Users	alincoln	Abraham	Lincoln	Human Resources
7	PowerLab Users	bobama	Barack	Obama	Human Resources
8	PowerLab Users	tjefferson	Thomas	Jefferson	IT
9	PowerLab Users	bclinton	Bill	Clinton	IT
10	PowerLab Users	gbush	George	Bush	IT
11	PowerLab Users	rreagan	Ronald	Reagan	IT

Рис. 17.1. Таблица Users

	A	B	C
1	OUName	GroupName	Type
2	PowerLab Groups	Accounting	DomainLocal
3	PowerLab Groups	Human Resources	DomainLocal
4	PowerLab Groups	IT	DomainLocal

Рис. 17.2. Таблица Groups

Теперь воспользуемся командой `Import-Excel` для анализа таблиц:

```
PS> Import-Excel -Path 'C:\Program Files\WindowsPowerShell\Modules\PowerLab\
ActiveDirectoryObjects.xlsx' -WorksheetName Users | Format-Table -AutoSize
OUName         UserName   FirstName LastName  MemberOf
------         --------   --------- --------  --------
PowerLab Users jjones     Joe       Jones     Accounting
PowerLab Users abertram   Adam      Bertram   Accounting
PowerLab Users jhicks     Jeff      Hicks     Accounting
PowerLab Users dtrump     Donald    Trump     Human Resources
PowerLab Users alincoln   Abraham   Lincoln   Human Resources
PowerLab Users bobama     Barack    Obama     Human Resources
PowerLab Users tjefferson Thomas    Jefferson IT
PowerLab Users bclinton   Bill      Clinton   IT
PowerLab Users gbush      George    Bush      IT
PowerLab Users rreagan    Ronald    Reagan    IT

PS> Import-Excel -Path 'C:\Program Files\WindowsPowerShell\Modules\PowerLab\
ActiveDirectoryObjects.xlsx' -WorksheetName Groups | Format-Table -AutoSize

OUName          GroupName       Type
------          ---------       ----
PowerLab Groups Accounting      DomainLocal
PowerLab Groups Human Resources DomainLocal
PowerLab Groups IT              DomainLocal
```

С помощью параметров `Path` и `WorksheetName` можно легко извлечь необходимые данные. Обратите внимание на команду `Format-Table`: она заставляет PowerShell отображать вывод в табличном формате. Параметр `AutoSize` сообщает PowerShell, что нужно попытаться «уложить» каждую строку в одну строку консоли.

Разработка плана

Теперь вы можете считывать данные из электронной таблицы Excel. Осталось понять, что с ними делать. В модуле PowerLab мы создадим функцию, которая считывает каждую строку и выполняет требуемое действие. Весь описываемый здесь код доступен с помощью `New-PowerLabActive DirectoryTestObject` в соответствующем модуле PowerLab.

Эта функция немного сложнее наших предыдущих сценариев, поэтому давайте разберем ее нестандартным образом, чтобы позже вы могли обращаться к этой информации. Этот шаг может показаться неважным, но в случае с более крупными функциями предварительное планирование значительно облегчит вам работу в долгосрочной перспективе. В этой функции вам необходимо сделать следующее:

1. Считать оба листа в электронной таблице Excel и получить все строки пользователей и групп.
2. Прочитать каждую строку на обоих листах и проверить, существует ли OU (подразделение), частью которого должен быть пользователь или группа.
3. Если OU не существует, его нужно создать.
4. Если пользователь/группа не существует, их нужно создать.
5. Добавить пользователя в указанную группу (этот шаг выполняется только для пользователя).

Теперь, когда у нас есть этот неформальный план, давайте приступим к написанию кода.

Создание AD-объектов

Для начала не будем все усложнять: сосредоточимся на обработке одного объекта. Нам незачем сейчас думать обо всем сразу. Ранее вы уже устанавливали Windows-компонент `AD-Domain-Services` на LABDC, поэтому у вас уже установлен модуль `ActiveDirectory`. Как вы убедились при изучении главы 11, в этом модуле есть большой набор полезных команд . Напомним, что многие команды следуют тому же соглашению об именах, что и `Get/Set/New-AD`.

Давайте откроем пустой сценарий .ps1 и приступим к работе. Начнем с написания всех необходимых команд (листинг 17.2) на основе предыдущего плана.

Листинг 17.2. Пишем код для проверки и создания новых пользователей и групп

```
Get-ADOrganizationalUnit -Filter "Name -eq 'OUName'" ❶
New-ADOrganizationalUnit -Name OUName ❷

Get-ADGroup -Filter "Name -eq 'GroupName'" ❸
New-ADGroup -Name GroupName -GroupScope GroupScope -Path "OU=OUName,DC=powerlab,
                                                          DC=local" ❹

Get-ADUser -Filter "Name -eq 'UserName'" ❺
New-ADUser -Name $user.UserName -Path "OU=$($user.OUName),DC=powerlab,DC=local" ❻

UserName -in (Get-ADGroupMember -Identity GroupName).Name ❼
Add-ADGroupMember -Identity GroupName -Members UserName ❽
```

Из нашего плана следует, что сначала нужно проверить, существует ли OU ❶, и при отсутствии таковой создать ее ❷. Будем делать то же самое с каждой группой: проверять ее наличие❸ и создавать ее, если она отсутствует ❹. Сделаем то же самое и для каждого пользователя: проверим наличие ❺ и создадим его ❻. У пользователей дополнительно проверяем, входят ли они в группу, имеющуюся в электронной таблице ❼, и по необходимости добавляем их в эту группу ❽.

Не хватает лишь условной структуры, которую мы добавим в листинге 17.3.

Листинг 17.3. Создание пользователей и групп в случае, если их еще нет

```
if (-not (Get-ADOrganizationalUnit -Filter "Name -eq 'OUName'")) {
    New-ADOrganizationalUnit -Name OUName
}

if (-not (Get-ADGroup -Filter "Name -eq 'GroupName'")) {
    New-ADGroup -Name GroupName -GroupScope GroupScope -Path "OU=OUName,
                                                DC=powerlab,DC=local"
}

if (-not (Get-ADUser -Filter "Name -eq 'UserName'")) {
    New-ADUser -Name $user.UserName -Path "OU=OUName,DC=powerlab,DC=local"
}

if (UserName -notin (Get-AdGroupMember -Identity GroupName).Name) {
    Add-ADGroupMember -Identity GroupName -Members UserName
}
```

Теперь, когда ваш код делает все, что нужно для отдельного пользователя или группы, вам нужно выяснить, как сделать это для всех. Сначала необходимо

считать рабочие листы. Вы уже видели подходящие команды, осталось лишь сохранить все эти строки в переменных. Технически это не требуется, но код таким образом получается более внятным. Мы будем использовать циклы `foreach` для чтения всех пользователей и групп, как показано в листинге 17.4.

Листинг 17.4. Код, проходящий по каждой строке листа Excel

```
$users = Import-Excel -Path 'C:\Program Files\WindowsPowerShell\Modules\
PowerLab\ActiveDirectoryObjects.xlsx' -WorksheetName Users
$groups = Import-Excel -Path 'C:\Program Files\WindowsPowerShell\Modules\
PowerLab\ActiveDirectoryObjects.xlsx' -WorksheetName Groups

foreach ($group in $groups) {

}

foreach ($user in $users) {

}
```

Теперь, когда у вас есть структура для обхода каждой строки, давайте добавим в нее алгоритм для каждой из строк, как показано в листинге 17.5.

Листинг 17.5. Выполнение задач для всех пользователей и групп

```
$users = Import-Excel -Path 'C:\Program Files\WindowsPowerShell\Modules\PowerLab\
ActiveDirectoryObjects.xlsx' -WorksheetName Users
$groups = Import-Excel -Path 'C:\Program Files\WindowsPowerShell\Modules\PowerLab\
ActiveDirectoryObjects.xlsx' -WorksheetName Groups

foreach ($group in $groups) {
    if (-not (Get-ADOrganizationalUnit -Filter "Name -eq '$($group.OUName)'")) {
        New-ADOrganizationalUnit -Name $group.OUName
    }
    if (-not (Get-ADGroup -Filter "Name -eq '$($group.GroupName)'")) {
        New-ADGroup -Name $group.GroupName -GroupScope $group.Type
        -Path "OU=$($group.OUName),DC=powerlab,DC=local"
    }
}

foreach ($user in $users) {
    if (-not (Get-ADOrganizationalUnit -Filter "Name -eq '$($user.OUName)'")) {
        New-ADOrganizationalUnit -Name $user.OUName
    }
    if (-not (Get-ADUser -Filter "Name -eq '$($user.UserName)'")) {
        New-ADUser -Name $user.UserName -Path "OU=$($user.
                            OUName),DC=powerlab,DC=local"
    }
    if ($user.UserName -notin (Get-ADGroupMember -Identity $user.MemberOf).Name) {
        Add-ADGroupMember -Identity $user.MemberOf -Members $user.UserName
    }
}
```

Почти все! Сценарий готов к работе, но теперь вам нужно запустить его на сервере LABDC. Поскольку мы не будем запускать этот код непосредственно на самой виртуальной машине LABDC, вам нужно запаковать все это в блок сценария и позволить команде `Invoke-Command` запустить его удаленно на LABDC. Поскольку мы хотим создать и заполнить лес за один раз, возьмем весь «рабочий» код и переместим его в функцию `New-PowerLabActiveDirectoryTestObject`. Вы можете загрузить копию этой готовой функции из прилагаемых к этой главе файлов.

Сборка и запуск тестов Pester

Сейчас у нас есть весь код, который нужен для создания нового леса AD и его заполнения. Теперь создадим несколько тестов Pester и убедимся, что все идет по плану. Вам предстоит многое проверить, поэтому здесь тесты Pester будут более сложными, чем прежде. Как и перед созданием сценария `New-PowerLab ActiveDirectoryTestObject.ps1`, сначала создадим сценарий тестирования Pester, а затем подумаем, какими должны быть тестовые примеры. Если вам нужно больше узнать о Pester, перечитайте главу 9. Я также включил все тесты Pester для этой главы в прилагаемые файлы.

Что конкретно нужно протестировать? В этой главе мы сделали следующее:

- Создали новый лес AD.
- Создали новый домен AD.
- Создали пользователей AD.
- Создали группы AD.
- Создали организационные подразделения AD.

После проверки наличия всех этих элементов вам необходимо убедиться, что у них правильные атрибуты (атрибуты, переданные в качестве параметров командам, которые их создали). Вот то, что вам нужно.

Таблица 17.1. Атрибуты AD

Объект	Атрибуты
Лес AD	`DomainName`, `DomainMode`, `ForestMode`, пароль администратора безопасного режима
Пользователь AD	Путь к OU, имя, член группы
Группа AD	Путь к OU, имя
Организационное подразделение AD	Имя

Теперь у нас есть некий набросок того, что мы хотим обнаружить с помощью тестов Pester. Если вы посмотрите на сценарий Creating an Active Directory Forest.Tests.ps1, то увидите, что я решил разбить каждую из этих сущностей на контексты и протестировать все связанные атрибуты как отдельные тесты.

Чтобы дать вам представление о том, как создаются эти тесты, в листинге 17.6 я привел фрагмент тестового кода.

Листинг 17.6. Часть кода теста Pester

```
context 'Domain' {
❶  $domain = Invoke-Command -Session $session -ScriptBlock { Get-AdDomain }
    $forest = Invoke-Command -Session $session -ScriptBlock { Get-AdForest }

❷  it "the domain mode should be Windows2016Domain" {
        $domain.DomainMode | should be 'Windows2016Domain'
    }

    it "the forest mode should be WinThreshold" {
        $forest.ForestMode | should be 'Windows2016Forest'
    }

    it "the domain name should be powerlab.local" {
        $domain.Name | should be 'powerlab'
    }
}
```

В этом контексте мы хотим убедиться, что домен и лес AD созданы правильно. Итак, сначала мы создаем домен и лес ❶, а затем проверяем правильность их атрибутов ❷.

Запуск всего теста должен выдать вам что-то вроде этого:

```
Describing Active Directory Forest
   Context Domain
    [+] the domain mode should be Windows2016Domain 933ms
    [+] the forest mode should be WinThreshold 25ms
    [+] the domain name should be powerlab.local 41ms
   Context Organizational Units
    [+] the OU [PowerLab Users] should exist 85ms
    [+] the OU [PowerLab Groups] should exist 37ms
   Context Users
    [+] the user [jjones] should exist 74ms
    [+] the user [jjones] should be in the [PowerLab Users] OU 35ms
    [+] the user [jjones] should be in the [Accounting] group 121ms
    [+] the user [abertram] should exist 39ms
    [+] the user [abertram] should be in the [PowerLab Users] OU 30ms
    [+] the user [abertram] should be in the [Accounting] group 80ms
    [+] the user [jhicks] should exist 39ms
```

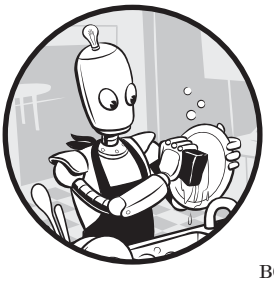
```
  [+] the user [jhicks] should be in the [PowerLab Users] OU 32ms
  [+] the user [jhicks] should be in the [Accounting] group 81ms
  [+] the user [dtrump] should exist 45ms
  [+] the user [dtrump] should be in the [PowerLab Users] OU 40ms
  [+] the user [dtrump] should be in the [Human Resources] group 84ms
  [+] the user [alincoln] should exist 41ms
  [+] the user [alincoln] should be in the [PowerLab Users] OU 40ms
  [+] the user [alincoln] should be in the [Human Resources] group 125ms
  [+] the user [bobama] should exist 44ms
  [+] the user [bobama] should be in the [PowerLab Users] OU 27ms
  [+] the user [bobama] should be in the [Human Resources] group 92ms
  [+] the user [tjefferson] should exist 58ms
  [+] the user [tjefferson] should be in the [PowerLab Users] OU 33ms
  [+] the user [tjefferson] should be in the [IT] group 73ms
  [+] the user [bclinton] should exist 47ms
  [+] the user [bclinton] should be in the [PowerLab Users] OU 29ms
  [+] the user [bclinton] should be in the [IT] group 84ms
  [+] the user [gbush] should exist 50ms
  [+] the user [gbush] should be in the [PowerLab Users] OU 33ms
  [+] the user [gbush] should be in the [IT] group 78ms
  [+] the user [rreagan] should exist 56ms
  [+] the user [rreagan] should be in the [PowerLab Users] OU 30ms
  [+] the user [rreagan] should be in the [IT] group 78ms
Context Groups
  [+] the group [Accounting] should exist 71ms
  [+] the group [Accounting] should be in the [PowerLab Groups] OU 42ms
  [+] the group [Human Resources] should exist 48ms
  [+] the group [Human Resources] should be in the [PowerLab Groups] OU 29ms
  [+] the group [IT] should exist 51ms
  [+] the group [IT] should be in the [PowerLab Groups] OU 31ms
```

Итоги

В этой главе мы сделали еще один шаг в создании PowerLab и добавили лес Active Directory, а затем заполнили его несколькими объектами. Мы это сделали вручную и автоматически и в процессе подробнее познакомились с Active Directory. Наконец, мы немного углубились в тестирование Pester, изучив более детально создание собственных тестов. В следующей главе мы продолжим работу над проектом PowerLab и узнаем, как автоматизировать установку и настройку SQL-сервера.

18

Создание и настройка SQL-сервера



Итак, вы создали модуль, который может настроить виртуальную машину, подключить к ней VHD, установить Windows, а также создать и заполнить лес Active Directory. Добавим в этот список еще одну вещь — развертывание SQL-сервера. Мы создали виртуальную машину, установили ОС и настроили контроллер домена, поэтому большая часть тяжелой работы позади! Теперь нам остается взять пару готовых функций, и после нескольких настроек вы сможете установить SQL-сервер.

Исходные требования

Я предполагаю, что вы изучили часть III и создали по крайней мере одну виртуальную машину под названием LABDC, которая работает на вашем хосте Hyper-V. Эта виртуальная машина должна быть контроллером домена, и, поскольку вы снова будете подключаться к нескольким ВМ через PowerShell Direct, вам понадобятся еще и учетные данные домена, сохраненные на хосте Hyper-V (см. главу 17).

На примере сценария ManuallyCreatingASqlServer.ps1 из материалов этой главы мы объясним, как правильно автоматизировать развертывание сервера SQL. Этот сценарий содержит все этапы, описанные в этой главе, поэтому он станет отличным справочным источником.

Как всегда, запустим сценарий тестирования исходных требований, включенный в эту главу, чтобы убедиться в их соблюдении.

Создание виртуальной машины

Термин *«SQL-сервер»*, вероятно, наводит на мысли о базах данных, задачах и таблицах. Но прежде чем вы вообще до этого доберетесь, необходимо проделать кучу подготовительной работы: для начала, каждая база данных SQL должна находиться на сервере, каждому серверу нужна операционная система, а каждой операционной системе для работы нужна физическая или виртуальная машина. К счастью, последние несколько глав мы посвятили настройке среды для создания SQL-сервера.

Хороший автоматизатор начинает проект с составления списка необходимых зависимостей. Он автоматизирует эти зависимости, а затем и все остальное. В результате получается модульная, несвязанная архитектура, которую можно относительно легко изменить в любое время.

В итоге нам нужно получить функцию, которая будет запускать любое количеств серверов SQL с некой стандартной конфигурацией. Но для достижения этого следует представить проект в виде слоев. Первый уровень — это виртуальная машина. Сначала разберемся с этим.

В вашем модуле PowerLab уже есть функция, которая создает виртуальную машину, — ее-то мы и будем использовать. Поскольку все наши лабораторные среды будут одинаковыми, у нас уже есть готовые параметры по умолчанию для создания новой виртуальной машины. В функции `New-PowerLabVM` единственное нужное для передачи значение — это имя ВМ:

```
PS> New-PowerLabVm -Name 'SQLSRV'
```

Установка операционной системы

И вот, словно по щелчку, у вас появилась готовая к работе виртуальная машина. Это было просто. Давайте повторим. Используем команду, которую вы написали в главе 16, чтобы установить Windows на ВМ:

```
PS> Install-PowerLabOperatingSystem -VmName 'SQLSRV'
Get-Item : Cannot find path 'C:\Program Files\WindowsPowerShell\Modules\
powerlab\SQLSRV.xml' because it does not exist.
At C:\Program Files\WindowsPowerShell\Modules\powerlab\PowerLab.psm1:138 char:16
+     $answerFile = Get-Item -Path "$PSScriptRoot\$VMName.xml"
```

```
+                    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo : ObjectNotFound: (C:\Program File...rlab\SQLSRV
                     .xml:String) [Get-Item], ItemNotFoundException
```

Ой! Мы использовали готовую функцию `Install-PowerLabOperatingSystem` в модуле PowerLab, чтобы установить операционную систему на будущий SQL-сервер, но произошел сбой, поскольку функция ссылается на файл с именем SQLSRV.xml в папке модуля. При создании этой функции предполагалось, что в папке модуля будет файл .xml. Подобные проблемы, такие как несовпадение путей и несуществующие файлы, часто встречаются при создании таких крупных проектов автоматизации. У вас будет много зависимостей, с которыми надо будет разобраться. Единственный способ избавиться от всех этих ошибок — выполнить код в максимально возможном количестве сценариев.

Добавление файла автоматического ответа Windows

Функция `Install-PowerLabOperatingSystem` предполагала, что в папке модуля PowerLab всегда будет файл с именем .xml. Это означает, что перед развертыванием нового сервера вы должны убедиться в наличии этого файла. К счастью, теперь, когда вы создали файл ответов автоматической установки LABDC, все будет просто. Первое, что вам нужно сделать, это скопировать уже существующий файл LABDC.xml и назвать его SQLSRV.xml:

```
PS> Copy-Item -Path 'C:\Program Files\WindowsPowerShell\Modules\PowerLab\LABDC.xml'
-Destination
'C:\Program Files\WindowsPowerShell\Modules\PowerLab\SQLSRV.xml'
```

После создания копии нужно выполнить несколько настроек — имя хоста и IP-адрес. Поскольку вы не развернули DHCP-сервер, следует использовать статические IP-адреса и впоследствии изменить их (в противном случае вам придется изменить только имя сервера).

Откройте файл C:\Program Files\WindowsPowerShell\Modules\SQLSRV.xml, найдите место, где задается имя хоста, и измените значение `ComputerName`. Это должно выглядеть примерно так:

```
<component name="Microsoft-Windows-Shell-Setup" processorArchitecture="amd64"
publicKeyToken="31bf3856ad364e35" language="neutral" versionScope="nonSxS"
    xmlns:wcm="http://schemas.microsoft.com/WMIConfig/2002/State"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ComputerName>SQLSRV</ComputerName>
    <ProductKey>XXXXXXXXXXXXX</ProductKey>
</component>
```

Затем найдите узел `UnicastIPAddress`. Это будет выглядеть как код, представленный ниже. К слову, я использую сеть 10.0.0.0/24 и решил, что у моего SQL-сервера будет IP-адрес 10.0.0.101:

```
<UnicastIpAddresses>
    <IpAddress wcm:action="add" wcm:keyValue="1">10.0.0.101</IpAddress>
</UnicastIpAddresses>
```

Сохраните файл SQLSRV.xml и попробуйте снова запустить команду `Install-PowerLabOperating System`. На этом этапе она должна успешно выполняться и развертывать Windows Server 2016 на виртуальной машине SQLSRV.

Добавление SQL-сервера в домен

Мы только что установили операционную систему, поэтому нужно запустить виртуальную машину. Это достаточно просто сделать с помощью командлета `Start-VM`:

```
PS> Start-VM -Name SQLSRV
```

Теперь вам нужно подождать, пока виртуальная машина подключится к сети — это может занять некоторое время. Сколько? Ну, тут есть много факторов. Как вариант, можно использовать цикл `while`, чтобы постоянно проверять наличие подключения к виртуальной машине.

Давайте разберемся, как это сделать. В листинге 18.1 мы получаем локально сохраненные учетные данные для виртуальной машины. Как только они будут у вас, создайте цикл `while`, который продолжает выполнение `Invoke-Command` до тех пор, пока что-то не будет возвращено.

Обратите внимание, что мы используем значение `Ignore` для параметра `ErrorAction`. Без него будет возвращаться сообщение о незавершающей ошибке, если `Invoke-Command` не сможет подключиться к компьютеру. Чтобы ожидаемые ошибки не наводнили вашу консоль (а они будут, вы об этом знаете, и это нормально), мы будем их игнорировать.

Листинг 18.1. Проверка работоспособности сервера и игнорирование сообщений об ошибках

```
$vmCred = Import-CliXml -Path 'C:\PowerLab\VMCredential.xml'
while (-not (Invoke-Command -VmName SQLSRV -ScriptBlock { 1 } -Credential
$vmCred -ErrorAction Ignore)) {
    Start-Sleep -Seconds 10
    Write-Host 'Waiting for SQLSRV to come up...'
}
```

Как только виртуальная машина появится, добавим ее в домен, который вы создали в предыдущей главе. Воспользуйтесь командой `Add-Computer`, которая добавит компьютер в домен. Поскольку вы запускаете все команды с самого хоста Hyper-V, команду `Add-Computer` следует заключить в блок сценария и выполнить через PowerShell Direct, чтобы запустить ее непосредственно в самом SQLSRV.

Обратите внимание, что в листинге 18.2 мы используем учетные записи и локального пользователя на ВМ, и домена. Для этого мы сначала устанавливаем соединение с самим сервером SQLSRV с помощью команды `Invoke-Command`. После подключения передаем контроллеру учетные данные домена для аутентификации, что позволит добавить учетную запись компьютера.

Листинг 18.2. Получение учетных данных и добавление компьютера в домен

```
$domainCred = Import-CliXml -Path 'C:\PowerLab\DomainCredential.xml'
$addParams = @{
    DomainName = 'powerlab.local'
    Credential = $domainCred
    Restart    = $true
    Force      = $true
}
Invoke-Command -VMName SQLSRV -ScriptBlock { Add-Computer ❶@using:addParams }
                                                          -Credential $vmCred
```

Обратите внимание, что мы используем оператор `$using` ❶. Он позволяет передать локальную переменную `$addParams` удаленному сеансу на вашем сервере SQLSRV.

Поскольку мы использовали параметр переключателя `Restart` в `Add-Computer`, ВМ будет перезагружена сразу после добавления в домен. Опять же, поскольку мы еще не закончили работу, нужно подождать. Однако на этот раз ждем до тех пор, пока машина отключится *и* вернется в рабочее состояние (листинг 18.3): сценарий работает настолько быстро, что если вы сначала не дождетесь отключения ВМ, то рискуете продолжить выполнение сценария, потому что он может решить, будто машина снова работает, хотя она даже еще не отключалась!

Листинг 18.3. Ожидание перезагрузки сервера

```
❶ while (Invoke-Command -VmName SQLSRV -ScriptBlock { 1 } -Credential $vmCred
  -ErrorAction Ignore) {
  ❷ Start-Sleep -Seconds 10
  ❸ Write-Host 'Waiting for SQLSRV to go down...'
  }

❶ while (-not (Invoke-Command -VmName SQLSRV -ScriptBlock { 1 } -Credential
  $domainCred -ErrorAction Ignore)) {
  ❷ Start-Sleep -Seconds 10
  ❸ Write-Host 'Waiting for SQLSRV to come up...'
}
```

Сначала мы проверяем, отключен ли SQLSRV, просто возвращая число 1 в SQLSRV ❶. Если он получает выходные данные, это означает, что удаленное управление PowerShell доступно и, следовательно, SQLSRV еще включен. Если результат возвращается, мы подождем 10 секунд ❷, выводим сообщение на экран ❸ и пробуем снова.

Затем мы производим обратное действие при тестировании, чтобы проверить, когда SQLSRV снова будет доступен. Как только сценарий освободит управление консолью, SQLSRV должен быть включен и добавлен в домен Active Directory.

Установка SQL-сервера

Теперь, когда вы создали виртуальную машину с Windows Server 2016, на нее можно установить SQL-сервер 2016. На этот раз вам придется подумать самим! До сих пор вы просто использовали готовый код, но здесь мы будем самостоятельно прокладывать тропу.

Установка SQL-сервера через PowerShell состоит из нескольких этапов:

1. Копирование и настройка файла ответов SQL-сервера.
2. Копирование ISO-файла SQL-сервера на будущий SQL-сервер.
3. Монтирование файла ISO на будущем SQL-сервере.
4. Запуск установщика SQL-сервера.
5. Демонтирование файла ISO.
6. Очистка всех временных скопированных файлов на SQL-сервере.

Копирование файлов на SQL-сервер

Согласно нашему плану, первым делом нужно загрузить несколько файлов на будущий SQL-сервер. Вам нужен файл ответов для установщика SQL-сервера, а также файл ISO, содержащий установочный контент SQL-сервера. Поскольку мы предполагаем, что у вас нет сетевого подключения к виртуальным машинам от хоста Hyper-V, для копирования этих файлов снова будем использовать PowerShell Direct. Сначала нужно создать сеанс на удаленной виртуальной машине, чтобы использовать его для копирования файлов. В следующем коде мы используем параметр `Credential` для аутентификации в SQLSRV. Если бы сервер находился в том же домене Active Directory, что и локальный компьютер, параметр `Credential` не понадобился бы.

```
$session = New-PSSession -VMName 'SQLSRV' -Credential $domainCred
```

Затем мы создадим копию файла шаблона SQLServer.ini из модуля PowerLab:

```
$sqlServerAnswerFilePath = "C:\Program Files\WindowsPowerShell\Modules\
PowerLab\SqlServer.ini"
$tempFile = Copy-Item -Path $sqlServerAnswerFilePath -Destination "C:\Program
Files\WindowsPowerShell\Modules\PowerLab\temp.ini" -PassThru
```

После этого изменим файл под нужную конфигурацию. Напомним, что ранее, когда нам нужно было изменить некоторые значения, мы вручную открывали файл ответов XML. Это слишком сложно и долго. Верите или нет, но вы можете автоматизировать даже этот шаг!

В листинге 18.4 считывается содержимое скопированного файла шаблона и находятся строки SQLSVCACCOUNT=, SQLSVCPASSWORD= и SQLSYSADMINACCOUNTS=, которые заменяются конкретными значениями. Когда вы закончите с этим, перепишите скопированный файл шаблона с новыми измененными строками.

Листинг 18.4. Замена строк

```
$configContents = Get-Content -Path $tempFile.FullName -Raw
$configContents = $configContents.Replace('SQLSVCACCOUNT=""',
                                'SQLSVCACCOUNT="PowerLabUser"')
$configContents = $configContents.Replace('SQLSVCPASSWORD=""',
                                'SQLSVCPASSWORD="P@$$w0rd12"')
$configContents = $configContents.Replace('SQLSYSADMINACCOUNTS=""',
                                'SQLSYSADMINACCOUNTS="PowerLabUser"')
Set-Content -Path $tempFile.FullName -Value $configContents
```

После создания файла ответов скопируйте его, а также ISO-файл на будущий SQL-сервер, и установщик будет готов к работе:

```
$copyParams = @{
    Path        = $tempFile.FullName
    Destination = 'C:\'
    ToSession   = $session
}
Copy-Item @copyParams
Remove-Item -Path $tempFile.FullName -ErrorAction Ignore
Copy-Item -Path 'C:\PowerLab\ISOs\en_sql_server_2016_standard_x64_dvd_8701871.iso'
-Destination 'C:\' -Force -ToSession $session
```

Запуск установщика SQL-сервера

Теперь мы готовы к установке SQL-сервера. Код приведен в листинге 18.5.

Листинг 18.5. Использование Invoke-Command для монтирования, установки и демонтирования образа

```
$icmParams = @{
    Session      = $session
    ArgumentList = $tempFile.Name
    ScriptBlock  = {
        $image = Mount-DiskImage -ImagePath 'C:\en_sql_server_2016_standard_x64_
                                                                    dvd_8701871
        .iso' -PassThru ❶
        $installerPath = "$(($image | Get-Volume).DriveLetter):"
        $null = & "$installerPath\setup.exe" "/CONFIGURATIONFILE=
                                    C:\$($using:tempFile.Name)" ❷
        $image | Dismount-DiskImage ❸
    }
}
Invoke-Command @icmParams
```

Сначала мы переносим скопированный ISO-файл на удаленный компьютер ❶, затем запускаем установщик с присвоением выводу переменной `$null` ❷ (он вам не нужен) и демонтируем образ после завершения установки ❸. В листинге 18.5 мы используем `Invoke-Command` и PowerShell Direct для удаленного выполнения этих команд.

После установки SQL-сервера мы выполняем очистку временных файлов, как показано в листинге 18.6.

Листинг 18.6. Очистка временных файлов

```
$scriptBlock = { Remove-Item -Path 'C:\en_sql_server_2016_standard_x64_dvd
_8701871.iso', "C:\$($using:tempFile.Name)" -Recurse -ErrorAction Ignore }
Invoke-Command -ScriptBlock $scriptBlock -Session $session
$session | Remove-PSSession
```

На этом этапе SQL-сервер настроен и готов к работе! Всего за 64 строки в PowerShell вы создали SQL-сервер Microsoft на хосте Hyper-V. Это большой прогресс, но еще есть куда расти.

Автоматизация SQL-сервера

Большую часть тяжелой работы мы уже проделали. Сейчас у вас уже есть сценарий, который делает все необходимое. Теперь нужно объединить все эти возможности в нескольких функциях модуля PowerLab: `New-Power`, `LabSqlServer` и `Install-PowerLabOperatingSystem`.

Будем следовать базовому шаблону автоматизации: создадим функции вокруг всех общих действий и вызовем их вместо использования жестко

запрограммированных значений. В результате останется одна функция, которую может вызвать пользователь. В листинге 18.7 мы используем готовые функции для создания ВМ и виртуального жесткого диска, а также вторую функцию `Install-PowerLabSqlServer` для размещения кода установки SQL-сервера.

Листинг 18.7. Функция New-PowerLabSqlServer

```
function New-PowerLabSqlServer {
    [CmdletBinding()]
    param
     (
        [Parameter(Mandatory)]
        [string]$Name,

        [Parameter(Mandatory)]
        [pscredential]$DomainCredential,

        [Parameter(Mandatory)]
        [pscredential]$VMCredential,

        [Parameter()]
        [string]$VMPath = 'C:\PowerLab\VMs',

        [Parameter()]
        [int64]$Memory = 2GB,

        [Parameter()]
        [string]$Switch = 'PowerLab',

        [Parameter()]
        [int]$Generation = 2,

        [Parameter()]
        [string]$DomainName = 'powerlab.local',

        [Parameter()]
        [string]$AnswerFilePath = "C:\Program Files\WindowsPowerShell\Modules
                                                                    \PowerLab
        \SqlServer.ini"
    )

    ## Build the VM
    $vmparams = @{
        Name       = $Name
        Path       = $VmPath
        Memory     = $Memory
        Switch     = $Switch
        Generation = $Generation
    }
```

```
    New-PowerLabVm @vmParams
    Install-PowerLabOperatingSystem -VmName $Name
    Start-VM -Name $Name
    Wait-Server -Name $Name -Status Online -Credential $VMCredential
    $addParams = @{
        DomainName = $DomainName
        Credential = $DomainCredential
        Restart    = $true
        Force      = $true
    Invoke-Command -VMName $Name -ScriptBlock { Add-Computer @using:addParams }
                                                                    -Credential
    $VMCredential
    Wait-Server -Name $Name -Status Offline -Credential $VMCredential
    Wait-Server -Name $Name -Status Online -Credential $DomainCredential
    $tempFile = Copy-Item -Path $AnswerFilePath
    -Destination "C:\Program Files\WindowsPowerShell\Modules\PowerLab\temp.ini"
                                                                     -PassThru

    Install-PowerLabSqlServer -ComputerName $Name -AnswerFilePath $tempFile.
                                                                    FullName
}
```

Большую часть этого кода вы наверняка узнали: это тот же самый код, который мы рассмотрели совсем недавно, но запакованный в функцию! Его можно будет использовать много раз. Я взял тот же код, но вместо использования жестко закодированных значений я параметризовал многие атрибуты — это позволит установить SQL-сервер с другими параметрами без изменения самого кода.

Превращение конкретных сценариев в общую функцию сохраняет работоспособность вашего кода, а также обеспечивает большую гибкость на случай, если в будущем вы захотите изменить способ развертывания серверов SQL.

Давайте посмотрим на основные части функции `Install-PowerLabSqlServer` в листинге 18.8.

Листинг 18.8. Функция модуля Install-PowerLabSqlServer PowerLab

```
function Install-PowerLabSqlServer {
  ❶ param
    (
        [Parameter(Mandatory)]
        [string]$ComputerName,

        [Parameter(Mandatory)]
        [pscredential]$DomainCredential,

        [Parameter(Mandatory)]
        [string]$AnswerFilePath,
```

```
        [Parameter()]
        [string]$IsoFilePath = 'C:\PowerLab\ISOs\en_sql_server_2016_standard
         _x64_dvd_8701871.iso'
    )

    try {
        --пропуск--

      ❷ ## Test to see if SQL Server is already installed
        if (Invoke-Command -Session $session
        -ScriptBlock { Get-Service -Name 'MSSQLSERVER' -ErrorAction Ignore }) {
            Write-Verbose -Message 'SQL Server is already installed'
        } else {
         ❸ PrepareSqlServerInstallConfigFile -Path $AnswerFilePath
            --пропуск--
    } catch {
        $PSCmdlet.ThrowTerminatingError($_)
    }
}
```

Мы параметризуем все входные данные для установки SQL-сервера ❶, затем добавляем обработку ошибок ❷, чтобы проверить, установлен ли SQL-сервер ранее. Это позволяет запускать функцию снова и снова, а если SQL-сервер уже установлен, функция просто пропустит его.

Обратите внимание на вызов новой функции — `PrepareSqlServerInstall ConfigFile` ❸. Она выполняет *вспомогательную роль*: реализует некоторые функциональные возможности, которые вы, вероятно, будете периодически использовать (вспомогательные функции обычно скрыты от пользователя и используются заочно). Такие маленькие необязательные детали делают код более читабельным. Как правило, функции должны делать только одну *«вещь»*. Это, конечно, весьма относительный термин, но чем больше вы программируете, тем больше будете интуитивно понимать, что ваша функция выполняет слишком много вещей одновременно.

В листинге 18.9 показан код функции `PrepareSqlServerInstallConfigFile`.

Листинг 18.9. Вспомогательная функция PrepareSqlServerInstallConfigFile

```
function PrepareSqlServerInstallConfigFile {
    [CmdletBinding()]
    param
    (
        [Parameter(Mandatory)]
        [string]$Path,

        [Parameter()]
        [string]$ServiceAccountName = 'PowerLabUser',
```

```
        [Parameter()]
        [string]$ServiceAccountPassword = 'P@$$w0rd12',

        [Parameter()]
        [string]$SysAdminAccountName = 'PowerLabUser'
    )

    $configContents = Get-Content -Path $Path -Raw
    $configContents = $configContents.Replace('SQLSVCACCOUNT=""',
    ('SQLSVCACCOUNT="{0}"' -f $ServiceAccountName))
    $configContents = $configContents.Replace('SQLSVCPASSWORD=""',
    ('SQLSVCPASSWORD="{0}"' -f $ServiceAccountPassword))
    $configContents = $configContents.Replace('SQLSYSADMINACCOUNTS=""',
    ('SQLSYSADMINACCOUNTS="{0}"' -f $SysAdminAccountName))
    Set-Content -Path $Path -Value $configContents
}
```

Вы, должно быть, узнали код из листинга 18.4; здесь он не особо изменился. Мы добавили параметры `Path`, `ServiceAccountName`, `ServiceAccountPassword` и `SysAdminAccountName` для представления каждого атрибута вместо прежде использованных жестко заданных значений.

Теперь, когда у нас есть все функции, создание SQL-сервера с нуля выполняется всего за пару команд. Запустите для этого следующий код!

```
PS> $vmCred = Import-CliXml -Path 'C:\PowerLab\VMCredential.xml'
PS> $domainCred = Import-CliXml -Path 'C:\PowerLab\DomainCredential.xml'
PS> New-PowerLabSqlServer -Name SQLSRV -DomainCredential $domainCred
                          -VMCredential $vmCred
```

Запуск тестов Pester

И снова настало время запуска тестов Pester для проверки внесенных изменений. В этой главе вы установили SQL-сервер на существующую виртуальную машину SQLSRV. Мы не особо его настраивали и приняли большинство параметров установки по умолчанию. Поэтому вы проведете всего несколько тестов Pester: следует убедиться, что SQL-сервер установлен и что при установке правильно считался файл автоматической конфигурации. Вы можете сделать это, если у `PowerLabUser` задана роль системного администратора сервера, и если SQL-сервер работает под учетной записью `PowerLabUser`:

```
PS> Invoke-Pester 'C:\PowerShellForSysAdmins\Part II\Creating and Configuring
SQL Servers\Creating and Configuring SQL Servers.Tests.ps1'

Describing SQLSRV
   Context SQL Server installation
```

```
    [+] SQL Server is installed 4.33s
   Context SQL Server configuration
    [+] PowerLabUser holds the sysadmin role 275ms
     [+] the MSSQLSERVER is running under the PowerLabUser account 63ms
Tests completed in 6.28s
Passed: 3 Failed: 0 Skipped: 0 Pending: 0 Inconclusive: 0
```

Проверка пройдена успешно — мои поздравления!

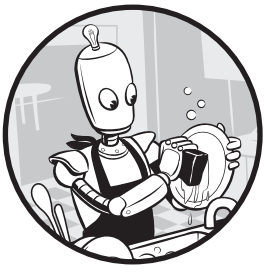
Итоги

В этой главе вы наконец увидели более подробный пример работы PowerShell. Основываясь на работе, проделанной в предыдущих главах, мы добавили последний уровень автоматизации — установку программного обеспечения (SQL-сервер) поверх операционной системы, которая была «наложена» поверх виртуальной машины. Это делается почти так же, как и в предыдущих нескольких главах. Вы использовали единственный пример, чтобы выяснить, какой код вам нужен, затем упаковали этот код в формат для многоразового использования и поместили его в модуль PowerLab. И теперь вы можете создать сколько угодно SQL-серверов с помощью всего лишь нескольких строк кода!

В следующей главе мы сделаем нечто иное: пересмотрим уже написанный код и проведем его рефакторинг. Вы узнаете о лучших практиках программирования и убедитесь, что ваш модуль находится в нужном месте, до того как мы добавим последний кусочек пазла в главе 20.

19

Рефакторинг кода



В предыдущей главе мы создали виртуальную машину с работающим SQL-сервером, имея лишь гипервизор, операционную систему, ISO-файл с ОС и немного кода. Для этого нужно было связать вместе функции из предыдущих глав. Здесь мы сделаем нечто иное: вместо того, чтобы добавлять новые функции в модуль PowerLab, мы углубимся в код и посмотрим, сможем ли мы сделать модуль более «модульным».

Под *«модульностью»* я имею в виду разделение функциональности кода на функции многократного использования, которые могут обрабатывать множество ситуаций. Более модульный код является более универсальным, и чем шире можно его применить, тем полезнее он будет. Модульность кода позволяет повторно использовать функции вроде `New-PowerLabVM` или `Install-PowerLabOperatingSystem` для установки разных серверов (это вы увидите в следующей главе).

Еще раз о функции New-PowerLabSqlServer

В главе 18 мы создали две основные функции — `New-PowerLabSqlServer` и `Install-PowerLabSqlServer`, — которые позволяют настроить SQL-сервер. Но что, если они должны применяться более широко? В конце концов, у разных

серверов есть много общих компонентов с SQL-серверами: виртуальная машина, виртуальный диск, ОС Windows и т. д. Вы можете просто скопировать имеющуюся у вас функцию и заменить все ссылки SQL на ссылки нужного типа сервера.

Но я не советую так делать, да и необходимости во всем этом дополнительном коде нет. Вместо этого мы просто проведем рефакторинг существующего кода. *Рефакторинг* — это изменение внутренней части кода без затрагивания его функциональности. Другими словами, рефакторинг помогает сделать код более читаемым и позволяет вам продолжать работу над проектом без слишком уж многих организационных проблем.

Давайте начнем с рассмотрения созданной вами функции `New-PowerLabSqlServer`, показанной в листинге 19.1.

Листинг 19.1. Функция New-PowerLabSqlServer

```
function New-PowerLabSqlServer {
    [CmdletBinding()]
  ❶ param
    (
        [Parameter(Mandatory)]
        [string]$Name,

        [Parameter(Mandatory)]
        [pscredential]$DomainCredential,

        [Parameter(Mandatory)]
        [pscredential]$VMCredential,

        [Parameter()]
        [string]$VMPath = 'C:\PowerLab\VMs',

        [Parameter()]
        [int64]$Memory = 4GB,

        [Parameter()]
        [string]$Switch = 'PowerLab',

        [Parameter()]
         [int]$Generation = 2,

        [Parameter()]
        [string]$DomainName = 'powerlab.local',

        [Parameter()]
     ❷  [string]$AnswerFilePath = "C:\Program Files\WindowsPowerShell\Modules
        \PowerLab\SqlServer.ini"
```

```
    )

❸ ## Build the VM
    $vmparams = @{
        Name       = $Name
        Path       = $VmPath
        Memory     = $Memory
        Switch     = $Switch
        Generation = $Generation
    }
    New-PowerLabVm @vmParams

    Install-PowerLabOperatingSystem -VmName $Name
    Start-VM -Name $Name

    Wait-Server -Name $Name -Status Online -Credential $VMCredential

    $addParams = @{
        DomainName = $DomainName
        Credential = $DomainCredential
        Restart    = $true
        Force      = $true
    }
    Invoke-Command -VMName $Name -ScriptBlock { Add-Computer
    @using:addParams } -Credential $VMCredential

    Wait-Server -Name $Name -Status Offline -Credential $VMCredential

❹  Wait-Server -Name $Name -Status Online -Credential $DomainCredential

    $tempFile = Copy-Item -Path $AnswerFilePath -Destination "C:\Program
    Files\WindowsPowerShell\Modules\PowerLab\temp.ini" -PassThru

    Install-PowerLabSqlServer -ComputerName $Name -AnswerFilePath $tempFile
    .FullName -DomainCredential $DomainCredential
}
```

Как провести рефакторинг этого кода? Вы знаете, что каждому серверу нужна виртуальная машина, виртуальный диск и операционная система. Это реализуется в блоке кода между ❸ и ❹.

Если вы посмотрите на этот код повнимательнее, то увидите, что нельзя просто взять и вставить его в новую функцию. Параметры определены в функции `New-PowerLabSqlServer` ❶, которую вы используете в этих строках. Обратите внимание, что единственный параметр, специфичный для SQL, — это `AnswerFilePath` ❷.

Теперь, когда вы определили код, не зависящий от SQL, давайте вытащим его и создадим новую функцию `New-PowerLabServer` (листинг 19.2).

Листинг 19.2. Более общая функция New-PowerLabServer

```
function New-PowerLabServer {
    [CmdletBinding()]
    param
    (
        [Parameter(Mandatory)]
        [string]$Name,

        [Parameter(Mandatory)]
        [pscredential]$DomainCredential,

        [Parameter(Mandatory)]
        [pscredential]$VMCredential,

        [Parameter()]
        [string]$VMPath = 'C:\PowerLab\VMs',

        [Parameter()]
        [int64]$Memory = 4GB,

        [Parameter()]
        [string]$Switch = 'PowerLab',

        [Parameter()]
        [int]$Generation = 2,

        [Parameter()]
        [string]$DomainName = 'powerlab.local'
    )

    ## Build the VM
    $vmparams = @{
        Name       = $Name
        Path       = $VmPath
        Memory     = $Memory
        Switch     = $Switch
        Generation = $Generation
    }
    New-PowerLabVm @vmParams

    Install-PowerLabOperatingSystem -VmName $Name
    Start-VM -Name $Name

    Wait-Server -Name $Name -Status Online -Credential $VMCredential

    $addParams = @{
        DomainName = $DomainName
        Credential = $DomainCredential
        Restart    = $true
        Force      = $true
    }
```

```
    Invoke-Command -VMName $Name
    -ScriptBlock { Add-Computer @using:addParams } -Credential $VMCredential

    Wait-Server -Name $Name -Status Offline -Credential $VMCredential

    Wait-Server -Name $Name -Status Online -Credential $DomainCredential
}
```

Сейчас у нас есть общая функция создания сервера, но нет способа указать его тип. Давайте исправим это, воспользовавшись параметром `ServerType`:

```
[Parameter(Mandatory)]
[ValidateSet('SQL', 'Web', 'Generic')]
[string]$ServerType
```

Обратите внимание на новый параметр `ValidateSet`: позже я подробнее объясню его действия. Пока что вам достаточно знать, что он предоставляет возможность пользователю выбрать серверы из предоставленного набора.

Теперь, когда у вас есть этот параметр, давайте им воспользуемся. Вставьте оператор `switch` в конце функции для выполнения разного кода в зависимости от типа сервера, который вводит пользователь:

```
switch ($ServerType) {
    'Web' {
        Write-Host 'Web server deployments are not supported at this time'
        break
    }
    'SQL' {
        $tempFile = Copy-Item -Path $AnswerFilePath -Destination "C:\Program
        Files\WindowsPowerShell\Modules\PowerLab\temp.ini" -PassThru
        Install-PowerLabSqlServer -ComputerName $Name -AnswerFilePath
        $tempFile.FullName -DomainCredential $DomainCredential
        break
    }
    'Generic' {
        break
    }
 ❶ default {
        throw "Unrecognized server type: [$_]"
    }
}
```

Как видно из этого примера, мы обрабатываем три типа сервера (и используем блок `default` по умолчанию для обработки любых исключений ❶). Но есть проблема: чтобы заполнить код для SQL, мы скопировали и вставили код из функции `New-PowerLabSqlServer` и теперь пытаемся использовать то,

чего нет, — переменную AnswerFilePath. Когда мы переместили свой общий код в новую функцию, мы оставили эту переменную, а это означает, что мы не можем использовать ее здесь... Или можем?

Использование наборов параметров

Для ситуаций, подобных этой, у PowerShell есть удобная вещь — *наборы параметров*. Они позволяют использовать условную логику для управления параметрами, введенными пользователем.

В этом примере вы будете использовать три набора параметров: набор для серверов SQL, набор для веб-серверов и набор по умолчанию.

Вы можете определить наборы параметров, используя атрибут ParameterSetName, за которым следует имя. Например:

```
[Parameter(Mandatory)]
[ValidateSet('SQL', 'Web', 'Generic')]
[string]$ServerType,

[Parameter(ParameterSetName = 'SQL')]
[string]$AnswerFilePath = "C:\Program Files\WindowsPowerShell\Modules\PowerLab
                                                              \SqlServer.ini",

[Parameter(ParameterSetName = 'Web')]
[switch]$NoDefaultWebsite
```

Обратите внимание, что вы не назначили набор параметров для ServerType. Параметры вне набора можно использовать где угодно, поэтому вы можете использовать ServerType либо с AnswerFilePath, либо с вновь созданным параметром, который будет использоваться для подготовки веб-сервера CreateDefaultWebsite.

Видно, что большинство параметров остаются без изменений, но последний меняется в зависимости от того, что было передано в ServerType:

```
PS> New-PowerLabServer -Name WEBSRV -DomainCredential CredentialHere
-VMCredential CredentialHere
-ServerType 'Web' -NoDefaultWebsite
PS> New-PowerLabServer -Name SQLSRV -DomainCredential CredentialHere
-VMCredential CredentialHere
-ServerType 'SQL' -AnswerFilePath 'C:\OverridingTheDefaultPath\SqlServer.ini'
```

Если вы попытаетесь одновременно использовать параметры из двух разных наборов, то ничего не выйдет:

```
PS> New-PowerLabServer -Name SQLSRV -DomainCredential CredentialHere
-VMCredential CredentialHere
-ServerType 'SQL' -NoDefaultWebsite -AnswerFilePath 'C:\OverridingTheDefaultPath
\SqlServer.ini'

New-PowerLabServer : Parameter set cannot be resolved using the specified named
parameters.
At line:1 char:1
+ New-PowerLabServer -Name SQLSRV -ServerType 'SQL' -NoDefaultWebsite - ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [New-PowerLabServer],
ParameterBindingException
    + FullyQualifiedErrorId : AmbiguousParameterSet,New-PowerLabServer
```

А что если бы вы поступили наоборот и не использовали параметр `NoDefaultWebsite` или `AnswerFilePath`?

```
PS> New-PowerLabServer -Name SQLSRV -DomainCredential CredentialHere -VMCredential
CredentialHere -ServerType 'SQL'
New-PowerLabServer : Parameter set cannot be resolved using the specified named
parameters.
At line:1 char:1
+ New-PowerLabServer -Name SQLSRV -DomainCredential $credential...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [New-PowerLabServer],
ParameterBindingException
    + FullyQualifiedErrorId : AmbiguousParameterSet,New-PowerLabServer

PS> New-PowerLabServer -Name WEBSRV -DomainCredential CredentialHere
-VMCredential CredentialHere -ServerType 'Web'
New-PowerLabServer : Parameter set cannot be resolved using the specified named
parameters.
At line:1 char:1
+ New-PowerLabServer -Name WEBSRV -DomainCredential $credential...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [New-PowerLabServer],
ParameterBindingException
    + FullyQualifiedErrorId : AmbiguousParameterSet,New-PowerLabServer
```

Вы получите ту же ошибку о невозможности разрешения набора параметров. Почему? PowerShell не знает, какой набор параметров использовать! Ранее я сказал, что мы будем использовать три набора, но определили-то мы только два. Вам необходимо определить набор параметров по умолчанию. Как вы убедились ранее, параметры, которые явно не присвоены набору, могут использоваться вместе с любыми параметрами набора. Однако если вы определите набор параметров по умолчанию, PowerShell будет использовать их при отсутствии иных, где-либо заданных.

Вы можете выбрать определенный набор параметров SQL либо веб-параметров в качестве значения по умолчанию или же просто определить неспецифический набор, например `blah blah`, который создаст собственный набор для всех параметров без явного набора:

```
[CmdletBinding(DefaultParameterSetName = 'blah blah')]
```

Если вы не хотите устанавливать заданный набор параметров по умолчанию, можно установить что угодно, и тогда PowerShell будет игнорировать оба набора, *если ни один из их параметров не используется*. Совершенно нормально не использовать определенный набор параметров, потому что вы уже используете `ServerType`, который указывает, что у нас будет — веб-сервер или SQL-сервер.

Теперь параметры функции `New-PowerLabServer` выглядят как в листинге 19.3.

Листинг 19.3. Новая функция New-PowerLabServer

```
function New-PowerLabServer {
    [CmdletBinding(DefaultParameterSetName = 'Generic')]
    param
    (
        [Parameter(Mandatory)]
        [string]$Name,

        [Parameter(Mandatory)]
        [pscredential]$DomainCredential,

        [Parameter(Mandatory)]
        [pscredential]$VMCredential,

        [Parameter()]
        [string]$VMPath = 'C:\PowerLab\VMs',

        [Parameter()]
        [int64]$Memory = 4GB,

        [Parameter()]
        [string]$Switch = 'PowerLab',

        [Parameter()]
        [int]$Generation = 2,

        [Parameter()]
        [string]$DomainName = 'powerlab.local',

        [Parameter()]
        [ValidateSet('SQL', 'Web')]
        [string]$ServerType,
```

```
        [Parameter(ParameterSetName = 'SQL')]
        [string]$AnswerFilePath = "C:\Program Files\WindowsPowerShell\Modules
        \PowerLab\SqlServer.ini",

        [Parameter(ParameterSetName = 'Web')]
        [switch]$NoDefaultWebsite
    )
```

Обратите внимание, что у вас есть ссылка на функцию `Install-PowerLabSqlServer`. Это похоже на функцию `New-PowerLabSqlServer`, которая сбила нас с толку. Вместо создания виртуальной машины и установки операционной системы функция `Install-PowerLabSqlServer` берет на себя функции `New-PowerLabServer`, устанавливает программное обеспечение SQL-сервера и выполняет базовую настройку. Возможно, вы захотите провести подобный рефакторинг для этой функции. Это, конечно, возможно, но вы вскоре поймете, что между установкой SQL-сервера и других типов серверов нет почти ничего общего. Это уникальный процесс, и его будет сложно «обобщить».

Итоги

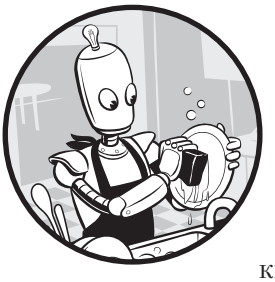
Теперь, когда мы обновили и переработали код, у нас появилась функция, которая создает SQL-серверы. То есть ничего нового? А вот и не так! Несмотря на то что мы ничего не изменили в функциональности кода, мы создали основу для реализации кода создания веб-сервера. Что мы и сделаем в следующей главе.

Мы убедились в том, что рефакторинг кода PowerShell — это сложный процесс. Знание способов рефакторинга кода и реализации вашей текущей ситуации — это навык, который приходит с опытом. Пока вы следуете *принципу DRY* («не повторяйтесь»), вы будете на правильном пути. В первую очередь следование DRY означает отсутствие дублирования кода и избыточной функциональности. Мы наблюдали это в этой главе, когда решили создать общую функцию, которая создает новые серверы. Она является полной противоположностью функции `New-PowerLabInsertServerTypeHereServer`.

Весь этот труд был не напрасен. В следующей главе мы вернемся к автоматизации и добавим код, необходимый для создания веб-сервера IIS.

20

Создание и настройка веб-сервера IIS



Остался последний этап автоматизации — веб-сервер. В этой главе мы будем использовать IIS, встроенную службу Windows, которая предоставляет веб-службы клиентам. *IIS* — это сервер, с которым вы будете сталкиваться достаточно часто в сфере ИТ. Другими словами, это область, которая требует автоматизации!

Как и в предыдущих главах, сначала мы развернем веб-сервер IIS с нуля, а затем сосредоточимся на установке службы и применим некоторые базовые настройки.

Исходные требования

Сейчас вы уже должны знать о создании и настройке новой виртуальной машины, поэтому не будем на этом останавливаться. Я предполагаю, что у вас уже есть виртуальная машина с Windows Server. Если нет, возьмите готовый модуль PowerLab и выполните следующую команду:

```
PS> New-PowerLabServer -ServerType Generic
-DomainCredential (Import-Clixml -Path C:\PowerLab\DomainCredential.xml)
-VMCredential (Import-Clixml -Path C:\PowerLab\VMCredential.xml) -Name WEBSRV
```

Обратите внимание, что на этот раз мы указали тип сервера `Generic`, потому что еще не добавили полную поддержку веб-серверов. Но мы это сделаем в этой главе!

Установка и настройка

После того как вы создали виртуальную машину, пришло время настроить IIS. IIS — это компонент Windows, и, к счастью, в PowerShell есть встроенная команда для установки компонентов Windows `Add-WindowsFeature`. Если бы установку IIS нужно было сделать только один раз, можно было бы использовать одну строку, но поскольку мы встраиваем эту автоматизацию в более крупный проект, будем устанавливать IIS так же, как и SQL, — с помощью новой функции. Назовем ее `Install-PowerLabWebServer`.

Эта функция будет соответствовать ранее созданной модели, когда мы писали функцию `Install-PowerLabSqlServer`. Когда мы начнем добавлять в проект поддержку второго вида серверов, вы заметите, что создание функции даже для одной строки кода делает использование модуля и его изменение намного, намного проще!

Самый простой способ максимально точно отразить функцию `Install-PowerLabSqlServer` — взять «скелет» функции и удалить любой код, специфичный для SQL-сервера. Обычно я бы рекомендовал повторно использовать имеющуюся функцию вместо создания новой, но мы работали с другим «объектом» — SQL-сервером, а не сервером IIS. Имеет смысл написать другую функцию. В листинге 20.1 мы копируем функцию `Install-PowerLabSqlServer`, удаляем ее содержимое (убирая параметры `AnswerFilePath` и `IsoFilePath`, поскольку они не нужны для работы с IIS) и сохраняем при этом все общие параметры.

Листинг 20.1. «Скелет» функции Install-PowerLabWebServer

```
function Install-PowerLabWebServer {
    param
    (
        [Parameter(Mandatory)]
        [string]$ComputerName,

        [Parameter(Mandatory)]
        [pscredential]$DomainCredential
    )

    $session = New-PSSession -VMName $ComputerName -Credential $DomainCredential

    $session | Remove-PSSession
}
```

Что касается фактической настройки службы IIS, то это совсем несложно: необходимо просто запустить одну команду, которая устанавливает `Web-Server`. Добавьте эту строку в функцию `Install-PowerLabWebServer` (листинг 20.2).

Листинг 20.2. Функция Install-PowerLabWebServer

```
function Install-PowerLabWebServer {
    param
    (
        [Parameter(Mandatory)]
        [string]$ComputerName,

        [Parameter(Mandatory)]
        [pscredential]$DomainCredential
    )

    $session = New-PSSession -VMName $ComputerName -Credential $DomainCredential

    $null = Invoke-Command -Session $session -ScriptBlock { Add-WindowsFeature
                                                            -Name 'Web-Server' }

    $session | Remove-PSSession

}
```

Начало функции `Install-PowerLabWebServer` положено! Давайте добавим еще кода.

Создание веб-сервера с нуля

Теперь, когда у вас есть функция установки для IIS, пришло время обновить функцию `New-PowerLabServer`. Вспомните главу 19: при рефакторинге `New-PowerLabServer` нам пришлось использовать заполнитель для части кода, отвечающей за веб-сервер, из-за отсутствия необходимой функциональности. В качестве заглушки мы использовали строку `Write-Host 'Web server deployments are not supported at this time'` («В настоящее время развертывания сервера не поддерживаются»). Теперь давайте заменим этот текст вызовом вашей вновь созданной функции `Install-PowerLabWebServer`:

```
PS> Install-PowerLabWebServer –ComputerName $Name –DomainCredential
$DomainCredential
```

Как только это будет сделано, вы сможете запускать веб-серверы так же, как и серверы SQL!

Модуль веб-администрирования

Когда вы запустите веб-сервер, надо будет что-то с ним сделать. Когда на сервере включена функция веб-сервера, устанавливается PowerShell-модуль под названием `WebAdministration`. У этого модуля есть множество команд, необходимых для обработки объектов IIS. Инструмент `Web-Server` также создает диск PowerShell под названием IIS, который позволяет управлять объектами IIS (веб-сайтами, пулами приложений и т. д.).

Виртуальный диск PowerShell позволяет вам перемещаться по источникам данных так же, как и в файловой системе. Далее вы увидите, что веб-сайтами, пулами приложений и многими другими объектами IIS можно управлять точно так же, как и обычными файлами и папками, используя общие командлеты `Get-Item`, `Set-Item` и `Remove-Item`.

Чтобы сделать диск IIS доступным, сначала нужно импортировать модуль `Web-administration`. Давайте подключимся к вашему недавно созданному веб-серверу и немного поиграем с модулем — посмотрим, что можно с ним сделать.

Для начала создадим новый сеанс PowerShell Direct и интерактивно войдем в него. Раньше для отправки команд виртуальным машинам мы использовали команду `Invoke-Command`. Теперь, когда вы просто изучаете возможности IIS, воспользуйтесь командой `Enter-PSSession` для интерактивной работы внутри сеанса:

```
PS> $session = New-PSSession -VMName WEBSRV
-Credential (Import-Clixml -Path C:\PowerLab\DomainCredential.xml)
PS> Enter-PSSession -Session $session
[WEBSRV]: PS> Import-Module WebAdministration
```

Обратите внимание на текст `[WEBSRV]` перед последним вводом. Это говорит о том, что вы теперь работаете с хостом `WEBSRV` и можете импортировать модуль `WebAdministration`. После импортирования модуля в сеанс убедитесь в создании диска IIS, запустив `Get-PSDrive`:

```
[WEBSRV]: PS> Get-PSDrive -Name IIS | Format-Table -AutoSize

Name Used (GB) Free (GB) Provider          Root     CurrentLocation
---- --------- --------- --------          ----     ---------------
IIS                     WebAdministration \\WEBSRV
```

Вы можете просматривать этот диск (как и любой другой диск PowerShell) как файловую систему, с помощью команд вроде `Get-ChildItem` для перечисления элементов на диске, `New-Item` для создания новых элементов и `Set-Item` для их

изменения. Но это не автоматизация, а просто управление IIS через командную строку. А наша цель — автоматизация! Единственная причина, по которой я сейчас упоминаю диск IIS, заключается в том, что позже он пригодится для задач автоматизации. К тому же всегда полезно знать, как делать что-то вручную на случай, если вам придется устранять какие-либо проблемы.

Сайты и пулы приложений

Команды в модуле `WebAdministration` позволяют автоматизировать практически все аспекты IIS. Мы начнем с изучения работы с веб-сайтами и приложениями, поскольку веб-сайты и пулы приложений — это два наиболее распространенных компонента, с которыми работают системные администраторы.

Сайты

Начнем с простой команды `Get-Website`, которая запрашивает IIS и возвращает все веб-сайты, имеющиеся на веб-сервере в настоящее время:

```
[WEBSRV]: PS> Get-Website -Name 'Default Web Site'

Name             ID   State      Physical Path                 Bindings
----             --   -----      -------------                 --------
Default Web Site 1    Started    %SystemDrive%\inetpub\wwwroot http *:80:
```

Видно, что тут уже есть какой-то сайт. Это связано с тем, что при установке IIS создает веб-сайт Default Web Site. Предположим, что вам не нужен этот веб-сайт по умолчанию и вы предпочитаете создать свой. Можно удалить его, направив вывод `Get-Website` в команду `Remove-Website`:

```
[WEBSRV]: PS> Get-Website -Name 'Default Web Site' | Remove-Website
[WEBSRV]: PS> Get-Website
[WEBSRV]: PS>
```

Если вы хотите создать веб-сайт, сделать это так же легко, используя команду `New-Website`:

```
[WEBSRV]: PS> New-Website -Name PowerShellForSysAdmins
-PhysicalPath C:\inetpub\wwwroot\

Name             ID   State      Physical Path                 Bindings
----             --   -----      -------------                 --------
PowerShellForSys 1052 Stopped    C:\inetpub\wwwroot\           http *:80:
Admins           6591
```

Если привязки веб-сайта отключены и вы хотите их изменить (скажем, вы хотите привязать его к нестандартному порту), можно использовать команду `Set-WebBinding`:

```
[WEBSRV]: PS> Set-WebBinding -Name 'PowerShellForSysAdmins'
-BindingInformation "*:80:" -PropertyName Port -Value 81
[WEBSRV]: PS> Get-Website -Name PowerShellForSysAdmins

Name             ID   State      Physical Path                 Bindings
----             --   -----      -------------                 --------
PowerShellForSys 1052 Started    C:\inetpub\wwwroot\           http *:81:
Admins           6591
                 05
```

Это уже немалая часть того, что можно делать с веб-сайтами. Давайте теперь посмотрим, что можно делать с пулами приложений.

Пулы приложений

Пулы приложений позволяют изолировать приложения друг от друга, даже если они работают на одном сервере. Таким образом, если в одном приложении появляется ошибка, другие не будут отключены.

Команды для пулов приложений аналогичны командам для веб-сайтов, что видно из кода ниже. Поскольку у меня был только один пул приложений, в моем случае отображается только `DefaultAppPool`. Если вы запустите эту команду на собственном веб-сервере, то сможете увидеть больше:

```
[WEBSRV]: PS> Get-IISAppPool

Name                Status       CLR Ver Pipeline Mode Start Mode
----                ------       ------- ------------- ----------
DefaultAppPool      Started      v4.0    Integrated    OnDemand

[WEBSRV]: PS> Get-Command -Name *apppool*

CommandType     Name                                Version    Source
-----------     ----                                -------    ------
Cmdlet          Get-IISAppPool                      1.0.0.0    IISAdministration
Cmdlet          Get-WebAppPoolState                 1.0.0.0    WebAdministration
Cmdlet          New-WebAppPool                      1.0.0.0    WebAdministration
Cmdlet          Remove-WebAppPool                   1.0.0.0    WebAdministration
Cmdlet          Restart-WebAppPool                  1.0.0.0    WebAdministration
Cmdlet          Start-WebAppPool                    1.0.0.0    WebAdministration
Cmdlet          Stop-WebAppPool                     1.0.0.0    WebAdministration
```

Поскольку мы уже создали веб-сайт, посмотрим, как создать и назначить ему пул приложений. Чтобы создать пул приложений, воспользуйтесь командой `New-WebAppPool`, как показано в листинге 20.3.

Листинг 20.3. Создание пула приложений

```
[WEBSRV]: PS> New-WebAppPool -Name 'PowerShellForSysAdmins'

Name                    State        Applications
----                    -----        ------------
PowerShellForSysAdmins  Started
```

К сожалению, не для всех задач IIS есть встроенный командлет. Чтобы назначить пул приложений существующему веб-сайту, следует использовать команду `Set-ItemProperty` и изменить веб-сайт на диске IIS ❶ (как показано ниже). Чтобы применить это обновление, нужно остановить ❷ и перезапустить веб-сайт ❸.

```
❶ [WEBSRV]: PS> Set-ItemProperty -Path 'IIS:\Sites\PowerShellForSysAdmins'
-Name 'ApplicationPool' -Value 'PowerShellForSysAdmins'
❷ [WEBSRV]: PS> Get-Website -Name PowerShellForSysAdmins | Stop-WebSite
❸ [WEBSRV]: PS> Get-Website -Name PowerShellForSysAdmins | Start-WebSite
   [WEBSRV]: PS> Get-Website -Name PowerShellForSysAdmins |
Select-Object -Property applicationPool
applicationPool
---------------
PowerShellForSysAdmins
```

Вы также можете убедиться, что пул приложений изменился, посмотрев на возвращаемый при запуске команды `Get-Website` атрибут `applicationPool`.

Настройка SSL на веб-сайте

Теперь, когда вы ознакомились с командами для работы с IIS, вернемся к нашему модулю PowerLab и напишем функцию, которая установит сертификат IIS и изменит привязку к порту 443.

Вы можете получить «настоящий» сертификат от действующего центра сертификации или создать самоподписанный сертификат с помощью функции `New-SelfSignedCertificate`. Давайте пока что просто создадим и используем последний, поскольку я просто описываю это понятие.

Сначала разместите функцию и укажите все необходимые параметры (листинг 20.4).

Листинг 20.4. Начало функции New-IISCertificate

```
function New-IISCertificate {
    param(
        [Parameter(Mandatory)]
        [string]$WebServerName,

        [Parameter(Mandatory)]
        [string]$PrivateKeyPassword,

        [Parameter()]
        [string]$CertificateSubject = 'PowerShellForSysAdmins',

        [Parameter()]
        [string]$PublicKeyLocalPath = 'C:\PublicKey.cer',

        [Parameter()]
        [string]$PrivateKeyLocalPath = 'C:\PrivateKey.pfx',

        [Parameter()]
        [string]$CertificateStore = 'Cert:\LocalMachine\My'
    )
    ## The code covered in the following text will go here
}
```

Первое, что эта функция должна делать — создавать самоподписанный сертификат. Это можно сделать с помощью команды `New-SelfSignedCertificate`, которая импортирует сертификат в *хранилище сертификатов* `LocalMachine` на локальном компьютере. Когда вы вызываете команду `New-SelfSignedCertificate`, ей можно передать параметр `Subject` для хранения строки, которая будет содержать информацию о сертификате. При создании сертификата он также импортируется на локальный компьютер.

В листинге 20.5 представлена строка, которую нужно использовать для генерации сертификата с использованием переданного субъекта (`$CertificateSubject`). Помните, что вы можете использовать переменную `$null` для хранения результатов команды, если вам ничего не нужно выводить в консоль.

Листинг 20.5. Создание самоподписанного сертификата

```
$null = New-SelfSignedCertificate -Subject $CertificateSubject
```

После создания сертификата вам необходимо сделать две вещи — получить отпечаток сертификата и экспортировать закрытый ключ из сертификата. *Отпечаток* сертификата — это строка, которая однозначно идентифицирует сертификат. *Закрытый ключ* сертификата используется для шифрования

и дешифровки данных, отправляемых на ваш сервер (не будем вдаваться в подробности).

Вы могли получить отпечаток из вывода `New-SelfSignedCertificate`, но предположим, что этот сертификат будет использоваться на другом компьютере, так как это более вероятно. Чтобы решить эту задачу, нужно сначала экспортировать открытый ключ из вашего самоподписанного сертификата. Это можно сделать с помощью команды `Export-Certificate`:

```
$tempLocalCert = Get-ChildItem -Path $CertificateStore |
    Where-Object {$_.Subject -match $CertificateSubject }
$null = $tempLocalCert | Export-Certificate -FilePath $PublicKeyLocalPath
```

В результате получится файл открытого ключа .cer, который можно использовать для временного импорта сертификата и получения отпечатка, применив немного магии .NET:

```
$certPrint = New-Object System.Security.Cryptography.X509Certificates.
                                                        X509Certificate2
$certPrint.Import($PublicKeyLocalPath)
$certThumbprint = $certprint.Thumbprint
```

Теперь, когда у вас есть отпечаток сертификата, нужно экспортировать закрытый ключ, который нужен для прикрепления к привязке SSL на веб-сервере. Ниже приведены команды для экспорта закрытого ключа:

```
$privKeyPw = ConvertTo-SecureString -String $PrivateKeyPassword -AsPlainText -Force
$null = $tempLocalCert | Export-PfxCertificate -FilePath $PrivateKeyLocalPath
                                                    -Password $privKeyPw
```

Если у вас есть закрытый ключ, вы можете импортировать свой сертификат в хранилище сертификатов на веб-сервере с помощью команды `Import-PfxCertificate`. Однако сначала следует проверить, был ли он уже импортирован. Вот почему вам раньше нужно было получить отпечаток сертификата. Можно использовать уникальные отпечатки для проверки наличия сертификатов на веб-сервере.

Чтобы импортировать сертификат, нужно использовать несколько команд, которые вы ранее видели в этой главе. Мы создадим прямой сеанс PowerShell, импортируем модуль `WebAdministration`, проверим существование сертификата, а затем добавим его при отсутствии такового. Все, кроме последнего этапа, делается в листинге 20.6.

Листинг 20.6. Проверка наличия сертификата

```
$session = New-PSSession –VMName $WebServerName
–Credential (Import-CliXml –Path C:\PowerLab\DomainCredential.xml)

Invoke-Command –Session $session –ScriptBlock {Import-Module –Name
WebAdministration}

if (Invoke-Command –Session $session –ScriptBlock { $using:certThumbprint –in
(Get-ChildItem –Path Cert:\LocalMachine\My).Thumbprint}) {
    Write-Warning –Message 'The Certificate has already been imported.'
} else {
    # Code for importing the certificate
}
```

Первые две строки кода вам должны быть знакомы из предыдущих частей этой главы. Но обратите внимание, что нужно использовать `Invoke-Command` для удаленного импорта модуля. Аналогично, поскольку внутри блока сценария в операторе `if` находится локальная переменная, для ее расширения на удаленном компьютере необходимо использовать префикс `$using`.

Давайте заполним код для оператора `else` в листинге 20.7. Чтобы завершить настройку сертификата IIS, следует сделать четыре вещи. Сначала необходимо скопировать закрытый ключ на веб-сервер. Затем нужно импортировать закрытый ключ с помощью `Import-PfxCertificate`. Наконец, стоит установить привязку SSL, а затем заставить ее использовать закрытый ключ.

Листинг 20.7. Привязка сертификата SSL к IIS

```
Copy-Item -Path $PrivateKeyLocalPath -Destination 'C:\' -ToSession $session

Invoke-Command -Session $session -ScriptBlock { Import-PfxCertificate
-FilePath $using:PrivateKeyLocalPath -CertStoreLocation
$using:CertificateStore -Password $using:privKeyPw }

Invoke-Command -Session $session -ScriptBlock { Set-ItemProperty "IIS:\Sites\
PowerShellForSysAdmins" -Name bindings
-Value @{protocol='https';bindingInformation='*:443:*'} }

Invoke-Command -Session $session -ScriptBlock {
    $cert = Get-ChildItem -Path $CertificateStore |
        Where-Object { $_.Subject -eq "CN=$CertificateSubject" }
    $cert | New-Item 'IIS:\SSLBindings\0.0.0.0!443'
}
```

Обратите внимание, что в этом коде сайт настроился на использование порта 443 вместо порта 80. Мы делаем это, чтобы убедиться, что веб-сайт придерживается типичного порта SSL 443, ведь благодаря этому веб-браузеры понимают, что включено шифрование веб-трафика.

На этом все! Вы успешно установили самоподписанный сертификат на веб-сервере, создали привязку SSL для своего сайта и заставили ее использовать ваш сертификат! Единственное, что осталось сделать, это очистить сеанс, в котором вы работали:

```
$session | Remove-PSSession
```

После очистки сеанса вы можете перейти по адресу https://<webservername>, где вам будет предложено подтвердить сертификат. Браузеры будут делать это, поскольку вы выпустили самоподписанный сертификат вместо сертификата из публичного центра сертификации. Если вы доверяете сертификату, появится веб-страница IIS по умолчанию.

Не забудьте проверить функцию `New-IISCertificate` внутри модуля PowerLab, чтобы увидеть все эти команды в одном месте.

Итоги

В этой главе мы рассмотрели еще один тип сервера — веб-сервер. Мы узнали, как создать его с нуля. Мы также изучили некоторые команды модуля `WebAdministration`, который поставляется с IIS. Вы узнали, как использовать встроенные команды для выполнения многих базовых задач, а также посмотрели на созданный диск IIS PowerShell. Чтобы завершить главу, вы подробно следовали практическому сценарию, который требовал объединения многих рассмотренных ранее команд и методов.

Если вы прочитали всю книгу, то я вас поздравляю! Мы прошли большой путь, и я рад, что вы не сошли с него. Полученные навыки и созданные проекты должны дать вам основу для решения задач с помощью PowerShell. Сохраните эти знания, закройте книгу и приступайте к написанию сценариев. Возьмите любую задачу и автоматизируйте ее с помощью PowerShell. Единственный способ по-настоящему овладеть понятиями, изложенными в этой книге, — это практика.

Нет лучше времени начать, чем сейчас!

Экономьте время — автоматизируйте!

Адам Бертрам

PowerShell для сисадминов

Перевел с английского С. Черников

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*А. Юринова*
Научный редактор	*А. Логунов*
Художественный редактор	*В. Мостипан*
Литературный редактор	*С. Ворожцов*
Корректоры	*С. Беляева, М. Молчанова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 05.2021. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева,
д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 29.04.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 33,540. Тираж 500. Заказ 0000.

Отпечатано в полном соответствии с качеством предоставленных материалов в ООО «Фотоэксперт».
109316, г. Москва, Волгоградский проспект, д. 42, корп. 5, эт. 1, пом. I, ком. 6.3-23Н.

Пол Тронкон, Карл Олбинг

BASH И КИБЕРБЕЗОПАСНОСТЬ: АТАКА, ЗАЩИТА И АНАЛИЗ ИЗ КОМАНДНОЙ СТРОКИ LINUX

Командная строка может стать идеальным инструментом для обеспечения кибербезопасности. Невероятная гибкость и абсолютная доступность превращают стандартный интерфейс командной строки (CLI) в фундаментальное решение, если у вас есть соответствующий опыт.

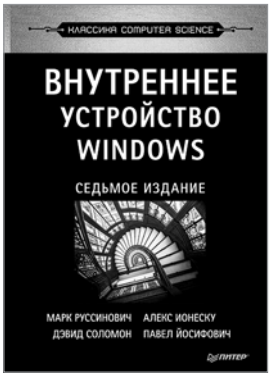
Авторы Пол Тронкон и Карл Олбинг рассказывают об инструментах и хитростях командной строки, помогающих собирать данные при упреждающей защите, анализировать логи и отслеживать состояние сетей. Пентестеры узнают, как проводить атаки, используя колоссальный функционал, встроенный практически в любую версию Linux.

КУПИТЬ

Марк Руссинович, Дэвид Соломон,
Алекс Ионеску, Павел Йосифович

ВНУТРЕННЕЕ УСТРОЙСТВО WINDOWS

7-е издание



С момента выхода предыдущего издания этой книги операционная система Windows прошла длинный путь обновлений и концептуальных изменений, результатом которых стала новая стабильная архитектура ядра Windows 10.

Книга «Внутреннее устройство Windows» создана для профессионалов, желающих разобраться во внутренней жизни основных компонентов Windows 10. Опираясь на эту информацию, разработчикам будет проще находить правильные проектные решения, создавая приложения для платформы Windows, и решать сложные проблемы, связанные с их эксплуатацией. Системные администраторы, зная, что находится у операционной системы «под капотом», смогут разобраться с поведением системы и быстрее решать задачи повышения производительности и диагностики сбоев. Специалистам по безопасности пригодится информация о борьбе с уязвимостями операционной системы.

Прочитав эту книгу, вы будете лучше разбираться в работе Windows и в истинных причинах того или иного поведения ОС.