ВТОРОЕ ИЗДАНИЕ

ИЗУЧАЕМ СКРИПТИНГ POWERSHELL

ЗА МЕСЯЦ, ЗАНИМАЯСЬ ОДИН ЧАС В ДЕНЬ

Создание скриптов и инструментов

- Язык скриптов
- Среда разработки скриптов
- Конвейер PowerShell
- Привязка параметров

- Как избежать ошибок
- Базовые функции
- Продвинутые функции
- Модули для скриптов

- Объекты
- Заполнение манифеста
- .NET Framework
- Конвейеры

- Управление исходным кодом с помощью Git
- Комментарии
- Создание профессиональных скриптов

ДЖЕЙМС ПЕТТИ · ДОН ДЖОНС
И ДЖЕФФРИ ХИКС

MANNING

Learn PowerShell Scripting in a Month of Lunches, Second Edition

WRITE AND ORGANIZE SCRIPTS AND TOOLS

JAMES PETTY, DON JONES AND JEFFERY HICKS

MANNING
SHELTER ISLAND

Изучаем скриптинг PowerShell за месяц, занимаясь один час в день

Создание скриптов и инструментов

Джеймс Петти, Дон Джонс, Джеффри Хикс

ПИТЕР®

Санкт-Петербург • Москва • Минск

2025

ББК 32.988.02-018
УДК 004.42:004.738.5
П31

Петти Джеймс, Джонс Дон, Хикс Джеффри

П31 Изучаем скриптинг PowerShell за месяц, занимаясь один час в день. 2-е изд. — СПб.: Питер, 2025. — 352 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-4184-5

Скрипты PowerShell можно писать для автоматизации практически любых административных задач в Windows, Linux и macOS. Эта книга покажет вам как! Всего за 27 коротких уроков, каждый из которых можно выполнить менее чем за час, вы научитесь создавать, тестировать и публиковать скрипты и инструменты, которые помогут сэкономить часы рабочего времени.

«Изучаем скриптинг PowerShell за месяц» — практическое руководство по автоматизации PowerShell и созданию инструментов. Этот тщательно переработанный бестселлер, обновленный до последней версии PowerShell, научит вас писать эффективные скрипты, находить и устранять ошибки и организовывать свои инструменты в библиотеки. Попутно вы получите советы по обеспечению безопасности и работе в Linux и macOS.

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

ББК 32.988.02-018
УДК 004.42:004.738.5

ISBN 978-1633438989 англ.

ISBN 978-5-4461-4184-5

Краткое содержание

Содержание

ЧАСТЬ II

ЧАСТЬ III

ОТЗЫВЫ О ПЕРВОМ ИЗДАНИИ

Ясное и лаконичное описание лучших возможностей PowerShell.

Джастин Коулстон (Justin Coulston), Intellectual Technology

Отличный источник информации для тех, кто хочет создавать скрипты, позволяющие автоматизировать задачи.

Бруно Соннино (Bruno Sonnino), Revolution Software

Реальные примеры, лучшие практики и советы от двух авторитетных специалистов по PowerShell.

Роман Левченко (Roman Levchenko), Microsoft MVP

Книга заставляет остановиться и подумать, а не просто «читать и кивать».

Река Хорват (Reka Horvath), Wirecard CEE

Книга, которую стоит прочитать тем, кто хочет стать экспертом по скриптам PowerShell.

Шанкар Свами (Shankar Swamy), Stealth Mode IoT Device Startup

Посвящается Кейсилин, неиссякаемому источнику поддержки и вдохновения. Спасибо тебе за безграничное терпение и понимание. Твоя любовь стала надежной опорой, помогающей мне реализовывать амбициозные проекты вроде этого. Данная книга — доказательство прочности нашего партнерства.

Кроме того, посвящаю эту книгу двум нашим дочерям. Ваши смех и любопытство наполняют дом радостью. Вы постоянно напоминаете мне о том, как важны простота и красота обучения. Пусть эта книга вдохновит вас реализовывать ваши стремления с таким же энтузиазмом, с каким вы относитесь к миру.

Джеймс Петти

Предисловие

Как человек, непосредственно ощутивший всю мощь PowerShell на практике, я с радостью познакомлю вас с его возможностями в области скриптинга. Кем бы вы ни были: опытным IT-профессионалом или новичком в этой сфере, — я постарался сделать так, чтобы ваше обучение было информативным и интересным.

Каждая глава писалась так, чтобы вы могли проработать материал, уделяя этому час в день. Благодаря этому с книгой удобно работать людям, имеющим плотный рабочий график. Ее назначение — помочь вам стать профессионалом в области скриптинга PowerShell. Занимаясь всего час в день, за месяц вы узнаете все необходимое и овладеете нужными навыками.

На страницах книги вы найдете практические примеры, упражнения и методы решения задач реальной сложности, которые улучшат ваши навыки взаимодействия с PowerShell и позволят уверенно применять их в ходе решения повседневных рабочих задач. Здесь описаны различные темы, начиная с основ скриптинга и заканчивая его продвинутыми техниками, благодаря чему вы можете познакомиться со всеми возможностями PowerShell.

Пусть это путешествие по освоению скриптов PowerShell будет интересным и результативным для вас.

Джеймс Петти

Благодарности

Хочу выразить сердечную благодарность всем людям, которые поддерживали создание данной книги и принимали участие в этом процессе. Отдельное спасибо моим дочерям и супруге за их безграничную поддержку.

Кроме того, выражаю признательность издательству Manning за возможность поделиться своими знаниями и поддержку, которую его сотрудники оказали мне в процессе написания книги.

В частности, хочу поблагодарить выпускающего редактора Фрэнсиса Лефковица (Frances Lefkowitz), технического корректора Кшиштофа Камычека (Krzysztof Kamyczek) и всех причастных к выпуску книги за их поддержку.

Отдельная благодарность научному редактору Уэсу Сталеру (Wes Stahler), CISSP, GCWN, GCIH, GSTRT, MCSD, который является заместителем директора Медицинского центра Векснера при Университете штата Огайо. Уэс с радостью рекламирует достоинства PowerShell и выступал как на государственном уровне в Microsoft Health Users Group, так и на местном для Central Ohio PowerShell Users Group и Central Ohio ISSA.

Выражаю признательность всем рецензентам: Элу Пезевски (Al Pezewski), Дэйву Коруну (Dave Corun), Глену Томпсону (Glen Thompson), Джеффри Яо (Jeffrey Yao), Киту Киму (Keith Kim), Кенту Спиллнеру (Kent Spillner), Марии Ане (Maria Ana), Оливеру Кортену (Oliver Korten), Питеру А. Шотту (Peter A. Schott), Пити Чампитонгу (Piti Champeethong), Ранджиту Сахайу (Ranjit Sahai), Роману Левченко (Roman Levchenko) и Сатеджу Кумар Саху (Satej Kumar Sahu). Ваши рекомендации помогли улучшить эту книгу.

Джеймс Петти

О книге

«Изучаем скриптинг PowerShell за месяц» — полноценное руководство, благодаря которому вы сможете погрузиться в удивительный мир скриптинга. Книга поможет вам выработать и усовершенствовать навыки написания скриптов PowerShell. В первой части описываются предварительные условия и основные сведения, которые нужны для создания скриптов. Во второй части вы перейдете к практической реализации и начнете составлять надежные скрипты PowerShell, сосредоточившись на принципах проектирования и стратегическом мышлении. В третьей части будут представлены более сложные профессиональные практики, которые бросают вызов традиционным моделям мышления и в которых особое внимание уделяется вопросам безопасности. В заключительной, четвертой части вы познакомитесь с продвинутыми приемами скриптинга, освоение которых позволит вам отточить этот навык до совершенства.

ДЛЯ КОГО ЭТА КНИГА

Данное издание предназначено для IT-специалистов, системных администраторов и тех, кто желает получить практические навыки в работе со скриптами PowerShell. Эта книга вполне подойдет и для новичков. Тем не менее мы советуем им начать изучение скриптинга с книги *Learn PowerShell in a Month of Lunches, Fourth Edition*[1] (Manning, 2022). В нашей книге, рассчитанной на читателей с небольшим опытом написания скриптов, используется структурированный практический подход к освоению PowerShell. Но независимо от того, являетесь вы новичком в мире скриптинга или же хотите освоить автоматизацию и системное администрирование, она станет для вас ценным ресурсом.

---

[1] *Петти Дж. и др.* Изучаем PowerShell за месяц, занимаясь один час в день. 4-е издание. — М., 2023.

Авторы предлагают структурированный и практический подход к обучению, особенно в части решения повседневных задач, автоматизации и системного администрирования. Книга станет для вас бесценным источником знаний, а благодаря особой подаче материала обучение будет удобным и последовательным, и вы сможете эффективно интегрировать PowerShell в свой рабочий процесс. Независимо от того, работаете вы в среде Windows или же используете другие технологии Microsoft, с помощью этой книги вы сможете получить важные знания и освоить практические навыки, необходимые для использования потенциала PowerShell при создании скриптов и автоматизации задач.

О КОДЕ

Для того чтобы улучшить читабельность и облегчить понимание кода, приведенного в книге, мы оформили его в соответствии с четкими правилами. Примеры кода, скрипты и дополнительные ресурсы можно найти в нашем репозитории GitHub. Каждый фрагмент кода сопровождается подробными пояснениями, чтобы читатели могли понять его синтаксис и внутренние принципы составления скриптов.

В книге содержится много примеров исходного кода, который находится как в пронумерованных листингах, так и внутри текста. В обоих случаях код оформлен `шрифтом с фиксированной шириной`, чтобы выделить его на фоне обычного текста. При этом в некоторых фрагментах кода **`жирным шрифтом`** выделены изменения, внесенные в предыдущие его версии, например, новая функциональность.

Во многих случаях изначальный исходный код был переформатирован. Мы добавили разрывы строк и переработали отступы, чтобы полноценно использовать пространство страниц. В редких случаях и этого оказалось недостаточно, в связи с чем некоторые листинги содержат маркеры переноса строки (➥). Кроме того, зачастую из кода удалены комментарии, если он описывается в тексте. При этом многие листинги сопровождаются аннотациями с описанием важных концепций.

Выполняемые фрагменты кода можно найти в онлайн-версии книги по ссылке https://livebook.manning.com/book/learn-powershell-scripting-in-a-month-of-lunches-second-edition. Весь код примеров доступен для скачивания на сайте Manning по адресу www.manning.com/books/learn-powershell-scripting-in-a-month-of-lunches-second-edition, а также в репозитории GitHub: https://github.com/psjamesp/MOL-Scripting.

Об авторах

Джеймс Петти (James Petty) — директор отдела информационных технологий в TextRequest. Он четыре раза получал награду Microsoft MVP, а в рамках своей профессиональной деятельности также занимает должность генерального директора в DevOps Collective Inc., некоммерческой организации, предоставляющей услуги образования в сфере технологий. Основная деятельность компании посвящена PowerShell, автоматизации и DevOps. Кроме того, компания получила широкое признание за предоставление серии бесплатных онлайн-ресурсов, в частности PowerShell.org (http://powershell.org/).

Джеймс — основной автор двух книг: «Изучаем PowerShell за месяц, занимаясь один час в день» (четвертое издание на русском языке вышло в 2023 году) и «Изучаем скриптинг PowerShell за месяц», второе издание которой вы читаете прямо сейчас.

Основной профессиональный интерес Джеймса составляет сфера автоматизации, для которой он разрабатывает такие инструменты, как PowerShell, Azure, а также платформы Windows Server.

Джеймс вместе со своей любимой женой, дочерьми, двумя собаками и двумя кошками живет в окрестностях Чаттануги, штата Теннесси.

Дон Джонс (Don Jones) — сооснователь PowerShell.org (http://powershell.org/) и The DevOps Collective. Он 16 раз получал награду Microsoft MVP, а также написал более 60 книг на технологические темы, в числе которых *Own Your Tech Career*[1] (Manning) и несколько книг из популярной серии *In a Month of Lunches*. Кроме того, Дон написал с десяток романов в жанре фэнтези и научной фантастики. Связаться с ним можно через его личный сайт DonJones.com (http://donjones.com/).

---

[1] *Джонс Д.* Soft skills для IT-специалистов. Прокачай карьеру и получи работу мечты. — М., 2022.

Джеффри Хикс (Jeffery Hicks) — ветеран в сфере IT с более чем 30-летним опытом работы. Джеффри специализируется на технологиях Microsoft Server, уделяя особое внимание автоматизации и эффективности. При этом он многократно получал награду Microsoft MVP Award. Джефф — преподаватель и консультант: за последние 20 лет он научил многих профессионалов IT-индустрии по всему миру тонкостям работы с PowerShell и преимуществам автоматизации. Он также является создателем Pluralsight, автором, соавтором и редактором нескольких книг, пишет статьи для множества сайтов и печатных изданий и часто выступает на технологических конференциях и в сообществах пользователей. Подробнее о нем можно узнать по адресу https://github.com/jdhitsolutions/jdhitsolutions.github.io.

От издательства

Мы выражаем огромную благодарность клубу рецензентов IT-литературы ReadIT Club за помощь в работе над русскоязычным изданием книги и их вклад в повышение качества переводной литературы.

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.

О НАУЧНОМ РЕДАКТОРЕ РУССКОЯЗЫЧНОГО ИЗДАНИЯ

Никита Каравцев — DevOps-инженер компании «Московская Биржа», занимается проектированием, установкой и настройкой IT-инфраструктуры, а также автоматизацией и оптимизацией процессов разработки программного обеспечения.

Часть I

Наше путешествие по сложному и удивительному миру скриптинга начинается. Уверенно перемещаться по нему вам помогут знания, которые вы получите в первой части книги. В главе 1 мы поговорим о том, какие условия и факторы необходимо учитывать при создании скриптов, а также какой образ мышления поможет в этом процессе. В главе 2 вы узнаете, как настроить среду для эффективного скриптинга, и поймете, насколько важна правильная конфигурация. В главе 3 вы изучите философию PowerShell, разобрав принципы и процессы принятия решений, которые лежат в основе этого языка, и попутно получите ценную информацию о создании скриптов. В главах 4–7 мы снова затронем фундаментальные понятия, начиная с привязки параметров и конвейера PowerShell и заканчивая кратким знакомством с языками написания скриптов и различными формами самих скриптов. Основное внимание при этом мы будем уделять важнейшему аспекту — безопасности.

Итак, мы приступаем к освоению теории и практики скриптинга.

1 Вводная информация

PowerShell используется в IT-индустрии уже более 15 лет, и за этот срок он прошел невероятный путь развития. Если вы пропустили вступительную часть, то повторю: на сегодня PowerShell является кросс-платформенным инструментом, то есть доступен не только в Microsoft Windows. До сих пор не верится, что Microsoft решила открыть его код. Изначально он создавался для решения конкретной задачи по автоматизации административных задач Windows, но для этого хватило бы более простого языка обработки «пакетных файлов». Создатель PowerShell, Джеффри Сновер (Jeffrey Snover), вместе со своей командой разработки смотрел на ситуацию гораздо масштабнее. Они хотели получить инструмент, который будет интересен разнообразной аудитории. По их мнению, администраторы могут начинать с простого использования команд для быстрого выполнения административных задач — как раз об этом наша предыдущая книга, «Изучаем PowerShell за месяц, занимаясь один час в день». Кроме того, разработчики PowerShell представили возможность автоматизации более сложных задач и процессов с помощью различных скриптов, чему посвящена уже эта книга.

Команда PowerShell также предполагала, что разработчики будут использовать их инструмент для создания новой функциональности, о чем мы тоже поговорим. Аналогично тому, как у вашей микроволновки наверняка есть кнопки, которые вы никогда не нажимали, в PowerShell есть инструменты, которые вы никогда не использовали, поскольку они вам не нужны. Но, прочитав эту книгу, вы сможете освоить самую сложную функциональность этой оболочки: скриптинг или — если вы согласны с нашим видением — *создание инструментов*.

1.1. ЧТО ТАКОЕ СОЗДАНИЕ ИНСТРУМЕНТОВ

По нашему опыту, многие подходят к скриптингу PowerShell так же, как подошли бы к работе с пакетными файлами, VBScript, Python и т. д. — и в этом нет ничего плохого. PowerShell позволяет использовать множество различных стилей

и подходов. Но в итоге вы просто усложняете себе жизнь до тех пор, пока не разберетесь в том, как эта оболочка *хочет* работать. Мы считаем, что PowerShell предназначен именно для создания инструментов.

PowerShell можно использовать для разработки универсальных, не зависящих от контекста *инструментов*, которые называются *командами*. Последние обычно выполняют какую-то небольшую задачу и делают это очень хорошо. В отдельных случаях одной команды может быть недостаточно, но PowerShell разработан так, что позволяет упростить «связывание» нескольких команд. Играть с одним кирпичиком «Лего» вряд ли будет весело, но, когда у вас есть целая коробка деталей, собирать конструктор может быть невероятно увлекательно. Именно этот подход мы используем в данной книге и именно поэтому описываем его как *создание инструментов*. Лучше всего вкладывать свои силы и время в разработку небольших автономных инструментов, которые можно будет соединять с другими. Такой подход делает код пригодным к использованию во множестве ситуаций, не только экономя ваше время, но и сокращая издержки, связанные с отладкой и сопровождением.

Скриптинг с помощью PowerShell подразумевает написание последовательностей команд и инструкций в текстовом файле, обычно с расширением `.ps1`. Эти скрипты, по сути, являются программами, написанными на языке PowerShell, предназначенном для автоматизации задач и управления конфигурацией в системах Windows. Скрипты PowerShell можно использовать для решения широкого спектра задач, начиная с выполнения административных заданий и заканчивая автоматизацией сложных рабочих потоков.

Давайте рассмотрим, чем различаются скриптинг PowerShell и работа с командами в консоли PowerShell.

- *Возможность повторного использования*. В скрипте PowerShell можно определять набор инструкций или функций, доступных для повторного использования в разных задачах. Это позволяет составлять модульный и удобный в сопровождении код. Если же вы работаете в командной строке через консоль PowerShell, то зачастую вводите команды интерактивно и возможность их повторного использования ограничена историей ввода или ручным копированием и вставкой.
- *Структура скрипта*. Скрипты PowerShell имеют структурированный формат и содержат такие элементы, как переменные, циклы, условия и функции, поэтому позволяют решать более сложные задачи. Использование же командной строки в консоли PowerShell обычно подразумевает ввод единичных команд, что затрудняет управление в случае сложных операций.
- *Автоматизация*. Скрипты PowerShell прекрасно подходят для автоматизации. Прописывая последовательности команд, вы можете автоматизировать повторяющиеся задачи, выполнять масштабные операции и планировать запуск скриптов в назначенное время. Такого уровня автоматизации сложно добиться интерактивным использованием команд в консоли.

- *Интерактивность.* Работая в консоли PowerShell, вы можете интерактивно вводить команды и видеть непосредственные результаты. Скрипты же, напротив, обычно не интерактивны и выполняют серию команд без вмешательства пользователя. Тем не менее их можно разрабатывать так, чтобы пользователь запрашивал ввод данных или получал параметры. Это увеличивает гибкость скриптов.
- *Политика выполнения скриптов.* К скриптам PowerShell могут применяться политики выполнения, определяющие возможность их запуска. Такие политики помогают исключить случайное выполнение вредоносных скриптов. При работе с командами в консоли по умолчанию аналогичной политики выполнения нет, поскольку каждая команда выполняется отдельно.
- *Обработка ошибок.* Скрипты PowerShell могут включать в себя механизмы обработки ошибок, позволяя управлять ошибками и исключениями. При вводе команд в консоли возможности обработки ошибок более ограниченны, и в случае сбоя зачастую приходится прибегать к ручному вмешательству или отладке.

Таким образом, скриптинг в PowerShell — это создание повторно используемых, структурированных последовательностей команд, призванных автоматизировать задачи. В свою очередь, работа с командами в консоли PowerShell носит более интерактивный характер и обычно подходит при решении разовых задач. Благодаря скриптам PowerShell системные администраторы и IT-специалисты получают мощный инструмент, позволяющий оптимизировать и автоматизировать задачи по управлению Windows.

1.2. ПОДХОДИТ ЛИ ЭТА КНИГА ВАМ

Книга предназначена для тех, кто уже умеет работать с PowerShell из командной строки и создавать повторно используемые скрипты. Основное внимание в ней уделяется не только самому процессу, но и синтаксису, поэтому хорошо, если вы уже умеете писать скрипты и просто хотите улучшить свои навыки или оценить их уровень. Следовательно, эта книга *не является* вводным руководством по PowerShell. Чтобы ваша дальнейшая работа с ней была успешной, попробуйте ответить на следующие вопросы, не раздумывая над ответами слишком долго.

- Какую команду вы бы использовали для запроса всех экземпляров `Win32_LogicalDisk` с удаленного компьютера? (**Подсказка:** если вы ответили `Get-WmiObject`, то имейте в виду, что ваши знания устарели и вам нужно освежить их, чтобы эта книга оказалась для вас полезной.)
- Какими двумя способами PowerShell может передавать данные из одной команды в другую в рамках конвейера?
- Грамотно составленные команды PowerShell не выводят текст. А что они выводят? Какие команды можно использовать, чтобы улучшить внешний вид вывода на экране?

- Как бы вы выяснили варианты правильного использования команды `Get-WinEvent`, если бы прежде с ней не сталкивались?
- Какие политики выполнения скриптов имеются и что означает каждая из них?

Мы не будем приводить ответы на эти вопросы — если вы в них не уверены, значит, вам стоит начать с чтения книги «Изучаем PowerShell за месяц, занимаясь один час в день». После того как вы ее прочтете и выполните приведенные в ней упражнения, можете переходить к нашей книге. Кроме того, мы предполагаем, что вы уже имеете опыт работы с операционной системой Windows, поскольку наши примеры будут иметь отношение именно к ней.

1.3. ЧТО ВАМ НУЖНО ДЛЯ РАБОТЫ С ЭТОЙ КНИГОЙ

Теперь быстро пробежимся по списку всего, что вам потребуется для эффективной работы с книгой.

1.3.1. Версия PowerShell

Мы писали книгу, используя PowerShell 7.2, но 99 % ее материала применимо и к более ранним версиям Windows PowerShell. Скачайте PowerShell по ссылке https://docs.microsoft.com/en-us/PowerShell/. Обратите внимание: не нужно устанавливать новые версии оболочки, если вы предварительно не изучили этот вопрос. Многие серверные приложения (например, Exchange Server) очень привередливы в отношении версии PowerShell, с которой они будут работать, и установка неподходящей может привести к тому, что все выйдет из строя. Кроме того, имейте в виду, что каждая версия PowerShell совместима только с конкретными версиями Windows — для этой книги мы используем Windows 11 и macOS.

Несмотря на использование PowerShell 7.2 (или более новой версии по мере их выхода), примеры из книги будут работать и в Windows PowerShell (5.1), хотя в ней мы ничего не тестировали. Функциональность, о которой мы будем говорить, настолько фундаментальна и стабильна, что не меняется даже после выхода новых версий оболочки. Для знакомства с новейшими возможностями PowerShell, относящимися к ее конкретным версиям, мы используем материалы с сайта https://PowerShell.org. А эта книга полностью посвящена основам, которые всегда остаются неизменными.

1.3.2. Права администратора

Вам потребуется возможность запустить на своем компьютере консоль PowerShell и ваш редактор от имени администратора (что будет показано в меню Пуск). В основном это нужно для того, чтобы вы смогли проработать примеры, которые приводятся в книге. Если вы не знаете, как запустить PowerShell от имени администратора, то, скорее всего, вам будет сложно читать эту книгу.

1.3.3. Редактор скриптов

Наконец, вам потребуется редактор скриптов. В клиентские версии Windows входит Integrated Script Editor (ISE), который будет работать только с Windows PowerShell. Мы советуем удалить его, поскольку команда PowerShell прекратила поддержку этого инструмента с момента выхода Windows 7. На сегодня Microsoft рекомендует бесплатный кросс-платформенный редактор Visual Studio Code (VS Code). Скачайте его по адресу https://code.visualstudio.com/, и в главе 2 мы покажем, как настроить этот редактор для работы с PowerShell.

ПРИМЕЧАНИЕ

VS Code и PowerShell являются кросс-платформенными. Все приведенные в этой книге принципы и практики применимы к PowerShell, работающей не только в Windows, но и в других системах. Тем не менее на момент написания книги примеры выполняются на компьютерах, работающих только под управлением Windows. В связи с этим рекомендуем использовать именно эту операционную систему (ОС). В противном случае вам придется переводить все наши рабочие примеры в форму, которая будет работать в других ОС.

1.4. КАК РАБОТАТЬ С КНИГОЙ

Предполагается, что вы будете читать по главе в день и для этого вам должно быть вполне достаточно одного часа. (Исключением станет специальная бонусная глава, о которой мы скажем отдельно, когда дойдем до нее.) Уделите дополнительное время — возможно, день или два — выполнению упражнений, приведенных в конце некоторых глав. *Не поддавайтесь* соблазну прочесть несколько глав за раз, даже если у вас больше одного часа свободного времени. Поясним почему. Мы будем приводить множество новых фактов, а вашему мозгу требуется время (и сон!), чтобы качественно упорядочить эти факты, связать их с уже известными вам сведениями и начать преобразовывать все это в *знания*. Когнитивисты (ученые, изучающие мышление) выяснили, что человеческий мозг способен усвоить за день определенный объем информации, и при составлении каждой главы мы старались учитывать эти ограничения. Так что просто читайте по одной главе в день. Попробуйте осваивать таким образом хотя бы по три-четыре главы в неделю, чтобы удерживать в сознании нить повествования. При этом не забывайте выполнять упражнения.

СОВЕТ

Рекомендуем повторять главу и приведенные в ней упражнения в течение двух-трех дней. Так вы лучше запомните материал. Не нужно спешить и читать по нескольку глав за один-два дня.

К слову, об упражнениях — *не нужно* просто их пролистывать и переходить сразу к нашим решениям. Опять же, когнитивисты утверждают, что человеческий мозг эффективнее всего усваивает материал, когда не просто узнает новые факты,

а сразу же применяет полученные знания. Вы можете посчитать какое-то упражнение слишком трудным, но благодаря этой сложности ваш мозг сосредоточится и начнет сопоставлять факты. Прежде чем вы захотите найти легкий ответ и откроете наше решение, вернитесь к предыдущим главам и перечитайте их. Такой поиск ответа позволит информации закрепиться. Да, потребуется приложить чуть больше усилий, но в итоге они окупятся. Если же вы выберете ленивый подход, то просто обманете сами себя, чего мы вам точно не желаем.

1.5. ОЖИДАНИЯ

Прежде чем переходить к основному повествованию, мы хотим убедиться, что вы знаете, чего ожидать. Вы уже можете представить, что выбранная тематика довольно обширна и мы могли бы добавить в книгу очень много материала. Но она была написана так, чтобы вы могли прочесть ее за месяц, уделяя этому час в день, и потому мы были вынуждены ограничить ее объем. Мы постарались предоставить базовую информацию, которую необходимо знать каждому, кто хочет начать писать скрипты и создавать простые инструменты PowerShell. Эта книга не задумывалась как всеобъемлющее руководство.

1.6. КАК ОБРАТИТЬСЯ ЗА ПОМОЩЬЮ

Вы можете обращаться на онлайн-форумы Manning, доступные по адресу http://mng.bz/rjgE, но мы советуем использовать и форум PowerShell, расположенный на https://PowerShell.org. Мы отслеживаем разделы вопросов и ответов на этих форумах, но самое главное: там вы найдете сотни единомышленников, которые делятся своими знаниями и тоже просят совета. Если вы работаете с PowerShell, то очень важно, чтобы вы познакомились с коллегами и единомышленниками, а со временем стали полноценным членом сообщества. Ресурс PowerShell.org предлагает видео с различными советами и рекомендациями, бесплатные электронные книги. Кроме того, вы можете принимать участие в ежегодных живых конференциях и использовать множество других возможностей.

ИТОГИ ГЛАВЫ

Надеемся, к этому моменту вы готовы погрузиться в книгу и начать писать скрипты или, что еще лучше, *создавать инструменты*. Вы должны установить и настроить все необходимые программы, а также иметь представление о том, сколько времени сможете уделять чтению каждую неделю.

2 Настройка среды для скриптинга

Итак, начнем погружение в глубины скриптинга. Но для начала нужно убедиться, что у вас есть подходящая среда, которую вы сможете использовать в ходе работы с книгой. На прочтение этой главы у вас может уйти полтора часа, но вы должны выполнить каждый описанный в ней шаг, чтобы организовать среду, в которой сможете выполнять упражнения, приводимые в конце большинства глав.

2.1. ОПЕРАЦИОННАЯ СИСТЕМА

Несмотря на кросс-платформенность PowerShell, мы будем работать преимущественно в Windows, поскольку этот язык используется в основном на устройствах, работающих под ее управлением. Вам потребуется компьютер с Windows 10 или 11. Можно использовать и Windows 7, однако имейте в виду, что на данный момент она уже не поддерживается, поэтому по возможности выполните обновление. Вопреки уже упомянутой кросс-платформенности PowerShell, некоторые из наших примеров будут ориентированы именно на Windows, поэтому мы рекомендуем использовать машину именно с этой системой. Если же у вас нет ПК с Windows (возможно, вы предпочитаете Mac или Linux), то можете запустить виртуальную машину (ВМ) с Windows 11, воспользовавшись услугами подходящего вам облачного провайдера. Включайте ее, когда потребуется, и выключайте, когда закончите работу. Кроме того, при желании вы можете использовать Windows Server (2019 или новее).

2.2. POWERSHELL

Вполне очевидно, что для работы с книгой вам потребуется установить PowerShell. Но имейте в виду, что мы будем использовать PowerShell 7, а не Windows PowerShell (5.1), которая установлена в вашей системе по умолчанию. Так что вам нужно установить именно PowerShell 7 (далее просто PowerShell). Если вы читали

«Изучаем PowerShell за месяц, занимаясь по часу в день», то наверняка уже устанавливали оболочку. Если нет, то описание данной процедуры легко найти в Интернете. А в официальном репозитории GitHub содержатся актуальные инструкции: https://github.com/PowerShell/PowerShell. Кроме того, вы можете использовать подходящий вам пакетный менеджер наподобие Chocolatey или Winget. Мы не рекомендуем устанавливать предрелизную версию, превью или бету. Мы будем использовать PowerShell 7.2, поскольку эта версия поддерживается давно. Вы же можете установить 7.3 или более новую, и никаких проблем при этом возникнуть не должно.

2.3. ПРАВА АДМИНИСТРАТОРА И ПОЛИТИКА ВЫПОЛНЕНИЯ

Вам необходимо убедиться, что вы можете запустить PowerShell от имени администратора на своем компьютере. Это может быть невозможно сделать на вашем рабочем компьютере, так что данный нюанс стоит проверить. Для начала запустите консоль PowerShell (нажмите Windows+R, введите `PowerShell` и нажмите Enter). Если в заголовке окна не отобразится надпись Administrator, то щелкните правой кнопкой мыши на значке PowerShell, расположенном на панели задач, и выберите Run as Administrator. Должно открыться новое окно Administrator (перед этим может появиться всплывающее окно User Access Control, в котором нужно будет подтвердить запрос на управление доступом пользователя). Если эта последовательность шагов не сработает, то *можете не продолжать*. У вас точно возникнут сложности с выполнением примеров из книги, и сначала нужно получить права на доступ от имени администратора.

Открыв оболочку от имени администратора, выполните команду `Get-ExecutionPolicy`. В результате должно вернуться что-то кроме `AllSigned`, например `RemoteSigned`, `Unrestricted` или `Bypass`. В противном случае попробуйте выполнить команду `Set-ExecutionPolicy RemoteSigned`. Если сработает, значит, все хорошо. Если же возникнут ошибки или вы получите предупреждения, то ваша политика выполнения наверняка не изменится. Прежде чем вы продолжите работать с книгой, обратитесь за помощью к IT-специалистам вашей компании. Если при решении этого вопроса возникнут сложности, то вы можете поискать ответ на форумах ресурса PowerShell.org (https://forums.PowerShell.org/).

2.4. РЕДАКТОРЫ СЦЕНАРИЕВ

Для выполнения примеров, упражнений и заданий вам потребуется редактор сценариев. В 2017 году компания Microsoft объявила, что прекращает поддержку Integrated Script Editor (ISE), и PowerShell 7 в нем не работает. Мы советуем (и будем использовать) другой бесплатный кросс-платформенный редактор от Microsoft — Visual Studio Code. Скачать его можно с сайта https://code.visualstudio.com. Как обычно, мы рекомендуем скачивать и устанавливать последние стабильные

релизы, а не превью или инсайдерскую сборку. Вы можете использовать любой подходящий вам редактор, а мы, как уже сказали, для этой книги выбрали VS Code и советуем вам сделать то же самое.

После установки окно запуска VS Code будет выглядеть следующим образом (рис. 2.1). Мы изменили тему на Light+, чтобы снимок экрана был лучше виден на бумаге.

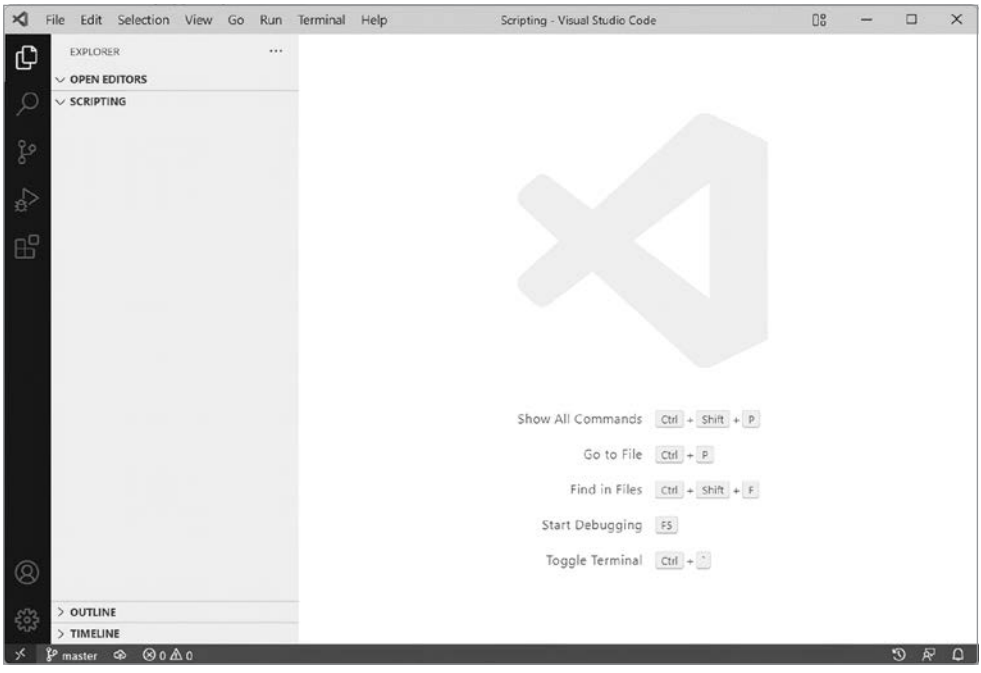


Рис. 2.1. Окно запуска VS Code

Иногда вы будете получать от VS Code оповещение о том, что он обновился и хочет перезапуститься. Позвольте ему сделать это. Обновление займет считаные секунды, зато вы можете быть уверены, что используете наиболее стабильную версию.

Сразу же после установки редактора нужно будет установить расширение, которое позволит ему понимать PowerShell. В вертикальном столбце слева нижний значок предоставляет доступ к *расширениям* (extensions) VS Code. Щелкнув на нем, вы увидите следующий экран (рис. 2.2). Здесь вы заметите, что несколько расширений уже установлены.

Расширение для PowerShell еще не было установлено, так что нужно это сделать. На панели поиска введите `PowerShell` (рис. 2.3). Щелкните на расширении PowerShell (убедитесь, что это не превью-версия), после чего нажмите кнопку Install.

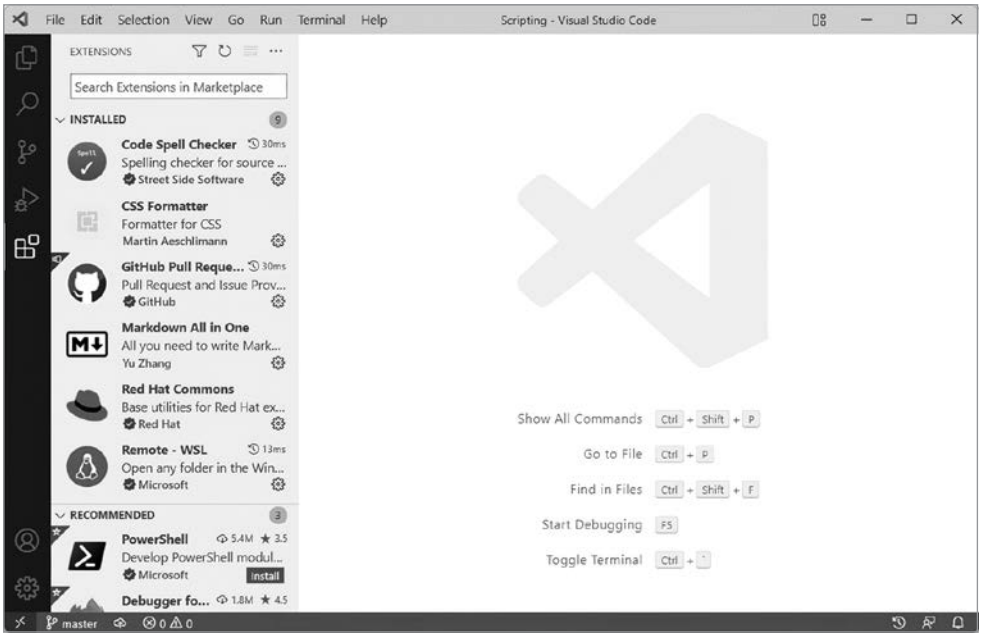


Рис. 2.2. Панель Extensions позволяет устанавливать расширения и управлять ими

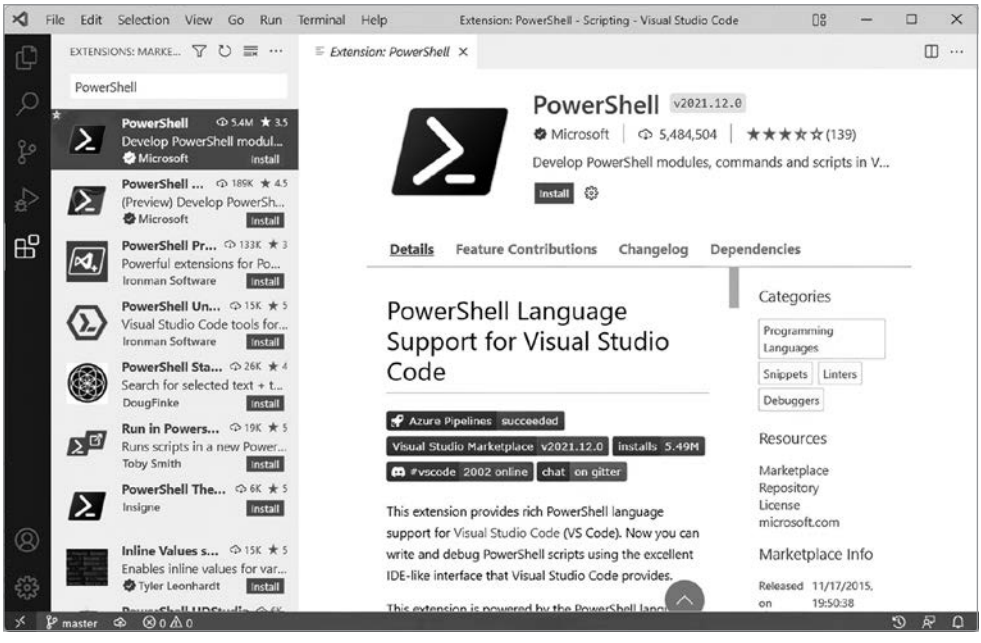


Рис. 2.3. Установка расширения PowerShell

Расширение PowerShell для VS Code часто обновляется, и вы будете получать соответствующие уведомления. Они будут появляться в нижнем правом углу окна редактора. Мы рекомендуем всегда обновлять это расширение, как только появляется новая версия. Еще несколько полезных настроек описаны ниже.

1. Чтобы легко найти файл `settings.json`, откройте Command Palette, щелкнув на View a Command Palette. Любители все делать с клавиатуры могут нажать Ctrl+Shift+P.
2. В качестве расширения по умолчанию можете установить PS1, добавив в файл `settings.json` следующее: `"files.defaultLanguage": "powershell"` (значение `powershell` *должно быть* написано строчными буквами).
3. Настоятельно рекомендуем сделать так, чтобы скобки имели разные цвета. Добавьте в файл `settings.json` следующий код (эта настройка может быть включена по умолчанию):

```
"editor.bracketPairColorization.enabled": true
```

Эта книга, конечно, не является руководством по VS Code, но по мере продвижения вперед мы будем давать полезные советы и подсказывать приемы, которые позволят работать с PowerShell в этом редакторе более эффективно.

ПРИМЕЧАНИЕ

Мы думаем, что многие до сих пор работают в Windows PowerShell. Если вы действительно используете PowerShell ISE и Windows PowerShell (5.1), то можете продолжать делать это, но имейте в виду: в таком случае диапазон доступного вам функционала сильно сузится (даже при наличии надстроек наподобие ISESteroids); особенно это касается отладки. VS Code — официальный редактор для PowerShell, и мы не видим причин его не использовать, но выбор за вами.

2.5. ЛАБОРАТОРНАЯ СРЕДА

Для написания книги мы использовали конфигурацию из четырех машин. Вот их имена и операционные системы:

- Srv01 — Windows Server 2022;
- Srv02 — Windows Server 2022;
- DC01 — Windows Server 2022;
- Client1 — Windows 11.

Предлагаем и вам по возможности настроить аналогичную среду, чтобы упростить работу, ведь тогда ваш экран будет выглядеть как тот, который мы использовали для выполнения примеров и написания этой книги. Лабораторию можно создать

несколькими способами. Естественно, если вы можете развернуть четыре виртуальные машины (ВМ) в лаборатории или отдельном пространстве на работе, то лучше всего так и сделать. Если на ваших машинах установлены Windows 10 или 11 Pro Enterprise, то вы можете включить Hyper-V и создать на локальной машине виртуальную лабораторию. Существуют и другие бесплатные, с открытым исходным кодом или платные варианты создания ВМ на локальной машине. Выбор за вами.

Для создания нашей лаборатории мы использовали AutomateLab. Это бесплатный проект с открытым исходным кодом, и он отлично подходит для решения наших задач. Файл определения лаборатории находится в папке `Resources` в дополнительных материалах к данной книге.

2.6. КОД ПРИМЕРОВ

Наконец, мы настоятельно советуем вам скачать код примеров книги. Издательство Manning разместило его в виде ZIP-архива в соответствующем репозитории: https://github.com/psjamesp/MOL-Scripting. Архив упорядочен по главам. Для каждой главы есть текстовый файл, содержащий все листинги кода. По ходу работы с книгой мы будем вводить кое-какие модули, которые тоже будут упорядочены по главам.

Вы можете скачать указанный ZIP-архив с GitHub либо клонировать репозиторий на свой компьютер. Мы предлагаем использовать клонирование, чтобы вы всегда могли убедиться в актуальности файлов, поскольку мы наверняка будем вносить в них какие-либо изменения. Просматривая примеры кода, вы увидите, что имена модулей повторяются. Дело в том, что последующие главы строятся на основе предыдущих. Мы не ожидаем, что вы обязательно импортируете и используете модули, но соответствующие инструкции дадим.

Напоследок уточним, что все примеры кода в книге приведены исключительно в целях *обучения*. Ни один из них не следует считать готовым к использованию в производственной среде, хотя такой соблазн может возникнуть.

2.7. УПРАЖНЕНИЯ

Скачайте код примеров, установите VS Code и его расширение PowerShell. Если VS Code *работает*, то у вас должно получиться сохранить пустой файл с расширением `.ps1` и затем в редакторе ввести что-нибудь наподобие `Get-P`. В этот момент должна сработать система IntelliSense и выдать варианты автодополнения команды, например `Get-Process`. Если все работает именно так, то вы готовы продолжать. Если нет, то остановитесь и настройте все должным образом. Как мы уже говорили, мы будем следить за вопросами, появляющимися на форумах https://forums.PowerShell.org, так что смело обращайтесь за помощью.

ИТОГИ ГЛАВЫ

Мы завершаем эту главу, посвященную настройке среды для написания скриптов. Напомним ее ключевые моменты.

Первым делом вы должны были убедиться, что ваша среда подходит для выполнения заданий и упражнений, приведенных в книге. Вы должны были получить доступ к компьютеру, работающему под управлением Windows 10 или 11, с правами администратора, а также выполнить установку PowerShell 7. А еще вам нужно было настроить в качестве редактора сценариев Visual Studio Code, добавив в него расширение PowerShell. Это значительно облегчит работу со скриптами.

Кроме того, мы поделились советами о том, как создать лабораторную среду, аналогичную нашей, которая позволяет перемещаться по примерам и упражнениям. Независимо от того, как вы развертываете виртуальные машины: локально или с помощью инструментов наподобие AutomatedLab, — наличие лабораторной конфигурации, похожей на нашу, значительно упростит ваше обучение.

Наконец, вам важно скачать примеры кода из этой книги и познакомиться с функциональностью VS Code, такой как IntelliSense. Еще раз напомним, что примеры предназначены исключительно для обучения. Если вы все же решитесь реализовывать их в производственной среде, то будьте осторожны.

Если в процессе чтения у вас возникнут сложности, не стесняйтесь обращаться с вопросами к онлайн-сообществу — его участники всегда готовы помочь. Удачного скриптинга!

3 Что бы сделал PowerShell

Прежде чем начать, поговорим о «правильном способе» работы с PowerShell. Одним из преимуществ этого инструмента, как и одним из его недостатков, является то, что он позволяет использовать разные подходы к программированию. Если до знакомства с ним вы работали с VBScript, то PowerShell позволит вам писать скрипты, очень похожие на скрипты этого языка. Если вы приверженец C#, то PowerShell без проблем будет выполнять C#-подобные сценарии. Но при этом PowerShell не является ни VBScript, ни C#. Если вы хотите использовать преимущества данного инструмента и переложить на него максимум основной работы, то должны понять базовые принципы. Мы будем *часто* напоминать об этом, начиная с текущего момента. Но при этом важно помнить: если мы выполняем задачи определенным способом, это не означает, что он единственно верный. Просто мы предпочитаем делать именно так. Когда дело касается скриптинга, мы обычно следуем рекомендациям сообщества.

Чтобы было понятнее, приведем такую метафору. Из пункта А в пункт Б можно доехать на машине, но при этом использовать разные подходы. Например, можно поставить на нейтральную передачу, выйти из машины и толкать до самого пункта Б. Еще можно привязать машину к лошади, которая будет ее тянуть. Этих животных на протяжении многих веков успешно использовали для транспортировки, зачем придумывать что-то новое, не так ли? Но эффективнее всего воспользоваться тем способом, который и предполагался при создании машины: заправить ее бензином, сесть за руль и нажать педаль газа. Так вы будете двигаться намного быстрее любой лошади, потратите меньше сил, чем если бы толкали автомобиль, а сами в этом путешествии будете чувствовать себя более счастливым и здоровым.

Именно для этого и нужен PowerShell — чтобы отвязать лошадь, сесть в машину *и ехать*.

3.1. ОДИН ИНСТРУМЕНТ, ОДНА ЗАДАЧА

Обратите внимание на рис. 3.1, поскольку это самое важное правило, которое вы должны запомнить, читая книгу.

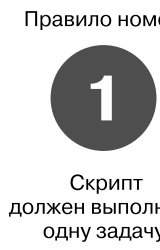


Рис. 3.1. Самое важное правило, которое вам нужно запомнить

PowerShell основывается на использовании небольших однозадачных инструментов (вам они знакомы как командлеты и функции), которые вы можете объединять в одно выражение, чтобы получить значительный результат, приложив минимум усилий. Если вы пришли из другой сферы программирования или скриптинга, то знаете, насколько длинным бывает код некоторых команд. Одна только `Sort-Object` на некоторых языках может занимать десятки строк кода и, например, выглядеть так:

```
Get-ciminstance win32_logicaldisk -filter 'drivetype=3' `
➥ -computername SRV1| Select PSComputername,DeviceID,`
➥ Size,FreeSpace | Select-Object FreeSpace
```

Так что вам нужно запомнить правило «Скрипт должен выполнять одну задачу» и соблюдать его при составлении инструментов. Это правило невероятно важное, и вы неоднократно убедитесь в этом на собственном опыте. Не нужно пытаться создать один гигантский скрипт, который будет выполнять десяток задач. Пишите небольшие, однозадачные инструменты, которые делают что-то одно, причем хорошо. Принцип действия и поведения инструментов, создаваемых вами, не должен отличаться от любой другой штатной команды PowerShell.

НАПУТСТВИЕ ПО СОЗДАНИЮ ОДНОЗАДАЧНЫХ ИНСТРУМЕНТОВ

Принцип «однозадачного инструмента» бывает тяжело усвоить. Мы понимаем, что для многих разработчиков это новая концепция и к ней трудно привыкнуть. В главе 17 будут приведены примеры до и после, которые помогут прояснить данный момент, но кое-что по этой теме мы хотим сказать именно сейчас.

Легко подумать, например, что инициализация нового пользователя — это одна задача, но это не так. Это *процесс*. И если вы представите себе его реализацию вручную, то тут же поймете, что он состоит из нескольких задач. Вам нужно создать пользователя, настроить базовую папку, присвоить лицензию Microsoft 365 (M365), создать почтовый ящик, затем пользовательскую библиотеку в SharePoint и т. д. Если вы начнете описывать этот процесс в коде, то в итоге создадите по одному инструменту на каждую задачу: нового пользователя, новую базовую папку, задачи M365, аккаунт SharePoint и т. д. (многие из этих задач можно выполнить с помощью готовых инструментов Microsoft). После этого вы «соедините» эти инструменты в процесс, написав *скрипт-контроллер*. О них мы поговорим позднее.

Даже такое простое действие, как запись информации в CSV-файл, — это одна задача (и в PowerShell имеется специальный инструмент для ее выполнения). Если у вас есть скрипт, который производит новую информацию и которому нужно время на то, чтобы форматировать ее в CSV и записать в файл, — вы не только допускаете ошибку в реализации, но и сильно затрудняете свою же работу.

Начинайте исходить из того, что *все нужно делать небольшим*. Если вам необходимо автоматизировать какой-то отдельный процесс, то спросите себя: «Какие наименьшие единицы работы можно создать, чтобы реализовать каждую задачу? Можно ли уменьшить что-либо или разделить на несколько частей?» В этом и заключается суть создания инструментов, и мы поступаем так по двум причинам. Первая — чтобы сделать код используемым повторно, а вторая — чтобы увеличить скорость работы.

3.2. ИМЕНОВАНИЕ ИНСТРУМЕНТОВ

В книге мы разберем немало острых тем. Если вы полистаете форумы PowerShell (https://forums.PowerShell.org) или побеседуете с участниками из соответствующих каналов Slack или Discord, то встретите разработчиков, которые будут спорить с вами, отстаивая свое видение того, как «правильно» писать код. И обычно в этом нет ничего удивительного. Каждое такое видение не является ошибочным и имеет право на существование. Одно из достоинств PowerShell состоит в том, что это инструмент с открытым исходным кодом, развиваемый сообществом. Реализовывать одни и те же задачи в данной оболочке можно разными способами. И когда мы показываем в книге какой-то конкретный вариант, это не означает, что он *единственный*. Просто по нашим ощущениям он является наилучшим.

Какое соглашение следует использовать при именовании инструментов[1]? Например, название `ListAllIISWebServersInTheIISWebFarm` вроде бы говорит само за себя, но не вписывается в модель PowerShell. Как уже говорилось в книге «Изучаем PowerShell за месяц, занимаясь по часу в день», в PowerShell для именования используется синтаксис `Verb-Noun`.

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Выполните команду `Get-Verb` и взгляните на вывод.

- Для существительных всегда используйте единственное число: *User*, а не *Users*.
- Предваряйте существительное каким-то обозначением, которое характерно для вашей компании (никогда не используйте *PS*). Если компания называется Globalmantics, то инструмент можно назвать `Get-GlobalUser`.

[1] То, что в PowerShell называется *командой* (общим словом, которое обозначает командлеты, функции и другие исполняемые артефакты), мы называем *инструментом*.

- Начинайте с глагола — но не любого, поскольку в PowerShell есть список утвержденных глаголов. Имейте в виду, что они скорее рекомендуемые, чем строго утвержденные: по факту вы можете использовать любой глагол, но лучше брать из списка. Время от времени разработчики PowerShell добавляют в него новые глаголы.

Такие требования небезосновательны: значительная часть кода PowerShell создана на базе этого соглашения об именовании и утвержденных глаголов. Например, `Get-Command` понимает разницу между глаголом и существительным, помогая обнаруживать соответствующие им команды. Еще один пример: `Import-Module` знает список утвержденных глаголов и отправляет вам предупреждения, когда вы пытаетесь загрузить неутвержденные.

Полный список утвержденных глаголов можно получить, выполнив команду `Get-Verb` или перейдя по ссылке http://mng.bz/VRzr.

3.3. ИМЕНОВАНИЕ ПАРАМЕТРОВ

Можете верить или нет, но именование параметров не менее важно (кто-то может сказать, что даже важнее), чем именование команд. Как вы вскоре узнаете, это ключевой элемент, позволяющий соединять команды в цепочку (конвейер). Кроме того, именование параметров — решающий фактор обнаружения команд с помощью `Get-Command`.

ПРОВЕРОЧНЫЕ ВОПРОСЫ

1. Если вы напишете команду, которая может подключаться к удаленным компьютерам, то параметр с каким именем будет получать имена или адреса этих компьютеров?
2. Если вы напишете команду, которая может выводить файл данных, то параметр с каким именем будет получать информацию о месте расположения файла и его имя?
3. Какое имя подойдет параметру, который будет получать объект сеанса для использования, если вы напишете команду, способную выполняться через существующий сеанс удаленного управления PowerShell?

Чтобы получить ответы на эти вопросы, вам, возможно, придется углубиться в изучение темы — в этом и смысл.

При выборе имени параметра сосредоточьтесь на основных, нативных командах PowerShell (а не на дополнительных модулях наподобие Active Directory). Что используется *в них* в аналогичных ситуациях? Основные команды неизменно

используют параметры `-ComputerName`, а не альтернативы вроде `-Host`, `-MachineName` или какие-либо другие. Вот еще несколько примеров.

1. Здесь основные команды немного непоследовательны, но в *большинстве* из них используется параметр `-FilePath` или `-Path`. В качестве рабочего примера нам послужит команда наподобие `Out-File`, которая использует `-FilePath`.
2. Основные команды для удаленной работы, такие как `Invoke-Command`, выполняют эту задачу с помощью параметра `-PSSession`.

Сомневаетесь в том, удачное ли имя выбрали для параметра? Посмотрите, используется ли оно другими командами PowerShell:

```
get-command -CommandType Cmdlet -ParameterName ComputerName
```

Если вы не найдете совпадений, это еще не значит, что такое имя *не следует* использовать, но наверняка есть более удачная альтернатива.

Идея в том, чтобы действовать *последовательно* и *согласованно*. Опять же, вы увидите, что это очень важно для того, чтобы команды можно было соединять в конвейер. Согласованное именование параметров играет значимую роль во многих внутренних процессах, поэтому не думайте, что у вас есть хорошая причина отклониться от нормы.

ПРОВЕРОЧНЫЙ ВОПРОС

1. Почему вы считаете, что использование параметра `-Host` — неудачная идея?
   a. При использовании команды `Invoke-Command` или `Enter-PSSession` можно задействовать параметр `-Computer-Name` или `-hostname`. `ComputerName` реализует подключение к удаленным машинам через Windows Remote Management (WinRM), а `Hostname` — через SSH.

3.4. ПОЛУЧЕНИЕ ВЫВОДА

Вывод PowerShell — это весьма туманная область, в которой можно растеряться, так как здесь задействовано очень много неочевидных процессов. Если вы читали книгу «Изучаем PowerShell за месяц, занимаясь один час в день», то кое-что в этом понимаете. Разработчикам, которые ее не читали, искренне рекомендуем сделать это. Если коротко, то о выводе PowerShell нужно знать следующее.

1. Как вы узнаете из этой книги, команды PowerShell создают в качестве вывода объекты. Это форма структурирования данных, похожая на электронную таблицу Excel. Объект представляет строку в этой таблице, а каждый столбец, по сути, является его *свойством*. Обращаясь к именам свойств, вы можете получать

доступ к их содержимому. Структурированный вывод данных — объекты — составляет основу PowerShell. Если вы не будете учитывать эту идею, то процесс работы с оболочкой окажется весьма непродуктивным.

2. Объекты выводятся и помещаются в *конвейер* PowerShell, по которому передаются очередной команде. В связи с этим, для того чтобы работать в подобном режиме, команды во многих случаях должны получать ввод из самого конвейера. Вы можете продолжать этот процесс сколько захотите, но должны понимать, что объекты могут менять конвейер в зависимости от используемых вами командлетов.
3. Когда последняя команда выдала свои команды конвейеру, он передает эти объекты в систему форматирования. В этот момент объекты все еще являются просто структурированными данными. Их свойства не представляются в каком-то конкретном порядке и не имеют явных предпосылок для отображения каким-то определенным образом.
4. Система форматирования, используя весьма сложный набор правил, о которых мы говорили в книге «Изучаем PowerShell за месяц, занимаясь один час в день», решает, как отобразить объекты на экране. Она выбирает, отображать ли их в виде списка или таблицы, придумывать ли заголовки столбцов и т. д.
5. В результате работы такой системы форматирования появляется множество специализированных директив, в связи с чем изначальные структурированные данные исчезают. Эти директивы полезны только для отображения вывода на экране или отправки его эквивалента в текстовый файл, на принтер или устройство вывода.

Ваши инструменты не должны ничего делать на этапах 4 или 5. То есть вам нужно сосредоточиться на выводе полезных, структурированных данных в виде объектов — и *не беспокоиться* о том, как все это выглядит *на экране*. За время своей работы мы много раз наблюдали, как разработчики стараются создать «привлекательный» вывод, прикладывая для этого слишком много усилий. Мы покажем вам, как сделать это *с помощью PowerShell*. Этот способ, по сути, подразумевает обучение системы форматирования, которая начинает работать на этапе 4. Что касается ваших инструментов, то сосредоточьтесь на получении в выводе правильных данных, не беспокоясь о том, как они будут выглядеть на экране.

3.5. НЕ ГАДАЙТЕ

Мы провели много лет, занимаясь обучением работе с PowerShell, написанием книг и просто разговорами об этой оболочке и ее значимости для IT-профессионалов по всему миру. Если обозначить одну неизменную проблему, с которой сталкиваются разработчики, то это предположения о том, что такое PowerShell и как она должна функционировать. Процитируем древнегреческого философа Эпиктета: «Человек не может начать изучать то, что, по его мнению, он знает».

Работая с PowerShell, вы встретите много знакомых паттернов, особенно если у вас уже есть опыт программирования или скриптинга. Этого следует ожидать. Разрабатывая данную оболочку, создатели анализировали различные языки, чтобы перенять идеи и принципы, подходящие для реализуемой ими парадигмы. (Прочитайте книгу Дона Джонса (Don Jones) *Shell of an Idea* (2020), если хотите больше узнать об истории разработки PowerShell.) Но одно только распознание вами чего-то, напоминающего код Python, еще не говорит о том, что данный элемент будет работать так же, как в Python. Есть разработчики, которые приступают к освоению PowerShell, думая, что с этим инструментом можно обращаться так, словно это знакомый им язык программирования. Исходя из нашего опыта, мы можем сказать, что эти люди в итоге сильно разочаровываются. Вам нужно помнить следующие нюансы.

- В PowerShell есть надежный конвейер, обладающий широкими функциональными возможностями. Тем не менее это не Bash. В данной оболочке конвейер работает иначе.
- Несмотря на то что выполнение команды может приводить к определенному оформлению вывода на экране, это не значит, что работа данной команды ограничивается только этим. Видимые результаты работы PowerShell не всегда в точности отражают «внутренние» процессы.
- В PowerShell есть конструкции скриптинга, такие как `If` и `ForEach`, и все же это не полноценный язык программирования.
- Хоть в PowerShell основная функциональность и завязана на .NET Framework, это не C#. С годами PowerShell стал больше походить на язык программирования, но все равно бывают случаи, когда правильным решением будет «Просто сделать это на C#». Вы можете оказаться в такой ситуации, если будете писать код практически полностью в классах .NET, а не в виде команд PowerShell.

Пожалуй, самая важная рекомендация звучит так: не пытайтесь воспринимать PowerShell с точки зрения исключительно вашего прежнего опыта в PowerShell. Это не VBScript, Perl, Python, KiXtart или batch. Чем больше вы будете пытаться рассматривать PowerShell как что-то из перечисленного, тем больше будете разочарованы и тем сложнее вам будет работать. Не пытайтесь заставить PowerShell соответствовать каким-то вашим ожиданиям. Это самодостаточный инструмент. Если вы читали книгу «Изучаем PowerShell за месяц, занимаясь по часу в день», то уже должны понимать, как нужно использовать этот инструмент. А наша книга подготовит вас к *расширению* PowerShell подходящим для него образом.

3.6. ИЗБЕГАЙТЕ ИННОВАЦИЙ

Эту главу мы закончим советом: *«Не пытайтесь изобретать новые способы делать то или иное»*. Вы наверняка сейчас думаете: «А разве смысл книги не в том, чтобы создавать собственные скрипты?» Да, в нем, но, поверьте, вы не первый человек, кто столкнулся с аналогичной задачей. Сила PowerShell — буквально весь смысл

его существования — состоит в создании из хаоса согласованных плоскостей администрирования. Поэтому не стоит приумножать хаос, придумывая новые подходы. Даже если вы думаете, что в Microsoft упустили какой-то момент и вам известен намного более удачный способ его реализации, перестаньте так думать. Цель создания инструментов в PowerShell — не сделать их *лучше*, чем Microsoft, а сохранять *согласованность* с тем, что было создано ранее.

И ВСЕ ЖЕ ВНОСИТЕ СВОЙ ВКЛАД

Мы не хотим вас ограничивать. Если у вас есть интересная идея или предложение в отношении того, что специалисты Microsoft могут улучшить, выскажите ее.

В данный момент PowerShell — проект с открытым исходным кодом на GitHub (https://github.com/PowerShell/PowerShell). Есть идеи? Составляйте запрос (issue). А лучше создайте новое ответвление в репозитории, разработайте свое улучшение и отправьте запрос на внесение изменений. Так вы можете внести свой вклад в облик будущих версий PowerShell.

ИТОГИ ГЛАВЫ

В этой главе мы хотели подчеркнуть, что вам нужно внимательно понаблюдать за тем, как PowerShell решает задачи, и попытаться повторить эти подходы, а не изобретать свои. Тогда ваши результаты окажутся более понятными для других, потребуют меньших усилий и обеспечат гораздо большую согласованность в рамках оболочки.

В отличие от автомобиля, который вы видите в повседневной жизни — предположительно без лошади впереди, — подход к использованию PowerShell не всегда оказывается очевиден. Хуже того, он не всегда оказывается последовательным, поскольку многие разработчики, даже сотрудники Microsoft, не прислушались к нашему совету из этой главы. Стоит уделить время изучению темы, особенно основных команд, которые предоставлены разработчиками оболочки, чтобы выявить подход PowerShell к решению задач и повторить его наилучшим возможным образом.

4 Привязка параметров и конвейер PowerShell

Возьмем традиционное поведение конвейера из таких оболочек, как Bash и Cmd.exe, смешаем его с уникальной объектно-ориентированной природой PowerShell и добавим немного парсинга команд в стиле Linux. В итоге получится конвейер PowerShell, сложное и мощное средство для создания инструментов в административных решениях. Наша цель — превратить вас из создателя скриптов в создателя инструментов. Для этого вы должны понять устройство конвейера и создавать такие инструменты, которые будут задействовать конвейер полноценно. Мы говорили об этих концепциях в книге «Изучаем PowerShell за месяц, занимаясь по часу в день», однако в этой главе пойдем дальше и будем рассматривать конвейеры не как инструмент, а как то, для чего инструменты нужно *писать*.

4.1. ОПЕРАЦИОННАЯ СИСТЕМА

Начнем с небольшого упражнения. Возьмите лист бумаги (или планшет, на котором можно писать) и нарисуйте несколько рамок, как на рис. 4.1. Теперь впишите в эти рамки какие-нибудь названия команд: возможно, `Get-Process` в первую рамку, `ConvertTo-HTML` во вторую и `Out-File` в третью.

Рис. 4.1. Визуализация конвейера

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Дорисуйте эти рамки. Мы могли бы просто повторить здесь законченные фигуры, и редактор настаивал именно на этом, но ценность в том, чтобы вы сделали это сами.

Это задание может показаться немного глупым, но в нем можно наглядно увидеть, как PowerShell выполняет команды в конвейере: когда одна команда создает объекты, они *по одному* поступают в конвейер и передаются следующей команде. Когда конвейер заканчивается, то есть команд не остается, любые его объекты передаются в систему форматирования PowerShell для вывода на экран.

Разделители на схеме (|) скрывают огромный объем внутренней функциональности, и это очень важно понимать. Легко сказать, что PowerShell передает объекты из одной команды в другую, но *как именно* это происходит?

4.2. ВСЕ ДЕЛО В ПАРАМЕТРАХ

PowerShell использует два метода для выяснения того, как динамически получить данные — объекты — из конвейера и отправить их в команду по другую сторону разделителя. В обоих методах предусматривается получение параметров команды. Иными словами (и это важно учесть), *единственный* вариант для команды получить данные — из параметров. Это говорит о том, что когда вы создаете команду и продумываете ее параметры, то решаете, как она будет получать информацию, в том числе то, как она будет получать ее из конвейера. Таким образом, в этом процессе нет никакой магии — это наука, которая заранее определяется разработчиком, который пишет команду.

Но все же это может выглядеть как магия. Рассмотрите такой вариант:

```
Get-Service | Where Status -eq "Running" | ConvertTo-HTML | Out-File stats.html
```

Мы не хотим, чтобы вы переходили к другим главам, пока не поймете, почему эта команда работает. Вам нужно уяснить, что все команды получают свой ввод только с помощью параметров. Точка. Никаких исключений. Проблема же в том, что зачастую вы не *вводите* имена параметров. Вместо этого PowerShell позволяет вам использовать *позиционные* параметры, когда порядок предоставления значений соответствует порядку параметров, для которых они предоставляются. Чтобы развеять этот ореол таинственности, можно переписать команду, указав каждый параметр полностью:

```
Get-Service | Where-Object -Property Status -eq -Value "Running" | ConvertTo-HTML
```

Команда `Where-Object` здесь особенно интересна. Мы использовали в ней три параметра: `-Property`, оператор `eq` (будучи таковым, он не нуждается в значении) и `-Value`. В реальности вы *никогда* не увидите, чтобы они указывались таким образом, но этот способ написания помогает понять, что все действия команды определяются параметрами.

Далее рассмотрим, как конвейер передает объекты данных от одной команды другой. Для этого в PowerShell используются два приема: `ByValue` (по значению) и `ByPropertyName` (по имени свойства).

4.3. КОНВЕЙЕР: BYVALUE

В PowerShell есть жестко закодированная установка передавать *объекты* из конвейера в команду *целиком*. В связи с этим оболочка всегда будет сначала пытаться передать объекты полностью, прежде чем делать что-то еще. Для этого должны выполняться следующие условия:

- принимающая команда должна определять параметр, который поддерживает получение ввода конвейера `ByValue`;
- этот параметр должен иметь возможность получать объект любого вида, который находится в конвейере.

К примеру, взгляните на вашу схему с командой `Get-Process` в разделе 4.1. Какой вид объекта производит эта команда? В PowerShell попробуйте выполнить `Get-Process | Get-Member` — вывод первой строки будет содержать `TypeName`, идентифицирующий вид объекта, созданного командой. Оказывается, это объект `System.Diagnostics.Process`.

А теперь используйте функциональность справки применительно ко второй предложенной нами команде. Сначала вам нужно выполнить `Update-Help`, чтобы получить файлы справки, а затем — `Help ConvertTo-HTML -ShowWindow`, чтобы изучить нужный раздел справки полностью. Видите ли вы какие-либо параметры команды, которые могут получать объект `[Process]`? Не беспокойтесь, вы его не упустили, поскольку таких не существует.

Возможно, вы видите параметр, который может получать `[Object]` (или `[Object[]]`), верно? В Microsoft .NET Framework `System.Object` служит своеобразным родительским типом для всего остального. Это значит, что все *наследуется* от типа `Object`. А `PSObject` (то есть PowerShell Object) — своеобразный эквивалент для `Object` в PowerShell. Поэтому, когда вы видите, что этот параметр получает `PSObject`, знайте: он может получать, по сути, что угодно. В разделе справки для `ConvertTo-HTML` вы найдете параметр `-InputObject`, который соответствует двум нашим критериям. Он может получать:

- ввод из конвейера, используя метод `ByValue`;
- объекты типа `System.Diagnostics.Process`, поскольку допускает более обобщенный `PSObject`.

Таким образом, PowerShell будет получать вывод `Get-Process` и привязывать его к параметру `-Input-Object` в `ConvertTo-HTML`. В справке по второй команде сказано, что этот параметр «задает объекты, которые должны быть представлены в HTML». Поэтому все, что вы добавляете в `ConvertTo-HTML`, будет захвачено `ByValue` через параметр `-InputObject` и представлено в формате HTML.

4.3.1. Введение команды трассировки

Поначалу работу конвейера непросто понять, поскольку это сложный механизм. В PowerShell есть возможность увидеть процесс передачи объектов с помощью команды `Trace-Command`. Это очень полезный способ отладки привязки параметров конвейера, показывающий решения, которые принимает оболочка, и действия, которые она пытается совершить. Чтобы запустить эту команду, ее нужно выполнить по шаблону `Trace-Command -Name Parameterbinding -Expression { Ваша команда } -PSHost`. Учтите, что ваша команда *действительно выполнится*, поэтому будьте внимательны и не выполняйте просто ради эксперимента что-либо, что может навредить, например, не стоит удалять ряд пользовательских аккаунтов или каждую базовую папку. Это не замена `-Whatif`.

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Запустите PowerShell, выполните следующую команду и взгляните на ее вывод:

```
Trace-Command -Name ParameterBinding -Expression {get-process | select
-first 1} -PSHost
```

4.3.2. Отслеживание привязки параметров ByValue

Выполним `Trace-Command` для текущего примера. Мы выполнили следующую команду, и вам тоже нужно это сделать:

```
PS C:\> trace-command -Expression { get-process | convertto-html | out-null }
  -Name ParameterBinding -PSHost
```

Как видите, мы завершили команду инструкцией `Out-Null`; таким образом мы заглушили стандартный вывод `ConvertTo-HTML`, чтобы сделать его чище. Но вы увидите, что PowerShell обрабатывает передачу объектов из `ConvertTo-HTML` в `Out-Null`, так что это полезный пример.

Сначала вы увидите попытку оболочки *привязать* все именованные аргументы (`NAMED`) для команды `Get-Process`. В нашем случае мы не указывали в ней никаких параметров вручную:

```
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Get-Process]
```

Затем PowerShell ищет позиционные параметры (`POSITIONAL`), которых у нас тоже нет. Далее она проверяет, были ли предоставлены все обязательные параметры (`MANDATORY`), — эту проверку наша команда проходит. Вот что вы увидите:

```
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args [Get-Process]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
[Get-Process]
```

Весь этот процесс — проверка именованных, позиционных и затем обязательных параметров — повторяется для команд `ConvertTo-HTML` и `Out-Null`. Это важный момент: команда может быть по-разному подключена для получения ввода из конвейера, но указание именованных или позиционных параметров *всегда* имеет приоритет, поскольку PowerShell привязывает сначала их. Если бы мы вручную указали, например, `-InputObject`, то помешали бы привязке параметров `ByValue`, поскольку «привязали» бы параметр сами еще до этого. Вот что вы увидите:

```
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [ConvertTo-Html]
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args
[ConvertTo-Html]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
[ConvertTo-Html]
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Out-Null]
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args [Out-Null]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
[Out-Null]
```

Далее PowerShell для каждой из трех команд вызывает код `BEGIN`. Он выполняется один раз до обработки объектов конвейера. Не во всех командах указывается какой-либо код `BEGIN`, но PowerShell дает им всем такую возможность. Вот что вы увидите:

```
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
```

Следующий этап слегка вас удивит, поскольку здесь PowerShell пытается привязать объект конвейера к параметру `Out-Null`. Вот что вы увидите:

```
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to parameters:
[Out-Null]
```

Как вообще что-либо смогло здесь попасть в конвейер? Что ж, предыдущая команда, `ConvertTo-HTML`, явно воспользовалась возможностью скрытно создать вывод из кода `BEGIN`. Теперь PowerShell нужно что-то с ним сделать, несмотря на то, что еще не выполнялась даже первая команда `Get-Process`!

Далее интересный момент. Вы увидите следующее:

```
DEBUG: ParameterBinding Information: 0 : PIPELINE object TYPE = [System.String]
DEBUG: ParameterBinding Information: 0 : RESTORING pipeline parameter's original
values
```

PowerShell идентифицирует тип объекта конвейера как `System.String`. Не поленитесь и прочтите раздел справки по `Out-Null`. Видите ли вы параметры, которые могут принять из конвейера `String`, используя метод `ByValue`?

PowerShell выяснит, что параметр `-InputObject` команды `Out-Null` принимает либо `Object`, либо `PSObject`, поэтому привяжет вывод `ConvertTo-HTML` к этому параметру:

```
DEBUG: ParameterBinding Information: 0 : Parameter [InputObject] PIPELINE INPUT
ValueFromPipeline NO COERCION DEBUG: ParameterBinding Information: 0 : BIND arg
[<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">] to parameter [InputObject]
DEBUG: ParameterBinding Information: 0 : BIND arg [<!DOCTYPE html PUBLIC "-//W3C//
DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">] to param [InputObject]
SUCCESSFUL
```

По факту команда получила из конвейера пару объектов `String`. Они похожи на строки заголовков HTML-файла, что вполне разумно: `ConvertTo-HTML`, вероятно, обрабатывает их как шаблонные операции, прежде чем приступить к своей реальной работе.

Далее мы видим, что обязательная проверка (`MANDATORY`) для `Out-Null` проходит успешно, и продолжаем работать с изначальным шаблонным выводом `ConvertTo-HTML`:

```
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet [Out-Null]
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to parameters: [Out-Null]
DEBUG: ParameterBinding Information: 0 : PIPELINE object TYPE = [System.String]
DEBUG: ParameterBinding Information: 0 : RESTORING pipeline parameter's original values
DEBUG: ParameterBinding Information: 0 : Parameter [InputObject] PIPELINE INPUT
ValueFromPipeline NO COERCION DEBUG: ParameterBinding Information: 0 : BIND arg [<html
xmlns="http://www.w3.org/1999/xhtml">] to parameter [InputObject]
DEBUG: ParameterBinding Information: 0 : BIND arg [<html
xmlns="http://www.w3.org/1999/xhtml">] to param [InputObject]
SUCCESSFUL
```

Продвинемся немного вперед, пропустив все шаблонные заголовки HTML. Мы перейдем к точке, где выполняется `Get-Process` и PowerShell распознает тип объекта, созданного этой командой:

```
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to parameters:
[ConvertTo-Html]
DEBUG: ParameterBinding Information: 0 : PIPELINE object TYPE =
[System.Diagnostics.Process#HandleCount]
```

Далее мы увидим, что эти объекты `Process` привязываются к параметру `InputObject` команды `ConvertTo-HTML`:

```
DEBUG: ParameterBinding Information: 0 : Parameter [InputObject] PIPELINE INPUT
ValueFromPipeline NO COERCION
DEBUG: ParameterBinding Information: 0 : BIND arg [System.Diagnostics.Process]
to parameter [InputObject]
DEBUG: ParameterBinding Information: 0 : BIND arg [System.Diagnostics.Process]
to param [InputObject] SUCCESSFUL
```

Вывод трассировки, естественно, продолжается, но мы ищем именно это: доказательство, что PowerShell выполняет ожидаемые действия. Вы могли заметить, что в предыдущих выводах часто появляется фраза `NO COERCION`. Она указывает на то, что PowerShell смогла привязать вывод как есть, не пытаясь преобразовать его во что-то еще. Приведение (coercion) — одно из действий, которое может еще больше запутать процесс привязки параметров, и именно трассировка выполнения команды помогает все прояснить. Например, PowerShell может *привести* (то есть преобразовать) число в строку так, чтобы итоговая строка могла привязываться к параметру, получающему `String`.

4.3.3. Когда ByValue не подходит

Это был довольно подробный разбор метода привязки `ByValue`, но что делать, если он не работает? Вернитесь к схеме, которую вы нарисовали на бумаге или планшете. Сотрите или зачеркните `ConvertTo-HTML` и `Out-Null`, а во второй ячейке запишите `Stop-Service`. Пока не выполняйте полученную команду — нам нужно поговорить о том, что в итоге произойдет.

Вы знаете, что первая команда создает объекты `Process`. Просмотрите ее справочный файл. Видите ли вы там параметры `Stop-Service`, которые реализуют два следующих пункта?

- Получают ввод из конвейера с помощью метода `ByValue`.
- Получают тип ввода `Process`, `Object` или `PSObject`.

Мы не видим в справке параметров, которые бы подходили под этот критерий, значит, метод `ByValue` не сработает. Что же делать?

4.4. BYPROPERTYNAME

Вы можете заметить в команде `Stop-Service` параметр `-Name`, получающий ввод из конвейера `ByPropertyName` (по имени свойства). Кроме того, он получает ввод `ByValue`, но этот момент мы уже обсудили, и теперь нас интересует конкретно `ByPropertyName`. А означает он следующее: поскольку *параметр* называется `NAME`, PowerShell просматривает объекты конвейера, проверяя, есть ли у них *свойство* `NAME`. Если да, то оболочка берет значения из этого свойства и передает их данному параметру — просто потому, что их названия совпадают.

Используйте команду `Trace-Command` для выполнения `Get-Process | Stop-Service -whatif` (мы добавили `-whatif`, просто чтобы исключить возможные неполадки). Видите, как PowerShell пытается привязать свойство объекта `Name` к параметру `-Name` команды?

PowerShell попытается «связать» таким образом все возможные пары свойств и параметров. Если объект в конвейере содержит свойства `Name`, `ID`, `Description` и `Status`,

а следующая команда этого конвейера содержит параметры `-Name` и `-Status`, то два свойства объекта будут привязаны к этим параметрам (при условии, что `-Name` и `-Status` были запрограммированы принимать ввод `ByPropertyName`). Этот прием может оказаться крайне полезным. Предположим, у вас есть CSV-файл `Users.csv`, который содержит столбцы `samAccountName`, `Name`, `Title`, `Department` и `City`. Загляните в файл справки для `New-ADUser` (`Get-Help New-Aduser -Online`, если у вас не установлен модуль Active Directory). Как вы думаете, что произойдет, если выполнить следующую команду?

```
Import-CSV Users.csv | New-ADUser
```

Поразмышляйте над этой ситуацией. Если у вас есть тестовая площадка, то создайте таким образом CSV-файл и добавьте в него несколько строк информации о вымышленных пользователях. Запустите команду и посмотрите, выполнит ли она ожидаемые действия.

4.4.1. Трассировка ByPropertyName

Рассмотрим еще один пример привязки `ByPropertyName`, обратив внимание на части трассировки, в которых происходит сама эта привязка. Вот наша команда (мы ограничиваем действие команды `Get-Process` — она будет выбирать процессы, имена которых начинаются с буквы *O*; мы делаем это, поскольку знаем, что у нас всего один такой процесс, и такое ограничение сократит вывод):

```
PS C:\> trace-command -Expression { Get-Process -Name o* | Stop-Job } -PSHost -Name
ParameterBinding
```

Посмотрим, что происходит. Начнем с того, что мы проходим через привязку параметров для команды `Get-Process`. На этот раз у нас есть именованный параметр `NAMED` с именем `-Name`, для которого мы предоставили значение `o*`. Но есть и проблема: этот параметр ожидает *массив* строк (показанный в его справочном файле как `[string[]]`), а мы предоставили лишь одну. Поэтому PowerShell создает массив, добавляет в него `o*` и прикрепляет этот одноэлементный массив к параметру:

```
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Get-Process]
DEBUG: ParameterBinding Information: 0 : BIND arg [o*] to parameter [Name]
DEBUG: ParameterBinding Information: 0 : COERCE arg to [System.String[]]
DEBUG: ParameterBinding Information: 0 : Trying to convert argument value from
System.String to System.String[]
DEBUG: ParameterBinding Information: 0 : ENCODING arg into collection
DEBUG: ParameterBinding Information: 0 : Binding collection parameter Name:
argument type [String], parameter type
[System.String[]], collection type Array, element type [System.String],
coerceElementType
DEBUG: ParameterBinding Information: 0 : Creating array with element type
[System.String] and 1 elements
DEBUG: ParameterBinding Information: 0 : Argument type String is not IList,
treating this as scalar
```

```
DEBUG: ParameterBinding Information: 0 : COERCE arg to System.String]
DEBUG: ParameterBinding Information: 0 : Parameter and arg types the same, no
coercion is needed.
DEBUG: ParameterBinding Information: 0 : Adding scalar element of type String to
array position 0
DEBUG: ParameterBinding Information: 0 : Executing VALIDATION metadata:
[System.Management.Automation.ValidateNotNullOrEmptyAttribute]
DEBUG: ParameterBinding Information: 0 : BIND arg [System.String[]] to param [Name]
SUCCESSFUL
```

Далее выполняется стандартная проверка позиционных параметров, сопровождаемая проверкой обязательных:

```
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args
[Get-Process]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
[Get-Process]
```

Теперь мы переходим к команде `Stop-Job`, снова обрабатывая параметры `NAMED`, `POSITIONAL` и `MANDATORY`:

```
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Stop-Job]
DEBUG: ParameterBinding Information: 0 : BIND POSITIONAL cmd line args [Stop-Job]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
[Stop-Job]
```

Затем PowerShell дает каждой из двух команд возможность выполнить свой код `BEGIN`, если они таковой содержат:

```
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
```

В нашем случае возвращается единственный процесс `OSDUIHelper`, который отображается рядом с выводом трассировки:

```
DEBUG: ParameterBinding Information: 0 : BIND arg [System.Diagnostics.Process
(OSDUIHelper)] to parameter [Job]
```

Посмотрим, что PowerShell делает с ним, поскольку мы уверены, что метод привязки `ByValue` не сработает:

```
DEBUG: ParameterBinding Information: 0 : Binding collection parameter Job:
argument type [Process], parameter type
[System.Management.Automation.Job[]], collection type Array, element
type [System.Management.Automation.Job], no coerceElementType
DEBUG: ParameterBinding Information: 0 : Creating array with element type
[System.Management.Automation.Job] and 1 elements
DEBUG: ParameterBinding Information: 0 : Argument type Process is not IList,
treating this as scalar
DEBUG: ParameterBinding Information: 0 : BIND arg [System.Diagnostics.Process
(OSDUIHelper)] to param [Job] SKIPPED
```

Элемент `SKIPPED` в конце (выделен жирным) сообщает нам, что привязка `ByValue` в итоге не сработала. PowerShell пытался! Параметр `-Job` команды `Stop-Job` получает ввод с помощью метода привязки `ByValue`, поэтому PowerShell попробовал его выполнить. Этот параметр ожидает один или более объектов типа `Job`, в связи с чем PowerShell создал массив и добавил в него наш объект `OSDUI-Helper`, имеющий тип `Process`. Но он никак не смог превратить `Process` в `Job`, поэтому прекратил попытки. Пора переходить к плану Б:

```
DEBUG: ParameterBinding Information: 0 : Parameter [Id] PIPELINE INPUT
ValueFromPipelineByPropertyName NO COERCION
DEBUG: ParameterBinding Information: 0 : BIND arg [5248] to parameter [Id]
DEBUG: ParameterBinding Information: 0 : Binding collection parameter Id: argument
type [Int32], parameter type [System.Int32[]],
collection type Array, element type [System.Int32], no coerceElementType
DEBUG: ParameterBinding Information: 0 : Creating array with element type
[System.Int32] and 1 elements
DEBUG: ParameterBinding Information: 0 : Argument type Int32 is not IList,
treating this as scalar
DEBUG: ParameterBinding Information: 0 : Adding scalar element of type Int32 to
array position 0
DEBUG: ParameterBinding Information: 0 : Executing VALIDATION metadata:
[System.Management.Automation.ValidateNotNullOrEmptyAttribute]
DEBUG: ParameterBinding Information: 0 : BIND arg [System.Int32[]] to param [Id]
SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet
[Stop-Job]
```

Объект `Process` содержит свойство ID, а параметр `-Id` команды `Stop-Job` получает ввод конвейера с помощью привязки `ByPropertyName`. Это свойство содержит, а параметр получает целое число, несмотря на то что по факту параметр ожидает их массив. Поэтому PowerShell создает массив с одним элементом, добавляет наш ID 5248 и привязывает его к `-Id`. Все вроде бы работает. Хотя не совсем. Мы знаем, и вы наверняка догадались, что `Stop-Job` ожидает номер ID *задачи*, в то время как мы предоставляем ID *процесса* — а это совсем разные вещи. Это все равно что использовать номер вашего дома вместо номера телефона — оба являются числами, но относятся к разным субъектам. Вот поэтому в результате мы получаем ошибку:

```
Stop-Job : The command cannot find a job with the job ID 5248. Verify the value of
the Id parameter and then try the command again.
At line:1 char:52
+ trace-command -Expression { Get-Process -Name o* | Stop-Job } -PSHost ...
+ CategoryInfo : ObjectNotFound: (5248:Int32) [Stop-Job], PSArgumentException
+ FullyQualifiedErrorId : JobWithSpecifiedSessionNotFound,Microsoft.
PowerShell.Commands.StopJobCommand
```

Результат трассировки показывает, что PowerShell пытается создать запись об ошибке, которая в итоге отображается на экране. Причем это оказывается довольно сложной задачей, которая ведет к формированию еще нескольких десятков строк

вывода трассировки. Команда `Trace-Command` может оказаться удобным командлетом для устранения неполадок, поэтому рекомендуем подробно ознакомиться с ее справкой и примерами.

4.4.2. Когда метод ByPropertyName не работает

Что, если вы окажетесь в ситуации, когда у вас в конвейере будет объект и команда, готовая его получить, но не сработает ни привязка `ByValue`, ни привязка `ByPropertyName`? Такое вполне возможно: команда, к примеру, может не уметь обрабатывать подобный тип объектов или же вообще не получать ввод из конвейера. Это довольно редкий случай, и для его демонстрации мы создали простую команду PowerShell:

```
PS C:\> "frances" | set-foo
set-foo : The input object cannot be bound to any parameters for the command either
because the command does not take pipeline input or the input and its properties do
not match any of the parameters that take pipeline input.
At line:1 char:13
+ "frances" | set-foo
+             ~~~~~~~
    + CategoryInfo            : InvalidArgument: (frances:String) [Set-Foo],
ParameterBindingException
    + FullyQualifiedErrorId   : InputObjectNotBound,Set-Foo
```

Как видите, выполнение всего конвейера дает сбой. Объекты *не могут* быть переданы в команду, и PowerShell не хочет просто их отбросить, поэтому сообщает об ошибке и останавливает выполнение.

4.4.3. Заблаговременное планирование

Рассмотрим ключевой принцип разработки инструментов, особенно таких, в которых используется привязка параметров. В идеале вам нужно указывать лишь один параметр для получения ввода из конвейера по значению (`ByValue`). Представьте, что есть два параметра, `Foo` и `Bar`, предназначенные для получения ввода по значению. Если вы выполните команду как `Get-content data.txt | get-magic`, то возникнет вопрос: куда должны отправиться входящие значения — в `-Foo` или в `-Bar`? PowerShell не может ответить на этот вопрос, что подчеркивает важность наличия единственного параметра для ввода по значению. Технически есть возможность использовать несколько таких параметров с помощью их наборов, но это уже более сложный прием.

И напротив, у вас может быть множество параметров для ввода по имени свойства (`ByPropertyName`). Возможно даже такое, что один параметр будет получать ввод как по значению, так и по имени свойства, но для этого потребуется использовать подход, над которым придется хорошо подумать. Поразмышляйте о наиболее вероятном применении ваших инструментов. Будут ли пользователи часто передавать

результаты в вашу команду или же станут выполнять ее в конвейере как первичную? Очень важно тестировать различные сценарии использования, чтобы убедиться в том, что привязка параметров работает должным образом. Если возникнут проблемы, то применяйте `Trace-Command`, чтобы получить представление о внутренних процессах конвейера.

ИТОГИ ГЛАВЫ

В этой главе у нас была двойная цель. Во-первых, мы хотели, чтобы вы поняли, как объекты конвейера переходят из команды в команду. Во-вторых, нам было важно, чтобы вы осознали, насколько трассировка команд помогает визуализировать этот процесс и диагностировать неожиданное поведение конвейера. Вскоре вы начнете создавать собственные команды, которые будут получать ввод из конвейера, и мы хотим, чтобы вы всегда продумывали этот процесс, попутно учитывая принцип его работы.

5 Язык написания скриптов: экспресс-курс

По ходу чтения этой книги вы заметите, что сначала мы что-то используем, а потом даем об этом общую информацию. Тем не менее в текущей главе мы хотим сделать исключение. Работая с книгой, вы будете писать скрипты, а это означает добавление определенного объема кода. Язык написания скриптов PowerShell очень прост и содержит несколько десятков ключевых слов. При этом в книге будут использоваться около дюжины. Но вам нужно хорошо запомнить самые важные из них, чтобы при необходимости применять. В текущей главе будет дано лишь общее описание этих элементов. Вы начнете лучше понимать их смысл, когда станете использовать их в работе.

СОВЕТ

Дополнительно углубиться в тему этой главы можно в первую очередь с помощью системы справки PowerShell. Значительная ее часть задокументирована в разделах About. Например, вы можете просмотреть информацию касательно about_if и about_comparison_operators. Кроме того, вы можете прочесть книгу *PowerShell in Depth, 2nd ed.* (Manning, 2014; http://mng.bz/xjzq).

5.1. СРАВНЕНИЯ

Практически во всех представленных в этой главе элементах скриптинга используются *сравнения*. Это значит, что вы сообщаете им какое-то выражение, которое нужно оценить как `True` или `False`, после чего конструкции скриптинга основывают свое поведение на результате этой оценки. Для выполнения сравнения в PowerShell используется *оператор сравнения*. В отличие от традиционных языков скриптинга или программирования, в PowerShell используются не общепринятые

операторы (<, >, +, =, –), а англоязычные сокращения. Основные операторы, которые вы, скорее всего, будете использовать по ходу работы с книгой, представлены ниже:

- `-eq` — равно;
- `-ne` — не равно;
- `-gt` — больше чем;
- `-ge` — больше чем или равно;
- `-lt` — меньше чем;
- `-le` — меньше чем или равно.

При сравнении строк эти операторы нечувствительны к регистру, то есть строки `"Hello"` и `"HELLO"` считаются равнозначными. Если же вам нужно выполнить сравнение с учетом регистра, то добавьте перед именем оператора букву `c`, например `-ceq` или `-cne`.

Если использовать эти операторы, то PowerShell будет возвращать значение `True`/`False`:

```
PS C:\> 1 -eq 1
True
PS C:\> 5 -gt 10
False
PS C:\> 'James' -eq 'Jim'
False
PS C:\> 'James' -eq 'james'
True
PS C:\> 'james' -ceq 'James'
False
```

Диапазон операторов в PowerShell куда более широк, чем в некоторых языках. Например, невыполнение сравнения «в точности равно» запрещает парсеру оболочки приводить тип данных к другому типу.

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Чему будет равно 5 -eq «Five» — true или false?

5.1.1. Подстановочные символы

Есть сравнение с подстановочными символами, `-like` и `-not like`, а также его версии, чувствительные к регистру. Оно позволяет использовать при составлении сравнений строк типичные символы подстановки, такие как * (с дополнительными символами или без) и `?` (один символ):

```
PS C:\> 'james' -eq 'Jim'
False
PS C:\> 'james' -eq 'James'
True
PS C:\> 'james' -ceq 'James'
False
PS C:\> 'PowerShell'-like '*shell'
True
PS C:\> 'james' -notlike 'james*'
False
PS C:\> 'james' -like 'jam?s'
True
PS C:\> 'james' -like 'Jim'
False
```

Этих подстановочных символов не так много, как в языке полных регулярных выражений. PowerShell поддерживает регулярные выражения с помощью оператора `-match`, но здесь мы не будем углубляться в его изучение. Если эта тема вам интересна, можете почитать главу *PowerShell and regular expressions* книги *PowerShell in Depth, 2nd ed.* (Manning, 2014; http://mng.bz/xjzq).

5.1.2. Коллекции

Операторы PowerShell `-contains` и `-in` работают с коллекциями объектов. Порой они формируют довольно сложные конструкции, и разработчики почти всегда путают их с операторами подстановочных символов. Например, мы часто видим такое выражение:

```
If ("DC" -in $ServerList) {
  $IsDomainController = $True
}
```

Причем оно работает не так, как вы могли подумать. На английском оно прекрасно читается, но оператор делает не это. Если вы начинаете с массива, то можете использовать эти операторы для определения того, содержит ли данный массив (или коллекция) конкретный объект:

```
$array = @("one", "two", "three")
$array -contains "one"
$array -contains "five"
"two" -in $array
"bob" -in $array
```

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Выполните эти пять строк кода в PowerShell столбиком, набирая их по одной и нажимая после каждой Enter.

5.1.3. Устранение неполадок при использовании сравнений

Около 50 % багов в скриптах вызваны сравнением, которое работает не так, как ожидалось. Устранять подобные проблемы мы рекомендуем так: нужно прекратить работать со скриптом, перейти в консоль PowerShell и проверить это сравнение.

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Какой результат даст «55» -eq 55? Мы не дадим ответ. Попробуйте сами выполнить это задание и попытайтесь объяснить себе, почему получился именно такой результат.

5.2. КОНСТРУКЦИЯ IF

В некоторых случаях вам потребуется конструкция `If`, которая позволяет коду принимать логические решения. В своей полной форме она выглядит так:

```
If (<expression>) {
    # код
} ElseIf (<expression>) {
    # код
} ElseIf (<expression>) {
    # код
} Else {
    # код
}
```

Вам нужно знать следующую информацию.

- В PowerShell `<expression>` — это любое выражение, приводящее либо к `$True`, либо к `$False`. Например, `$something -eq 5` будет `$True`, если переменная `$something` равна 5. В статье PowerShell под названием *about_comparison_operators* приводится список допустимых операторов сравнения, в том числе `-eq`, `-ne`, `-gt`, `-like` и др.
- Выражения в вашей конструкции `If` могут иметь любую необходимую степень сложности. Главное, помните: чтобы скрипт можно было выполнить, все выражение в итоге должно вычисляться как `True`:

```
$now = Get-Date
if ($now.DayOfWeek -eq 'Monday' -AND $now.hour -gt 18) {
  #выполнение чего-либо
}
```

- Часть `If` в конструкции является обязательной и должна сопровождаться `{script block}`, который выполнится, если выражение окажется `True`.
- В вашей конструкции может присутствовать нужное количество разделов `ElseIf`, каждый из которых будет представлять собственное выражение и блок скрипта, выполняясь, если выражение оказывается `True`. Но здесь нужно помнить важный момент: будет выполнен только блок скрипта первого выражения,

которое окажется `True`. Поэтому в предыдущем простом примере выполнится только первый блок скрипта, если первое выражение будет `True`, и никакое из выражений `Elseif` вычисляться не будет. Если у вас несколько операторов `Elseif`, то PowerShell продолжит вычислять их, пока не найдет то, которое окажется `True`. Когда это произойдет, оболочка перейдет к команде после соответствующей структуры `If`.

- В конце вашей конструкции может быть необязательный раздел `Else`. Он определяет блок скрипта, который выполнится, если никакие из его выражений не окажутся `True`.
- Здесь нет оператора `End If`, который можно встретить в других языках.
- В предыдущем примере вы заметите строки, начинающиеся с символа `#`. Это комментарии. PowerShell будет игнорировать все содержимое, указанное после `#` и до конца текущей строки.

PowerShell довольно снисходителен в отношении форматирования. К примеру, мы считаем, что это хороший способ форматирования конструкции:

```
If (<expression>) {
    # код
} ElseIf (<expression>) {
    # код
} ElseIf (<expression>) {
    # код
} Else {
    # код
}
```

При этом некоторые предпочитают ставить открывающую { на отдельной строке. И ни один из этих способов не является более правильным, но вам следует иметь в виду, что некоторые разработчики могут писать так:

```
If (<expression>)
{
    # код
}
```

Как бы то ни было, PowerShell позволит вам писать код и так:

```
If (<expression>) { # код }
ElseIf (<expression>) { # код }
ElseIf (<expression>) { # код }
Else { # код }
```

Такой код читать сложнее, особенно если какой-либо из блоков скрипта должен содержать несколько строк. Мы точно не советуем использовать этот метод, но вы увидите, что другие разработчики иногда так делают. Суть в том, что хоть PowerShell этот нюанс и не волнует, он должен волновать вас. Выберите стиль форматирования, который облегчит чтение кода, и придерживайтесь его.

СОВЕТ

Форматирование кода очень важно. Оно может казаться незначительной эстетической деталью, но при этом делает код более понятным, а значит, и снижает количество ошибок. Поверьте нашему опыту. Взгляните, к примеру, на этот код:

```
If ($user) { ForEach ($u in $user) {
Set-ADUser -Identity $user -Pass $True }
```

Сложно понять, правильный ли он (по факту нет), учитывая беспорядочность расстановки фигурных скобок и то, что `ForEach` начинается на одной строке с `If`.

Если вы работаете в хорошем редакторе, например Visual Studio Code, то поддерживать порядок в коде довольно легко: позвольте редактору делать то, что нужно. Когда вы открываете конструкцию, используя {, и нажимаете Enter, VS Code автоматически добавляет закрывающую } и помещает указатель мыши — добавляя отступ в четыре пробела — внутри этой конструкции. Пусть VS Code делает свою работу — например, чтобы добавить отступ в строке, используйте клавишу Tab, а не многократное нажатие пробела. Кроме того, если вы еще этого не сделали, включите в редакторе цветное выделение скобок.

Если содержимое не выравнивается вертикально, то вот вам совет: выделите нужную область (во всем документе скрипта), щелкните правой кнопкой и выберите Format Selection. VS Code наведет порядок, добавив внутри каждой конструкции правильные отступы. Кроме того, вы можете открыть панель команд и выбрать Format Document, чтобы отформатировать весь документ.

Разберем практический пример этой конструкции. Предположим, у вас в переменной `$proc` есть объект `Process` и вы хотите, чтобы в момент, когда свойство виртуальной памяти процесса (`VM`) превышает определенное значение, выполнялось заданное действие:

```
If ($proc.vm -gt 4) {
  # выполнить действие
}
```

Обратите внимание, что мы добавили комментарий (напоминаем, что все после символа # и до конца строки игнорируется), чтобы указать, на что переключится выполняющий действие код. А если вы, напротив, хотели бы выполнить действие для значений `VM`, которые меньше 2, но больше 4?

```
If ($proc.vm -gt 4 -or $proc.vm -lt 2) {
  # выполнить действие
}
```

Логический оператор `-or` позволяет объединять два условия. Обратите внимание, что сравнение с обеих сторон `-or` an- должно быть *завершенным*. Например, такой вариант не будет работать:

```
If ($proc.vm -gt 4 -or -lt 2) {
  # выполнить действие
}
```

В этом «ошибочном» примере сравнение «меньше» неполное, так как слева отсутствует необходимая часть. PowerShell в этом случае озадачится: «Что конкретно должно быть меньше 2?» — и выдаст ошибку. Если это вам поможет, используйте скобки, чтобы зрительно отделить каждое сравнение:

```
If ( ($proc.vm -gt 4) -or ($proc.vm -lt 2) ) {
  # выполнить действие
}
```

Далее мы разберем пример, в котором есть дополнительные действия:

```
If ($proc.vm -gt 4) {
  # выполнить действие
} ElseIf ($proc.vm -lt 2) {
  # выполнить иное действие
} Else {
  # ничто не оказалось true, выполнить это
}
```

Как мы уже говорили, PowerShell выполнит первое из этих действий, условие которого будет оценено как `True`, прекратив дальнейшее вычисление.

5.3. КОНСТРУКЦИЯ FOREACH

Вы будете часто использовать конструкцию `ForEach`, которую иногда называют *нумератором*. Она есть практически во всех языках программирования, так что наверняка будет вам знакома. Работает эта конструкция примерно так же, как команда PowerShell `ForEach-Object`, но имеет другой синтаксис:

```
ForEach ($item in $collection) {
    # код для выполнения каждого объекта, указанного в $item
}
```

Смысл здесь в том, чтобы взять коллекцию или массив объектов и перебрать их по одному. Каждый объект, в свою очередь, помещается в отдельную переменную, чтобы к нему было удобно обращаться. После перебора всех объектов коллекции или массива цикл автоматически завершается, и выполняется оставшаяся часть скрипта.

Вторая переменная этой конструкции, `$collection`, может содержать некоторое количество элементов или не содержать их вовсе. *Цикл* `ForEach` будет выполнять свой `{script block}` по разу *для каждого* (for each) элемента из второй переменной. Иначе говоря, если вы предоставите в `$collection` три имени компьютеров, то данный цикл выполнится три раза. При каждом его выполнении из второй переменной берется один элемент и помещается в первую. Таким образом, в предыдущем блоке скрипта `$item` будет одновременно содержаться один элемент из `$collection`.

СОВЕТ

Мы придумали имена переменных $item и $collection. Обычно вы будете использовать свои названия, указывающие на то, что должны содержать эти переменные.

Вы часто будете видеть, как разработчики используют в своих циклах `ForEach` слова в единственной и множественной форме:

```
$names = Get-Content names.txt
ForEach ($name in $names) {
    # код для каждого $name
}
```

Такой подход помогает запомнить, что `$name` содержит один элемент из `$names`, но это сделано исключительно для того, чтобы облегчить чтение кода человеком. PowerShell не будет учитывать, что *name* — форма единственного числа для *names*. Для оболочки это неважно. Предыдущий пример можно легко переписать так:

```
$names = Get-Content names.txt
ForEach ($purple in $unicorns) {
    # код
}
```

PowerShell сочла бы такой вариант полностью рабочим. Этот код будет намного сложнее читать и отслеживать, но если вы любите единорогов, то можете делать так. Тем не менее в некоторых случаях вы заметите, что вторая переменная указана не во множественном числе, хотя как будто должна быть такой:

```
foreach ($computer in $computername) {
```

Зачастую это объясняется тем, что `$ComputerName` — один из входных параметров функции. В соответствии с соглашением об именовании PowerShell для названий команд и параметров используется единственное число. Вы не встретите параметр `-ComputerNames`, только `-ComputerName`. Вам нужно придерживаться этого соглашения, поэтому в данном случае цикл `forEach` не будет строиться по принципу «единственное/множественное число». Опять же, для PowerShell это не имеет значения, и важнее, чтобы имена ваших команд и параметров, которые видят другие люди, были заданы с учетом соглашения об именовании, используемого оболочкой.

РЕКОМЕНДАЦИЯ

В скрипте мы предпочитаем использовать конструкцию ForEach вместо команды ForEach-Object. У этого решения есть ряд преимуществ: вам нужно просто именовать переменную, представляющую элемент коллекции, а не использовать $_ или $PSItem, что делает код более удобным для чтения; при этом конструкция в отношении больших коллекций часто выполняется быстрее, чем команда. Если же коллекции очень велики, конструкция может вынуждать вас использовать больше памяти, поскольку весь массив или коллекция должны находиться в одной переменной. При этом в случае команды объекты могут передаваться и обрабатываться по одному, что в некоторых сценариях снижает потребление памяти.

У конструкции `ForEach` есть один *подвох*: она не выполняет запись в конвейер после закрытия фигурных скобок. Мы неоднократно видели, как разработчики, пытаясь создать что-то подобное примеру ниже, раз за разом терпят неудачу:

```
$numbers=1..10
foreach ($n in $numbers) {
 $n*3
} | out-file data.txt
```

Если вы попробуете выполнить это в VS Code или других редакторах, то наверняка получите ошибку, связанную с пустым конвейером. В конвейер пишется все *внутри* блока скрипта. Вы не можете добавить в него ничего *после*. Но можно написать такой код:

```
$numbers=1..10
$data = foreach ($n in $numbers) {
 $n*3
}
$data | out-file data.txt
```

Он сработает ожидаемым образом. Во втором примере `$n*3` неявно записывает свой вывод в конвейер (`Write-Output` — это стандартная команда PowerShell), и конечный результат конструкции `ForEach` оказывается в переменной `$data`, которая затем передается в `Out-File`. Бóльшая часть этой путаницы происходит потому, что псевдонимом для *объекта* `ForEach` является `ForEach`, хотя работает он не так, как *конструкция* `ForEach`. Рассматриваемая нами конструкция всегда имеет синтаксис (`$x in $y`), в то время как команда `ForEach-Object` его не использует.

Учитывая все это, мы настоятельно рекомендуем основательно подумать над тем, когда использовать нумератор `ForEach`, поскольку легко привыкнуть к действиям, противоречащим принципам работы PowerShell. Ниже мы привели пример кода, который видели либо у разработчиков — новичков в скриптинге, либо у тех, кто так и не понял модель PowerShell:

```
$services = Get-Service -name bits,lanmanserver,spooler
Foreach ($service in $services) {
  Restart-service $service -passthru
}
```

Этот скрипт определенно сработает, и мы делали так, когда пользовались языком программирования VBScript, но такое написание противоречит PowerShell. Не нужно искажать код, когда вполне хорошо работает вот такой:

```
$services | restart-service -passthru
```

5.4. КОНСТРУКЦИЯ SWITCH

Конструкция `switch` отлично заменяет собой огромный блок `If`, содержащий несколько разделов `ElseIf`. Ее прототип выглядит так:

```
switch (<expression>) {
  <condition> { <script block> }
  <condition > { <script block> }
  <condition > { <script block> }
  default { <script block }
}
```

А вот как она работает.

1. Выражением обычно выступает переменная, содержащая *одно значение или объект*. Это важно, поскольку в одной только `switch` нельзя перебрать коллекции или массивы.
2. Каждое условие является значением, которое, на ваш взгляд, может содержаться в выражении. При этом каждое условие сопровождается блоком скрипта (который можно разбить на строки), и если выражение содержит условие, то выполняется связанный с ним блок скрипта.
3. Блок `default` выполняется, если никакие условия не совпадают. Если он вам не нужен, то можете его опустить.

Каждое совпадающее условие будет выполнено. Если таких совпадений несколько, то выполняется каждый совпадающий блок скрипта. Это может казаться нерациональным, пока вы не познакомитесь с некоторыми расширенными параметрами конструкции:

```
$x = "d1234"
switch -wildcard ($x)
    {
        "*1*" {"Contains 1"}
        "*5*" {"Contains 5"}
        "d*" {"Starts with 'd'"}
        default {"No matches"}
    }
```

Параметр `-wildcard` делает возможным совпадение нескольких условий. Конкретно в этом примере, если бы `$x` содержала *1 of 5 dying worms*, то вы бы получили

две строки вывода: `"Contains 1"` и `"Contains 5"`. Третий паттерн в данном случае не подходит, и блок `default` тоже не выполняется. Рекомендуем прочесть статью *about_switch* в справочной системе PowerShell.

5.5. КОНСТРУКЦИЯ DO/WHILE

Эту конструкцию вы начнете использовать чуть позже. `While` позволяет указывать блок операторов скрипта, который будет выполняться, пока (while) некоторое условие истинно. Два основных варианта выглядят так:

```
While (<condition>) {
 # код
}
Do {
 # код
} While (<condition>)
```

Оба они, по сути, делают одно и то же: повторяют код внутри конструкции, пока указанное `<condition>` не перестанет быть истинным. Различие же состоит в следующем.

- В первой версии код конструкции *может никогда так и не выполниться*. Он выполняется, только если `<condition>` верно.
- Во второй версии код внутри конструкции *в любом случае выполнится как минимум раз*. Проверка `<condition>` начинает производиться только после первого выполнения.

Вам нужно быть внимательным при написании этих циклов, поскольку из них нельзя выйти автоматически, как при использовании конструкций `Switch`, `If` или `ForEach`. Если вы не уверены, что ваше `<condition>` в итоге изменится и будет вычислено как ложное, то конструкция `Do/While`, по сути, может выполняться вечно, становясь *бесконечным циклом*. В большинстве средств выполнения PowerShell, таких как консоль, вы можете прервать этот цикл, нажав Ctrl+C, если понимаете, что он получился бесконечным.

5.6. КОНСТРУКЦИЯ FOR

Последняя из конструкций скриптинга, `For`, используется настолько редко, что мы долго думали, стоит ли вообще упоминать ее в книге (вы вполне можете пропустить этот раздел, если чувствуете, что перегружены информацией). Тем не менее, чтобы не расстраивать тех, кому эта конструкция все же интересна, мы ее продемонстрируем. Выглядит она обычно так:

```
For (<start>; <condition>; <action>) {
  # код
}
```

Этот цикл должен повторять код внутри конструкции *несколько раз*, и его будет чуть проще объяснить на более конкретном примере:

```
For ($i = 0; $i -lt 3; $i++) {
  Write $i
}
```

Идея в том, что элемент `<start>` выполняется до начала выполнения конструкции, устанавливая `$i` на значение `0`. Далее `<condition>` заставляет конструкцию выполняться, пока не окажется `True`. Наконец, *после* выполнения блока скрипта конструкции каждый раз выполняется `<action>`. То есть конкретно в этом примере скрипт выполнится четыре раза.

1. `$i` изначально устанавливается равным 0, после чего выполняется блок скрипта.
2. `$i` меньше 3, поэтому увеличивается на 1, и блок скрипта выполняется.
3. `$i` все еще меньше 3, поэтому снова увеличивается на 1 (теперь становясь 2), и блок скрипта опять выполняется.
4. `$i` все так же меньше 3, поэтому опять увеличивается на 1 (становясь 3), и блок скрипта в очередной раз выполняется.
5. Теперь `$i` равен 3, то есть уже не меньше 3, а значит, блок скрипта не выполняется, и конструкция выполняет выход.

Эта схема аналогична использованию оператора диапазона, предоставляемого PowerShell, и команды `ForEach-Object`:

```
0..3 | ForEach-Object { Write $_ }
```

Конструкцию `For` проще читать, и она воспринимается как более декларативная. И если нам вдруг потребуется выполнить подобную задачу, то мы предпочтем использовать именно ее, а не оператор диапазона. Но по факту мы применяем `For` редко, поскольку нечасто сталкиваемся с ситуациями, в которых нужно выполнить что-либо несколько раз. Чаще мы все же используем `ForEach`, так как у нас есть коллекция объектов, с каждым из которых нужно выполнить определенное действие. Это можно реализовать и с помощью `For`, но код получится не очень лаконичным. Если предположить, что `$objects` содержит коллекцию объектов, то перебрать их можно двумя способами:

```
For ($i = 0; $i -lt $objects.Count; $i++) {
  Write $objects[$i]
}
ForEach ($thing in $objects) {
  Write $thing
}
```

Второй пример проще читать. Мы подозреваем, что любители первого подхода до использования PowerShell писали код на языке, в котором нет конструкции для перебора наподобие `ForEach`, поэтому по привычке склонны задействовать знакомую им `For`.

5.7. КЛЮЧЕВОЕ СЛОВО BREAK

В скриптинге есть еще один элемент, о котором вам следует знать, — ключевое слово `Break`. Оно позволяет выйти из любой конструкции кода. Но при этом нужно иметь в виду несколько нюансов.

- Использование `Break` в конструкциях `For`, `ForEach`, `While` или `Switch` приводит к мгновенному выходу из них.
- При использовании `Break` в скрипте вне конструкции выполняется выход из скрипта.
- Использование `Break` в конструкции `If` не приводит к выходу. В таких случаях ключевое слово выполняет выход из того, что *содержит* саму эту конструкцию: внешних `For`, `ForEach`, `While`, `Switch` или самого скрипта. Конструкция `If` невидима для `Break`, поэтому это ключевое слово видит лишь то, внутри чего находится `If`.

С помощью `Break` можно отменять операции. Представим, что у вас в переменной `$computers` находится список компьютеров. Вы хотите просмотреть каждый и проверить их доступность, используя пинг. Если же по какой-то причине один из компьютеров не ответит, то вы хотите прекратить процесс и тут же выполнить выход. Это можно записать так:

```
ForEach ($comp in $computers) {
  If (-not (Test-Ping $comp -quiet)) {
    Break
  }
}
```

Вам нужно знать, что существует антипаттерн. Некоторые разработчики намеренно пишут бесконечный цикл. Вместо того чтобы указать естественное условие для его завершения, они используют `Break`. Вот краткий пример:

```
While ($true) {
  $choice = Read-Host "Enter a number."
  If ($choice -eq 0) { break }
}
```

У нас часто возникает вопрос: неужели эти разработчики не знали о других возможностях реализации цикла? К примеру, в данном случае они, похоже, хотели сделать так, чтобы содержимое цикла выполнилось как минимум раз, но не знали, как изначально попасть в него. Этот код можно переписать так:

```
Do {
  $choice = Read-Host "Enter a number."
} While ($choice -ne 0)
```

Этот вариант немного чище в плане выполнения. Проблема использования `Break` состоит в том, что это ключевое слово предоставляет альтернативный выход из

конструкции, создавая вторичный поток логики, усложняющий понимание. Поскольку `Break` зачастую используется внутри конструкции `If` (как было показано чуть выше), то становится сложно спрогнозировать, как поведет себя скрипт, если не будет выполняться. Это, в свою очередь, создает всевозможные проблемы с отладкой, которых нам хотелось бы избежать. Если коротко, то мы пытаемся писать конструкции, содержащие естественную конечную точку, по возможности избегая использования `Break`.

СОВЕТ

Старайтесь обходиться без ключевого слова Break. Оно порождает то, что мы называем неестественным выходом из цикла. То есть при использовании Break цикл не приходит к естественному завершению. Это слово легко проглядеть в циклах, содержащих большое количество кода, в результате чего бывает сложно понять, почему цикл завершается раньше времени. Если же вам все же приходится использовать Break, тогда до и после него нужно добавлять пустые строки и подробные комментарии с описанием происходящего.

ИТОГИ ГЛАВЫ

Конструкции, которые мы рассмотрели в текущей главе, формируют основу языка PowerShell. В отличие от команд, они нужны для того, чтобы скрипты имели логику и структуру. Если вы сможете запомнить эти четыре основные конструкции (`If`, `ForEach`, `Switch`, `Do/While`), то наверняка поймете, что из них состоит весь код, который вам потребуется для написания большинства скриптов.

6 Множество форм скриптинга (и как из них выбирать)

Вы наверняка думаете, что стали жертвой маркетинговых приемов издательства. Мы используем слова *«инструменты»* и *«создание инструментов»*, но о скриптинге почти не говорим. При этом книга называется «Изучаем скриптинг PowerShell». Что ж, мы можем ответить так: в данном случае скриптинг равнозначен созданию инструментов. Слово «скриптинг» довольно обобщенное, и в мире PowerShell оно охватывает два конкретных и ценных элемента, о которых мы поговорим в текущей главе.

6.1. ИНСТРУМЕНТЫ И КОНТРОЛЛЕРЫ

Представьте молоток. Это инструмент, и даже если сами вы им не пользовались, то как минимум видели. Молоток вполне автономен. Его основное назначение — бить по другим предметам. Он не имеет никакого представления ни о контексте своей жизни, ни о своей судьбе. В одном случае он может использоваться при строительстве дома, в другом — для разбивания окна, а в третьем — для удара по большому пальцу. Пока молоток лежит в ящике для инструментов и им никто не машет, он бесполезен.

В данном случае именно вы будете махать молотком, чтобы бить по гвоздям (или по своему большому пальцу). Но подумайте о нем еще секунду. Представьте, насколько сильно вы будете им замахиваться и куда будете наносить удар (будь то гвоздь, окно или палец). Ваша цель — ударить по гвоздю и загнать его в дерево, услышав при этом звонкий звук удара молотка по шляпке гвоздя. Но что произойдет, если ударить по гвоздю под углом? Он либо погнется, либо сломается, что приведет к нежелательным последствиям.

Так вот PowerShell похож на молоток. Он получает от вас ввод и создает вывод. Иногда он работает так, как вы хотите, а иногда нет, но всегда *в точности*

выполняет те указания, которые вы ему даете, — ни больше ни меньше. Этим он и хорош. То, что в PowerShell называется *командой* (общим словом, которое обозначает командлеты, функции и другие исполняемые артефакты), мы называем *инструментом*. Он должен выполнять только одну задачу. Именно поэтому у нас есть инструменты `Get-Process`, `Start-Process`, `Stop-Process` и т. д. — каждый из них делает лишь что-то одно. У нас нет универсального инструмента, который бы запускал, останавливал или перебирал процессы в зависимости от указанных параметров. Подобный многозадачный инструмент противоречит идее PowerShell, согласно которой для одной задачи должен использоваться один инструмент.

Возьмем, к примеру, `Stop-Process`. Сам по себе этот инструмент особого смысла не имеет — как и в случае молотка, для его использования нужны контекст и цель. Его нужно контролировать. Применение инструмента обретает смысл и цель, когда он используется в рамках скрипта-контроллера.

В данной главе мы поговорим о том, как научиться разделять эти два одинаково важных вида скриптов. Одни приемы предназначены для инструментов, другие — для контроллеров. Каждый набор приемов создан для того, чтобы снижать рабочую нагрузку, упрощать отладку и сопровождение и при этом улучшать читабельность кода и увеличивать возможность его повторного использования. Если вы будете понимать, к какому виду относится скрипт, который вы пишете, то сможете выбрать подходящий набор приемов и создать успешный скрипт, а в перспективе — стать мастером создания инструментов.

6.2. ОБ ИНСТРУМЕНТАХ

В мире PowerShell инструменты имеют несколько важных характеристик.

- *Инструменты выполняют какое-то одно действие, которое должно описываться в глагольной части их имени*. Лучше создать пять небольших инструментов, выполняющих по одному действию, чем один большой, который будет делать все пять. Мелкие, узкоспециализированные инструменты проще писать, тестировать, отлаживать и сопровождать.
- *Инструменты не знают, где генерируются их входные данные, как и молоток не знает заранее, как будет использоваться: будет им бить рука или же какой-то роботизированный механизм*. Инструменты получают вводные данные из своих параметров, как и молоток получает вводную информацию только от того, что держит его рукоятку. (Да, мы достаточно вольно используем эту аналогию, но вы наверняка понимаете смысл.) Для создания ввода и его последующей передачи в параметры можно использовать другие инструменты.
- *Инструменты не знают, как будет использован их вывод, и им это неважно, как и молотку неважно, ударят им по гвоздю или пальцу*. Одни инструменты формируют вывод. А другие отслеживают, как он будет выглядеть и куда будет направлен.

Неформально мы склонны представлять себе несколько типов инструментов. Это не строгая классификация, но она помогает понять, как они связаны друг с другом.

- *Инструменты ввода* предназначены для создания данных, потребляемых преимущественно другими инструментами. Вы можете написать инструмент, который получает множество компьютерных имен, например, от Microsoft Entra ID. Распространенный глагол для имен инструментов ввода — `Get`, но вы также будете встречать `Import` и `ConvertFrom`.
- *Инструменты действия* обычно требуют дополнительного ввода для выполнения неких действий, и эти «некие действия» могут быть чем угодно. Во многих командах есть такие глаголы, как `Set` и `Remove`.
- *Инструменты вывода* обычно создаются в целях получения вывода из инструмента ввода или действия, чтобы этот вывод можно было использовать в конкретной задаче. Они могут создавать специально отформатированный файл данных, отрисовывать конкретный вывод на экране и т. д. Для их именования часто используются такие глаголы, как `Out`, `Format`, `ConvertTo` и `Export`.

Представьте, что вам нужно написать скрипт, который будет сообщать возраст пароля всех аккаунтов сервиса в вашей среде и форматировать его в CSV, чтобы инструмент Security Information and Event Management (SIEM), используемый вашим департаментом кибербезопасности, мог анализировать журналы записей и определять, где используются эти аккаунты. Руководство хочет, чтобы во время ежедневных заседаний у них на видеоэкране показывался красивый HTML-отчет. Сколько инструментов нужно написать? Вам надо продумать каждую отдельную задачу и посмотреть, какие из них уже решены с помощью инструментов, имеющихся в PowerShell.

- Для начала нужно найти команду, которая будет получать возраст пароля аккаунта. В этом случае мы хотим использовать `Get-AdUser`. Мы возьмем некоторые атрибуты, сохраним их в виде массива, экспортируем в CSV-файл и затем проделаем нужные HTML-манипуляции. Если у вас нет этой команды, то установите ее:

```
Install-WindowsFeature -Name "RSAT-AD-PowerShell" `
-IncludeAllSubFeature
```

- Далее нужно выяснить, как выполнять фильтрацию на основе возраста пароля. К счастью, это может делать команда PowerShell `Where-Object`, так что вам надо написать специальный виджет.
- Вам нужно конвертировать эти результаты в CSV и сохранять в файл, что может проделать команда `Export-CSV` — вам не понадобится делать дополнительную работу.
- Вам требуется способ составления HTML-отчета. Если команды `ConvertTo-HTML` будет недостаточно, то в модуле `EnhancedHTML2` PowerShell Gallery (подробнее об этом сайте поговорим в главе 22) есть `ConvertTo-EnhancedHTML`, которая должна с этим справиться. Вам потребуется разобраться, как ее использовать, но программировать ничего не придется.

Итак, вам нужно для всего этого написать всего один инструмент. Этим и хорош подход на основе инструментов: в PowerShell и цифровой среде в целом есть много готовых универсальных инструментов, поэтому зачастую вам достаточно сосредоточиться только на том, что характерно для вашей среды. Сделайте это правильно, и ваши пользовательские инструменты будут легко внедряться в уже существующие решения.

Но предполагаемый инструмент `Get-ServiceAccountPassword` сам по себе бесполезен. Нужны цель и контекст его использования, а значит, нужен контроллер.

ПРИМЕЧАНИЕ

Обратите внимание: вы можете не встретить термины «инструмент» и «контроллер» в документации Microsoft или записях сообщества PowerShell. Для многих людей все это просто скриптинг. Но разбираться в этих двух понятиях важно, поскольку именно это помогает понять способ автоматизации процессов в PowerShell. Многие начинающие студенты не могут написать повторно используемый код PowerShell, так как пытаются делать все одновременно. Очень важно отделять определение инструмента от того, как он будет использоваться.

6.3. О КОНТРОЛЛЕРАХ

Если инструменты являются обобщенными и лишены контекста, то контроллеры ориентированы именно на него. Цель контроллера — применить инструмент конкретным образом в конкретной ситуации. И для вас это хорошо, поскольку создаваемый вами инструмент под управлением контроллера может использоваться в разных ситуациях. Мы не используем для контроллеров имена в стиле команд, состоящие из глаголов и существительных. Вместо этого мы даем им более удобные англоязычные названия. Например, `Password-HistoryReport.ps1` — это скрипт, созданный для генерации HTML-отчета по клиентам, не проявлявшим активность в течение года и более. Этот скрипт может быть откровенно прост и содержать всего один конвейер:

```
Get-ServiceAccountPassword |
Where-Object { $_.passwordlength -lt (Get-Date).AddDays(-365) } |
ConvertTo-HTML |
Out-File \\intranet\www\reports\inactive-customers.html
```

Скрипт получается несложным, и в этом весь смысл. Контроллеры, как правило, *просты*, поскольку отвечают лишь за объединение в конвейер нескольких инструментов. Ни один из этих инструментов не знал заранее, что будет участвовать в создании HTML-отчетов о клиентах, но контроллер дал им определенную цель. При этом нам потребуется еще один скрипт, `PasswordHistoryReportCSVDataFile.ps1`, который будет отвечать за генерацию необходимого CSV-файла данных. Забавы ради мы можем создать еще и `DisplayPasswordHistoryReport.ps1`, который будет

запрашивать информацию о неактивных клиентах и форматировать вывод, чтобы он красиво отображался на экране. Никогда не повредит сделать чуть больше, чем требуется!

Как и у инструментов, у контроллеров тоже есть несколько характеристик.

- *Контроллер привязан к контексту*. Он автоматизирует бизнес-процесс, взаимодействует с человеком или выполняет прочие ситуативные действия.
- *Контроллер зачастую содержит жестко прописанные данные*, например, имя файла, который будет считываться в качестве ввода, или строку для связи с базой данных, которая будет указывать место отправки вывода.
- *Контроллер отвечает за отправку своего вывода в конкретной форме, которая может не представлять структурированные данные*. Например, он может выводить информацию на экран или отправлять ее на принтер. Инструмент же записывает объекты в конвейер.
- *Инструмент выполняет задачу, а контроллер решает проблему*. Зачастую эта «проблема» отображает потребность бизнеса или указание руководства.

Разработчики часто спрашивают о написании в PowerShell «графических скриптов» с помощью либо предоставляемой .NET Framework библиотеки Windows Forms, либо более новой библиотеки оболочки Windows Presentation Foundation (WPF). Это можно сделать, и мы считаем такие скрипты *контроллерами*. Они должны содержать минимум кода и использовать преимущественно работающие инструменты. Парадигма PowerShell состоит в том, что команды, выполняемые из графического контроллера, — это те же команды, которые можно выполнить из интерактивной консоли. Графические скрипты лишь дают «картинку» для человека, который может ее увидеть или как-то отредактировать с клавиатуры.

КОНТРОЛЛЕРЫ ИЗ КОМАНД

Если вы посмотрите на предыдущий образец скрипта-контроллера, который использует наш вымышленный инструмент `Get-ServiceAccountPassword`, то увидите, что это просто команда PowerShell. В роли такого «контроллера» можете выступать вы сами, интерактивно вводя команды в консоли. Это отличный способ, гарантирующий, что ваш инструмент будет делать ровно то, что нужно.

Поместив команды в скрипт-контроллер, вы избавите себя от необходимости вводить большой объем кода и сделаете выполнение команд согласованным. К тому же скрипт-контроллер при желании можно слегка усложнить. Имея файл скрипта, выполнить его может любой разработчик. При этом результаты будут согласованными и предсказуемыми.

6.4. СРАВНЕНИЕ ИНСТРУМЕНТОВ И КОНТРОЛЛЕРОВ

Представьте конвейерную линию сборки автомобилей. В наше время на них преимущественно трудятся специальные роботы. Один красит машину, другой сваривает детали и т. д. Эти роботы являются инструментами — сами по себе они бесполезны. Только когда вы добавляете контроллер — производственную линию, которая расставляет роботов в нужной последовательности и координирует их действия, вы получаете какую-то пользу. В табл. 6.1 представлены некоторые ключевые различия инструментов и контроллеров.

Таблица 6.1. Различия инструментов и контроллеров

Инструмент	Контроллер
Делает что-то одно, и ничего больше	Связывает несколько инструментов
Получает ввод через параметры	Может содержать жестко прописанный ввод и использовать инструменты, чтобы извлекать данные, которые будут переданы другим инструментам
Создает данные в виде объектов	Может создавать выходные данные любого вида, в том числе форматированные данные, особые файлы и т. д.
Выполняет задачу	Решает проблему или удовлетворяет потребность
Сам по себе зачастую оказывается бесполезным или приносит минимальную пользу	Автономен
Может применяться во множестве разных ситуаций	Используется только в конкретных условиях

В книге мы уделим много внимания созданию инструментов. Их использование ничем не отличается от использования любой другой команды PowerShell, такой как `Get-Eventlog`. Все, у кого есть доступ к вашим инструментам, могут создать собственный контроллер.

6.5. КОНКРЕТНЫЕ ПРИМЕРЫ

В данном разделе мы разберем несколько реальных примеров инструментов, сравнив их работу с принципом создания контроллеров. Кода в этих примерах будет немного, но мы проговорим лежащий в их основе ход мысли.

6.5.1. Уведомление пользователей об истечении срока действия их пароля

Это прекрасный пример, с которым мы еще поработаем в книге позднее. А пока предположим, что вы хотите отправить email-уведомление пользователям, пароли которых через день или два перестанут действовать. Что входит в процесс уведомления?

Вы начнете с получения списка пользователей, срок действия паролей которых вскоре истечет: тех, чьи аккаунты не отключены и у кого не задана настройка «бессрочный пароль». Кроме того, вам потребуется вычислить точный срок окончания действия пароля и исключить тех пользователей, пароль которых не входит в установленный вами диапазон времени. После этого нужно уведомить пользователей по электронной почте и, возможно, внести эту информацию в журнал, чтобы впоследствии можно было выполнить проверку.

Вам нужно создать пять различных инструментов, каждый из которых будет выполнять одну *задачу* из общего *процесса*:

- получать аккаунты пользователей, пароли которых скоро перестанут действовать;
- получать дату истечения срока действия пароля;
- исключать аккаунты, исходя из установленного количества оставшихся дней;
- отправлять письма;
- создавать журнал контроля.

Если вы все сделали правильно, то скрипт вашего «контроллера» мог получиться таким:

```
Get-EnabledExpiringUser |
Add-ExpiryDataToUser |
Where-Object { $_.DaysToExpire -lt 2 } |
Send-PasswordExpiryMessageToUser |
Export-CSV report.csv
```

Три из перечисленных инструментов являются новыми, которые вам нужно создать, а два относятся к стандартному набору PowerShell. Возможно, для создания этих трех инструментов вам предстоит написать около сотни строк кода. При этом некоторые из инструментов наверняка можно будет использовать в других бизнес-процессах. Например, получение аккаунтов активных пользователей, пароль которых скоро перестанет работать, может оказаться полезным где-то еще. Получение списка всех пользователей и добавление даты истечения срока действия их пароля тоже может пригодиться в каких-то ситуациях. Имеет смысл модулировать эти задачи в виде *инструментов*, которые в последующем будут вызываться из контроллера.

Помните, что контроллер необязательно должен быть скриптом. Выступать в его роли можете вы, выполняя команды в сеансе PowerShell. Использование скрипта лишь избавляет вас от лишнего ввода и обеспечивает согласованность.

6.5.2. Инициализация новых пользователей

Это наш классический пример демонстрации различий инструментов и контроллеров. Подумайте, какие действия необходимо выполнить при инициализации нового пользователя в вашей организации. Вам наверняка нужно завести новый аккаунт, активировать его через электронную почту, настроить базовую папку, возможно, добавить пользователя в среду наподобие SharePoint и т. п. Каждое из этих действий является отдельной задачей в рамках процесса, и каждое должно выполняться отдельным инструментом. При этом многие из таких инструментов, например `New-ADUser`, уже предоставлены Microsoft.

И здесь есть возможность действовать рационально. Например, *где* вы настроите новую базовую папку? Какая у вас стандартная бизнес-логика? Вы можете ответить так: «Обычно мы не добавляем в базовую папку файлового сервера более 1000 пользователей. То есть мы просматриваем эти серверы, находим тот, на котором используется менее 1000 базовых папок, и задействуем его. Если же выбранный сервер имеет меньше 75 Гбайт свободного пространства, то вместо него мы выбираем другой». Все это является *задачей*, и ее нужно автоматизировать. Возможно, вы создадите инструмент `Select-UserHomeFolderFileServer`, который будет выполнять анализ и возвращать список приемлемых серверов, а также инструмент `New-UserHomeFolder`, который будет использовать для создания базовой папки первый подходящий сервер. Это две отдельные задачи, и для них нужны отдельные инструменты.

ОРИЕНТИРУЙТЕСЬ НА ГЛАГОЛЫ

Однажды нам нужно было получить из Active Directory множество пользователей. С этим прекрасно справлялась команда `Get-ADUser`, но мы хотели расширить объекты пользователей дополнительными данными. В частности, нам нужно было добавить свойство, которое бы указывало, сколько времени прошло с того момента, когда пользователь выполнял в своем аккаунте некие действия. К тому же мы хотели отфильтровать аккаунты, которые *никогда* не использовались для авторизации. В итоге встал вопрос, как должен называться такой инструмент.

Мы всегда начинаем подобные рассуждения с изучения документации PowerShell на странице Microsoft Learn (http://mng.bz/XqQG), где перечислены официальные глаголы, допустимые для использования в именах команд. В данном случае подходящим казался глагол `Add`. Мы же *добавляли* (adding) информацию к объектам пользователей, а этот глагол имеет значение «[присоединять] один элемент к другому». Но *добавление* не отражает процесс *фильтрации*, который мы также хотели

реализовать. В итоге решение мы нашли не сразу. Можно ли использовать в качестве глагола слово `Process`? Мы же реально обрабатываем (processing) эти объекты. Нет, этот глагол недопустим. Тогда, возможно, `Evaluate`? Тоже нет.

И тут нас осенило. Мы столкнулись с этой проблемой, поскольку наш инструмент выполнял *две задачи*: добавлял в объект дополнительную информацию и отфильтровывал объекты из очереди обработки. Для подобной фильтрации уже существует команда `Where-Object` — нам не нужно было дублировать ее внутри другого инструмента.

Когда мы перестали пытаться подобрать глаголы, все встало на свои места. Нам нужно было: 1) создать один инструмент, позволяющий добавлять в объекты пользователей некие данные; 2) использовать существующий инструмент для фильтрации тех объектов, которые были нам нужны. Вместо того чтобы выполнять эти два действия в одном инструменте, мы выполняли одно — и такой подход оказался более эффективным. Изучив список глаголов PowerShell и уяснив их назначение, вы сможете принимать более взвешенные решения при создании инструментов.

6.5.3. Настройка разрешений файлов

Теперь перейдем к более сложной задаче: нужно установить разрешение для всей иерархии файлов, но при этом исключить определенные их типы. Какие задачи задействованы в этом процессе? В таких случаях иногда полезно подумать о том, как бы вы сделали это вручную. И мы имеем в виду реально вручную, а не с помощью графического интерфейса пользователя (graphical user interface, GUI). Представьте, что *сами* являетесь Windows, — как бы вы тогда это реализовали?

Сначала вы получите список файлов. Для этого в PowerShell есть инструмент, который может рекурсивно перебирать подпапки и даже исключать файлы на основе заданного критерия. Далее вам нужно получить их существующий объект разрешений или список управления доступом (access control list, ACL). Опять же, в PowerShell уже есть команда для этого. Затем вам нужно добавить разрешение в каждый ACL. И для этого тоже есть готовая команда. Итак, ваш «скрипт» в данном случае может быть сложным однострочником. И это будет контроллер, поскольку все нужные вам инструменты уже существуют.

ПРИМЕЧАНИЕ

Этот пример обнажает следующую ситуацию (которую порой трудно принять): если вы не слишком хорошо понимаете, как работает Windows (или любая другая среда, которую вы используете), то вам будет сложно автоматизировать ее функционирование в PowerShell. GUI скрывает значительную часть действий, выполняемых Windows, а PowerShell — нет. Начните активно использовать PowerShell — и быстро осознаете степень своего мастерства.

6.5.4. Помощь службе технической поддержки

Предположим, в вашем отделе технической поддержки работают преимущественно новички, у которых меньше опыта, чем у вас. Чтобы помочь им с решением типичных проблем и выполнением рутинных задач, вы создаете набор инструментов. Новички пока не слишком хорошо пользуются инструментами командной строки, поэтому вы решаете создать графический интерфейс пользователя, используя WPF или коммерческие инструменты наподобие PowerShell Studio.

Как мы уже говорили, GUI — вид контроллера. Это значит, что он должен содержать минимум кода, а конкретно только тот, который *необходим* ему для работы. Нажатие кнопки в GUI может запускать отдельный *скрипт-контроллер*, созданный для автоматизации указанного процесса. Для выполнения задач в рамках этого процесса скрипт может вызывать несколько *инструментов*. Может показаться, что получается много слоев, но польза этого подхода подтверждается рядом доводов.

- *GUI сложно писать и отлаживать.* Чем меньше в них кода, тем проще для вас.
- *GUI никогда не являются единственным местом, где выполняется конкретная задача.* Они должны выступать способом ее *активизации*, но не местом ее фактического нахождения. GUI, который запускает скрипт-контроллер, хорош тем, что тот же скрипт можно запустить откуда-то еще.
- *Отдельный скрипт-контроллер, вызывающий отдельные инструменты, проще разрабатывать и отлаживать.* Вы можете сосредоточиться на решении отдельных задач, используя при этом отдельные инструменты, потом объединить эти инструменты в скрипте-контроллере и затем вызывать его из любого созданного вами GUI.
- *Разделив все по слоям, вы поможете службе технической поддержки лучше выполнять свои обязанности.* GUI — это *интерфейс*, предназначенный для доступа к функциональности. И консоль PowerShell — тоже: это интерфейс командной строки (Command Line Interface, CLI). Предположим, что ваша служба технической поддержки может задействовать функциональность из обоих этих интерфейсов. В таком случае вы можете постепенно переводить их на использование CLI. В результате по мере накопления опыта они смогут расширять свои профессиональные возможности и увеличивать зону контроля. Создание функциональности, не зависящей от интерфейса, — прекрасная идея.

6.6. РАСШИРЯЙТЕ КОНТРОЛЬ

Обсудим еще один нюанс темы инструментов и контроллеров: вам нужно помнить о доступности других инструментов. Эта книга посвящена PowerShell, именно поэтому мы разбираем команды и принципы PowerShell. Но если у вас есть другой инструмент (например, Microsoft Windows Resource Kit Tool или предоставляемый поставщиком инструмент командной строки) и вы видите, что он вам подходит,

то используйте его. Ничто не обязывает вас применять в скрипте-контроллере исключительно PowerShell.

Представьте, что в целях соответствия требованиям вы должны создать отчет для каждого сервера домена, используя инструмент командной строки `MSInfo32.exe`. Какие инструменты вам понадобятся? Возможно, получать учетные записи компьютеров вы будете с помощью `Get-ADComputer` из модуля Active Directory. Вдобавок вы можете решить сначала проверить связь с этими компьютерами, используя `Test-Connection`, а затем, если компьютер находится онлайн, выполнить команду `MSInfo32`. Кроме того, руководитель может попросить вас записать имена недоступных серверов в отдельный текстовый файл. В итоге может оказаться так, что вам не нужно писать какие-либо новые инструменты, а придется создать лишь скрипт-контроллер, который соберет воедино эту коллекцию инструментов: как из PowerShell, так и сторонних. Выглядеть он может так:

```
#GetComplianceInfo.ps1
Get-ADComputer -filter * | foreach {
 if (Test-Connection $_.name -quiet) {
    msinfo32 /computer "\\$($_.name)" /report "c:\work\
  $($_.name)-msinfo.txt"
 }
 else {
    $_.name | out-file c:\logs\offline.txt -Append
 }
}
```

6.7. ПРОВЕРКА ЗНАНИЙ

При прочтении этой главы у вас должен был возникнуть основной вопрос по скриптингу: какой вид скрипта создавать? Явно мы об этом не писали, но зачастую первый этап скриптинга связан не с написанием кода, а с подробным описанием того, что вы хотите реализовать. Если мы справились со своей задачей в данной главе, то вы уже начали воспринимать *инструменты* и *контроллеры* так же, как это делает PowerShell, и начинаете видеть, как они взаимодействуют в целях реализации бизнес-задач. Если вы можете осознать различие этих двух концепций и учитывать их разное предназначение, то сможете достичь успеха в скриптинге PowerShell.

С учетом всего сказанного давайте проверим, насколько хорошо вы поняли весь изложенный в этой главе материал. Подумайте, какие инструменты вам понадобятся для решения описанных ниже бизнес-задач. Определите, какие из этих инструментов вам потребуется создать, а какие уже существуют. Наконец, нарисуйте (на бумаге или в планшете) примерную схему их использования. Это необязательно должен быть реальный код.

1. Вам нужно проанализировать ресурсы отдела и выявить файлы, которые не изменялись уже более года. Начальник хочет получить таблицу Excel, в которой

будет указан путь к файлу, его размер, дата создания и последнего изменения, а также владелец. Совет: не используйте автоматизацию Excel. Все, что вам нужно, — это CSV-файл, который можно открывать и сохранять в этой программе.

2. Каждую неделю вы получаете список аккаунтов пользователей, которые нужно удалить. Если вы делаете это вручную, то должны выполнить следующие действия: деактивировать аккаунт в Active Directory, добавить к нему комментарий с датой удаления, добавить аккаунт в группу Terminated-Users и отправить электронное письмо руководителю удаленного пользователя.

Мы не будем приводить какие-либо ответы или решения, поскольку в данном случае процесс важнее результата.

ИТОГИ ГЛАВЫ

Эта глава была посвящена инструментам и контроллерам, используемым в скриптинге PowerShell. Очень важно понимать, чем они различаются: с помощью инструментов можно выполнять конкретные действия, а контроллеры предназначены для координации работы этих инструментов в более широком контексте, что позволяет более эффективно закрывать потребности бизнеса. Мы использовали различные сценарии, такие как инициализация пользователей и настройки разрешений файлов, чтобы подчеркнуть модульную природу скриптинга PowerShell и ценность применения имеющихся инструментов. Кроме того, мы поговорили об использовании графических интерфейсов в качестве контроллеров, особо выделив универсальность автоматизации в PowerShell. В результате работы над этой главой вы должны были понять, что только тщательное планирование и детальная разработка скриптов помогут вам стать настоящим мастером скриптинга PowerShell, а понимание работы базовых механизмов оболочки позволит эффективно автоматизировать процессы.

7 Скрипты и безопасность

В этой главе мы обсудим вопросы безопасности. Мы уже говорили о ней в нашей предыдущей книге «Изучаем PowerShell за месяц, занимаясь один час в день», но эта тема заслуживает более тщательного рассмотрения. Если анализировать безопасность PowerShell через призму нашего опыта, то кажется, что лучше просто отключить эту систему на всех настольных ПК и серверах в мире. Все из-за его широких возможностей и простоты использования. Именно из-за них ведущие корпорации устанавливают внутренний запрет на применение PowerShell.

Вирусы и вредоносное ПО становятся все более сложными и трудноуловимыми. Специалисты по защите информации, работающие в крупнейших компаниях, хорошо умеют выявлять новые векторы атак и помогают устранять проблемы. Тем не менее в мире все еще много злоумышленников, у которых много свободного времени, что позволяет им находить уязвимости и векторы атак быстрее, чем их успевают выявить и устранить (поэтому очень важно своевременно патчить свои системы). Хорошая новость в том, что многие злоумышленники выбирают в качестве вектора атаки PowerShell. Вы наверняка удивитесь, почему мы считаем это хорошей новостью. Дело в том, что Microsoft и другие доверенные участники сообщества помогли усилить встроенные в PowerShell функции безопасности. Можете рассматривать их как углубленные программы защиты с множеством уровней безопасности. PowerShell нельзя назвать средством противодействия киберугрозам, и он не защитит, если в вашей системе появится вредоносное программное обеспечение. Поэтому важно понимать принципы работы встроенных в PowerShell решений по защите безопасности, чтобы не переоценивать их.

7.1. БЕЗОПАСНОСТЬ ВАЖНЕЕ ВСЕГО

Начнем с главного. PowerShell выполняет ваши указания. Мы пишем скрипт для `Get-Process | Stop-Process`, и оболочка сделает ровно это. Она не выдает никаких предупреждений, никаких подтверждений и никаких уточнений наподобие

«Вы уверены, что хотите сделать это?» — она просто выполнит данную команду. Таким образом, нет способа помешать пользователю намеренно выполнить код, который он скопировал из Интернета. И это проблема кадровиков, а не технологии. Но мы можем ввести меры безопасности, предотвращающие непреднамеренное выполнение кода, который может оказаться вредоносным. Главный предмет интереса Microsoft — *случайное* или *непреднамеренное* выполнение скриптов PowerShell.

Вы будете создавать инструменты и скрипты для себя и других, предполагая, что они будут выполняться (безопасно) в организации. Для этого вам необходимо знать принципы безопасности скриптов.

POWERSHELL КАК ВЕКТОР АТАКИ ДЛЯ ЗЛОУМЫШЛЕННИКОВ

Нет сомнений в том, что некоторые злоумышленники воспринимают PowerShell как удобное средство доставки вредоносных скриптов в вашу систему. Но вам нужно помнить один важный нюанс: все, что взломщики могут проделывать в PowerShell, они могут с той же скоростью осуществлять и без PowerShell.

На своем глубочайшем уровне PowerShell является оберткой для .NET Framework. Если бы оболочки не было, эти внутренние механизмы все равно бы существовали и атакующие использовали бы их. Даже если ваша организация полностью заблокирует использование PowerShell, это создаст лишь ложное чувство безопасности, поскольку взломщикам по-прежнему будет доступна вся внутренняя функциональность.

Изначально PowerShell должен был предоставлять простой способ использования объектной модели компонентов (component object model, COM), .NET Framework и Windows Management Instrumentation (WMI). PowerShell не добавляет в вашу среду новую функциональность. Он добавляет лишь новые *способы использования* тех возможностей, которые в нем уже были. В связи с этим блокировка PowerShell не избавит ни от чего, кроме определенных способов применения чего-либо — это «что-либо» по-прежнему останется в системе.

Полагать иначе все равно что сказать, будто в ваш дом невозможно попасть, поскольку вы закопали все ключи от его дверей. Но в дом можно попасть и другими способами. Открыть дверь ключом — просто самый удобный. Злоумышленник все еще может подобрать отмычку, выбить дверь, разбить окно, и если вы закопаете ключи, то *вам самим* придется использовать эти менее удобные способы.

Как неоднократно говорил участник команды разработчиков продукта Ли Холмс (Lee Holmes), «если вас взломали, то вас взломали». Если в вашу среду попал злоумышленник, то вы уже в печальном положении, а PowerShell при этом вызывает наименьшее беспокойство. Вашей первой целью должно быть удержание злоумышленников *за пределами системы*, а второй — ограничение областей, *к которым они могут получить доступ* в случае успешного проникновения. С точки зрения безопасности блокировка инструментов, которые могут использовать взломщики, является ложным средством.

7.2. ПОЛИТИКА ВЫПОЛНЕНИЯ

По умолчанию PowerShell не будет выполнять свои файлы скриптов в клиентских операционных системах, кем бы вы ни были и какими бы разрешениями ни обладали. Эти файлы имеют расширения `.ps1`, `.psm1`, `.pssc` или `.ps1xml`. Данным поведением управляет политика выполнения на уровне машины. Технически существуют некоторые частные исключения, но для наших задач они неактуальны. Политики устанавливаются один раз и начинают действовать сразу. Чтобы узнать свои текущие настройки, выполните команду `Get-ExecutionPolicy`. Вы должны увидеть одно из значений, указанных в табл. 7.1.

Таблица 7.1. Политики выполнения

Политика	Описание
`Restricted`	На клиентах Windows это установка по умолчанию. Она означает, что никакие файлы скриптов PowerShell, в том числе скрипты профилей, выполняться не будут
`AllSigned`	Эта установка требует, чтобы любой файл скрипта PowerShell содержал корректную цифровую подпись из сертификата, выданного доверенной службой сертификатов. О подписании скриптов мы будем говорить в главе 21
`RemoteSigned`	PowerShell будет выполнять любой локально созданный скрипт, подписанный или нет, но от других скриптов потребует наличия цифровой подписи. Начиная с Windows Server 2012 R2 это установка по умолчанию
`Unrestricted`	PowerShell будет выполнять любой скрипт, задавая минимум вопросов. Вы можете получить запрос от оболочки при выполнении скрипта, который PowerShell определит как скачанный извне вашей машины. Эта установка по умолчанию действует в системах UNIX и Linux
`Bypass`	PowerShell будет выполнять все, не задавая вопросов. Эта политика подразумевает, что вы предпримете отдельные меры по обеспечению безопасности и целостности скриптов
`Undefined`	Политика выполнения отсутствует. PowerShell будет пошагово перемещаться по списку и использует первую активную политику, которую найдет. Чуть позже мы поговорим об этом более подробно

Помните, что выполнение нужно разрешать, лишь когда вы собираетесь выполнять скрипты, которые относятся к вашей настольной системе или централизованному серверу управления. У вас должна быть возможность оставить серверы с их базовыми настройками и изменять только настройки локальных клиентов. Кроме того, вы можете подумать над установкой на системах конечных пользователей политики `Restricted`, если только вам не нужно, чтобы они выполняли скрипты.

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Какая политика выполнения у вас установлена? Выясните это с помощью команды Get-ExecutionPolicy.

Рекомендуется устанавливать политику выполнения с применением Group Policy. Это позволит быть уверенным в том, что каждая машина в среде получила корректную политику выполнения, которую вы и ваша компания решили использовать. Подробнее об указанных выше политиках читайте в разделе справки *about_execution_policies*.

ПОЛИТИКИ ВЫПОЛНЕНИЯ И СЕРВЕРЫ

По умолчанию PowerShell запрещает выполнение скриптов на клиентских компьютерах, поскольку за ними работают обычные пользователи, как правило не обладающие специфическими техническими знаниями. Такие люди зачастую просто ходят по сайтам и пользуются электронной почтой.

Иначе обстоят дела с серверами. Пользователи не должны иметь интерактивный доступ к ним (за исключением серверов Remote Desktop, которые в этом плане скорее являются мультиклиентским компьютером, нежели сервером). Даже администраторы не должны интерактивно авторизовываться на серверах (это не опечатка). Прекратите развертывать серверы, используя опцию Servers Desktop Experience, и начните применять опцию Server Core. Современные версии PowerShell по умолчанию разрешают выполнение скриптов на серверных операционных системах, зачастую из-за инструментов конфигурирования, таких как Server Manager или Windows Admin Center, которым PowerShell необходима для функционирования.

Это возвращает нас к цели PowerShell в отношении безопасности: *воспрепятствовать непреднамеренному* выполнению скриптов *неосведомленным* пользователем. *Неосведомленность* и *непреднамеренное выполнение* на сервере недопустимы. Если такое происходит, значит, налицо, как мы это называем, «проблема с человеческими кадрами».

Итак, что же использует *мы*? По очевидным причинам Microsoft не скажет, какие конкретно настройки нужно использовать в вашей среде. Но вы и ваша команда должны обсудить оптимальную схему. В Microsoft не просто так сделали базовыми определенные установки — может быть, и вам стоит начать именно с них.

7.2.1. Область выполнения

Политику выполнения PowerShell можно установить на одном из трех уровней области действия в следующем порядке приоритетности.

- *LocalMachine* — применяется ко всей машине и сохраняется в JSON-файле конфигурации по адресу `C:\Program Files\PowerShell\7\PowerShell.config.json`.
- *CurrentUser* — применяется только к текущему пользователю и сохраняется в его JSON-файле конфигурации, находящемся в папке `Documents` (`\Documents\PowerShell`). Исключением будет случай, когда такой файл не определен.

- *Process* — управляет текущим сеансом и сохраняется в системной переменной `$env:PSExecutionPolicyPreference`. Эта установка аннулируется сразу после завершения сеанса PowerShell.

Выбранная настройка продолжает действовать, пока активен сеанс PowerShell. Вы можете задать ее, установив переключатель политики выполнения при запуске PWSH.exe. Иными словами, легко осведомленный пользователь, имеющий четкое намерение обойти политику выполнения, может сделать это: что бы вы ни делали где-то еще, кто-то другой при желании сможет запустить оболочку с помощью политики `Bypass`.

Эти политики применяются в том порядке, в каком мы их перечислили, даже если более ограничительная находится ниже. Например, скрипты будут выполняться, если вы установите текущую политику как `RemoteSigned`, а политику машины — как `Restricted`. С практической точки зрения для большинства организаций должно быть достаточно установки политики на уровне машины. На наш взгляд, другие настройки предназначены для особых и исключительных случаев.

ПРИМЕЧАНИЕ

Прежде чем у вас возникнет реальный повод для беспокойства, имейте в виду, что если кто-либо может вносить неавторизованные изменения в политику выполнения, то уже можно беспокоиться. Если это вторжение, то злоумышленник уже может выполнять произвольный код вне PowerShell, и изменение политики выполнения должно волновать вас меньше всего.

Такой порядок применения политик выполнения показывает, что они никогда не задумывались как средство защиты. Мы считаем, что политика выполнения больше похожа на маленькую пластиковую крышечку, которая прикрывает большую красную кнопку, отвечающую за запуск ядерных ракет. Политика выполнения, как и эта крышка, предназначена для того, чтобы не дать запустить ракеты кому-то, кто просто облокотился не туда и не в то время. Они не предназначены для того, чтобы не дать кому-то совершить запланированное действие, как и остановить злоумышленника, который ворвется в ракетный бункер. Проникший внутрь злоумышленник может откинуть защитную крышку так же быстро, как и авторизованный пользователь, то есть она не является механизмом безопасности. Такими механизмами можно считать двери с замком, открыть который можно с помощью специальной карты, и вооруженную охрану, но никак не маленькую крышечку.

7.2.2. Выяснение ваших политик

Узнать текущие настройки политик выполнения можно с помощью команды `Get-ExecutionPolicy`:

```
PS C:\> Get-ExecutionPolicy
Restricted
```

Этот командлет вернет действующую политику на основе выбранной области действия. Иными словами, он вернет политику, которой будет следовать текущий экземпляр оболочки, откуда бы эта политика ни была установлена. Можно получить текущие настройки и для всех областей:

```
PS C:\> Get-ExecutionPolicy -List
        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy      Undefined
   UserPolicy      Undefined
      Process      Undefined
  CurrentUser      Undefined
LocalMachine       Restricted
```

Области *политик* устанавливаются через Group Policy, которую мы не используем. Кроме того, стоит отметить, что этот перечень *представлен не в порядке применения* — очередность пунктов данного списка не имеет никакого смысла. В данной ситуации будет применена политика `Restricted`, в чем мы можем убедиться:

```
PS C:\> C:\work\test.ps1
C:\work\test.ps1 : File C:\work\test.ps1 cannot be loaded because running scripts
is disabled on this system. For more information, see about_Execution_Policies at
https:/go.microsoft.com/fwlink/?LinkID=135170.
```

Естественно, если мы хотим, чтобы наши скрипты выполнялись, то нужно внести изменение.

7.2.3. Установка политики выполнения

Для изменения политики используется командлет `Set-ExecutionPolicy`. Вам нужно указать саму политику и при желании ее область действия. По умолчанию будет использована локальная машина. Иными словами, если вы пытаетесь изменить установку локальной машины, то вам нужно запустить оболочку с параметром As Administrator, поскольку установка локальной машины хранится в JSON-файле ее конфигурации, расположенном в Program Files, куда могут делать запись только администраторы. Имейте в виду, что вы не можете изменить таким образом ни одну из установок, управляемых Group Policy. Вам, очевидно, нужно использовать для этого данную группу. Кроме того, вы не можете изменить политику выполнения, установленную для определенной области. Она должна устанавливаться при запуске PowerShell, а не когда она уже запущена и вы в ней работаете:

```
PS C:\> 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 https:/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"):
```

Чтобы внести изменения, на запрос оболочки нужно ответить Y. Изменение произойдет сразу. Обратите внимание: обычный пользователь может менять политику выполнения для себя или процесса. Как раз поэтому все эти приемы не считаются обеспечивающими безопасность.

7.3. POWERSHELL НЕ ЯВЛЯЕТСЯ ПРЕДУСТАНОВЛЕННЫМ ПРИЛОЖЕНИЕМ

Напоминаем: эти установки препятствуют случайному выполнению скриптов PowerShell. Итак, что произойдет, когда Мисси щелкнет на вложении в электронном письме, чтобы проверить статус доставки заказа с Amazon? Если это скрипт PowerShell, то он не выполнится автоматически. По умолчанию с файлами `.ps1` ассоциируется приложение Блокнот, а не PowerShell. В итоге, когда Мисси щелкнет на вложении, скрипт откроется в Блокноте. Естественно, вы можете изменить ассоциированное с этими файлами приложение, а некоторые редакторы скриптов будут автоматически ассоциироваться с `.ps1` и другими расширениями файлов, предназначенных для редактирования скриптов. Это касается и любого файла PowerShell в Проводнике Windows: дважды щелкнув на нем, вы откроете файл в Блокноте.

Можно ассоциировать с этими расширениями файлов действие Выполнить (в противоположность действию Редактировать). В результате файлы будут выполняться, если на них щелкнуть дважды. Но мы считаем, что это ужасная идея.

7.4. ВЫПОЛНЕНИЕ СКРИПТОВ

Наконец, если вы настроили все для выполнения скриптов, вам нужно указать путь к их файлу, даже если он находится в том же каталоге. Предположим, у нас в текущем каталоге есть тестовый скрипт, который мы хотим выполнить:

```
PS C:\work> test
test: The term 'test' is not recognized as the name of a cmdlet, function, script
file, or operable program. Check the spelling of the name, or if a the path was
included; verify that the path is correct and try again.
```

Ничего не получится. Так задумано для предотвращения *перехвата* команд, когда кто-то или что-то помещает вредоносный скрипт в каталог с типичным именем команды наподобие `dir`. Вам нужно сообщить PowerShell, что вы хотите именно выполнить скрипт:

```
PS C:\work> .\test
Handles  NPM(K)     PM(K)       WS(K)       CPU(s)       Id   SI  ProcessName
-------  ------     -----       -----       ------       --   --  -----------
   2517     276   1146832     1082780     2,365.03     1328    1  dwm
      0       0      1884      748852       115.84     5408    0  Memory Compress…
   1538     208    992756      353264     4,789.05     8284    1  firefox
    483      62    215324      218868       169.59    12232    1  slack
   1999     111    202616      199176       945.55     4284    1  WINWORD
```

Вам необязательно добавлять расширение файла, но лишним оно не будет. В таком случае вы точно не ошибетесь со скриптом, который хотите запустить. Если вы используете функцию заполнения нажатием клавиши Tab, то PowerShell все равно добавит расширение файла.

ВЫ ЧАСТЬ СИСТЕМЫ БЕЗОПАСНОСТИ

Помните: когда дело касается безопасности, вы — часть общей системы. Вы можете сказать: «Но у нас есть администраторы, которые могут не понимать, когда можно безопасно выполнить скрипт». И это, повторимся, говорит о наличии проблемы с кадрами: такие люди не должны занимать должности инженеров DevOps.

Именно здесь в игру вступает перехват команд. Раньше это была реальная проблема в MS-DOS, и объяснялась она расстановкой в системе приоритетов при обработке команд. Если вы выполняли `Dir`, то сначала эта команда искала пакетные файлы с этим именем, затем исполняемые и потом уже внутренние команды — или что-то вроде этого. То есть можно было вбросить исполняемый или пакетный файл *с тем же именем*, что и у внутренней команды, обманув пользователей и заставив их выполнить вместо желаемой команды исполняемый либо пакетный файл.

В случае с PowerShell эта хитрость более очевидна. `Dir` практически всегда* будет выполнять команду `Get-ChildItem`, в то время как `./Dir` будет выполнять `Dir.ps1` из текущего каталога. Но именно *вы* должны понимать разницу. Если вы ее не знаете или игнорируете, то данная мера безопасности не будет работать. Вас все равно могут обхитрить, если вы будете недостаточно бдительными, поскольку именно *вы* являетесь тем проводником, благодаря которому работают эти меры безопасности.

---

* Можно заставить `Dir` выполнять нечто иное, нежели команду `Get-ChildItem`. Причем сделать это легко. Просто переопределите ее псевдоним для выполнения другой команды или загрузите альтернативную команду `Get-ChildItem`. Будет крайне просто, например, с помощью вредоносной программы внедрить ее в ваш скрипт профиля PowerShell, являющийся простым текстовым документом, расположенным в личной папке `Documents`, к которой у вас явно есть полный доступ. Он выполняется каждый раз, когда вы запускаете оболочку, и вы никогда не поймете, пошло ли что-нибудь не так. Это один из аргументов в пользу политики `AllSigned`: внедрение чего-либо в ваш профиль нарушит его подпись, вызвав ошибку при запуске оболочки. Если ваш сертификат для подписания кода не был установлен локально (что было бы очень неудобно) или вы защитили его паролем (более удачная идея), то подобное внедрение команды не сможет привести к повторному подписанию скрипта.

7.5. РЕКОМЕНДАЦИИ

Что мы обычно рекомендуем, когда вопрос касается политики выполнения? Вас это может удивить.

- Если для контроля релизов скриптов будут использоваться сертификаты, то мы предлагаем устанавливать политику `AllSigned`. И это не столько мера безопасности, сколько процедурное решение. Ваша компания решает, что подпись в скрипте будет определять его как готовый к использованию в производственной среде. Вдобавок это помогает усложнить процесс внедрения команды в скрипт профиля, о чем мы говорили в конце врезки, приведенной выше. Помимо этого, `Allsigned` может оказаться полезной на клиентских компьютерах, где вам нужно допустить выполнение скриптов (в противном случае используйте для них `Restricted`), а также там, где вы хотите наложить ограничения в отношении того, какие скрипты могут выполнять пользователи. Помните, что пользователь, выполняющий скрипт, по-прежнему *не может* сделать ничего, на что у него нет разрешения, и запуск скрипта — это не единственный путь, которым вредоносное ПО может обхитрить пользователя. Это не мера безопасности, а лишь небольшая преграда, не позволяющая кому-либо случайно выполнить что-то, о чем он в итоге может пожалеть.
- В большинстве случаев мы используем `RemoteSigned`. Эта политика позволяет соблюсти хороший баланс между неудобством и защитой от дурака. Скрипты, скачанные через приложение Microsoft, такое как Internet Explorer, Edge или Outlook, будут помечены этим приложением как удаленные, то есть PowerShell не будет их выполнять, предварительно не спросив пользователя. Естественно, это не мера *безопасности*, а лишь дополнительная преграда. Мы все знаем, что если система задает вопрос «Вы уверены?», то почти все пользователи рефлекторно отвечают «Да», так что эта мера вряд ли *остановит* кого-либо или заставит задуматься дважды.
- Мы не видим особой разницы между политиками `RemoteSigned` и `Unrestricted`, кроме того, что *большинство* скриптов, доступ к которым осуществляется через универсальное соглашение об именовании (universal naming convention, UNC), будут работать под управлением `RemoteSigned`, а не `Unrestricted`.
- Мы предлагаем политику `Bypass`, если вы не используете `AllSigned` по одной из указанных выше причин и если вас не устраивает ложное чувство безопасности, которое появляется при использовании `RemoteSigned` и `Unrestricted`. Устанавливая `Bypass`, вы как бы говорите: «Я знаю, что эта политика выполнения не является уровнем безопасности. Я настолько уверен (-а) в предпринятых мной мерах защиты, таких как строгий контроль доступа, что даже не желаю использовать данную политику, поскольку боюсь, что *некоторые* пользователи могут *посчитать* ее уровнем безопасности. А я хочу, чтобы эта мысль даже не приходила им в голову».

Вот почему важен последний пункт: многие так называемые профессионалы по информационной безопасности не уделяют внимания изучению принципа политик безопасности в PowerShell. Они размышляют примерно так.

1. В вузе мы изучали, что скрипты ужасны с точки зрения безопасности.
2. Политика выполнения позволяет мне исключить возможность скриптинга.
3. Вредоносным программам не нужен скриптинг, но защита в глубину подразумевает, что я должен исключить как можно больше угроз.
4. Значит, лучше всего использовать в качестве политики безопасности `Restricted`.

При таком рассуждении упускается мысль, что любой автор вредоносного ПО может обойти политику `Restricted`, проявив толику смекалки. Мы бросаем вызов таким «профессионалам» по безопасности, задавая вопрос: «Хорошо. А как бы вы защитили среду, если бы мы заставили вас установить политику выполнения `Bypass`?» Их ответы варьируются от совершенно правильных: «Убедились бы, что наши брандмауэры многоуровневые, а наша защита от вредоносных программ обновлена и тоже является многоуровневой» до просто возмутительных: «Выдернули бы все шнуры питания».

Исключите политику выполнения из уровней безопасности (поскольку он не один) и начните думать о реальных политиках безопасности.

ИТОГИ ГЛАВЫ

Настройки, связанные с выполнением скриптов в PowerShell, по умолчанию максимально ограничительные. Любые изменения, которые вы вносите, лишь ослабляют их. Вам следует рассмотреть другие типичные рекомендации Windows, такие как предоставление минимально необходимых привилегий для выполнения, фильтрация электронной почты и надежная защита файлов. Вы определенно захотите использовать скрипты PowerShell, ведь именно поэтому и читаете эту книгу. Так что ваша задача — использовать их максимально безопасно. Надеемся, мы смогли указать вам верное направление. Закончив читать данную главу, вы должны решить, как будете применять эти идеи в своей организации.

Часть II

Наше путешествие по миру скриптинга PowerShell продолжается. В этой части мы плавно перейдем от основных принципов к практической реализации и познакомим вас с поэтапным процессом создания надежных скриптов PowerShell. В главе 8 вы узнаете, почему так важно предварительное проектирование, а в главе 9 мы расскажем, как можно снизить количество ошибок и багов, возникающих при написании скрипта. В главах 10–16 вы сможете улучшить навыки разработки скриптов. Вы постепенно перейдете к созданию простых функций и модулей скриптов, подробно изучите некоторые расширенные функции и оцените потенциал объектов в качестве вывода скриптов. После этого вы научитесь мастерски использовать различные потоки для ввода и вывода, освоите искусство документирования с помощью комментариев и разберете проблемные ситуации, попутно познакомившись с эффективными способами их устранения. Завершит эту часть глава 16, в которой вы заполните манифест. Этот документ является важным компонентом, гарантирующим, что скрипт четко определен и готов к развертыванию. По мере чтения вы будете вырабатывать практические навыки и начнете лучше понимать принципы стратегического мышления, лежащего в основе эффективной разработки скриптов PowerShell.

8 Всегда начинайте с проектирования

Большинство наших скриптов начинаются как однострочные команды для выполнения некоего действия, например, создания нового пользователя. Такой скрипт будет считывать данные из CSV-файла, создавать новых пользователей и вносить всю информацию, полученную от отдела кадров. Написанный скрипт вы сохраняете где-нибудь на диске C и переходите к следующей задаче.

Звучит знакомо? В мире есть два типа создателей скриптов и разработчиков инструментов: одни все планируют, а другие действуют необдуманно. Наша цель — к концу книги (надеемся, даже к концу этой главы) научить вас проектировать инструменты. Взгляните на этот процесс с противоположной стороны: что является желаемым результатом и как его достичь? Поняв это, опишите и все остальные промежуточные этапы.

В этой главе мы изложим некоторые основные принципы проектирования инструментов PowerShell, чтобы далее вы придерживались правильного пути. Говоря точнее, мы будем отталкиваться от того, о чем говорили в части I. Теперь мы готовы привести более конкретные примеры.

8.1. ИНСТРУМЕНТЫ ДЕЛАЮТ ЧТО-ТО ОДНО

Если вы смотрели наши выступления на тему создания инструментов и скриптов, то видели, как каждый из нас заявляет, что скрипт должен выполнять одну задачу. Он не создает пользователя Active Directory (AD user) и почтовый ящик M365 — это две задачи, а значит, они должны быть разделены. Такой принцип проектирования использовался в PowerShell с самого начала. Взгляните на команды, которые изначально содержатся в этом языке. Они все делают что-то одно. Например, `Get-Service` не запускает и не останавливает сервисы. Эта команда получает о них информацию и возвращает ее вам. Если вы хотите остановить сервис, то можете использовать команду `Stop-Service`.

Именно этот принцип новички нарушают чаще всего. Например, мы видим, как разработчики создают команду с параметром `-ComputerName`, позволяющим получить имя удаленной машины, и параметром `-FilePath`, который нужен для того, чтобы считывать имена компьютеров из файла. С точки зрения PowerShell, да и нашей тоже, это абсолютно ошибочное решение, поскольку оно ведет к тому, что инструмент делает два действия, а не одно. Корректным вариантом проектирования, соответствующим парадигме PowerShell, будет использование только параметра `-ComputerName`, получающего строки (имена компьютеров) из конвейера. Кроме того, можно передавать эти имена из файла с помощью скобочной конструкции `-ComputerName (Get-Content servers.txt)` либо определить параметр `-ComputerName`, позволяющий получить ввод по значению:

```
Get-Content servers.txt | Get-ServerInfo
```

Команда `Get-Content` существует ради единственной цели: получать данные из файла. Так зачем создавать для этого что-то другое? Уделяйте время важным задачам и прекратите изобретать колесо. Сотрудники Microsoft потратили немало времени на создание продуманных команд PowerShell.

Пожалуй, стоит более подробно рассмотреть описанный антипаттерн. Вот пример использования фиктивной команды (это значит, что не нужно пробовать выполнять ее) двумя разными способами:

```
# Указываем три имени компьютеров
Get-CompanyStuff -ComputerName ONE,TWO,THREE
# Указываем файл с именами компьютеров
Get-CompanyStuff -FilePath ./names.txt
```

Такой подход излишне усложняет инструмент, который в итоге становится труднее писать, отлаживать, тестировать и сопровождать. А вот пример того, как аналогичный подход ведет к получению того же эффекта в более простом инструменте:

```
# Указываем три имени компьютеров
Get-CompanyStuff -ComputerName ONE,TWO,THREE
# Указываем файл с именами компьютеров
Get-CompanyStuff -ComputerName (Get-Content ./names.txt)
# Либо, если вы создали инструмент правильно…
Get-Content ./names.txt | Get-CompanyStuff
```

Эти паттерны отлично имитируют работу внутренних команд PowerShell. Тем не менее мы разберем еще один антипаттерн, суть которого можно описать примерно так: «У меня имена компьютеров находятся в файле особого формата, считывать который умею только я». Порой разработчики убеждают себя, что вполне можно сделать так:

```
# Указываем три имени компьютеров
Get-CompanyStuff -ComputerName ONE,TWO,THREE
# Указываем файл с именами компьютеров
Get-CompanyStuff -FilePath ./names.dat
```

СОВЕТ

А вы знали, что PowerShell раньше не умел считывать файл .DAT? Эта возможность появилась лишь в PowerShell 6.

8.2. ИНСТРУМЕНТЫ МОЖНО ТЕСТИРОВАТЬ

При создании инструментов следует учитывать еще один момент: если вы хотите делать это на профессиональном уровне, то вам потребуется создавать для них еще и автоматизированные модульные тесты. Подробно мы будем говорить об этом в главе 20. В сообществе PowerShell идут активные споры на тему тестирования. Одни участники заявляют, что сначала нужно писать тесты, а потом уже код. Таким образом, вы будете работать, двигаясь в направлении желаемого результата. Другие говорят, что сначала необходимо написать рабочий код, а потом уже тесты, чтобы убедиться в том, что он работает корректно в случае добавления новой функциональности.

Нельзя сказать, какой из этих подходов единственно верен, поскольку это вопрос личных предпочтений. Но с точки зрения проектирования вам нужно сделать так, чтобы проектируемые вами инструменты можно было тестировать. Опять же, одним из способов реализовать это является разработка узкоспециализированных инструментов, выполняющих строго что-то одно. Чем меньше функциональных возможностей предоставляет инструмент, тем меньше вам придется тестировать. Чем меньше разветвляется логика кода, тем проще проводить тщательное модульное тестирование.

Предположим, вам поставили задачу создать процесс для приема на работу новых сотрудников. Вы можете создать монолитный скрипт объемом более 100 строк, который будет:

- считывать полученный от кадровиков CSV-файл;
- создавать логин в соответствии со стандартами компании;
- проверять, не существует ли уже это имя в Active Directory, там же в AD создавать пользователя со всей полученной информацией;
- заводить для него почтовый ящик в Microsoft 365;
- добавлять этого пользователя в онлайн-группы SharePoint и т. д.

Сразу понятно, к чему все это ведет. А вот плохие новости: такой скрипт выполняет слишком много задач и его нужно разбить на четыре отдельных (рис. 8.1).

Вдобавок вам следует избегать встраивания в инструменты такой функциональности, которую будет трудно тестировать. Например, вы можете решить добавить определенный способ логирования ошибок. Это прекрасно. Но если это логирование будет происходить в базу данных SQL Server или инструмент SIM, то будет сложно его тестировать и проверять, корректно ли оно работает. Логирование в файл

обработать проще, поскольку он более доступен для проверки. Но еще проще будет написать специальный *отдельный инструмент*. Это позволит тестировать его независимо и *использовать* в других инструментах. Здесь мы возвращаемся к идее о том, что каждый инструмент должен выполнять что-то одно.

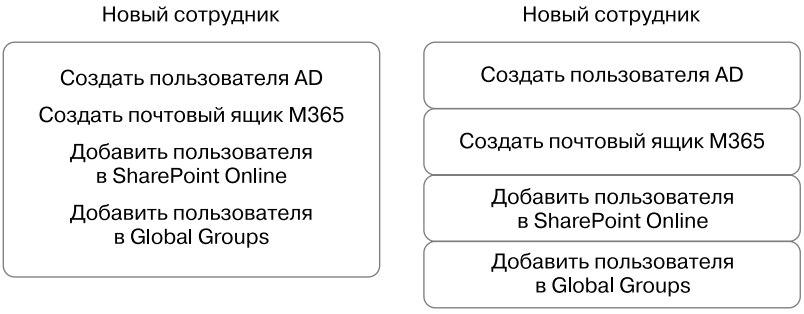


Рис. 8.1. Разделение скрипта на несколько частей

8.3. ИНСТРУМЕНТЫ ДОЛЖНЫ БЫТЬ ГИБКИМИ

Инструменты нужно проектировать так, чтобы их можно было применять в различных ситуациях. Зачастую это означает использование параметров для получения ввода из конвейера. Предположим, вы пишете инструмент `Set-MachineStatus`, который изменяет какую-то настройку компьютера. Вы можете указать параметр `-ComputerName`, позволяющий получить имена компьютеров. Сколько имен он будет получать в таком случае: одно или несколько? Откуда он будет их получать? Всегда по возможности предполагайте, что будет несколько имен, и не беспокойтесь о том, откуда они поступают. С точки зрения проектирования вам нужно сделать так, чтобы можно было применять разные подходы.

Будет нелишним написать несколько примеров использования команды, которые *должны работать*. Позднее они могут стать образцами файла справки (о котором речь пойдет в главе 14), однако на этапе проектирования такие шаблоны позволят убедиться, что каждый из них в итоге окажется рабочим. Например, вам может потребоваться возможность применения следующих паттернов:

```
Get-Content names.txt | Set-MachineStatus
Get-ADComputer -filter * | Select -Expand Name | Set-MachineStatus
Get-ADComputer -filter * | Set-MachineStatus
Set-MachineStatus -ComputerName (Get-Content names.txt)
```

Третий пример потребует от вас быть более внимательными при проектировании, поскольку вы не сможете соединить объект компьютера из Active Directory с тем же параметром `-ComputerName`, который получает объект `String` из `Get-Content`! Реализация обоих сценариев может потребовать двух наборов параметров, в одном из которых, возможно, используется `-ComputerName <string[]>`,

а во втором — `-InputObject <ADComputer>`. Создание двух наборов параметров слегка усложнит программирование и автоматизированное модульное тестирование — так что вам придется решить, стоит ли оно того. Будет ли этот третий пример использоваться настолько часто, что это оправдает лишние строки кода и тестирование, или же окажется достаточно редким сценарием, в связи с чем его можно заменить аналогичным вторым вариантом?

Суть в том, что каждое принимаемое вами решение по проектированию будет влиять на код вашего инструмента, его модульные тесты и т. д. Любые подобные решения стоит продумывать заранее, поэтому данный этап и зовется *этапом проектирования*.

8.4. ИНСТРУМЕНТЫ ДОЛЖНЫ ВЫГЛЯДЕТЬ НАТИВНЫМИ

Наконец, будьте осторожны с выбором имен инструментов и параметров. Мы уже говорили об этом в части I, но стоит повторить, поскольку мы регулярно видим, как разработчики «креативят» на эту тему. Названия инструментов должны всегда соответствовать принятому в PowerShell паттерну *«глагол — существительное»*, и в них стоит использовать только наиболее подходящие формы глаголов из списка, возвращаемого командой `Get-Verb`. Этот список дополнительно размещен на странице Microsoft Learn (http://mng.bz/A8rE); он содержит некорректные вариации и объяснения, которые вы можете использовать для проверки своих идей. Не корите себя слишком сильно за тонкие расхождения с утвержденными глаголами, например между *Get* и *Read*. Если вы ознакомитесь с указанным списком, то поймете, что `Get-Content` скорее должен быть `Read-Content`. По всей видимости, в Microsoft этот второй вариант появился, *уже когда* `Get-Content` активно использовался.

Кроме того, рекомендуем выработать привычку дополнять существительную часть команды кратким префиксом. Например, если вы работаете на Globomantics, Inc., то можете создавать команды с названием наподобие `Get-GloboSystemStatus`, а не просто `Get-SystemStatus`. Благодаря тому, что префикс расширяет название, исключается конфликт новых команд с уже существующими, а также упрощается поиск и определение команд или инструментов, созданных конкретно для вашей организации.

ПРИМЕЧАНИЕ

Одна из причин, по которой мы затронули тему нативных паттернов в части I, — их важность. Не забывайте, что существующие команды, особенно ключевые, которые разработала команда PowerShell из Microsoft, представляют видение их создателей в отношении того, как работает эта оболочка. Если решите идти вразрез с этим видением, то делайте это на свой страх и риск.

Имена параметров всегда должны соответствовать внутренним паттернам PowerShell. Если вам вдруг нужен какой-либо параметр, то обратитесь к огромному арсеналу готовых команд PowerShell, чтобы понять, какое имя параметра используется в них

для аналогичных целей. Например, если вам нужно получать имена компьютеров, то следует использовать `-ComputerName` (обратите внимание: имя в единственном числе), а не какую-либо вариацию наподобие `MachineName`. Если вам нужно имя файла, то в большинстве нативных команд для этого используется `-FilePath` или `-Path`.

ДИЛЕММА С ГЛАГОЛАМИ

Есть кое-что, в чем можно слегка запутаться. Это касается подбора правильного глагола для имени команды. Честно говоря, в Microsoft предоставляют слишком много глаголов на выбор. Мы уверены, что кто-то внутри компании отчетливо понимал разницу между ними, однако до пользователей PowerShell ее доносили не всегда достаточно хорошо. Например, если вы пишете команду для извлечения информации из базы данных SQL Server, то как назовете ее: `Get-MyWhatever-Data` или `Read-My-WhateverData`? В Microsoft по этому поводу есть руководство, в котором сказано: «Глагол `Get` используется для извлечения ресурса, такого как файл. Глагол `Read` служит для получения информации из источника, такого как файл». Таким образом, подразумевается, что `Get` нужно использовать для *получения* (get) *файла*, то есть объекта, представляющего сам файл, а `Read` — для получения *содержимого файла*. Но при этом мы имеем `Get-Content`, то есть в Microsoft не последовали собственной же рекомендации.

А что рекомендуем мы? Выбирайте тот вариант, который покажется наиболее согласованным с элементами, уже имеющимися в PowerShell. Если же самостоятельно разобраться не получится, то обратитесь с вопросом на форумы PowerShell.org, где вы сможете получить помощь профессионалов.

8.5. ПРИМЕР

Для того чтобы начать принимать решения, вам потребуется проанализировать бизнес-требования. После этого попробуйте перевести эти требования в примеры использования, чтобы вам с командой стало ясно, как конкретно можно будет применять готовый инструмент. Если есть и другие заинтересованные стороны (возможно, люди, которые будут использовать этот инструмент, например служба технической поддержки), то будет нелишним согласовать с ними функциональную спецификацию, чтобы вы могли перейти к этапу проектирования, имея общее видение итогового результата. Кроме того, постарайтесь четко сформулировать *задачи*, которые должен решать новый инструмент, поскольку они порой лучше отражают видение бизнеса, чем спецификация, которую мог писать кто-то другой. Разберем пример.

Задача бизнеса: у нас в компании используется множество компьютеров, в которых установлены комплектующие от разных поставщиков, разные версии Windows, настроены разные конфигурации и т. д. Когда пользователи обращаются в службу технической поддержки, ее специалистам зачастую трудно понять, о каком

компьютере идет речь. Пользователи не всегда знают такие детали, как номера моделей, версии ОС, объем установленной оперативной памяти и пр. У нас есть система управления конфигурацией, к которой может обратиться специалист техподдержки, но она не всегда оказывается актуальной или точной. Нам нужен инструмент, с помощью которого служба технической поддержки сможет быстро получать важную информацию о доступности компьютера, а также о его ОС и аппаратной конфигурации. Иногда у нас бывают периоды простоя, и мы можем запрашивать эту информацию от нескольких компьютеров, перепроверяя точность данных системы управления конфигурацией. Если ее база данных потребует обновления, то сотрудники техподдержки выполнят его.

БУДЬТЕ ВНИМАТЕЛЬНЫ К КОНТЕКСТУ

Когда вы начинаете проектировать инструменты, вполне естественно продумывать формулировки задач, которые эти инструменты решают на уровне бизнеса, ведь это значительная часть этапа проектирования. Очень хороши формулировки наподобие: «Когда пользователи звонят в техподдержку, ее специалистам зачастую трудно понять, о каком компьютере говорит человек».

Кроме того, вполне нормально описывать желаемый результат, как это делали мы, например: «Нам нужен инструмент, с помощью которого сотрудники техподдержки смогут быстро запрашивать информацию о компьютере». Он определяет потребность бизнеса. Но не каждую задачу бизнеса вы обязательно должны решать с помощью *одной* команды или инструмента. Вам может потребоваться набор инструментов, упакованных в модуль… Но не будем забегать вперед.

Мы говорили о том, что инструменты должны быть максимально отделены от конкретного контекста. Тем не менее наша бизнес-задача выражает контекст очень четко: «Нам нужно, чтобы техники могли запрашивать различную информацию». И эта формулировка ведет к конкретным допущениям, например: «Вывод должен быть понятным человеку» и, возможно, «Наши техники не очень опытные, поэтому для работы им потребуется графический интерфейс». Это хорошая исходная информация, но она не означает, что для решения поставленной задачи нужно использовать лишь один инструмент.

В целом нам требуется инструмент для извлечения данных и, возможно, скрипт-контроллер, предоставляющий службе техподдержки интерфейс ввода/вывода. При этом неважно, как именно специалист использует *инструмент* или что он увидит с его помощью. Все эти связанные с контекстом нюансы можно поручить *контроллеру*, а *инструмент* использовать внутри системы для получения данных.

Никогда не забывайте о паттерне проектирования «инструмент/контроллер». Привыкайте читать бизнес-задачи, которым в итоге потребуются инструменты *и* контроллеры, а также учитесь понимать, с помощью какого вида скриптов лучше всего реализуются те или иные элементы бизнес-задач.

Если более глубоко вникнуть в материал врезки, то могут возникнуть уточняющие вопросы относительно того, что конкретно должен запрашивать инструмент. Предположим, ответ будет следующий:

- имя хост-компьютера;
- данные о производителе;
- модель;
- версию ОС и номер сборки;
- версию пакета обновлений, если таковой имеется;
- информацию об установленной оперативной памяти;
- тип процессора;
- количество сокетов процессоров;
- общее количество ядер;
- свободное место на системном диске (обычно C: но не всегда).

Это нормально — мы знаем, что можем каким-то образом получить эту информацию. Мы знаем, что собираемся написать инструмент `Get-MachineInfo` и в нем наверняка будет не менее одного параметра `-Computer-Name`, получающего в качестве строк одно или несколько имен компьютеров. Если подумать наперед, то мы можем начать делать и заметки для команды `Update-OrgCMDatabase`, которая будет получать вывод от `Get-MachineInfo` и автоматически обновлять базу данных организации, предназначенную для управления конфигурацией. Об этом никто *не просил*, но сама постановка бизнес-задачи это подразумевает, и запрос обязательно появится, когда мы предоставим специалистам первый инструмент. Прозвучит это примерно так: «Послушайте, если инструмент получает всю эту информацию, то можно как-то переместить ее в конфигурационную базу данных?» Мы будем предусмотрительными и учтем это сразу при проектировании первого инструмента: нам нужно сделать так, чтобы он выводил то, что в дальнейшем сможет легко получить другая команда.

Предположим, некоторые компьютеры на запрос не ответят, и спроектируем на этот случай отдельный механизм. Допустим также, что некоторые компьютеры работают на старых версиях Windows, и сделаем так, чтобы инструмент был приспособлен для работы не только с последними версиями этой ОС, но и с настолько старой, насколько это возможно.

Примеры использования описанного проекта могут быть довольно простыми:

```
Get-MachineInfo -ComputerName CLIENT
Get-MachineInfo -ComputerName CLIENTA,CLIENTB
Get-MachineInfo -ComputerName (Get-Content names.txt)
Get-MachineInfo -ComputerName (Get-ADComputer -id CLIENTA |
Select -Expand name)
Get-Content names.txt | Get-MachineInfo
Get-ADComputer -id CLIENTA | Select -Expand name | Get-MachineInfo
```

Для второго набора примеров потребуются те же элементы, притом что в последнем наборе будут использоваться уже другие. И это не проблема. Их вывод должен быть вполне детерминированным. Передавая в инструмент конкретный набор входных данных, мы должны получать один и тот же вывод, что упростит написание модельных тестов для проверки данного проекта. Наша команда выполняет лишь одно действие с небольшим количеством параметров, что хорошо демонстрирует узкий охват ее структуры.

ПОЛЬЗА ДОБАВЛЕНИЯ В ПРОЕКТ ПРИМЕРОВ ИСПОЛЬЗОВАНИЯ

Описание примеров использования инструмента в рамках его проекта станет *прекрасной* идеей. Как минимум это поможет случайно не превратить проектирование *инструмента* в проектирование *контроллера*. Если ваши примеры использования начинают выглядеть сложно и занимать порядка десяти страниц, это говорит о том, что вы недостаточно сужаете область задачи инструмента, и желательно рассмотреть создание нескольких решений, а не одного.

Кроме того, примеры использования могут стать частью итогового файла справки. Есть мнение, что проектирование инструмента желательно *начинать* с написания файла справки. В итоге этот файл сможет существовать как функциональная спецификация, в соответствии с которой вы будете писать код. По той же логике написание примеров использования помогает поддерживать модель *разработки через тестирование* (test-driven development, TDD), в которой вы сначала создаете автоматизированные тесты для упорядочивания или определения работы инструмента, после чего пишете его код.

Изначальное написание примеров использования помогает избежать неудачных проектных решений. Если вы не можете прописать все нужные примеры и ваш список все еще слишком длинный или сложный, то это признак ошибочного пути. Возможно, стоит попросить коллегу помочь вам совместно выполнить рефакторинг всего проекта, чтобы его упростить.

Мы бы показали этот набор примеров участникам команды разработчиков и спросили, что они думают. Практически всегда это приведет к возникновению таких вопросов:

- Как мы узнаем, если компьютер выйдет из строя?
- Продолжит ли инструмент работать?
- Будет ли инструмент логировать информацию о сбое куда-либо?

Что ж, нам нужно немного доработать проект. Мы знаем, что в случае сбоя инструмент должен продолжать работать и у пользователя должна быть возможность логировать информацию о сбое, например, в текстовый файл:

```
Get-MachineInfo -ComputerName ONE,TWO,BUCKLE,SHOE ` -LogFailuresToPath errorlog.txt
```

Если текстовый файл в качестве журнала ошибок устроит команду, то можно добавить его в проект. Если же участники команды захотят что-то посложнее — возможность логировать информацию в базу данных или журнал событий, — то мы спроектируем для этого отдельный инструмент. Но чисто теоретически сейчас предположим, что нас устраивает текстовый файл. На данном этапе мы еще не выясняем, *как именно* все будет реализовываться, а просто проектируем.

Допустим, что команду устраивают эти дополнения и мы зафиксировали примеры желаемого использования. Теперь можно перейти к программированию. Но прежде чем мы это сделаем, почему бы вам не попробовать свои силы в проектировании?

ПРОЕКТИРОВАНИЕ НАБОРОВ КОМАНД

Текущая дискуссия прекрасно подходит в случаях, когда вы пишете команду для выполнения чего-либо автономного, например, извлечения административной информации из нескольких компьютеров. А вот при написании набора команд для управления крупной системой будут актуальны уже иные инструкции.

Например, предположим, что вам нужно написать набор команд для управления приложением, отслеживающим информацию пользователя. Какие команды могут потребоваться?

Начните с определения актуальных *существительных* в системе. С чем конкретно она работает? Пользователи? Клиенты? Заказы? Элементы в заказах? Адреса? Запишите все это где-нибудь в виде списка.

Затем взгляните на каждое существительное и решите, что система может с ним сделать. Например, какие возможности система предлагает в отношении пользователей? Создание новых? Их удаление? Изменение существующих? Перечисление всех? Из всего этого вы получите актуальные глаголы: *New*, *Remove*, *Set* и *Get*. В данном случае можно создать команды наподобие `New-SystemUser`, `Remove-SystemUser`, `Set-SystemUser` и `Get-SystemUser` (при условии, что термин *System* пригоден в качестве префикса для вашей организации).

Это небольшое упражнение поможет убедиться, что вы не упускаете никакую ключевую функциональность. Составление списка команд не означает, что вы обязательно должны *написать* все эти команды, но позволяет правильно расставить приоритеты и работать в соответствии с ними.

8.6. УПРАЖНЕНИЯ

Если вы работаете в команде, то сможете прекрасно подискутировать на эту тему. Вам не потребуется компьютер — достаточно доски или ручки с бумагой. Идея в том, чтобы ознакомиться со всеми бизнес-требованиями и составить отвечающие им примеры использования. Мы предоставим *все* бизнес-требования в одном выражении, так что вам не придется «возвращаться к команде» и собирать дополнительную информацию.

8.6.1. Вводная информация

Ваши коллеги попросили вас спроектировать инструмент PowerShell, который поможет им автоматизировать выполнение повторяющейся скучной задачи. У всех участников есть *опыт работы* с PowerShell, поэтому им нужна команда или набор команд, которые эту задачу автоматизируют.

Пароли для авторизации сервисов вы поленились менять. Многие были переключены на групповую управляемую учетную запись сервиса (group Managed Service Accounts, gMSA), поэтому вам это и не нужно, но у вас на многих сервисах (значительная часть которых выполняется на нескольких компьютерах в кластере) пароль не менялся годами. Нативная команда `Set-Service` здесь не поможет. Вам нужен инструмент, который бы позволил менять логин и пароль для одного сервиса на одном или нескольких компьютерах одновременно. Если какое-то устройство даст сбой, то вам нужно понимать, как устранить его вручную. Выводить информацию на экран и логировать ее в файл вполне нормально.

Этот инструмент должен работать на различных версиях Windows Server. Обычно не нужно прописывать такую информацию в скрипте, поэтому пароль можно передавать открытым текстом в командной строке в виде параметра. Вам нужно, чтобы команда в любом случае выводила что-то: имя каждого компьютера, уведомление об успехе/провале ее выполнения, измененный ею сервис и учетную запись, которую она сейчас использует (независимо от того, менялась она или нет). Обычно это нужно выводить на экран в виде простого HTML-отчета или в CSV-файл, который можно загрузить в Microsoft Excel.

8.6.2. Ваша задача

Вам нужно спроектировать инструмент, отвечающий бизнес-требованиям вашей команды. На данном этапе вы еще не пишете никакой код. При создании инструмента необходимо продумать, кто и как будет его использовать и каких действий от него будут ждать. При этом пользователем можете оказаться вы сами. Конечным результатом вашего проекта станет список примеров использования команды (выше мы описывали похожий), в котором показано, как инструмент будет решать поставленные командой бизнес-задачи. Вполне можно добавить в ваши

примеры существующие команды PowerShell, если они нужны для реализации заявленных требований.

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Прежде чем продолжать чтение, выполните это задание.

8.6.3. Наше решение

Мы сформулируем название команды как `Set-TMServiceLogon`. Сокращение `TM` будет означать *toolmaking* (создание инструментов), так как у нас нет конкретного названия компании. В качестве случаев использования предположим следующие:

```
Set-TMServiceLogon -ServiceName LOBApp
                   -NewPassword "P@ssw0rd"
                   -ComputerName SERVER1,SERVER2
                   -ErrorLogFilePath failed.txt
                   -Verbose
```

Мы предполагаем, что `-Verbose` будет генерировать экранные предупреждения о сбоях, а `-Error-LogFilePath` — записывать в файл имена вышедших из строя компьютеров. Обратите внимание: чтобы упростить восприятие этой спецификации, мы привели каждый параметр с новой строки. Команда не будет *исполняться* точно так, но это нормально — здесь важна ясность:

```
Set-TMServiceLogon -ServiceName OurService
                   -NewPassword "P@ssw0rd"
                   -NewUser "COMPANY\User"
                   -ComputerName SERVER1,SERVER2
```

Этот пример показывает, что `-ErrorLogFilePath` и `-Verbose` являются необязательными, как и `-NewUser`; если новый пользователь не указывается, то мы оставляем это свойство пустым. Вдобавок мы хотим продемонстрировать некоторые из гибких вариантов выполнения:

```
Get-Content servers.txt |
  Set-TMServiceLogon -ServiceName TheService -NewPassword "P@ssw0rd"
```

Это иллюстрирует нашу возможность получать из конвейера имена компьютеров.

Наконец, мы покажем еще две возможности:

```
Import-CSV tochange.csv | Set-TMServiceLogon | ConvertTo-HTML
```

Во-первых, эта команда показывает, что мы можем получать импортируемый CSV-файл, если в нем есть столбцы `ServiceName`, `NewPassword`, `ComputerName` и (необязательно) `NewUser`. Во-вторых, вывод этого инструмента можно передать стандартным командам PowerShell, таким как `ConvertTo-HTML`. В свою очередь, это говорит о том, что команды `Format-` и `Export-` тоже будут работать.

ОБЪЕМНЫЙ ПРОЕКТ НЕ ОБЯЗАТЕЛЬНО ОЗНАЧАЕТ МНОГО КОДА

Обычно изначально мы создаем проект, в котором есть все. Это не означает, что мы сразу начинаем реализовывать его целиком. В сфере программного обеспечения есть разница между общим *видением* и конкретным *выполнением*.

Мы говорим лишь о командах PowerShell, поэтому не будем начинать философскую дискуссию, но это важный момент. Вы можете не захотеть реализовывать в своей команде логирование ошибок прямо сейчас, и это нормально. Но это также не означает, что вы не можете *запланировать* внедрение этой функциональности позднее. Планирование (иными словами, *ви́дение* вашей программы в перспективе) означает, что в процессе написания кода, который вам нужен прямо сейчас, вы можете учитывать возможные нововведения.

Если на данный момент вы не планируете логировать информацию о компьютерах, на которых произошел сбой, но понимаете, что однажды реализуете такую возможность, то можете создать структуру кода, в которую будет проще добавить соответствующую функциональность в будущем. Иначе говоря, текущая схема выполнения необязательно должна отражать все видение целиком. Вы можете сформировать его сейчас, а затем поэтапно воплощать его элементы по мере появления времени и потребности.

ИТОГИ ГЛАВЫ

В этой главе мы познакомили вас с фундаментальными основами проектирования инструментов PowerShell, подчеркнув важность тщательного планирования и следования рекомендациям. Мы сделали акцент на том, что каждый инструмент должен выполнять одну конкретную задачу — это позволит избежать ловушки многофункциональных скриптов, которые усложняют написание кода, его отладку, тестирование и сопровождение. При этом мы призвали авторов скриптов предвидеть разнообразные ситуации и соответствующим образом оптимизировать скрипты. Сформулировав бизнес-требования в виде понятных примеров использования, мы показали вам структурированный подход к проектированию и заложили твердую основу для эффективной разработки скриптов PowerShell.

9 Предотвращение ошибок: начинайте с команды

Перед запуском редактора скриптов мы оказываемся в стартовом окне командной строки PowerShell. Оно является наиболее простой средой для тестирования и позволяет обеспечить корректность выполнения команд вашим инструментом. Намного легче отлаживать или исправлять одну команду из интерактивной консоли, нежели корректировать целый скрипт. Под одной командой мы подразумеваем выражение PowerShell — единый элемент, который можно вручную ввести в консоли, чтобы проверить корректность его синтаксиса. С этого момента вы будете замечать общую схему, которая будет выглядеть так: начинать с чего-то малого (с одной команды), делать элемент работоспособным, затем, используя его как основу, создавать новый элемент. Не пытайтесь написать весь скрипт за раз, иначе его будет практически невозможно отладить.

ТАК ЗАДУМАНО

PowerShell хороша в том числе тем, что вы можете открыть консоль, выполнить команды и сразу получить результаты (хорошие или плохие). Традиционно программистам приходилось долго писать код, компилировать его и, возможно, даже создавать под него тестовую программу. Преимущество мгновенности, которое предоставляет PowerShell, позволяет сократить рабочую нагрузку. Пользуйтесь им.

9.1. ЧТО ВАМ НУЖНО ВЫПОЛНЯТЬ

Если вы прочли предыдущую главу, то помните, что в сценарии примера вам была поставлена задача создать инструмент, который будет запрашивать следующую информацию:

- имя хост-компьютера;
- данные о производителе;
- модель;
- версию ОС и номер сборки;
- версию пакета обновлений;
- информацию об установленной оперативной памяти;
- тип процессора;
- количество сокетов процессора;
- общее количество ядер;
- свободное пространство на системном диске (обычно C, но не всегда).

Будет лучше, если вы запланируете использование общей информационной модели (Common Information Model, CIM), поскольку инструментарий Windows Management Instrumentation (WMI) из Windows PowerShell был удален. Вдобавок вам потребуется логировать информацию в текстовый файл и больше работать в отношении самого инструмента, но все это базовые элементы функциональности, с которыми предстоит разобраться.

В данный момент мы находимся на этапе проектирования. Цель данной главы — определить *подвижные части* скрипта. Да, скрипт будет содержать логику и прочие сопутствующие элементы, управляющие выполнением команд, но мы до этого момента еще не дошли. Для начала вам нужно выяснить, *какие* команды выполнять, *как* их выполнять и то, *правильный ли* синтаксис вы используете.

Говоря о целях, будем конкретны в отношении того, что нам нужно выяснить.

- Какая команда или команды необходимы?
- Какие классы данных нужно будет запрашивать?
- Какие изменения нужно вносить, чтобы опробовать оба протокола?
- Как логировать ошибки в текстовый файл?

Естественно, *многое* может пойти не так. Это начало процесса. Вы можете изначально выбрать ошибочную команду. Вы даже можете использовать такую команду, что тоже вполне нормально. Найдя подходящую команду, вы сделаете ошибочные предположения относительно результатов, которые она выдает, — и эти

предположения в дальнейшем приведут к появлению багов. Команда может вполне работать локально, а на удаленном компьютере сбоить — и вам нужно выяснить этот момент с самого начала. Она может работать только в определенных версиях Windows, и с этим тоже нужно будет разобраться. Все эти нюансы необходимо проработать *до того*, как вы запустите редактор скриптов. В мире было бы намного меньше багов, если бы разработчики, прежде чем писать код, сначала как следует тестировали все в интерактивной консоли.

ПРОЦЕСС ПОИСКА НУЖНОЙ КОМАНДЫ

Мы не станем разбирать весь процесс выяснения того, какую команду нужно выполнить, поскольку этой теме посвящена почти четверть книги «Изучаем PowerShell за месяц, занимаясь один час в день», и исходим из того, что вы либо читали ее, либо имеете соответствующие знания или опыт.

Но при этом *очень важно*, чтобы вы хорошо разбирались в процессе определения команд. Если каждый свой проект разработки инструмента вы будете вынуждены начинать с трехнедельных поисков в Google информации о том, какие команды вам потребуются, то это прямой путь к низкой эффективности и разочарованию. Честно говоря, прежде чем приступать к созданию инструментов, вам нужно получить больше опыта работы с PowerShell.

Еще один важный момент: вы должны уметь уверенно *экспериментировать* в командной строке. Читайте примеры из файлов справки и пробуйте. На занятиях и презентациях нас постоянно спрашивают: «А что, если я попробую использовать IP-адрес вместо имени компьютера?» Что тут сказать. Вы же сидите перед компьютером. Попробуйте — и сами увидите. Экспериментирование — первая половина процесса нашего обучения (вторую составляет всяческая «возня»), так что привыкайте экспериментировать и не бойтесь обрушить систему. Запустите виртуальную машину для тестов (VM) с Windows 11 или Windows Server 2022 — и действуйте.

ПРИМЕЧАНИЕ

Большинство разработчиков, работающих с .NET Framework, любят PowerShell, поскольку эта оболочка позволяет им интерактивно экспериментировать с .NET. Им не нужно писать огромную программу, компилировать ее и выполнять, чтобы понять, правильный ли код они создали, — они могут быстро выполнить команду в PowerShell, проверить свои предположения и писать код, уже будучи уверенными в его правильности. То же касается программистов на PowerShell: тестируйте все в консоли, добивайтесь необходимой работоспособности и только потом приступайте к написанию скрипта. Не беспокойтесь — позднее мы подробно поговорим об отладке скриптов с помощью Visual Studio Code.

9.2. ДЕЙСТВУЙТЕ ПОЭТАПНО И СЛЕДИТЕ ЗА РЕЗУЛЬТАТАМИ

Если консоль PowerShell у вас еще не открыта, то запустите ее (в том числе через Windows Terminal). Обратите внимание, что мы не сказали VS Code. Разберем хороший конкретный пример. Предположим, мы заходим в консоль PowerShell и выполняем такую команду:

```
Get-CimInstance -ClassName Win32_ComputerSystem
```

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Кстати, рекомендуем не откладывать, а сразу попробовать выполнить эти команды. Никакие из примеров текущей главы ничего не испортят в системе, но, выполнив их, вы получите полезный опыт.

Сработало? Удалось ли нам успешно протестировать команду тем способом, которым ее будет использовать наш скрипт? *Нет, не удалось!* Все дело в том, что наш скрипт должен будет выполнять ее *на удаленных компьютерах*, а не локально, как сделали мы. Это разные вещи, и выполнение команды на удаленном компьютере значительно усложняет процесс.

Ниже приводится более эффективный тест в консоли, который ближе к тому, как наш скрипт будет, скорее всего, выполняться (при условии, что SRV2 является действительным именем сервера в нашей среде, к которому у нас есть административный доступ, или же заменяет имя вашего компьютера):

```
Get-CimInstance -ClassName Win32_ComputerSystem -ComputerName SRV2
```

Задача — выявить подвижные части скрипта и проанализировать, *как* он будет их выполнять, чтобы в итоге протестировать эти элементы из консоли именно таким образом. Причем этот скрипт следует выполнить на нескольких компьютерах с разными версиями Windows. (Мы знаем, что некоторые из вас по каким-то причинам до сих пор используют Windows XP. Имейте в виду: если вы продолжаете работать с такими устаревшими системами, то скрипт не заработает.)

СОВЕТ

Сложно сказать, сколько раз на форумах PowerShell.org мы помогали разработчикам, которые запускали редактор скриптов и с ходу начинали что-то в нем программировать. Мы неизменно просим таких людей выполнять команду «из консоли», чтобы они могли четко понять, что именно делают не так. Вы сэкономите много времени, если не будете сильно спешить.

Но ваше участие не ограничивается одним только выполнением команд в надежде, что они сработают без ошибок. Вам нужно смотреть и на *результаты* работы этих команд. Например, вы ожидаете, что предыдущая команда вернет номер версии Windows. В таком случае вам нужно выполнить команду и посмотреть,

что произойдет. Многие команды только на экране выглядят отлично, поэтому мы всегда рекомендуем передавать их результат в `fl *` (`Format-List *`), чтобы можно было видеть полный и чистый вывод. Какие свойства вы увидите? Что они содержат? Знаете ли вы, что означает это содержимое? Есть ли в них различия между разными компьютерами, способные повлиять на скрипт, который вы собираетесь писать?

ВАЖНОСТЬ ТЕСТОВОЙ СРЕДЫ

Лучше всего, если у вас будет безопасное место для экспериментов.

Освоение использования команд PowerShell всегда сопряжено с пробами и ошибками, и сеть производственной среды вашей организации наверняка окажется не лучшим местом для этого. Виртуализация хороша тем, что с помощью таких продуктов, как VMware Workstation, VMware Fusion, VirtualBox, Parallels, Hyper-V, Azure Virtual Desktop и др., вы можете запускать несколько виртуальных компьютеров на одной машине в рамках собственной тестовой лаборатории. Кроме того, вы можете настраивать лаборатории (с разрешением) в виртуальной инфраструктуре организации, использовать облачные среды наподобие Microsoft Azure или Amazon Web Services (AWS) и т. д.

Иногда мы встречаем разработчиков, которые вынуждены изучать все это автономно, так как не могут позволить себе подписку на Azure или AWS. У них нет допуска к ресурсам организации, и, возможно, их домашние компьютеры не тянут запуск двух или трех виртуальных машин. К сожалению, это своего рода «плата за вход». Работа с PowerShell и создание инструментов подразумевают набор технологий бизнес-уровня, которые требуют соответствующих ресурсов. Освоить все это самостоятельно *может быть* сложно, но всегда есть доступный способ экспериментировать с подобными задачами.

Если у вас есть хорошее оборудование с 8–16 Гбайт оперативной памяти с достаточно большим свободным местом на диске, которое работает под управлением хотя бы Windows 10 или Windows Server 2016, то вы можете воспользоваться проектом AutomatedLab с сайта https://AutomatedLab.org, который упростит для вас запуск готовых тестовых сред.

9.3. ВЫПОЛНЕНИЕ КОМАНД И БОЛЕЕ ДЕТАЛЬНОЕ ИЗУЧЕНИЕ

Предположим, вы уже знаете, как выполнять команды PowerShell. Если же нет, то остановитесь и сначала прочитайте книгу «Изучаем PowerShell за месяц, занимаясь один час в день», поскольку она целиком посвящена поиску и выполнению подходящих команд. Наш посыл в том, чтобы вы, вручную выполняя команды

в командной строке, проводили тестирование и выясняли, как именно сделать так, чтобы инструмент выполнял то, что от него требуется.

Кроме того, конкретно в этом случае вам нужно убедиться, что вы умеете извлекать всю информацию по указанному списку, для чего потребуется не один класс CIM — вам как минимум не обойтись без `Win32_OperatingSystem` и `Win32_ComputerSystem`. Вдобавок вам придется с помощью одного из этих классов определять системный диск и извлекать его экземпляр `Win32_LogicalDisk`, чтобы уточнить объем его свободного пространства. Если вы читаете данную книгу, то вам желательно уже уметь проделывать все это, так что мы не будем проговаривать весь сопутствующий процесс.

Этап «поиска и тестирования» подразумевает не просто выяснение того, какие команды выполнять и какой синтаксис использовать. Как предполагалось в предыдущем разделе, мы уделяем время анализу вывода этих команд. В каком именно свойстве `Win32_ OperatingSystem` или `Win32_ComputerSystem` вы найдете системный диск? В каком формате он будет представлен: `C:`, `C` либо `C:/`? Или это число, например 0 или 1? С помощью какого значения вы сможете получить соответствующий экземпляр `Win32_LogicalDisk`? Идея состоит в том, чтобы вы могли ответить на *все* вопросы «Как мне сделать ...?» заранее, проверить свои ответы в консоли, а затем перейти к фактическому процессу скриптинга, используя рабочие команды, заметки и все остальное, что вам нужно для того, чтобы сделать все правильно с первого раза.

СОВЕТ

> Если вы не используете приложение для заметок, то заведите его. По мере того как вы начнете понимать, какие именно действия конкретно вам нужно совершать в скрипте, вам понадобится место, где вы сможете вести соответствующие записи. Вам зачастую понадобится копировать и вставлять какие-то фрагменты этих заметок, так что большая тетрадь с ручкой здесь не слишком помогут.

Вы будете использовать для запросов команду `Get-CimInstance`, и поскольку в итоге запрашивать потребуется множество классов, то и запросов понадобится множество. Окажется ли этот процесс медленным? Проверим. Кроме того, будет нелишним почитать справочные файлы — *целиком*, в том числе примеры. Это позволит найти способ создания и повторного использования постоянного соединения, ускоряющего множественные запросы. В связи с этим вы будете использовать команды `New-CimSession` и `Remove-CimSession` для создания и удаления этого постоянного соединения с каждым компьютером, позволяющего выполнять все запросы по одному каналу. Вдобавок вы должны будете обнаруживать ошибки, если с работой данного соединения возникнут проблемы. И если реализация этих задач вам незнакома, то самое время изучить раздел справки по `New-CimSession`, чтобы во всем разобраться.

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Мы говорим серьезно. Почитайте файл помощи. Прямо сейчас. Как вы реализуете создание и удаление постоянного соединения? Попробуйте это сделать — посмотрите, удастся ли заставить его работать, и запросите экземпляр Win32_LogicalDisk с удаленного компьютера или двух.

9.4. ПРОЦЕСС ВАЖЕН

Мы упоминали это в начале главы в качестве отступления, но вопрос настолько важный, что стоит повторить. Процесс поиска, тестирования и доработки нужных команд должен продолжаться на протяжении всей разработки. Мы видели, как студенты могут целый час писать код в VS Code, а затем пробуют его выполнить, но безуспешно. Естественно, это их сильно злит. Несмотря на все наши усилия, эти студенты игнорируют рекомендации проводить соответствующие исследования, тестировать и писать код *по ходу* разработки скрипта или инструмента. Цикл «поиск/тестирование/написание» — отличная причина использовать в VS Code расширение PowerShell. В редакторе удобнее находить нужные команды, вводить их и выполнять. Если что-то не работает, то тут можно внести исправления и повторить процесс. Как только вы все сделаете правильно, просто скопируйте и вставьте рабочий код в скрипт, после чего переходите к следующей его части. PowerShell дает мгновенную обратную связь. Пользуйтесь этим преимуществом.

9.5. РАЗБЕРИТЕСЬ, ЧТО ВАМ НУЖНО

У нас родился короткий нерадужный афоризм, который является горькой правдой: «PowerShell прост, Windows сложна». Суть в том, что многие из нас (благодаря тому, что прослойка GUI долгие годы отделена от операционной системы) не знают, что конкретно Windows делает внутри себя самой. Известно ли вам различие между разделом, диском, логическим диском и дисковым томом? Операционная система их знает, но не всегда открыто показывает эти различия в своем GUI. Если же вы сами их не знаете, то работать из PowerShell — формы контроля более низкого уровня, чем GUI, — будет трудно.

Эта тема *постоянно* всплывает на форумах PowerShell.org. Кто-нибудь просит помощи с блоком кода и прикрепляет к посту свое творение, которое напоминает программу C#, поскольку PowerShell в данном коде используется для доступа ко множеству необработанного материала .NET Framework. В этом случае вопрос не в PowerShell, а в особенностях .NET. Еще можно встретить вопрос наподобие: «Где найти список событий, связанных с подключением USB-устройств?» Это ситуативный вопрос. Он не имеет отношения к PowerShell, а подчеркивает другую сложность — работу с внутренней операционной системой.

Вот почему процесс нахождения/тестирования/написания кода так важен. В первую очередь вам нужно выяснить, *что* делать, а затем уже *как* это делать. Интерактивная консоль PowerShell — подходящее место для данного процесса, когда вы знаете, *что* делать и *как* собрать воедино то, что вы реализовали в скрипте с помощью редактора.

9.6. УПРАЖНЕНИЯ

В данной главе мы продолжим с того места, на котором остановились, выполняя упражнение предыдущей главы. Вам нужно попрактиковать подход, описанный в разделах выше. Благодаря этому вы сможете понять, как реализовать всю необходимую функциональность инструмента, начав работу с командной строки PowerShell. Если среди описанных задач есть *что-то*, что вы не умеете реализовывать, то разберитесь с этим *до* того, как доберетесь до конца текущей главы.

9.6.1. Вводная информация

Напоминаем, что вы разработали инструмент, который будет менять логины и пароли сервиса. Вы не сможете использовать для этого команду `Set-Service` (она не позволяет изменять указанную информацию). Вам потребуется задействовать общую информационную модель (CIM).

9.6.2. Ваша задача

Ваша основная задача — найти класс CIM, который позволит менять логин и пароль сервиса. Лучшим средством для этого, пожалуй, станет поисковый движок, и мы дадим подсказку: имя класса начинается с `Win32_`.

Кроме того, вам необходимо убедиться, что вы можете использовать этот класс для выполнения задачи. Потребуется *вызывать (invoke)* что-то в общей информационной модели: фоновую интеллектуальную службу передачи (Background Intelligent Transfer Service, BITS) или программу буферизации печати (Print Spooler). Экспериментирование с этими компонентами не приведет к сбою Windows, что очень хорошо. Но если вы работаете не на лабораторном компьютере, то помните, что BITS обеспечивает работу Windows Update и прочих важных программных компонентов. Когда вы закончите экспериментировать с BITS, сбросьте ее настройки так, чтобы она авторизовывалась как LocalSystem без установленного пароля.

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Прежде чем продолжать, выполните описанное задание.

9.6.3. Наше решение

Мы выяснили, что решить задачу сможет класс `Win32_Service`. Честно говоря, узнали мы это на странице Microsoft.com (http://mng.bz/1noL), которую нашли среди результатов такого запроса к Google: `change windows service password`.

Кроме того, мы выполнили в PowerShell `Get-Command -verb invoke`, учитывая, что в лабораторном задании часть *invoke* была указана явно. Мы нашли команду `Invoke-CimMethod`, которую и будем использовать. Вдобавок мы прочли ее раздел справки и составили такую команду для изменения стартового имени пользователя и пароля сервиса BITS:

```
$CimMethodParameters = @{
    Query = "SELECT * FROM Win32_Service WHERE Name='BITS'"
    Method = "Change"
    Arguments = @{
        'StartName'     = 'DOMAIN\User'
        'StartPassword' = 'P@ssw0rd'
    }
    ComputerName = $env:computername
}
Invoke-CimMethod @CimMethodParameters
```

Будем честны: продумывание этой команды потребовало проведения некоторых экспериментов и поисков (опять же в Google). Мы решили использовать `-Query`, поскольку нам нужен конкретный экземпляр `Win32_ Service`, а не *все* службы компьютера. Кроме того, мы обратили внимание на параметр `-Computer-Name`, который должен пригодиться позже, когда мы будем обращаться к другим машинам. Чтобы обеспечить его правильное использование, мы задействуем для имени локального компьютера переменную среды. Это должно гарантировать правильность полной синтаксической конструкции, которую мы в итоге встроим в наш инструмент.

В нашей тестовой среде `DOMAIN` является валидным именем, но вам, очевидно, потребуется использовать подходящее имя пользователя в формате *DOMAIN\USERNAME*. Мы заметили, что команда вернула объект. При этом `ReturnValue` оказалось `0` в случае успеха и `22` в случае передачи недействительного имени пользователя. Веб-страница метода `Change`, на который мы давали ссылку ранее, содержит все корректные коды возврата. Мы смогли добавить этот возвращаемый объект в переменную, обеспечив в процессе написания инструмента успешность связи с каждым компьютером.

Кстати, мы не забыли сбросить настройки службы:

```
Invoke-CimMethod -Query "SELECT * FROM Win32_Service WHERE Name='BITS'"
                 -Method Change
                 -Arguments @{'StartName'='LocalSystem'}
```

Реализуйте этот процесс внимательно и вдумчиво. Вам нужно начать вырабатывать своеобразную мышечную память на создание инструментов PowerShell.

ИТОГИ ГЛАВЫ

В этой главе мы подчеркнули, что важно начинать с простого и тщательно тестировать команды в PowerShell, прежде чем переходить к их скриптингу. Создавая одиночные команды в командной строке PowerShell, вы сможете обеспечить их корректность и снизить вероятность багов в скриптах. Кроме того, мы уделили внимание значимым элементам функциональности, необходимым в каждом проекте разработки инструмента, — запросу системной информации и логированию данных в текстовый файл.

Процесс поиска подходящих команд крайне важен, и мы призываем вас тщательно его освоить. Экспериментирование в командной строке, опробование различных сценариев и понимание вывода команд — все это очень значимые шаги процесса поиска. Кроме того, налаживание тестовой среды, с помощью виртуализации или облачных решений, позволит вам безопасно экспериментировать, не влияя на продакшен-системы.

В этой главе мы акцентировали ваше внимание на итеративной природе создания скриптов: поиск, тестирование и доработка команд должны продолжаться в ходе всего процесса. Благодаря преимуществам PowerShell, которые заключаются в ее интерактивности и выдаче мгновенной обратной связи, разработчики могут создавать надежные скрипты и избегать распространенных затруднений.

Наконец, мы привели упражнения для отработки описанных в главе принципов, таких как поиск классов CIM и тестирование команд, чтобы закрепить ваше понимание и помочь выработать важные навыки разработки инструментов PowerShell.

10 Создание простой функции и модуля скрипта

Помните инструмент, который мы создали в главе 8? Сейчас вам нужно запустить Visual Studio Code и открыть тот файл `.ps1`. В текущей главе мы возьмем этот инструмент и превратим его в средство, которым смогут пользоваться другие. Важно понимать: здесь мы не покажем вам, как создать целый инструмент или решить всю бизнес-задачу, поставленную в главе 8. Мы будем разбирать этапы, так как хотим подробно показать процесс создания инструмента.

10.1. НАЧИНАЕМ С ПРОСТОЙ ФУНКЦИИ

Простые функции были в PowerShell с самого начала и являются одним из множества типов *команд*, которые понимает эта оболочка (к другим относятся командлеты, приложения и пр.). Функции представляют собой отличную рабочую единицу для создания инструментов при условии, что вы следуете базовому принципу *сохранения их узкой специализации и автономности*. Мы уже писали, что ваши функции должны быть узкоспециализированными, то есть выполнять только что-то одно. *Автономность* в этом случае означает, что функция должна существовать в собственном маленьком мирке, становясь подобной черному ящику. В практическом смысле это означает две вещи.

- Информация для функции должна браться только из объявленных входных параметров. Естественно, некоторые функции могут находить данные где-то еще, например, в базе данных или реестре. И это нормально, если такова задача функции. Но они не должны использовать внешние переменные или источники, за исключением внутренних элементов наподобие PSDrives для файловой системы или переменных среды. Нужно, чтобы они были максимально автономными.

- Вывод функции должен предназначаться *только* для конвейера PowerShell. Так, создание файла на диске, обновление базы данных и пр. — не *вывод*, это *действия*. Функция может выполнять одно из таких действий, *если в этом и состоит ее задача*.

ПРОЕКТИРОВАНИЕ ВЫВОДА ФУНКЦИИ

На этот момент стоит обратить внимание, поскольку это одна из тех вещей, которые многие понимают неправильно. `Write-Output` — *стандартная команда* PowerShell. То есть если вы передаете оболочке какое-то обособленное выражение, то она использует `Write-Output`. Приведем пример. Зайдите в оболочку, введите `5+5` и нажмите Enter. На экране отобразится результат. По факту оболочка выполнила что-то вроде `Write-Output (5+5)` и отправила результат в конвейер (поскольку именно это делает команда `Write-Output`). В конвейере ничего другого не было, поэтому управление перешло к системе форматирования, которая вывела на экране то, что в нем находилось (надеемся, что 10).

Это означает, что ваш скрипт никогда не должен использовать `Write-Output` ни для чего, кроме вывода предполагаемого результата. И он всегда должен быть либо ничем, если такое возможно, либо какими-то структурированными данными — объектами — которые можно передать другой команде.

Команду `Write-Output` никогда не нужно использовать для коротких сообщений статуса, которые сообщают о том, что делает скрипт. Она никогда не должна выводить открытый, предварительно отформатированный текст (если только это не вывод или результат вашей команды). Мы будем обсуждать этот процесс проектирования вывода в течение нескольких глав. Сейчас же просто помните: *вывод важен*, и внутренняя структура PowerShell имеет определенные ожидания в отношении его формы и содержания.

10.1.1. Проектирование входных параметров

Если еще раз просмотреть структуру инструмента, то какая информация потребуется функции? Примеры использования уже дают четкое представление о том, какие параметры вам нужно создать. Собственно, это одна из причин, по которой вы создаете такие примеры в качестве основного результата проектирования. А теперь создадим базовые версии этих параметров:

```
function Get-MachineInfo {
 Param(
  [string[]]$ComputerName,
  [string]$LogFailuresToPath,
  [string]$Protocol = "wsman",
  [switch]$ProtocolFallback
 )}
```

Обратите внимание, как вдумчиво мы подходим к форматированию кода. Чтобы сэкономить место на странице книги, мы делаем небольшие отступы в коде внутри функции и блока `Param()`. Вы же обычно будете делать отступы в четыре пробела (в большинстве редакторов именно такой отступ делается при нажатии Tab). Форматируйте код *как следует*. Небрежное форматирование нередко является признаком того, что в коде есть баги и его будет сложно отлаживать.

СОВЕТ

Форматирование очень важно! VS Code будет автоматически форматировать ваш документ PowerShell. Просто введите на панели команд словосочетание Format Document.

В блоке `Param()` мы объявляем четыре параметра. Это простые объявления, и вы будете использовать их в последующих главах. Сейчас же мы просим обратить внимание на следующие моменты.

- Типы данных заключаются в квадратные скобки. К стандартным типам относятся `[string]`, `[int]` и `[datetime]`. Вы заметите здесь `[switch]`, который определяет параметр, содержащий `$True`, если команда выполнена с параметром, и `$False` — если нет.
- Внутри функции параметры становятся переменными, то есть имена предваряются символом `$`. Один важный момент: не пытайтесь создавать имя параметра, содержащее пробелы.
- В разделе `Param()` запятая отделяет каждый параметр от текста. Вам *необязательно* помещать их по одному на строку, как это сделали мы, но когда вы начнете использовать их в дальнейшем, то будет намного проще читать код, если разделить их именно так.
- Параметр `-ComputerName` будет получать или не получать значения в виде *массива*, о чем говорит `[string[]]`.
- Переменная `$Protocol` будет содержать Wsman, если только кто-нибудь явно не укажет что-то иное. Вы не ограничиваете выбор пользователя только Wsman или Dcom, но в итоге так и сделаете.

10.1.2. Написание кода

Теперь добавим в листинг 10.1 кое-какой функциональный код. Опять же, на этом миссия инструмента не заканчивается — вы только начали, и мы хотим провести вас через каждый этап. И вновь мы призываем вас уделить внимание процессу, а не обязательно результату. Все наши примеры приведены в целях обучения и могут не являться наилучшим способом реализации задачи.

Листинг 10.1. Базовый функциональный код

```
# Возможно, вам придется запустить Enable-PSRemoting или Winrm QC,
# чтобы включить удаленное взаимодействие
function Get-MachineInfo {
```

```
Param(
 [string[]]$ComputerName,
 [string]$LogFailuresToPath,
 [string]$Protocol = "Wsman",
 [switch]$ProtocolFallback
)
foreach ($computer in $computername) {
 # Установка протокола сеанса
 if ($protocol -eq 'Dcom') {
  $option = New-CimSessionOption -Protocol Dcom
 } else {
  $option = New-CimSessionOption -Protocol Wsman
 }
 # Открытие сеанса
 $session = New-CimSession -ComputerName $computer -SessionOption $option
 # Запрос данных
 $os = Get-CimInstance -ClassName Win32_OperatingSystem -CimSession $session
 # Закрытие сеанса
 $session | Remove-CimSession
 # Вывод данных
 # TODO
} #foreach
} #function
```

Обрабатывает каждый компьютер

Конструкция If

Закрывает функцию

СОВЕТ

Обратите внимание, что мы добавили к закрывающей скобке функции комментарий #function. Желательно выработать привычку так делать, поскольку она помогает не забыть, какую конструкцию закрывает скобка. Кроме того, вам следует освоить команды для редактора скриптов, позволяющие находить парные скобки. Если ваш редактор поддерживает сворачивание блоков кода, то такая функция тоже будет полезной. Разработчики сталкиваются с немалым количеством багов из-за отсутствующих или неверно поставленных закрывающих скобок.

Конструкция `If` поможет предотвратить проблемы, если кто-нибудь укажет в параметре `-Protocol` недействительный протокол. Если указать Dcom, то вы настроите сеанс объектной модели распределенных компонентов (Distributed Component Object Model, DCOM). Если же будет указано что-то еще, то запустится сеанс управления веб-сервисами (Web Services Management, WSMan).

Вы запрашиваете только один из классов, которые в итоге нужно будет запросить. Смысл в том, чтобы запустить, протестировать и затем, когда все заработает, добавить другие. Этот процесс слегка увеличит время разработки, но вместе с тем позволит защитить код от серьезных ошибок. Если вы попутно проводите тесты, то в случае обнаружения бага отлаживать придется наверняка всего несколько строк.

10.1.3. Проектирование вывода

Наконец, хорошо, если вы заставите команду выводить что-либо, как показано в листинге 10.2.

Листинг 10.2. Добавление вывода

```
function Get-MachineInfo {
 Param(
  [string[]]$ComputerName,
  [string]$LogFailuresToPath,
  [string]$Protocol = "Wsman",
  [switch]$ProtocolFallback
 )
 foreach ($computer in $computername) {
  # Установка протокола сеанса
  if ($protocol -eq 'Dcom') {
   $option = New-CimSessionOption -Protocol Dcom
  } else {
   $option = New-CimSessionOption -Protocol Wsman
  }
  $session = New-CimSession -ComputerName $computer -SessionOption $option
  # Запрос данных
  $os = Get-CimInstance -ClassName Win32_OperatingSystem -CimSession
➥ $session
  # Закрытие сеанса
  $session | Remove-CimSession
  # Вывод данных
  $os | Select-Object -Prop @{n='ComputerName';e={$computer}},
                            Version,ServicePackMajorVersion
 } #foreach
} #function
```

Подключает сеанс

Запрашивает данные ОС

Записывает вывод с помощью Select-Object

Это не слишком сложный вывод: вы просто берете имя компьютера и два свойства операционной системы, которые указали в проекте. В итоге по мере добавления запросов и внедрения их свойств вывод будет усложняться.

ПРИМЕЧАНИЕ

Обратите внимание: вы выводите структуру данных — объект — в конвейер. Вы не использовали явно команду Write-Output, но неявно она там, поскольку вы не присваивали результаты выражения переменной, как и не передавали явно объект куда-то еще. Вы передали $os в Select-Object, и результат этого выражения попадет в конвейер.

10.2. СОЗДАНИЕ МОДУЛЯ СКРИПТА

На последнем этапе мы сохраним весь этот код в виде *модуля скрипта*. Модули можно сохранять в разных местах, но лучше всего — в их основном каталоге. Он указан в переменной среды `PSModulePath` (`$env:PSModulePath -split ";"`). В Windows PowerShell и более поздних версиях этот путь по умолчанию содержит `C:\Program Files\WindowsPowerShell\Modules`, а в PowerShell v7 и выше — `C:\Program Files\PowerShell\7\Modules`, так что именно здесь вы будете создавать модули в подпапке `ScriptingMOL`. Если говорить конкретнее, то сохраните ее как `ScriptingMOL.psm1`.

Обратите внимание: *имя подпапки* и *имя файла* должны совпадать, чтобы PowerShell могла найти модуль и автоматически загрузить его по первому требованию.

СОВЕТ

По факту, когда мы просто экспериментируем, мы обычно сохраняем модуль куда-нибудь в папку Documents. В таком случае он воспринимается своего рода личным. Каталог в Program Files мы обычно оставляем для модулей, которые уже готовы к производственной среде. В данном же случае мы хотим, чтобы вы привыкли к существованию именно этого расположения, куда «реальные» модули отправляются после того, как будут завершены.

Мы добавили наш модуль в его текущем виде на данный момент в примеры кода для этой книги (упорядоченные по главам и доступные для скачивания на странице http://mng.bz/rjgE). Для загрузки модуля вам нужно вручную выполнить `Import-Module` и указать в файле `.psm1` на вашем компьютере полный путь из извлеченного ZIP-файла. Причина в том, что примеры кода содержат несколько версий модуля и вы не устанавливаете их в одно место, где PowerShell ищет модули автоматически. Предоставление полного пути к `Import-Module` гарантирует, что вы загрузите для своих целей правильную версию модуля. Когда закончите, используйте `Remove-Module` (или закройте и заново откройте консоль), чтобы провести очистку перед загрузкой следующей версии того же модуля. Кроме того, можете сопроводить `Import-Module` параметром `-Force`, чтобы принудительно переписать существующие команды.

СОВЕТ

В зависимости от того, как вы скачаете ZIP-файл, его заголовок может содержать отметку о том, что он получен из Интернета. Опять же, в зависимости от того, как вы его распакуете, отмеченными могут оказаться и отдельные файлы. Многие политики выполнения PowerShell не позволяют запускать скачанные файлы. В более новых версиях оболочки есть команда Unblock-File, которая удаляет флаг downloaded, делая скрипт доступным для выполнения (или для загрузки в качестве модуля).

10.3. ПРЕДВАРИТЕЛЬНАЯ ПРОВЕРКА

Прежде чем тестировать команду (особенно если вы планируете выполнять ее сами, следуя инструкциям), вам нужно кое-что проверить.

- Убедитесь, что в заголовке вашего окна PowerShell всегда указано Administrator. Если это не так, то запустите оболочку от имени администратора, щелкнув правой кнопкой на ее значке в панели задач и выбрав параметр As Administrator.
- Выполните `Get-ExecutionPolicy`; результатом должен быть `RemoteSigned`, `Bypass` или `Unrestricted`. Если это не так, то используйте `Set-ExecutionPolicy`, чтобы

изменить настройку на одно из этих значений (мы используем `Bypass` и в главе 7 описывали, почему вы можете выбрать ту или другую политику).

- Выполните `Get-CimInstance win32_service -computername localhost`, чтобы настроить и привести в готовность общую информационную модель (Common Information Model, CIM).

Если что-то из перечисленного в вашей системе не допускается, то *остановитесь*. Вам нужно это исправить. Мы говорили о первых двух пунктах. Последний может стать проблемой только в старых версиях Windows (до Windows 8), где CIM не активирована по умолчанию. Обычно это можно исправить, установив более свежую версию PowerShell (v3 или выше), после чего потребуется выполнить перезагрузку. При этом будьте уверены: если вы не заставите работать эти три элемента, то ничто другое из данной книги тоже работать не будет.

10.4. ВЫПОЛНЕНИЕ КОМАНДЫ

Теперь сам тест. Для начала закройте окно PowerShell, чтобы выполнить тест в чистой среде. Затем откройте новое (обязательно как администратор) и выполните следующую команду:

```
Get-MachineInfo -ComputerName localhost
```

> **НИКАКИХ КОРОТКИХ ПУТЕЙ**
>
> Мы предполагаем, что вы следовали инструкции и создавали модуль с нуля, а не просто выполняете тесты с помощью примера кода, который мы предоставляем. Как мы уже объясняли, выполнение `Get-MachineInfo` не будет работать автоматически, пока вы не создадите файл `.psm1` в правильном особом месте, где его будет искать PowerShell. Наши примеры кода *не будут* находиться в этом особом месте.
>
> Не пытайтесь пойти короткими путями. Следуйте инструкциям и пишите код. Это лучший способ научиться.

Если вы получите сообщение об ошибке `The client cannot connect to the destination specified in the request`, это означает, что у вас не подключено удаленное соединение. Чтобы это исправить, выполните команду `WinrQuickConfig` или `Enable-PSRemoting`, затем попробуйте еще раз.

Будет лучше всего, если вы получите при выполнении команд какой-нибудь вывод. У вас должна быть возможность ввести `Get-Machi`, нажать Tab, набрать пробел, ввести `-Comp`, нажать Tab, после чего ввести пробел и `localhost`. Если автозаполнение

с помощью Tab не работает, то перепроверьте скрипт на предмет правильности имен, опечаток в коде (в VS Code обозначаются красными волнистыми линиями) и т. д. Кроме того, убедитесь, что использовали путь, который указан в переменной окружения вашей машины `PSModulePath`:

```
$env:PSModulePath
```

Если команда выполняется без каких-либо проблем, то все хорошо. Убедитесь, что понимаете, *почему* каждая строка кода есть в команде, и можете объяснить причину каждого шага, выполненного к текущему моменту.

Если вы *внесете в модуль хоть какие-то изменения*, то вам следует знать, что PowerShell их не увидит. Причина в том, что оболочка загрузила модуль в память, когда вы впервые выполнили команду. После этого она выполняется полностью из памяти и не загружается повторно с диска. Итак, если вы внесете изменения в код, то вам нужно выполнить кое-какие из нижеперечисленных действий.

- Закрыть консоль PowerShell, где выполнялось тестирование, и открыть новое. Это гарантированный способ каждый раз начинать с чистого листа.
- Выгрузить свой модуль, а затем еще раз выполнить команду, чтобы его загрузить. В данном случае это означает выполнение `Remove-Module ScriptingMOL`, поскольку `ScriptingMOL` — имя модуля (согласно определению имени подпапки и имени файла `.psd1`).
- Попробовать вручную заставить PowerShell заново импортировать модуль с помощью команды `Import-Module ScriptingMOL -force`.

Кроме того, вы заметите, что мы обычно тестируем наши команды в стандартном окне консоли PowerShell, несмотря на то что выполняем разработку в чем-то вроде VS Code. Дело в том, что среды разработки иногда предполагают несколько иной способ выполнения скриптов, и окно консоли представляет стандартный метод выполнения скрипта в производственной среде. Консоль представляет производственную среду, поэтому в ней мы и проводим тестирование.

10.5. УПРАЖНЕНИЯ

Вернемся к инструменту, который мы просили вас спроектировать в главе 8. Пора начать писать его код.

10.5.1. Вводная информация

Напомним, что вы сформулировали имя команды как `Set-TMServiceLogon`. *TM* означает *toolmaking* (создание инструментов), поскольку у вас нет конкретного

имени компании или организации, которое можно было бы использовать. Далее вы спроектируете следующие случаи использования:

```
Set-TMServiceLogon -ServiceName LOBApp
                   -NewPassword "P@ssw0rd"
                   -ComputerName SERVER1,SERVER2
                   -ErrorLogFilePath failed.txt
                   -Verbose
```

Нам нужно, чтобы `-Verbose` генерировала выводимые на экран предупреждения о сбоях, а `-ErrorLogFilePath` записывала имена соответствующих компьютеров в файл. Обратите внимание: мы поместили каждый параметр на отдельную строку, чтобы спецификацию было проще читать. Команда не будет *выполняться* именно так, но это нормально — в данном случае для нас важна ясность.

Пример ниже показывает, что `ErrorLogFilePath` и `-Verbose` являются необязательными, как и`-NewUser`: если новый пользователь не указан, то данное свойство остается пустым:

```
Set-TMServiceLogon -ServiceName OurService
                   -NewPassword "P@ssw0rd"
                   -NewUser "COMPANY\User"
                   -ComputerName SERVER1,SERVER2
```

Кроме того, мы хотим показать вам кое-какие параметры выполнения:

```
Get-Content servers.txt |
➥ Set-TMServiceLogon -ServiceName TheService -NewPassword "P@ssw0rd"
```

В этой строке показана возможность получения имен компьютеров из конвейера. Наконец, здесь есть еще два момента:

```
Import-CSV tochange.csv | Set-TMServiceLogon | ConvertTo-HTML
```

Во-первых, эта строка показывает, что вы можете получать импортированный CSV-файл при условии, что в нем есть столбцы `ServiceName`, `NewPassword`, `ComputerName` и (необязательно) `NewUser`. Во-вторых, вывод могут получить стандартные команды PowerShell наподобие `ConvertTo-HTML`. Это, в свою очередь, говорит о том, что команды `Format-` и `Export-` тоже будут работать.

10.5.2. Ваша задача

Создайте базовую функцию `Set-TMServiceLogon`. Укажите все параметры, прописанные в проекте, хотя прямо сейчас использовать вы сможете не все из них. Напишите такой код, чтобы при получении имени компьютера, имени сервиса и нового пароля функция могла изменить текущий пароль. Если указывается новое

имя пользователя, то оно тоже должно устанавливаться. Вы будете использовать как конструкцию `If`, так и конструкцию `ForEach`. Прямо сейчас убедитесь, что эти два примера будут работать:

```
Set-TMServiceLogon -ServiceName OurService
                   -NewPassword "P@ssw0rd"
                   -NewUser "COMPANY\User"
                   -ComputerName SERVER1,SERVER2
Set-TMServiceLogon -ServiceName OurService
                   -NewPassword "P@ssw0rd"
                   -ComputerName SERVER1,SERVER2
```

Создайте в модуле скрипта функцию `MolTools`. Протестируйте ее в отношении службы BITS на локальном хосте. Помните: все необходимые команды следует предварительно выполнять в лабораторной среде, чтобы определить правильный синтаксис. Сейчас же предположим, что вам нужно реализовать только соединение WSMan (CIM). Кроме того, пока не беспокойтесь о логировании или прочих указанных в проекте возможностях.

Держите в памяти материал прошлой главы, касающийся вывода `Invoke-CimMethod`. Пока вполне допустимо выводить имя компьютера и его код возврата. Вы можете создать этот вывод с помощью `Select-Object` и пользовательских свойств, как делали в примере `Get-MachineInfo`. Позднее вы будете работать над сближением результата со своей спецификацией проекта.

Протестируйте команду в консоли PowerShell, а не в ISE или VS Code и помните о наших предостережениях, касающихся выгрузки модуля после внесения изменений.

10.5.3. Наше решение

Вот наше решение (листинг 10.3), с которым вы можете сравнить свое. Незначительные различия не должны вызывать особого беспокойства, если ваша команда исправно работает.

Листинг 10.3. Наше решение

```
function Set-TMServiceLogon {
    Param(
        [string]$ServiceName,
        [string[]]$ComputerName,
        [string]$NewPassword,
        [string]$NewUser,
        [string]$ErrorLogFilePath
    )
    ForEach ($computer in $ComputerName) {
        $option = New-CimSessionOption -Protocol Wsman
        $session = New-CimSession -SessionOption $option `
                                  -ComputerName $Computer
        If ($PSBoundParameters.ContainsKey('NewUser')) {
```

Использует PSBoundParameters

```
            $args = @{'StartName'=$NewUser;
                      'StartPassword'=$NewPassword}
        } Else {
            $args = @{'StartPassword'=$NewPassword}
        }
        Invoke-CimMethod -ComputerName $computer `
                         -MethodName Change `
                         -Query "SELECT * FROM Win32_Service WHERE Name =
        ➥ '$ServiceName'" `
                         -Arguments $args |
        Select-Object -Property @{n='ComputerName';e={$computer}},
                                @{n='Result';e={$_.ReturnValue}}
        $session | Remove-CimSession
    } #foreach
} #function
```

Запрос CIM

Результат метода, переданный в Select-Object

Обратите внимание, что мы не добавляли параметр `Verbose`. Это намеренное решение, и в следующих двух главах вы узнаете его причину.

Кроме того, мы использовали `$PSBoundParameters` для проверки того, был ли указан параметр `NewUser`. Это своеобразный трюк, который вы вряд ли знаете: вы могли уточнить, содержит ли `$NewUser` что-либо, с помощью `If ($NewUser -ne "")` или `if (-Not $NewUser)`, и это нормально. `$PSBoundParameters` — это хеш-таблица, содержащая все параметры, с которыми была выполнена команда. Она создается автоматически. Для этого не нужно ничего делать. Используя ее метод `ContainsKey()`, мы можем видеть, находится ли среди используемых параметров `NewUser`. Этот способ считается более правильным для проверки того, используется ли параметр. Но вы можете видеть, как строится конструкция `If` для создания хеш-таблицы с аргументами CIM: только с паролем или же с паролем и именем пользователя. Проблемы возникают, когда кто-нибудь не указывает новый пароль, но по мере развития функции мы эту вероятность исключаем.

В нашем запросе CIM (который в книге может быть представлен в сокращенном виде; загляните в примеры кода, чтобы увидеть его целиком) для вставки в запрос `$Service-Name` используется прием с двойными кавычками. Мы передаем результат `Invoke-CimMethod` — который, как вы узнали в предыдущей главе, возвращает объект со свойством `ReturnValue` — в `Select-Object`, чтобы можно было выстроить наш вывод. И для этого мы тоже создали манифест:

```
New-ModuleManifest -Path TMTools.psd1
                   -RootModule .\TMTools.psm1
                   -FunctionsToExport Set-TMServiceLogon
                   -ModuleVersion 1.0.0.0
```

К этому моменту мы добавили наше решение в примеры кода, которые можно найти в папке соответствующей главы книги (http://mng.bz/rjgE). Для загрузки модуля вам потребуется вручную выполнить `Import-Module` и передать полный путь к файлу `.psd1` на вашем компьютере. В примерах кода для этой главы

в качестве имени модуля мы использовали `MoLTools-Prelim`, чтобы избежать конфликта с «реальным» модулем `MoLTools`, который создадите вы сами. Наконец, когда протестируете функцию, сбросьте настройки сервиса BITS, как делали это в предыдущей главе.

ИТОГИ ГЛАВЫ

В текущей главе вы узнали, как превратить скрипт PowerShell в повторно используемый инструмент путем создания базовой функции и модуля скрипта. Начав с написания простой функции, вы узнали о важности узконаправленности и автономности функций, гарантирующих, что они используют только объявленные входные параметры и выводят данные в конвейер PowerShell. Особое внимание мы уделили проектированию входных параметров и выходных структур данных, а также написанию функционального кода и его правильному форматированию. Помимо этого, вы научились создавать модуль скрипта, сохранять его в подходящий каталог и делать так, чтобы PowerShell узнавала и загружала его. Глава завершилась упражнением, в котором вы создавали функцию `Set-TMServiceLogon`, указав в ней параметры для изменения учетных данных сервиса, и тестировали ее в отношении службы BITS на локальном хосте. В течение всего этого процесса мы призывали вас внимательно относиться к практикам написания кода и стараться понимать среду PowerShell, что позволяет создавать эффективные инструменты.

11 Расширенные функции

Прежде чем начать писать расширенные функции, сосредоточимся на блоке `Param()` примера функции из этой главы и обсудим кое-какие действия, которые можно с ним выполнять.

11.1. НЕСКОЛЬКО СЛОВ О CMDLETBINDING И ОБЩИХ ПАРАМЕТРАХ

В чем разница между простой функцией и расширенной? Вас может удивить, что она всего в одной строке кода: атрибуте `CmdletBinding()`. Он добавляет широкую функциональность. Чтобы наглядно показать разницу, начнем с простой функции:

```
function test {
    Param(
    [string]$ComputerName
    )
}
```

На этом все — никакого исполняемого кода. А теперь попросим PowerShell помочь с этой функцией:

```
PS C:\> help test
NAME
    test
SYNTAX
    test [[-ComputerName] <string>]
ALIASES
    None
```

Как и ожидалось, из-за полного отсутствия в функции каких-либо действий PowerShell предоставляет всю информацию, какую может. А теперь внесем в код всего одно изменение:

```
function test {
    [CmdletBinding()]
    Param(
         [string]$ComputerName
    )
}
```

Снова просим помощи:

```
PS C:\> help test
NAME
    test
SYNTAX
    test [[-ComputerName] <string>]  [<CommonParameters>]
ALIASES
    None
```

Добавление атрибута `[CmdletBinding()]` привело к тому, что PowerShell внесла в функцию стандартные параметры. Если вы прочитаете файл справки `about_CommonParameters`, то узнаете, что этот набор параметров поддерживают *все* команды PowerShell. Количество таких параметров с каждой версией оболочки увеличивалось, и на данный момент их 11. Авторам командлетов ничего не нужно делать, чтобы они работали, — PowerShell берет все действия на себя. И поскольку мы добавили `[CmdletBinding()]`, то функция будет поддерживать и все эти популярные параметры. Некоторые из самых лучших параметров (доступность которых отличается в зависимости от версии PowerShell) описаны ниже.

- `-Verbose` — активирует вывод `Write-Verbose` в вашей функции, переопределяя глобальную переменную `$VerbosePreference`.
- `-Debug` — позволяет использовать в функции `Write-Debug`.
- `-ErrorAction` — изменяет поведение функции в случае ошибки и переопределяет глобальную переменную `$ErrorActionPreference`.
- `-ErrorVariable` — позволяет указать имя переменной, в которой PowerShell будет отражать все генерируемые функцией ошибки.
- `-InformationAction`— переопределяет глобальную переменную `$InformationPreference` и активирует вывод `Write-Information`. Этот параметр был добавлен в PowerShell v5.
- `-InformationVariable` — указывает переменную, в которой будет отражаться вывод `Write-Information`. Этот параметр тоже появился в PowerShell v5.
- `-OutVariable` — указывает переменную, в которой PowerShell будет помещать копии вывода вашей функции при их отправке в основной конвейер.

- `-PipelineVariable` — указывает переменную, в которой PowerShell будет сохранять копию текущего элемента конвейера. Об этом мы поговорим в одной из следующих глав.

Есть и другие параметры, большинство из которых мы разберем в следующих главах.

11.1.1. Получение ввода конвейера

Если вы помните изначальный проект нашего примера инструмента, то мы указывали на необходимость получения вывода из конвейера. Для этого нужно изменить параметры и код функции. В предыдущей главе мы дошли до определенного этапа. В листинге 11.1 показано, откуда мы начинаем на этот раз, а в листинге 11.2 приводится измененная функция.

Листинг 11.1. Изначальная функция Get-MachineInfo

```
function Get-MachineInfo {
    Param(
        [string[]]$ComputerName,
        [string]$LogFailuresToPath,
        [string]$Protocol = "Wsman",
        [switch]$ProtocolFallback
    )
    foreach ($computer in $computername) {
        if ($protocol -eq 'Dcom') {
            $option = New-CimSessionOption -Protocol Dcom
        } else {
            $option = New-CimSessionOption -Protocol Wsman
        }
        $session = New-CimSession -ComputerName $computer
       ➥ -SessionOption $option
        #
        $os = Get-CimInstance -ClassName Win32_OperatingSystem
        ➥ -CimSession $session
        $session | Remove-CimSession
        $os | Select-Object -Prop @{n='ComputerName';e={$computer}},
                                  Version,ServicePackMajorVersion
    } #foreach
} #function
```

Установка протокола сеанса

Открытие сеанса

Запрос данных

Закрытие сеанса

Вывод данных

Листинг 11.2. Измененная функция Get-MachineInfo

```
function Get-MachineInfo {
    [CmdletBinding()]
    Param(
        [Parameter(ValueFromPipeline=$True)]
        [string[]]$ComputerName,
        [string]$LogFailuresToPath,
        [string]$Protocol = "Wsman",
```

Добавленный CmdletBinding

Добавленный декоратор [Parameter]

```
        [switch]$ProtocolFallback
    )
BEGIN {}  ◄── Добавленные блоки скриптов
PROCESS {
    foreach ($computer in $computername) {
        # Установка протокола сеанса
        if ($protocol -eq 'Dcom') {
            $option = New-CimSessionOption -Protocol Dcom
        } else {
            $option = New-CimSessionOption -Protocol Wsman
        }
        # Открытие сеанса
        $session = New-CimSession -ComputerName $computer `
                                  -SessionOption $option
        # Запрос данных
        $os = Get-CimInstance -ClassName Win32_OperatingSystem `
                              -CimSession $session
    # Закрытие сеанса
    $session | Remove-CimSession
    # Вывод данных
    $os | Select-Object -Prop @{n='ComputerName';e={$computer}},
                            Version,ServicePackMajorVersion
    } #foreach
} #PROCESS
END {}
} #function
```

Мы кое-что добавили:

- в `Param()` — блок `[CmdletBinding()]`;
- в параметр `[Parameter()]$ComputerName` — *декоратор*, или *атрибут*. Несмотря на то что мы физически поместили его на предыдущую строку, PowerShell будет считывать обе строки как одну;
- блоки скриптов `BEGIN{}`, `PROCESS{}` и `END{}`.

Чтобы понять, как все это совмещается, необходимо помнить, что функция должна выполняться в двух различных режимах, каждый из которых имеет несколько иные требования к PowerShell.

Выполнение команд в режиме без конвейера

Представьте, что выполняете такую команду:

```
Get-MachineInfo -ComputerName ONE,TWO,THREE
```

В этом режиме PowerShell будет игнорировать *метки* `BEGIN{}`, `PROCESS{}` и `END{}`, но не *код внутри них*. Иными словами, получится так, будто метки не существовали изначально. `$ComputerName` будет содержать массив из трех объектов `[string]`: `ONE`, `TWO` и `THREE`. Вся команда будет выполняться один раз, от первой строки кода и до последней. Цикл `ForEach` будет выполняться трижды.

Выполнение команд в режиме конвейера

А теперь представьте, что выполняете ту же команду так:

```
"ONE","TWO","THREE" | Get-MachineInfo
```

Сначала PowerShell создаст трехэлементный массив, поскольку так работают в оболочке списки, разделенные запятой. Затем она просканирует конвейер и выполнит блок `BEGIN{}` для каждой находящейся в нем команды. Это касается как расширенных функций, так и скомпилированных командлетов. Удачным местом для настройки задач наподобие открытия соединений с базой данных, настройки файлов логов или инициализации массивов является блок `Begin` (название которого *необязательно* должно писаться заглавными буквами и который можно опустить, если у вас нет для него кода). Все создаваемые в этом блоке переменные продолжат существовать в других местах вашей функции.

Далее PowerShell начнет *поочередно* передавать элементы из этого трехэлементного массива по конвейеру. То есть она вставит `"ONE"` в `$ComputerName`, а затем выполнит блок `PROCESS{}`. Цикл `ForEach` выполнится, но всего раз — в этом режиме он не нужен, только в режиме без конвейера. После этого оболочка передаст в `$ComputerName` элемент `"TWO"` и снова выполнит `PROCESS{}`. Далее в `$ComputerName` будет передан элемент `"THREE"`, и `PROCESS{}` выполнится последний раз.

Наконец, когда все объекты пройдут через конвейер, PowerShell заново просканирует его и попросит каждый элемент вернуть свой блок `END{}`. Опять же, вы можете опустить данный шаг, если добавить в этот блок будет нечего. Хотя для наглядности мы предпочитаем добавлять его, даже если он пуст. Как вариант, можно вставлять в пустые блоки `Begin` и `End` комментарий, чтобы не казалось, будто чего-то не хватает:

```
End {
 Write-Output "All Done"
}
```

Значения и PROPERTYNAMES

Обратите внимание, что в рассмотренном примере используется этот декоратор:

```
[Parameter(ValueFromPipeline=$True)]
```

Так мы включим режим привязывания вывода конвейера `ByValue` (по значению). Включить его можно только для одного параметра на каждый тип данных. Из-за того, что `$ComputerName` является `[string]`, это единственный параметр `[string]`, который можно обозначить как получающий вывод конвейера `ByValue`.

Кроме того, можно включить режим получения вывода `ByPropertyName` (по имени свойства):

```
[Parameter(ValueFromPipeline=$True,ValueFromPipelineByPropertyName=$True)]
```

Теперь, если объект в конвейере не является `System.String`, но содержит *свойство* `ComputerName`, переменная `$ComputerName` тоже его подхватит.

Если вы плохо знакомы с режимами ввода параметров конвейера `ByValue` и `ByPropertyName`, то рекомендуем прочесть книгу «Изучаем PowerShell за месяц, занимаясь один час в день». Это очень важная функциональность Windows PowerShell.

11.1.2. Обязательность (Mandatory)

Функция не может корректно выполняться без имени компьютера, поэтому всегда необходимо предоставлять ей хотя бы одно. Исправленный набор параметров выглядит так:

```
Param(
    [Parameter(ValueFromPipeline=$True,
               Mandatory=$True)]
    [string[]]$ComputerName,
    [string]$LogFailuresToPath,
    [string]$Protocol = "Wsman",
    [switch]$ProtocolFallback
)
```

А вот важные моменты, которые нужно учитывать при принятии решения.

- Есть смысл сделать `$ComputerName` обязательным. Если значение этого параметра не будет предоставлено, то PowerShell его запросит. Если же оно так и не будет указано, то оболочка выдаст ошибку. Важно помнить, что для обязательного параметра нельзя указать предустановленное значение, как мы делаем в случае с параметром `Protocol`.
- Делать обязательным `$LogFailuresToPath` не имеет смысла, поскольку не нужно заставлять разработчиков логировать ошибки. Мы будем проверять, предоставил ли пользователь путь для их логирования, и соответствующим образом настраивать этот процесс.
- Чисто технически `$Protocol` является обязательным, однако мы указываем для него предустановленное значение `"Wsman"`, чтобы не вынуждать разработчиков вводить его вручную, к чему привела бы установка `Mandatory=$True`. Вполне нормально, если пользователь не укажет протокол, так как у нас есть рабочее предустановленное значение.
- Параметр `[switch]` мы никогда не делаем обязательным, поскольку тогда он, по сути, окажется `$True` (либо пользователю придется выполнить `-ProtocolFallback:$false` для его отключения, что было бы весьма неудобно).
- Обязательными можно сделать любое количество параметров по вашему желанию.

11.1.3. Валидация параметров

Параметр `$Protocol` имеет слабое место: он получает любую строку. Его код плохо защищен от некорректных значений, и причиной тому — структура конструкции `If`. Но будет очень кстати предотвратить возможность получения ошибочных значений в принципе. Вдобавок будет хорошо предоставить пользователям некий ключ, позволяющий понять, какие значения являются корректными. Оба действия можно выполнить на одном этапе:

```
[CmdletBinding()]
Param(
    [Parameter(ValueFromPipeline=$True,
               Mandatory=$True)]
    [string[]]$ComputerName,
    [string]$LogFailuresToPath,
    [ValidateSet('Wsman','Dcom')]
    [string]$Protocol = "Wsman",
    [switch]$ProtocolFallback
)
```

Мы добавляем в параметр `$Protocol` атрибут `[ValidateSet()]`. В итоге PowerShell не допустит применения значений, указанных в этом списке, будет выводить в справке допустимые значения, а также автоматически генерировать и даже предоставлять возможность автозавершения этих значений с помощью клавиши Tab. Есть и другие методы валидации. Полный список можно найти в `about_functions_advanced_parameters`.

11.1.4. Псевдонимы параметров

И последнее. До этого вы следовали нативным паттернам PowerShell, используя в качестве имени параметра `-ComputerName`. Тем не менее вы можете найти ценным такое дополнение:

```
[CmdletBinding()]
Param(
    [Parameter(ValueFromPipeline=$True,
               Mandatory=$True)]
    [Alias('CN','MachineName','Name')]
    [string[]]$ComputerName,
    [string]$LogFailuresToPath,
    [ValidateSet('Wsman','Dcom')]
    [string]$Protocol = "Wsman",
    [switch]$ProtocolFallback
)
```

Здесь вы определяете для параметра три псевдонима, делая `-CN`, `-MachineName` и `-Name` рабочими альтернативами.

УСЛОЖНЕНИЕ

Параметры могут становиться крайне сложными, особенно когда вы начнете знакомиться с новейшими возможностями последних версий PowerShell. Мы разбирали их в более сложных книгах (*PowerShell in Depth* (Manning, 2014), *The PowerShell Scripting & Toolmaking Book* (2022)), но здесь мы о них не говорим, поскольку эта тема выходит за пределы «знакомства с разработкой инструментов», которому посвящена данная книга. Тем не менее мы хотим, чтобы вы знали о наличии таких возможностей.

Как вариант, вы можете определять несколько *наборов параметров*. Такие наборы зачастую содержат как определенные общие параметры, которые типичны для них всех, так и другие, которые делают их уникальными и взаимоисключающими. Ваш атрибут `CmdletBinding` даже может определять, какой набор будет являться предустановленным.

Еще одна тема, которая заслуживает отдельной книги, — *динамические параметры*. Они появляются или исчезают в зависимости от конкретной ситуации, в которой оказывается команда. Вы можете использовать определенные параметры, когда команда выполняется на локальном диске, но скрывать их, когда диск сетевой. Возможности практически безграничны, что серьезно усложняет работу с этими параметрами.

Параметры PowerShell позволяют реализовывать широкий спектр самых разных сценариев. Вы будете готовы погрузиться в эту тему еще больше, когда освоите основы, о которых идет речь на страницах этой книги.

11.1.5. Поддержка -Confirm и -WhatIf

Атрибут `[CmdletBinding()]` сопровождается еще двумя дополнительными элементами функциональности: параметрами `-WhatIf` и `-Confirm`. Предлагаем ненадолго выйти за рамки нашего рабочего примера и обсудить эту зачастую недопонимаемую, но весьма ценную возможность. Рассмотрим блок параметров:

```
Function Set-Something {
    [CmdletBinding(SupportsShouldProcess=$True,ConfirmImpact='Low')]
    Param(
    )
} #function
```

Атрибут `CmdletBinding` стал более сложным. Он объявил, что поддерживает `ShouldProcess`, функциональность PowerShell, позволяющую использовать параметры функции `-WhatIf` и `-Confirm`. Такая возможность подходит для функций, которые предполагают внесение изменений в систему. Если кто-либо выполнит нашу команду с помощью параметра `-WhatIf`, при том, что мы предварительно проделали все правильные шаги, то команда никаких действий не совершит: она

просто покажет, что сделала бы, если бы мы ей это позволили. Если же кто-нибудь выполнит ее с помощью параметра `-Confirm` и при этом в коде будут реализованы все правильные действия, то PowerShell попросит пользователя подтвердить каждую операцию, по сути спрашивая: «Вы уверены?»

Стоит отметить, что переключатели `-WhatIf` и `-Confirm` наследуются командами внутри нашей функции. То есть нам не нужно *ничего* делать, если мы просто выполняем какую-то другую команду, которая сама поддерживает `-WhatIf` и `-Confirm`. Выполнение *нашей* функции с одним или обоими параметрами приведет к их передаче внутрь команд. Но представим, что мы хотим выполнить команду, которая не поддерживает `-WhatIf` и `-onfirm`, — возможно, необработанный класс .NET Framework, способный нарушить работу системы:

```
Function Invoke-InfoTechExplosion {
  [CmdletBinding(SupportsShouldProcess=$True,ConfirmImpact='Low')]
  Param(
    [Parameter(Mandatory=$True)]
    [string[]]$DomainNameToCrash
  )
  ForEach ($Domain in $DomainNameToCrash) {
    If ($PSCmdlet.ShouldProcess($Domain)) {
      [System.Directory]::GetDomain($Domain).Destroy()
    }
  }
} #function
```

Это смешной пример, но суть вы наверняка поняли. Когда мы вызываем `$PSCmdlet.ShouldProcess()` и передаем описание того, чего мы хотим, PowerShell выполняет следующие действия.

- Если команда была выполнена без `-WhatIf` или `-Confirm`, то метод возвращает `True`, и выполняется то, что мы поместили в конструкцию `If`.
- Если команда была выполнена с `-WhatIf`, то выводится сообщение, метод возвращает `False`, и наш опасный код не выполняется.
- Если команда была выполнена с `-Confirm`, то система задает пользователю вопрос, и метод возвращает `True` либо `False`, исходя из ответа на этот вопрос, определяя, будет ли опасный код выполнен.

Установка `ConfirmImpact` действует во встроенной переменной `$ConfirmPreference` в оболочке, которая по умолчанию определена как `"High"`. В ней можно указать `"Low"`, `"Medium"` или `"High"`. Суть вот в чем: если указанная установка `ConfirmImpact` *равна* или *больше* содержимого `$ConfirmPreference`, то автоматически используется параметр `-Confirm`, даже если мы не вводим его явно.

В качестве лучшей практики вам следует поддерживать возможность `ShouldProcess` в любой команде, которая может изменить систему. Как правило, команды с глаголом `Get` этого не делают, а команды наподобие `Set`, `Invoke`, `Remove`, `Add` и т. д. могут и должны поддерживать такой набор возможностей. Если вы сопроводите свою команду

комментариями на основе справки (о чем мы скоро поговорим), то вам не придется документировать `-WhatIf` и `-Confirm`; они будут задокументированы автоматически.

Вторая лучшая практика заключается в том, чтобы *не объявлять* поддержку `ShouldProcess`, пока вы эту поддержку не *реализуете*. Как мы отметили, иногда вам не нужно ничего делать, кроме как позволить `-WhatIf` или `-Confirm` попасть в команды, которые вы уже выполняете. Но *этот момент нужно тестировать*: будет очень печально, если кто-нибудь выполнит команду с `-WhatIf`, а в итоге окажется, что код ошибочен и опасное действие будет совершено.

11.2. УПРАЖНЕНИЯ

Вернемся к команде, которую вы создали в предыдущей главе, и внесем кое-какие доработки.

11.2.1. Вводная информация

В листинге 11.3 показано то, на чем мы остановились. Можете использовать данный код в качестве отправной точки для получения своих результатов.

Листинг 11.3. Функция Set-TMServiceLogon

```
function Set-TMServiceLogon {
    Param(
        [string]$ServiceName,
        [string[]]$ComputerName,
        [string]$NewPassword,
        [string]$NewUser,
        [string]$ErrorLogFilePath
    )
    ForEach ($computer in $ComputerName) {
        $option = New-CimSessionOption -Protocol Wsman
        $session = New-CimSession -SessionOption $option `
                                  -ComputerName $Computer
        If ($PSBoundParameters.ContainsKey('NewUser')) {
            $args = @{'StartName'=$NewUser;
                      'StartPassword'=$NewPassword}
        } Else {
            $args = @{'StartPassword'=$NewPassword}
        }
        Invoke-CimMethod -ComputerName $computer `
                         -MethodName Change `
                         -Query "SELECT * FROM Win32_Service WHERE Name =
        ➥ '$ServiceName'" `
                         -Arguments $args |
        Select-Object -Property @{n='ComputerName';e={$computer}},
                                @{n='Result';e={$_.ReturnValue}}
        $session | Remove-CimSession
    } #foreach
} #function
```

11.2.2. Ваша задача

Сделайте эту функцию расширенной и реализуйте следующие возможности:

- `ServiceName`, `ComputerName` и `NewPassword` должны быть обязательными. Не делайте обязательным `NewUser`;
- `ComputerName` должна получать ввод по значению (`ByValue`);
- `ServiceName`, `ComputerName`, `NewPassword` и `NewUser` должны получать ввод из конвейера `ByPropertyName`.

11.2.3. Наше решение

В листинге 11.4 показано, к какому решению пришли мы. В частности, обратите внимание на добавленную в тело кода метку `PROCESS{}`.

ПРИМЕЧАНИЕ

Мы не реализовывали здесь команду ShouldProcess, хотя, пожалуй, стоило, поскольку она изменяет систему. Обратите внимание: наше изменение вносится с помощью использования Invoke-CimMethod. Поддерживает ли она ShouldProcess? То есть поддерживает ли она -WhatIf и -Confirm? Если да, то как нужно передавать их из нашей команды? Выполните все это в качестве упражнения и посмотрите, удастся ли вам разобраться.

Листинг 11.4. Измененная функция Set-TMServiceLogon

```
function Set-TMServiceLogon {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string]$ServiceName,
        [Parameter(Mandatory=$True,
                   ValueFromPipelineByPropertyName=$True,
                   ValueFromPipeline=$True)]
        [string[]]$ComputerName,
        [Parameter(Mandatory=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string]$NewPassword,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewUser,
        [string]$ErrorLogFilePath
    )
BEGIN{}
PROCESS{
    ForEach ($computer in $ComputerName) {
        $option = New-CimSessionOption -Protocol Wsman
        $session = New-CimSession -SessionOption $option `
                                  -ComputerName $Computer
        If ($PSBoundParameters.ContainsKey('NewUser')) {
            $args = @{'StartName'=$NewUser
```

```
                    'StartPassword'=$NewPassword}
        } Else {
            $args = @{'StartPassword'=$NewPassword}
        }
        Invoke-CimMethod -ComputerName $computer `
                         -MethodName Change `
                         -Query "SELECT * FROM Win32_Service WHERE Name =
        ➥ '$ServiceName'" `
                        -Arguments $args |
        Select-Object -Property @{n='ComputerName';e={$computer}},
                                @{n='Result';e={$_.ReturnValue}}
        $session | Remove-CimSession
    } #foreach
} #PROCESS
END{}
} #function
```

Мы добавили это решение в примеры кода книги (доступны на http://mng.bz/rjgE).

Наконец, после тестирования функции сбросьте настройки службы BITS, как делали это в предыдущей главе, чтобы исключить какие-либо проблемы с ней.

```
Get-Service Bits | Start-service
Get-Service Bits | Set-Service -StartupType Automatic
```

ИТОГИ ГЛАВЫ

В этой главе мы уделили особое внимание использованию атрибута `CmdletBinding()`. При добавлении его в функцию становятся доступны стандартные параметры наподобие `-Verbose`, `-Debug`, `-ErrorAction` и др., расширяющие ее функциональность и юзабилити. Кроме того, в этой главе мы рассмотрели различные аспекты расширенных функций PowerShell, в том числе получение ввода из конвейера, установку параметров как обязательных, валидацию параметров, их псевдонимы и поддержку параметров `-Confirm` и `-WhatIf`. И наконец, в ней вы узнали, как применять навыки скриптинга, изучив примеры и пояснения по созданию более гибких и мощных функций. Кроме того, мы порекомендовали вам ресурсы для дополнительного изучения темы динамических параметров и наборов параметров, которые могут быть полезны в более сложных сценариях.

12 Объекты как лучший вид вывода

Помните, как в главе 8 мы создали нашу первую структуру для `Get-MachineInfo`? До этого момента мы запрашивали информацию, но не всю нужную нам. Это решение было намеренным, чтобы вы могли сначала сформировать некую структуру инструмента. Кроме того, когда вы начнете запрашивать большой объем информации, вам придется использовать определенный подход для ее объединения, и мы хотели разобрать его в отдельной главе.

На данный момент «функциональная» часть инструмента выглядит так:

```
# Запрос данных
$Session = New-CIMSession -ComputerName SRV1
$os = Get-CimInstance -ClassName Win32_OperatingSystem `
                      -CimSession $session
# Закрытие сеанса
$session | Remove-CimSession
# Вывод данных
$os | Select-Object -Prop @{n='ComputerName';e={$computer}},
                          Version,ServicePackMajorVersion
```

Для создания нужных нам элементов вывода мы используем `Select-Object`. Некоторые могут назвать это ленивым решением, но мы лишь делаем то, что кто-то мог бы сделать и сам, — сокращаем объем собранной информации. Вернемся к списку информации, которую мы хотели получать изначально, и добавим данные о том, откуда она должна поступать:

- имя хоста (берется из параметра);
- данные о производителе (`Win32_ComputerSystem`);
- модель (`Win32_ComputerSystem`);
- версия и номер сборки ОС (`Win32_OperatingSystem`; `Version` и `BuildNumber`);
- версия пакета обновлений, если таковой имеется (`Win32_OperatingSystem`; `ServicePackMajorVersion`);

- установленная оперативная память (`Win32_ComputerSystem`; `TotalPhysicalMemory` указывается в байтах);
- тип процессора (`Win32_Processor`; `AddressWidth` может быть 32 либо 64);
- количество сокетов процессора (`Win32_ComputerSystem`; `NumberOfProcessors`);
- общее количество ядер (`Win32_ComputerSystem`; `NumberOfLogicalProcessors`);
- свободное пространство на системном диске (обычно `C:`, но не всегда). Здесь есть сложность. В `Win32_OperatingSystem` есть свойство `SystemDrive`, которое отражает системный диск, например `C:`; вам нужно запросить `Win32_LogicalDisk`, у которого значение свойства `DeviceId` совпадает со значением `SystemDrive`, и просмотреть его `FreeSpace`, указанное в байтах.

Теперь приступим к сборке этой информации.

12.1. СБОРКА ИНФОРМАЦИИ

В некоторых местах мы постараемся не использовать обратные кавычки, чтобы ширина столбцов кода вписывалась в формат страницы. Вместо них мы начнем применять прием *«сплэттинг»* (splatting). С его помощью мы создаем хеш-таблицу, ключи которой являются именами параметров, а значения — соответствующими им значениями. Можете дать переменной хеш-таблицы любое имя. Мы обычно используем описательное. Например:

```
$params = @{'ClassName'='Win32_OperatingSystem'
            'ComputerName'='CLIENT1'}
```

Разместите каждый параметр на новой строке. Параметрам-переключателям присвойте значение `$True`:

```
$params = @{'ClassName'='Win32_OperatingSystem'
            'ComputerName'='CLIENT1'
            'Verbose' = $True}
```

Далее мы передадим эти значения команде, заменив в имени переменной `$` на `@`:

```
Get-CimInstance @params
```

Готово!

Вот исправленный фрагмент кода, который запрашивает нужную вам информацию в переменных:

```
# Запрос данных
$os_params = @{'ClassName'='Win32_OperatingSystem'
               'CimSession'=$session}
$os = Get-CimInstance @os_params
$cs_params = @{'ClassName'='Win32_ComputerSystem'
               'CimSession'=$session}
$cs = Get-CimInstance @cs_params
$sysdrive = $os.SystemDrive  ◄── Получает только значение SystemDrive
```

```
$drive_params = @{'ClassName'='Win32_LogicalDisk'
                  'Filter'="DeviceId='$sysdrive'"
                  'CimSession'=$session}
$drive = Get-CimInstance @drive_params
$proc_params = @{'ClassName'='Win32_Processor'
                 'CimSession'=$session}
$proc = Get-CimInstance @proc_params |
        Select-Object -first 1
```

Выбирает первый процессор

Напоследок два важных примечания.

- Обратите внимание, откуда вы получаете букву системного диска в `$sysdrive`, после чего используете `$sysdrive` как часть фильтра в `Get-CimInstance`. Так вы гарантируете, что `$drive` будет содержать только один объект.
- Вы используете `Select-Object`, чтобы убедиться, что `$proc` содержит только один объект. Процессоры в компьютере не могут иметь другой `AddressWidth`, поэтому ограничение запроса одним результатом немного упростит работу с ним при сбое информации.

12.2. СОЗДАНИЕ И ОТПРАВКА ВЫВОДА

В этот момент *вам нужно избежать* вывода *текста*. Лучше всего будет не использовать `Write-Host` для генерируемого инструментом вывода, поскольку этот вывод отправляется в информационный поток и попадает в консоль. Вы не сможете перенаправлять или как-то иначе снова использовать этот вывод, что идет вразрез с самой сутью инструментов, которые применяются повторно. Вместо этого они должны *всегда* выводить структурированные данные в виде объектов, как это и полагается командам PowerShell:

```
# Вывод данных
$props = @{'ComputerName'=$computer
           'OSVersion'=$os.version
           'SPVersion'=$os.servicepackmajorversion
           'OSBuild'=$os.buildnumber
           'Manufacturer'=$cs.manufacturer
           'Model'=$cs.model
           'Procs'=$cs.numberofprocessors
           'Cores'=$cs.numberoflogicalprocessors
           'RAM'=($cs.totalphysicalmemory / 1GB)
           'Arch'=$proc.addresswidth
           'SysDriveFreeSpace'=$drive.freespace}
$obj = New-Object -TypeName PSObject -Property $props
Write-Output $obj
```

Есть несколько моментов, которые нужно учесть.

- В отличие от применения сплэттинга вы создаете в переменной `$props` хеш-таблицу, которая содержит вывод. Каждый ключ в ней представляет имя свойства, которое вам нужно вывести, а каждое значение — соответствующие этому свойству данные.

- Мы использовали более короткие имена свойств, чем обычно, в основном для того, чтобы блоки кода помещались на странице книги. Например, мы обычно используем `Architecture`, а не `Arch`, поскольку это имя более понятное. Ключ хеш-таблицы в итоге станет именем свойства. Постарайтесь не использовать имена с пробелами; имена с нижними подчеркиваниями (_) также смотрятся непрофессионально.
- Вы используете `New-Object` для создания пустого объекта и прикрепления к нему свойств и значений из хеш-таблицы.
- Вам *необязательно* сохранять объект в `$obj`, но мы обычно так делаем. Дело в том, что мы изменяем этот объект позднее и его нахождение в переменной упрощает задачу.
- Вы выводите объект *сразу же* в конвейер, используя `Write-Output`, а не накапливаете его в массиве или чем-то еще, чтобы вывести позже. Весь смысл конвейера в том, чтобы накапливать объекты и передавать их на следующий этап.

12.3. БЫСТРЫЙ ТЕСТ

После импорта модуля и выполнения команды мы получили такой вывод:

```
OSVersion         : 10.0.22621
SPVersion         : 0
SysDriveFreeSpace : 16500285440
Procs             : 1
Manufacturer      : HyperV
Cores             : 12
ComputerName      :
RAM               : 31.5475044250488
OSBuild           : 22621
Model             : 20MDCTO1WW
Arch              : 64
```

Обратите внимание: *эти свойства находятся в неправильном порядке*. Дело в том, что для создания их списка мы использовали обычную хеш-таблицу, а память .NET оптимизирует это хранилище, что может приводить к изменению порядка. И это нормально. На данном уровне создания инструмента вам не стоит беспокоиться о том, как конкретно выглядит вывод, — вы всегда можете указать порядок с помощью команды `Format` или `SelectObject`. Еще можно создать `[ordered]` (упорядоченную) хеш-таблицу, но мы редко так делаем. Излишнее беспокойство о необработанном выводе скрипта контрпродуктивно и противоречит нативным паттернам PowerShell. Постарайтесь сдержать свои порывы по оформлению вывода и позвольте ему быть таким, каким он должен быть.

ПРИМЕЧАНИЕ

Мы намеренно оставили SysDriveFreeSpace в байтах, поскольку это значение пригодится для демонстрации другого приема позднее.

В листинге 12.1 представлен код функции `Get-MachineInfo`.

Листинг 12.1. Get-MachineInfo

```
function Get-MachineInfo {
    [CmdletBinding()]
    Param(
        [Parameter(ValueFromPipeline=$True,
                   Mandatory=$True)]
        [Alias('CN','MachineName','Name')]
        [string[]]$ComputerName,
        [string]$LogFailuresToPath,
        [ValidateSet('Wsman','Dcom')]
        [string]$Protocol = "Wsman",
        [switch]$ProtocolFallback
    )
BEGIN {}
PROCESS {
    foreach ($computer in $computername) {
        # Установка протокола сеанса
        if ($protocol -eq 'Dcom') {
            $option = New-CimSessionOption -Protocol Dcom
        } else {
            $option = New-CimSessionOption -Protocol Wsman
        }
        # Открытие сеанса
        $session = New-CimSession -ComputerName $computer `
                                  -SessionOption $option
        # Запрос данных
        $os_params = @{'ClassName'='Win32_OperatingSystem'
                       'CimSession'=$session}
        $os = Get-CimInstance @os_params
        $cs_params = @{'ClassName'='Win32_ComputerSystem'
                       'CimSession'=$session}
        $cs = Get-CimInstance @cs_params
        $sysdrive = $os.SystemDrive
        $drive_params = @{'ClassName'='Win32_LogicalDisk'
                          'Filter'="DeviceId='$sysdrive'"
                          'CimSession'=$session}
        $drive = Get-CimInstance @drive_params
        $proc_params = @{'ClassName'='Win32_Processor'
                         'CimSession'=$session}
        $proc = Get-CimInstance @proc_params |
                Select-Object -first 1
        # Закрытие сеанса
        $session | Remove-CimSession
        # Вывод данных
        $props = @{'ComputerName'=$computer
                   'OSVersion'=$os.version
                   'SPVersion'=$os.servicepackmajorversion
                   'OSBuild'=$os.buildnumber
                   'Manufacturer'=$cs.manufacturer
                   'Model'=$cs.model
```

```
                    'Procs'=$cs.numberofprocessors
                    'Cores'=$cs.numberoflogicalprocessors
                    'RAM'=($cs.totalphysicalmemory / 1GB)
                    'Arch'=$proc.addresswidth
                    'SysDriveFreeSpace'=$drive.freespace}
        $obj = New-Object -TypeName PSObject -Property $props
        Write-Output $obj
    } #foreach
} #PROCESS
END {}
} #function
```

12.4. АЛЬТЕРНАТИВА ОБЪЕКТАМ

Надеемся, что к этому моменту вы уже запомнили, что PowerShell полностью базируется на объектах. Использование `New-Object`, как мы показывали, очень полезно. Но в качестве альтернативы можно использовать акселератор типов — `[pscustomobject]`. Если применить его перед определением хеш-таблицы, то PowerShell создаст пользовательский объект, как если бы вы применили `New-Object`:

```
 [pscustomobject]@{
Name = 'James'
Department = 'IT'
ComputerName = 'Laptop-QTP097'
Expires = (Get-Date).AddDays(90)
}
```

Этот код создаст объект таким образом:

```
Name   Department  ComputerName   Expires
----   ----------  ------------   -------
James  IT          Laptop-QTP097  3/30/2023 1:23:33 PM
```

Мы считаем, что использовать `[pscustomobject]` в консоли при тестировании привязок конвейера очень удобно, поскольку этот прием позволяет динамически создавать простые объекты:

```
[pscustomobject]@{Name='bits';computername='Laptop-256'} | get-service
```

В качестве бонуса этот акселератор типов будет использовать хеш-таблицу как упорядоченную. Это означает, что имена ваших свойств будут отображаться в том порядке, в котором вы их перечислили. Как мы уже говорили, об этом не стоит слишком сильно беспокоиться, но иногда такой прием оказывается кстати.

Теперь ответим на вопрос, который, надеемся, вы зададите: «Какой прием мне использовать?» Предпочтительно взять командлет наподобие `New-Object`, ведь если ваш код будет смотреть тот, кто мало знаком с PowerShell, то сможет запросить справку о `New-Object`. Использование `[pscustomobject]` может сделать код менее понятным, но в таком случае вы можете добавить пояснительный комментарий.

12.5. ДОПОЛНЕНИЕ ОБЪЕКТОВ

В рабочем примере вы используете пользовательские объекты для объединения информации из других полученных вами объектов. Это не единственный случай применения, с которым вы можете столкнуться, поэтому мы хотели ненадолго отойти от разбираемого примера и рассмотреть другой сценарий.

Предположим, что вы пишете команду для извлечения из Active Directory компьютерных аккаунтов, соответствующих указанному критерию фильтрации. Ваша задача — создать всю исходную информацию, которая есть в Active Directory для каждого компьютерного аккаунта. При этом вам нужно вернуть номер сборки Windows, запущенной на каждом компьютере, — по крайней мере, для тех подключенных к сети машин, на запрос которых у вас есть разрешение.

Вы *можете* использовать ту же модель, которую мы применяли до сих пор, и создать новый объект, содержащий объединенную информацию. Но эти объекты компьютеров из Active Directory содержат *множество* свойств, для воспроизведения которых потребуется *большой объем* кода. А вам нужно добавить всего одно маленькое свойство. Нельзя ли добавить его в существующий объект компьютера? Можно. Взгляните на листинг 12.2.

Листинг 12.2. Функция Add-ADComputerWindowsBuild

```
function Add-ADComputerWindowsBuild {
    [CmdletBinding()]
    Param(
        [Parameter(ValueFromPipeline=$True)]
        [object[]]$InputObject
    )
    PROCESS {
        ForEach ($comp in $inputobject) {
            $os = Get-CimInstance -ComputerName $comp.name `
                                  -Class Win32_OperatingSystem
            $comp | Add-Member -MemberType NoteProperty `
                               -Name OSBuild `
                               -Value $os.BuildNumber
        } #foreach
    } #process
} #function
```

Это довольно простой вариант — например, мы не обработали в нем ситуацию, когда компьютер не в сети. Ключевой функциональностью здесь является командлет `Add-Member`. Когда вы передаете в него объект, он позволяет вам добавить свойство. В этом случае мы добавляем `Note-Property`, являющееся статичным значением. Мы назвали это свойство `OSBuild` и указали в нем номер сборки операционной системы, который только что запросили из общей информационной модели. `Add-Member` автоматически изменяет объект, после чего передает его через конвейер. Данный вывод мы не «перехватывали», поэтому он становится выводом функции. Эта команда выполняется так:

```
Get-ADComputer -filter * | Add-ADComputerWindowsBuild
```

Мы по-прежнему используем ключевую команду `Get-ADComputer` для выполнения того, что она умеет лучше всего. Мы просто передаем ее во вторую команду, которая дополняет объекты, добавляя в них новую информацию. Опять же, этот процесс не слишком отличается от создания новых объектов и копирования в них нужных данных, но в данном случае добавить что-то одно намного проще, чем копировать десятки или сотни элементов. К тому же данный прием добавления элемента может оказаться *более быстрым*, поскольку в случае его применения вам не нужно создавать новый объект и копировать множество данных.

Тем не менее отметим, что с классической точки зрения на разработку ПО мы проделали нечто ужасающее. Объекты (точнее говоря, *классы*, которые определяют то, как выглядит класс) должны быть *контрактами*. Они фиксированны, неизменны и надежны. Прикрепляя информацию описанным способом, мы, возможно, *не нарушили* контракт, но, условно говоря, что-то написали карандашом на полях. Но это нормально: расширяемая система типов PowerShell (Extensible Type System, ETS), благодаря которой работает `Add-Member`, *создавалась* как раз для этой цели. PowerShell постоянно дополняет всевозможные объекты, и вы этого наверняка никогда даже не замечали. Так что смело используйте этот прием, когда он может решить ваши задачи.

12.6. УПРАЖНЕНИЯ

Как и в предыдущих главах, мы сосредоточимся на инструменте изменения сервисов. Вы можете предположить, что он не создает никакого вывода, но ошибетесь. Если вы еще раз взглянете на изначальную структуру, то поймете, что вам нужен *вывод* для каждого компьютера, независимо от того, как он работает: *успешно* или же *выдает сбой*. Сейчас вы наверняка просто создаете минимальный вывод с помощью `Select-Object`:

```
Select-Object -Property @{n='ComputerName';e={$computer}},
@{n='Result';e={$_.ReturnValue}}
```

Но скоро это изменится!

12.6.1. Вводная информация

В нашей версии функции мы остановились на следующем моменте (листинг 12.3). Используйте этот код в качестве отправной точки или возьмите собственный из предыдущей главы.

Листинг 12.3. Функция Set-TMServiceLogon

```
function Set-TMServiceLogon {
    [CmdletBinding()]
```

```
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipelineByPropertyName=$True)]
         [string]$ServiceName,
         [Parameter(Mandatory=$True,
                    ValueFromPipeline=$True,
                    ValueFromPipelineByPropertyName=$True)]
         [string[]]$ComputerName,
         [Parameter(ValueFromPipelineByPropertyName=$True)]
         [string]$NewPassword,
         [Parameter(ValueFromPipelineByPropertyName=$True)]
         [string]$NewUser,
         [string]$ErrorLogFilePath
    )
BEGIN{}
PROCESS{
    ForEach ($computer in $ComputerName) {
        $option = New-CimSessionOption -Protocol Wsman
        $session = New-CimSession -SessionOption $option `
                                  -ComputerName $Computer
        If ($PSBoundParameters.ContainsKey('NewUser')) {
            $args = @{'StartName'=$NewUser
                      'StartPassword'=$NewPassword}
        } Else {
            $args = @{'StartPassword'=$NewPassword}
        }
        Invoke-CimMethod -ComputerName $computer `
                         -MethodName Change `
                         -Query "SELECT * FROM Win32_Service WHERE Name =
        ➥ ?'$ServiceName'" `
                         -Arguments $args |
        Select-Object -Property @{n='ComputerName';e={$computer}},
                                @{n='Result';e={$_.ReturnValue}}
        $session | Remove-CimSession
    } #foreach
} #PROCESS
END{}
} #function
```

12.6.2. Ваша задача

Измените функцию так, чтобы она выводила объект для каждого компьютера, в отношении которого выполняется. Этот вывод должен содержать имя компьютера и его статус. Еще раз загляните в коды статуса на странице http://mng.bz/c05L и сделайте так, чтобы 0 в выводе означал Success, 22 — Invalid Account, а все остальное — Failed: *XX*, где *XX* — численное возвращаемое значение. В качестве усложнения попробуйте не добавлять в код дополнительные конструкции `If` — вместо этого примените конструкцию `Switch`. Лучше всего, если вы еще и поищете в коде места, где можно использовать сплэттинг.

12.6.3. Наше решение

В листинге 12.4 показана наша версия. (Помните, что файл с кодом вы можете найти в примерах, доступных для скачивания.)

Листинг 12.4. Наша версия Set-TMServiceLogon

```
function Set-TMServiceLogon {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string]$ServiceName,
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string[]]$ComputerName,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewPassword,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewUser,
        [string]$ErrorLogFilePath
    )
BEGIN{}
PROCESS{
    ForEach ($computer in $ComputerName) {
        $option = New-CimSessionOption -Protocol Wsman
        $session = New-CimSession -SessionOption $option `
                                  -ComputerName $Computer
        If ($PSBoundParameters.ContainsKey('NewUser')) {
            $args = @{'StartName'=$NewUser
                      'StartPassword'=$NewPassword}
        } Else {
            $args = @{'StartPassword'=$NewPassword}
        }
        $params = @{'CimSession'=$session
                    'MethodName'='Change'
                    'Query'="SELECT * FROM Win32_Service WHERE Name = '$ServiceName'"
                    'Arguments'=$args}
        $ret = Invoke-CimMethod @params
        switch ($ret.ReturnValue) {
            0  { $status = "Success" }
            22 { $status = "Invalid Account" }
            Default { $status = "Failed: $($ret.ReturnValue)" }
        }
        $props = @{'ComputerName'=$computer
                   'Status'=$status}
        $obj = New-Object -TypeName PSObject -Property $props
        Write-Output $obj
        $session | Remove-CimSession
    } #foreach
} #PROCESS
END{}
} #function
```

Надеемся, что вы заметили некоторые нюансы.

- Вместо имени компьютера мы использовали сеанс CIM. Вам наверняка стало интересно почему, не так ли? Надеемся, что да. Зачем же мы это сделали? Просто для того, чтобы проверить вашу внимательность.
- Мы использовали сплэттинг в строке 29, а также хеш-таблицу `$params` в строке 34. Заметьте, насколько это решение более интуитивно по сравнению со строкой 29 листинга 12.3.
- Обратите внимание: с помощью конструкции `switch` мы создали сообщение о статусе. Кроме того, при определении запроса мы использовали символ +. Сделали мы это лишь для того, чтобы отформатировать код под размеры страницы. В обычной ситуации вы бы написали этот запрос в одну строку: `"SELECT * FROM Win32_Service WHERE Name = '$Service- Name'"`, и именно такой вариант вы увидите в скачиваемом коде.

Накопление большого количества результатов приведет к тому, что команда заблокирует конвейер. Поочередный же вывод объектов позволяет конвейеру параллельно выполнять множество команд.

ИТОГИ ГЛАВЫ

В этой главе мы рассказали вам, как создавать и генерировать структурированный вывод в PowerShell, подчеркнув важность создания объектов, а не текста. Мы продолжили дорабатывать инструмент `Get-MachineInfo`, чтобы теперь он мог собирать всеобъемлющую информацию об оборудовании и программном обеспечении системы. Используя сплэттинг и команду `New-Object`, мы создали пользовательские объекты, свойства которых представляют различные атрибуты системы. Кроме того, мы познакомили вас с альтернативным методом создания объектов, таким как `[pscustomobject]`. После этого мы разобрали возможность динамического добавления свойств, показав, как вносить в существующие объекты дополнительную информацию, например, номер сборки Windows. Наконец, мы применили все эти приемы для расширения функции `Set-TMServiceLogon`, изменив ее так, чтобы она выводила структурированные данные для каждого обработанного компьютера. При этом мы использовали конструкцию `switch` для определения статуса этого объекта и сплэттинг для обработки параметров. За счет акцента на создании структурированного вывода в виде объектов наши инструменты PowerShell становятся более гибкими, повторно используемыми и подходящими для конвейерной обработки.

13 Использование всех потоков

В главе 11 мы говорили о ключевом слове `[CmdletBinding()]`. Его можно добавлять в блок `Param()`, который превращает функцию в расширенную, позволяющую использовать команды для получения информационного, оповещающего и прочего вывода. Пора использовать эту возможность, чтобы показать, какие выгоды она сулит.

13.1. ПОНИМАНИЕ СЕМИ ПОТОКОВ ВЫВОДА

Полезно знать, что у PowerShell есть семь *потоков вывода*, а не один, который мы обычно представляем. Первый, самый для вас знакомый — это поток *Success*, который вы привыкли считать «концом конвейера». К нему в механизме PowerShell особое отношение. Например, этот поток служит для передачи объектов из команды в команду. Кроме того, в конце конвейера PowerShell как бы незаметно добавляет командлет `Out-Default`, который прогоняет любые объекты конвейера через систему форматирования PowerShell. Какое бы основное приложение вы ни использовали — консоль PowerShell, Visual Studio Code и т. д., — оно отвечает за обработку этого вывода и его отображение на экране или выполнение иных действий.

Всего существует семь потоков.

1. Success (Успех), о котором мы только что говорили.
2. Error (Ошибка).
3. Warning (Предупреждение).
4. Verbose (Подробности).
5. Debug (Отладка).
6. Information (Информация).
7. Progress (Прогресс).

При обращении к каждому из них для перенаправления PowerShell использует соответствующий номер.

Все эти потоки представляют собой отдельные, независимые конвейеры передачи информации. При этом каждое базовое приложение решает, как именно ему работать с каждым конвейером. Например, консольные отображают элементы конвейера 4 (Verbose) желтым текстом с префиксом `VERBOSE:`. Другие могут логировать этот вывод в журнал событий или игнорировать.

Кроме того, оболочка определяет несколько переменных *настроек*, которые управляют выводом каждого конвейера. Так, `$VerbosePreference` управляет конвейером 4, `$WarningPreference` — конвейером 3 и т. д. Установка настройки на `SilentlyContinue` приведет к исключению вывода этого конвейера. Если установить в ней `Continue`, то мы увидим вывод таким, как его определяет базовое приложение. Типичные параметры переопределяют переменные настроек на основе отдельных команд. Например, добавление к команде `-Verbose` при ее выполнении активирует в ней `Write-Verbose`.

13.2. ДОБАВЛЕНИЕ ПОДРОБНОГО (VERBOSE) И ПРЕДУПРЕЖДАЮЩЕГО (WARNING) ВЫВОДА

Подробный вывод по умолчанию отключен, а предупреждающий, наоборот, включен. Учитывая это, мы выполняем с данными формами вывода следующие действия (листинг 13.1).

Листинг 13.1. Добавление вывода

```
function Get-MachineInfo {
    [CmdletBinding()]
    Param(
        [Parameter(ValueFromPipeline=$True,
                   Mandatory=$True)]
        [Alias('CN','MachineName','Name')]
        [string[]]$ComputerName,
        [string]$LogFailuresToPath,
        [ValidateSet('Wsman','Dcom')]
        [string]$Protocol = "Wsman",
        [switch]$ProtocolFallback
    )
BEGIN {}
PROCESS {
    foreach ($computer in $ComputerName) {
        if ($protocol -eq 'Dcom') {
            $option = New-CimSessionOption -Protocol Dcom
        } else {
            $option = New-CimSessionOption -Protocol Wsman
        }
        Write-Verbose "Connecting to $computer over $protocol"
```

[CmdletBinding()] добавляется перед блоком Param

Использует подробные сообщения

```
        $session = New-CimSession -ComputerName $computer `
                                  -SessionOption $option
        Write-Verbose "Querying from $computer"
        $os_params = @{'ClassName'='Win32_OperatingSystem'
                       'CimSession'=$session}
        $os = Get-CimInstance @os_params
        $cs_params = @{'ClassName'='Win32_ComputerSystem'
                       'CimSession'=$session}
        $cs = Get-CimInstance @cs_params
        $sysdrive = $os.SystemDrive
        $drive_params = @{'ClassName'='Win32_LogicalDisk'
                          'Filter'="DeviceId='$sysdrive'"
                          'CimSession'=$session}
        $drive = Get-CimInstance @drive_params
        $proc_params = @{'ClassName'='Win32_Processor'
                         'CimSession'=$session}
        $proc = Get-CimInstance @proc_params |
                Select-Object -first 1
        Write-Verbose "Closing session to $computer"
        $session | Remove-CimSession
        Write-Verbose "Outputting for $computer"
        $obj = [pscustomobject]@{'ComputerName'=$computer
                   'OSVersion'=$os.version
                   'SPVersion'=$os.servicepackmajorversion
                   'OSBuild'=$os.buildnumber
                   'Manufacturer'=$cs.manufacturer
                   'Model'=$cs.model
                   'Procs'=$cs.numberofprocessors
                   'Cores'=$cs.numberoflogicalprocessors
                   'RAM'=($cs.totalphysicalmemory / 1GB)
                   'Arch'=$proc.addresswidth
                   'SysDriveFreeSpace'=$drive.freespace}
        Write-Output $obj
    } #foreach
} #PROCESS
END {}
} #function
```

Использует [pscustomobject]

Внимательные читатели отметят здесь два момента.

- Мы изменили процесс создания `New-Object`, в основном потому, что хотели показать вам новый прием. Вместо того чтобы определять хеш-таблицу свойств и передавать ее в `New-Object`, мы проделали ту же работу в меньшем объеме кода, используя акселератор типов `[pscustomobject]`. Мы уже говорили о нем в предыдущей главе.
- Мы заменили множество внутристрочных комментариев подробным выводом. Теперь при выполнении кода с добавлением `-Verbose` человек будет видеть то же самое сообщение. Если же команда будет запущена без `-Verbose`, то строки `Write-Verbose` все равно выполнятся, но вывод вы не увидите.

Мы еще не добавили никакого предупреждающего вывода, поскольку он не был нужен. Но в конце концов он потребуется, поэтому не забывайте о `Write-Warning`. В итоге вы добавите оператор наподобие этого:

```
write-warning "Danger, Danger!"
```

13.3. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ, ДОСТУПНЫЕ БЛАГОДАРЯ -VERBOSE

Если вы немного подумаете, то поймете, что добавление в инструменты операторов `Write-Verbose` имеет огромный смысл. Мы советуем вам добавлять их с самого начала работы со скриптом. Не откладывайте это действие на финальный этап. Вставляйте подробные сообщения на протяжении всего скрипта, подчеркивая выполняемые командой действия или значения ключевых переменных. Это поможет вам находить и устранять проблемы в процессе разработки, поскольку вы сможете выполнять команду с префиксом `-Verbose`. Кроме того, подробные сообщения могут дублироваться как внутренняя документация. Наконец, если кто-то пытается использовать ваш инструмент и сталкивается с проблемами, то вы можете попросить этих людей запустить транскрибирование, выполнить команду с приставкой `-Verbose`, после чего завершить транскрибирование и отправить полученный файл вам. Если вы написали хорошие подробные сообщения, то сможете отследить происходящее и, весьма вероятно, выявить проблему.

Подумайте о добавлении в начало команды подробных сообщений наподобие такого:

```
Write-Verbose "Execution Metadata:"
Write-Verbose "User = $($env:userdomain)\$($env:USERNAME)"
$id = [System.Security.Principal.WindowsIdentity]::GetCurrent()
$IsAdmin = [System.Security.Principal.WindowsPrincipal]::new($id).IsInRole(
'administrators')
Write-Verbose "Is Admin = $IsAdmin"
Write-Verbose "ComputerName = $env:COMPUTERNAME"
Write-Verbose "OS = $((Get-CimInstance Win32_Operatingsystem).Caption)"
Write-Verbose "Host = $($host.Name)"
Write-Verbose "PSVersion = $($PSVersionTable.PSVersion)"
Write-Verbose "Runtime = $(Get-Date)"
```

При выполнении вы получите потенциально полезную информацию:

```
VERBOSE: Execution Metadata:
VERBOSE: User = Win10Laptop\James
VERBOSE: Is Admin = False
VERBOSE: ComputerName = Win10Laptop
VERBOSE: Perform operation 'Enumerate CimInstances' with following parameters,
''namespaceName' = root\cimv2,'className' = Win32_Operatingsystem'.
VERBOSE: Operation 'Enumerate CimInstances' complete.
VERBOSE: OS = Microsoft Windows 10.0 Professional
VERBOSE: Host = Microsoft Studio Code Host
VERBOSE: PSVersion = 7.3.2
VERBOSE: Runtime = 02/01/2023 13:57:25
```

Напомним, что при использовании `-Verbose` любые командлеты с поддержкой `-Verbose` (например, `Get-CimInstance`), вызываемые внутри вашей команды, будут генерировать и собственный подробный вывод. В связи с этим ваш подробный вывод будет содержать больше информации, чем могут сгенерировать операторы `Write-Verbose`.

Еще один совет: добавляйте в каждое подробное сообщение префикс, указывающий, какой блок вызывается:

```
Function TryMe {
[cmdletbinding()]
Param(
[string]$ComputerName
)
Begin {
    Write-Verbose "[BEGIN ] Starting: $($MyInvocation.Mycommand)"
    Write-Verbose "[BEGIN ] Initializing array"
    $a = @()
} #begin
Process {
    Write-Verbose "[PROCESS] Processing $ComputerName"
    # код
} #process
End {
    Write-Verbose "[END ] Ending: $($MyInvocation.Mycommand)"
} #end
} #function
```

Видите, как получается блок-комментарий? Он позволяет быстрее понять, где находится ваша команда. Обратите внимание на использование выровненных пробелов. Мы сделали так, чтобы облегчить чтение вывода в консоли:

```
PS C:\> tryme -ComputerName FOO -Verbose
VERBOSE: [BEGIN  ] Starting: TryMe
VERBOSE: [BEGIN  ] Initializing array
VERBOSE: [PROCESS] Processing FOO
VERBOSE: [END    ] Ending: TryMe
```

Подумайте о добавлении временно́й метки. Она будет особенно полезна, если команды выполняются длительное время:

```
Function TryMe {
[cmdletbinding()]
Param(
[string]$ComputerName
)
Begin {
    Write-Verbose "[$((get-date).TimeOfDay.ToString()) BEGIN  ] Starting:
     $($MyInvocation.Mycommand)"
    Write-Verbose "[$((get-date).TimeOfDay.ToString()) BEGIN  ] `
    Initializing array"
```

```
$a = @()
} #begin
Process {
    Write-Verbose "[$((get-date).TimeOfDay.ToString()) PROCESS] Processing
 $ComputerName"
    # код
} #process
End {
    Write-Verbose "[$((get-date).TimeOfDay.ToString()) END    ] Ending:
 $($MyInvocation.Mycommand)"
} #end
} #function
```

В итоге вы получите такой подробный вывод:

```
VERBOSE: [15:18:55.3840626 BEGIN  ] Starting: TryMe
VERBOSE: [15:18:55.4040871 BEGIN  ] Initializing array
VERBOSE: [15:18:55.4080634 PROCESS] Processing FOO
VERBOSE: [15:18:55.4090586 END    ] Ending: TryMe
```

Подробные сообщения можно использовать всевозможными способами. Только вам решать, какая информация окажется полезной. С учетом этого дадим последний совет: добавляйте подробное сообщение, указывающее имя команды. Именно его передавала строка `$myinvocation.mycommand`. Встроенная переменная `$MyInvocation` может сообщить полезную информацию. Свойство `MyCommand` указывает имя вашей команды. Это особенно полезно, если она вызывает другие команды. Добавление типа подробной информации существенно упрощает отслеживание потока вашего выражения PowerShell.

13.4. ВЫВОД ИНФОРМАЦИИ

Шестой поток появился в PowerShell v5, где командлет `Write-Host` претерпел ощутимые изменения, превратившись в оболочку вокруг `Write-Information`. Этот информационный поток немного отличается от других, которые могут передавать сообщения, поскольку предназначен для передачи *структурированных* сообщений. Правильное его использование предполагает предварительное планирование. Но в нем по-прежнему есть переменная `$InformationPreference`, которая может запрещать или разрешать вывод этого потока. По умолчанию она установлена на `SilentlyContinue` или `Off`. При выполнении команды вы можете указать `-InformationAction Continue`, чтобы активировать ее информационный вывод.

`$InformationPreference` и `-InformationAction` автоматически устанавливаются на `Continue`, когда вы используете `Write-Host`, чтобы `Write-Host` работала так, как в предыдущих версиях PowerShell.

На базовом уровне использование `Write-Information` не отличается от использования `Write-Verbose`. Параметр `-MessageData` находится в первой позиции, поэтому зачастую вы можете не указывать его имя и добавить нужное сообщение — как мы

делали с `Write-Verbose`. Но сообщения можно дополнять еще и *тегами*, такими как *information*, *instructions* и др. Это позволяет впоследствии выполнять по ним поиск потока. Кроме того, вы можете выполнять команды, используя параметр `-InformationVariable`, чтобы добавить в указанную переменную информационные сообщения. Так вы избегаете загромождения итогового вывода информационными сообщениями. Вот пример:

```
Function Example {
    [CmdletBinding()]
    Param()
    Write-Information "First message" -tag status
    Write-Information "Note that this had no parameters" -tag notice
    Write-Information "Second message" -tag status
}
Example -InformationAction Continue -InformationVariable x
```

Использование `Continue` подобным образом ведет к ее применению ко всем командам `Write-Information` *внутри* функции `Example`. Если вы выполните ее в PowerShell 5 и более новых версиях, то увидите, что информационные сообщения действительно выводятся. К тому же вы обнаружите их, если проверите `$x`. Сравните предыдущий пример со следующим:

```
function Example {
    [CmdletBinding()]
    Param()
    Write-Information "First message" -tag status
    Write-Information "Note that this had no parameters" -tag notice
    Write-Information "Second message" -tag status
}
Example -InformationAction SilentlyContinue -IV x
```

На этот раз сообщения не появляются, так как мы использовали `SilentlyContinue`. Но команды *по-прежнему* выполняются, и если вы проверите `$x`, то найдете все три сообщения. Обратите внимание: мы сократили `-InformationVariable` до ее псевдонима `-IV`, чтобы сэкономить место.

Продолжим и сделаем еще один шаг:

```
function Example {
    [CmdletBinding()]
    Param()
    Write-Information "First message" -tag status
    Write-Information "Note that this had no parameters" -tag notice
    Write-Information "Second message" -tag status
}
Example -InformationAction SilentlyContinue -IV x
$x | where tags -in @('notice')
```

В этом примере отобразится только второе сообщение, `Note that this had no parameters`, так как мы выделили его из `$x`, используя свойство сообщений `Tags`.

13.4.1. Пример подробного информационного потока

Как и в случае с подробным выводом, эффективное использование информационного потока требует планирования. При создании инструмента нужно определиться, что будет логироваться и как это можно использовать, а также реализовать команды `Write-Information`. В листинге 13.2 приведена простая функция, показывающая, как можно использовать `Write-Information`. Файл с этими тестовыми функциями находится в папке текущей главы по ссылке http://mng.bz/rjgE.

Листинг 13.2. Использование информационной переменной

```
Function Test-Me {
[cmdletbinding()]
Param()
Write-Information "Starting $($MyInvocation.MyCommand) " -Tags Process
Write-Information "PSVersion = $($PSVersionTable.PSVersion)" -Tags Meta
Write-Information "OS = $((Get-CimInstance Win32_operatingsystem).Caption)"`
-Tags Meta
Write-Verbose "Getting top 5 processes by WorkingSet"
Get-process | sort WS -Descending | select -first 5 -OutVariable s
Write-Information ($s[0] | Out-String) -Tags Data
Write-Information "Ending $($MyInvocation.MyCommand) " -Tags Process
}
```

Обычно выполнение этой команды приведет к выводу пяти верхних процессов по рабочему множеству (`Getting top 5 processes by WorkingSet`). А теперь выполните ее так:

```
PS C:\> test-me -InformationAction Continue
Starting Test-Me
PSVersion =  7.3.2
OS = Microsoft Windows 10 Pro Insider Preview
Handles   NPM(K)     PM(K)        WS(K)  VM(M)    CPU(s)     Id  SI  ProcessName
-------   ------     -----        -----  -----    ------     --  --  -----------
   2145      249    856976       883488  1931   7,151.38   5948   1  firefox
   2692      126    769444       396928 ...86   1,531.13   8552   1  PowerShell
    373       59    310584       390504  1421     446.03   7172   1  slack
    395       55    186628       361964  1391     590.89   7508   1  slack
   1181       95    335932       317060  1216     375.38   1004   1  PowerShell...
Handles   NPM(K)     PM(K)        WS(K)  VM(M)    CPU(s)     Id  SI  ProcessName
-------   ------     -----        -----  -----    ------     --  --  -----------
    2145     249    856976       883488   1931  7,151.38   5948   1  firefox
Ending Test-Me
```

Установив общий параметр `-InformationAction` на `Continue`, вы активируете поток Information, который тоже выводит информацию. Такой прием может оказаться полезным при создании сообщений, когда вы хотите увидеть, что конкретно они будут делать.

Далее выполните команду с параметром `-InformationVariable`:

```
PS C:\> test-me -InformationVariable inf
```

Здесь вы не получите информационные сообщения, поскольку команда выполняется с предустановленным параметром `SilentlyContinue`, отключающим их вывод. Вместо этого они направляются в переменную `inf`:

```
PS C:\> $inf
Starting Test-Me
PSVersion = 7.3.2
OS = Microsoft Windows 10 Pro
Handles   NPM(K)     PM(K)        WS(K)  VM(M)    CPU(s)     Id  SI  ProcessName
-------   ------     -----        -----  -----    ------     --  --  -----------
    2142     248     857768       883332   1904  7,155.00  5948   1  firefox
Ending Test-Me
```

Обратно вы получаете очень объемный объект:

```
PS C:\> $inf | get-member
   TypeName: System.Management.Automation.InformationRecord
Name             MemberType  Definition
----             ----------  ----------
Equals           Method      bool Equals(System.Object obj)
GetHashCode      Method      int GetHashCode()
GetType          Method      type GetType()
ToString         Method      string ToString()
Computer         Property    string Computer {get;set;}
ManagedThreadId  Property    uint32 ManagedThreadId {get;set;}
MessageData      Property    System.Object MessageData {get;}
NativeThreadId   Property    uint32 NativeThreadId {get;set;}
ProcessId        Property    uint32 ProcessId {get;set;}
Source           Property    string Source {get;set;}
Tags             Property    System.Collections.Generic.List[string] Tags...
TimeGenerated    Property    datetime TimeGenerated {get;set;}
User             Property    string User {get;set;}
```

Это означает, что вы можете работать с любыми нужными вам данными:

```
PS C:\> $inf.where({$_.tags -contains 'meta'}) |
select Computer,Messagedata
Computer    MessageData
--------    -----------
Win10-01    PSVersion =  7.3.2
Win10-01    OS = Microsoft Windows 10 Pro Insider Previe
```

Самое главное здесь в том, что информационные параметры неважны, если ваша команда не содержит команд `Write-Information`.

Но, как мы говорили ранее, в PowerShell v5 `Write-Host` подверглась рефакторингу, став оболочкой для `Write-Information`. Ознакомьтесь с этой доработанной версией функции (листинг 13.3).

Листинг 13.3. Доработанная функция

```
Function Test-Me2 {
[cmdletbinding()]
Param()
```

```
Write-Host "Starting $($MyInvocation.MyCommand) " -foreground green
Write-Host "PSVersion = $($PSVersionTable.PSVersion)" -foreground green
Write-Host "OS = $((Get-CimInstance Win32_operatingsystem).Caption)"
 -foreground green
Write-Verbose "Getting top 5 processes by WorkingSet"
Get-Process | sort WS -Descending | select -first 5 -OutVariable s
Write-Host ($s[0] | Out-String) -foreground green
Write-Host "Ending $($MyInvocation.MyCommand) " -foreground green
}
```

Одно из преимуществ использования `Write-Host` — возможность окрашивания вывода. К сожалению, даже если вы выполните команду как:

```
test-me2 -InformationVariable inf2
```

то вывод информации будет сохранен в `$inf2`. Но при этом информационные сообщения будут выведены на хосте зеленым цветом, что может оказаться нежелательным. Кроме того, данный прием лишает вас возможности добавлять теги.

В листинге 13.4 приведена заключительная версия, которая лучше всего подтверждает описанные принципы. Вам нужно будет выполнить ее самостоятельно, чтобы увидеть результаты.

Листинг 13.4. Подтверждение принципов

```
Function Test-Me3 {
[cmdletbinding()]
Param()
if ($PSBoundParameters.ContainsKey("InformationVariable")) {
 $Info = $True
 $infVar = $PSBoundParameters["InformationVariable"]
}
if ($info) {
 Write-Host "Starting $($MyInvocation.MyCommand) " -foreground green
  (Get-Variable $infVar).value[-1].Tags.Add("Process")
  Write-Host "PSVersion = $($PSVersionTable.PSVersion)" -foreground green
  (Get-Variable $infVar).value[-1].Tags.Add("Meta")
  Write-Host "OS = $((Get-CimInstance Win32_operatingsystem).Caption)"
 -foreground green
  (Get-Variable $infVar).value[-1].Tags.Add("Meta")
}
Write-Verbose "Getting top 5 processes by WorkingSet"
Get-process | sort WS -Descending | select -first 5 -OutVariable s
if ($info) {
 Write-Host ($s[0] | Out-String) -foreground green
 (Get-Variable $infVar).value[-1].Tags.Add("Data")
 Write-Host "Ending $($MyInvocation.MyCommand) " -foreground green
 (Get-Variable $infVar).value[-1].Tags.Add("Process")
}
}
```

Эта функция проверяет, была ли указана `-InformationVariable`; если да, то переменная (`$Info`) включается. Когда информацию нужно получить с помощью

`Write-Host`, строки `Write-Host` вызываются при `$Info = True`. Сразу после каждой строки в информационную переменную добавляется тег:

```
test-me3 -InformationVariable inf3
```

Эта команда окрашивает информационные сообщения зеленым цветом и генерирует информационную переменную:

```
PS C:\> $inf3 | Group {$_.tags -join "-"}
Count  Name                     Group
-----  ----                     -----
    2  PSHOST-Process           {Starting Test-Me3 , Ending Test-Me3 }
    2  PSHOST-Meta              {PSVersion = 7.3.2, OS = Mi...}
    1  PSHOST-Data              {...
```

Помните, что информационные переменные — это просто типы объектов. Вы можете экспортировать переменную, используя `Export-Clixml`, сохранить результаты в базе данных или создать пользовательский файл журнала из других свойств.

Подробный ввод по-прежнему является удачным решением, когда вы используете версии PowerShell до v5. Если же вы работаете с PowerShell v5, то имеет смысл начать переходить на информационные сообщения, поскольку они гибкие, содержат теги и в них можно выполнять поиск. Сейчас же мы стремимся к максимальной совместимости, поэтому оставляем в своих примерах подробный вывод.

13.5. УПРАЖНЕНИЯ

Как вы наверняка догадались, далее вам нужно будет добавить в инструмент какой-нибудь подробный вывод.

13.5.1. Вводная информация

В листинге 13.5 показано, на чем мы остановились в предыдущей главе. Можете продолжить, используя этот код (или наш пример кода из скачанных дополнительных материалов к книге) либо собственные результаты из предыдущей главы.

Листинг 13.5. Функция Set-TMServiceLogon

```
function Set-TMServiceLogon {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string]$ServiceName,
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string[]]$ComputerName,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewPassword,
```

```
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewUser,
        [string]$ErrorLogFilePath
    )
BEGIN{}
PROCESS{
    ForEach ($computer in $ComputerName) {
        $option = New-CimSessionOption -Protocol Wsman
        $session = New-CimSession -SessionOption $option `
                                  -ComputerName $Computer
        If ($PSBoundParameters.ContainsKey('NewUser')) {
            $args = @{'StartName'=$NewUser
                      'StartPassword'=$NewPassword}
        } Else {
            $args = @{'StartPassword'=$NewPassword}
        }
        $params = @{'CimSession'=$session
                    'MethodName'='Change'
                    'Query'="SELECT * FROM Win32_Service " +
                            "WHERE Name = '$ServiceName'"
                    'Arguments'=$args}
        $ret = Invoke-CimMethod @params
        switch ($ret.ReturnValue) {
            0  { $status = "Success" }
            22 { $status = "Invalid Account" }
            Default { $status = "Failed: $($ret.ReturnValue)" }
        }
        $props = @{'ComputerName'=$computer
                   'Status'=$status}
        $obj = New-Object -TypeName PSObject -Property $props
        Write-Output $obj
        $session | Remove-CimSession
    } #foreach
} #PROCESS
END{}
} #function
```

13.5.2. Ваша задача

Добавьте в свой инструмент подробный вывод, который будет иметь смысл. Если у вас есть возможность добавить предупреждение, то можете сделать и это.

13.5.3. Наше решение

У нас получился такой вариант (листинг 13.6).

Листинг 13.6. Наше решение

```
function Set-TMServiceLogon {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipelineByPropertyName=$True)]
```

```
        [string]$ServiceName,
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string[]]$ComputerName,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewPassword,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewUser,
        [string]$ErrorLogFilePath
    )
BEGIN{}
PROCESS{
    ForEach ($computer in $ComputerName) {
        Write-Verbose "Connect to $computer on WS-MAN"
        $option = New-CimSessionOption -Protocol Wsman
        $session = New-CimSession -SessionOption $option `
                                  -ComputerName $Computer
        If ($PSBoundParameters.ContainsKey('NewUser')) {
            $args = @{'StartName'=$NewUser
                      'StartPassword'=$NewPassword}
        } Else {
            $args = @{'StartPassword'=$NewPassword}
            Write-Warning "Not setting a new user name"
        }
        Write-Verbose "Setting $servicename on $computer"
        $params = @{'CimSession'=$session
                    'MethodName'='Change'
                    'Query'="SELECT * FROM Win32_Service " +
                            "WHERE Name = '$ServiceName'"
                    'Arguments'=$args}
        $ret = Invoke-CimMethod @params
        switch ($ret.ReturnValue) {
            0  { $status = "Success" }
            22 { $status = "Invalid Account" }
            Default { $status = "Failed: $($ret.ReturnValue)" }
        }
        $props = @{'ComputerName'=$computer
                   'Status'=$status}
        $obj = New-Object -TypeName PSObject -Property $props
        Write-Output $obj
        Write-Verbose "Closing connection to $computer"
        $session | Remove-CimSession
    } #foreach
} #PROCESS
END{}
} #function
```

Добавьте подробный вывод такого объема, который требуется для предоставления понятной обратной связи или информации. Вам ничего не стоит добавить команды `Write-Verbose`, и они не будут активированы, пока вы не выполните их с `-Verbose`.

ИТОГИ ГЛАВЫ

В этой главе мы разобрали особенности использования всех потоков вывода в PowerShell вдобавок к привычному потоку Success. В общей сложности вы изучили их семь: Success, Error, Warning, Verbose, Debug, Information и Progress. Каждый поток служит своей цели. Управлять им можно с помощью настроечных переменных и стандартных параметров. Вы узнали, как добавлять в функции подробный и предупреждающий вывод, благодаря которому можно предоставлять дополнительную информацию и выводить предупреждения в процессе выполнения скрипта. Кроме того, мы обсудили важность добавления подробных сообщений с самого начала разработки скрипта, что позволяет упростить последующую отладку и устранение проблем. Наконец, мы разобрали поток Information, который был добавлен в PowerShell v5, а также его возможности в плане создания структурированных сообщений. С помощью примеров и упражнений вы уяснили, как эффективно использовать эти потоки вывода в скриптах PowerShell, расширяя их функциональность и повышая качество пользовательского опыта.

14 Быстрая помощь: добавление комментариев

Одно из достоинств PowerShell — система справки. Подобно страницам `man` в Linux, файлы справки PowerShell предоставляют информацию, примеры, функции и не только. Создавая инструменты, мы стремимся сопровождать их вспомогательной информацией — и вам следует делать так же. Вы можете делать это двумя способами.

Первый и самый простой — добавление комментариев. Вы просто помещаете файлы справки в начало скриптов и функций, и PowerShell интерпретирует их соответственно. Второй вариант — создание внешнего файла справки, обычно в формате Markdown. Упростить создание таких файлов можно с помощью PlatyPS. В данном случае мы будем использовать более простой одноязычный вариант предоставления комментариев внутри функции.

14.1. ГДЕ РАЗМЕЩАТЬ СПРАВОЧНУЮ ИНФОРМАЦИЮ

Есть три места, в которых PowerShell будет искать ваши специально отформатированные комментарии, чтобы преобразовать их в справочные данные.

- *Непосредственно перед ключевым словом, открывающим вашу функцию, при условии, что между этой функцией и последним комментарием нет пустых строк*. Нам это место не нравится, и мы предпочитаем следующий вариант.
- *Просто внутри функции после открывающего слова* `function` *и перед фрагментами* `[CmdletBinding()]` *или* `Param`. Эта область нам нравится, поскольку при ее использовании проще перемещать справочную информацию о функции при копировании кода куда-то еще. Если вы используете редактор с возможностью сворачивания кода, то ваши комментарии тоже будут сворачиваться в функцию.

Как вы увидите в ходе своей работы, именно в этом месте большинство программистов размещают свою справочную информацию.

- *В виде последнего элемента функции перед закрывающей фигурной скобкой* }. Это место нам тоже не очень нравится, так как для будущего читателя удобнее, когда комментарии размещаются в начале функции.

14.2. НАЧАЛО

Со временем вы увидите, что в работе с комментариями нет ничего сложного. Чтобы помочь вам понять ее, разберем конкретный пример (листинг 14.1).

Листинг 14.1. Помощь в виде комментариев

```
function Get-MachineInfo {
<#
.SYNOPSIS
Извлекает конкретную информацию об одном или нескольких компьютерах, используя WMI
или CIM.
.DESCRIPTION
Эта команда использует WMI либо CIM для извлечения конкретной информации об одном
или нескольких компьютерах. Выполнять ее нужно от имени пользователя, имеющего
разрешение удаленно запрашивать CIM или WMI на целевых машинах. Вы можете задать
стартовый протокол (по умолчанию CIM) и указать, что в случае сбоя на отдельных
машинах может использоваться другой протокол.
.PARAMETER ComputerName
Одно или несколько имен компьютеров. При использовании WMI это могут быть
и IP-адреса. В случае CIM IP-адреса могут не работать.
.PARAMETER LogFailuresToPath
Путь и имя файла для записи имен компьютеров, с которыми не удалось установить
связь. Если этот параметр опустить, то записи в журнал вноситься не будут.
.PARAMETER Protocol
Допустимые значения: Wsman (использует CIM) или Dcom (использует WMI). Указанный
параметр будет применяться для всех машин. По умолчанию его значение установлено
как "Wsman".
.PARAMETER ProtocolFallback
Укажите этот параметр, чтобы автоматически попытаться использовать другой протокол,
если с помощью предустановленного связь с машиной установить не удается..
.EXAMPLE
Get-MachineInfo -ComputerName ONE,TWO,THREE
Этот пример будет опрашивать три машины.
.EXAMPLE
Get-ADComputer -filter * | Select -Expand Name | Get-MachineInfo
Этот пример попытается опросить все машины в AD.
#>
    [CmdletBinding()]
    Param(
        [Parameter(ValueFromPipeline=$True,
                   Mandatory=$True)]
```

```
        [Alias('CN','MachineName','Name')]
        [string[]]$ComputerName,
        [string]$LogFailuresToPath,
        [ValidateSet('Wsman','Dcom')]
        [string]$Protocol = "Wsman",
        [switch]$ProtocolFallback
    )
BEGIN {}
PROCESS {
    foreach ($computer in $computername) {
        if ($protocol -eq 'Dcom') {
            $option = New-CimSessionOption -Protocol Dcom
        } else {
            $option = New-CimSessionOption -Protocol Wsman
        }
        Write-Verbose "Connecting to $computer over $protocol"
        $session = New-CimSession -ComputerName $computer `
                                  -SessionOption $option
        Write-Verbose "Querying from $computer"
        $os_params = @{'ClassName'='Win32_OperatingSystem'
                       'CimSession'=$session}
        $os = Get-CimInstance @os_params
        $cs_params = @{'ClassName'='Win32_ComputerSystem'
                       'CimSession'=$session}
        $cs = Get-CimInstance @cs_params
        $sysdrive = $os.SystemDrive
        $drive_params = @{'ClassName'='Win32_LogicalDisk'
                          'Filter'="DeviceId='$sysdrive'"
                          'CimSession'=$session}
        $drive = Get-CimInstance @drive_params
        $proc_params = @{'ClassName'='Win32_Processor'
                         'CimSession'=$session}
        $proc = Get-CimInstance @proc_params |
                Select-Object -first 1
        Write-Verbose "Closing session to $computer"
        $session | Remove-CimSession
        Write-Verbose "Outputting for $computer"
        $obj = [pscustomobject]@{'ComputerName'=$computer
                   'OSVersion'=$os.version
                   'SPVersion'=$os.servicepackmajorversion
                   'OSBuild'=$os.buildnumber
                   'Manufacturer'=$cs.manufacturer
                   'Model'=$cs.model
                   'Procs'=$cs.numberofprocessors
                   'Cores'=$cs.numberoflogicalprocessors
                   'RAM'=($cs.totalphysicalmemory / 1GB)
                   'Arch'=$proc.addresswidth
                   'SysDriveFreeSpace'=$drive.freespace}
        Write-Output $obj
    } #foreach
} #PROCESS
END {}
} #function
```

Этот пример справки демонстрирует то, что, на наш взгляд, является необходимым минимумом. Здесь нужно учесть следующие моменты.

- Вам необязательно использовать только прописные буквы, но точка, с которой начинается любое ключевое слово справки (`.SYNOPSIS`, `.DESCRIPTION`), должна располагаться в первой позиции строки.
- Мы использовали блочный комментарий (`<#....#>`). Помимо этого, можно размещать слова построчно, предваряя каждую строку символом `#`. Этот блок комментариев выглядит более аккуратно и в некоторых редакторах скриптов трактуется как сворачиваемый.
- `.SYNOPSIS` предназначен для краткого описания действий команды.
- `.DESCRIPTION` содержит более подробное описание деталей, инструкций и полезной информации.
- `.PARAMETER` сопровождается именем параметра и далее описанием используемых параметров. При этом не нужно приводить описание для каждого отдельного параметра.
- `.EXAMPLE` должен сопровождаться самим примером. При выводе справки PowerShell будет добавлять перед этой строкой запрос к пользователю. Если ваш инструмент запрашивает информацию, скажем, в реестре, то можете вставить соответствующий запрос для иллюстрации примера. Последующий текст может объяснять этот пример.
- Между этими установками можно добавлять пустые строки комментариев, чтобы код было проще читать.
- Обычно нет нужды беспокоиться о длине строки. PowerShell при необходимости сворачивает строки в зависимости от размера консоли текущего хоста. Но если вы хотите разделить строки вручную, то лучше всего оставлять ширину 80 символов:

```
<#
.SYNOPSIS
Извлекает конкретную информацию об одном или нескольких компьютерах,
используя WMI или CIM.
.DESCRIPTION
Эта команда использует WMI либо CIM для извлечения конкретной информации об одном
или нескольких компьютерах. Выполнять ее нужно от имени пользователя, имеющего
разрешение удаленно запрашивать CIM или WMI на целевых машинах. Вы можете задать
стартовый протокол (по умолчанию CIM) и указать, что в случае сбоя на отдельных
машинах может использоваться другой протокол.
.PARAMETER ComputerName
Одно или несколько имен компьютеров. При использовании WMI это могут быть
и IP-адреса. В случае CIM IP-адреса могут не работать.
.PARAMETER LogFailuresToPath
Путь и имя файла для записи имен компьютеров, с которыми не удалось установить
связь. Если этот параметр опустить, то записи сделаны не будут.
.PARAMETER Protocol
```

```
Допустимые значения: Wsman (использует CIM) или Dcom (использует WMI).
Указанный параметр будет применяться для всех машин. По умолчанию его значение
установлено как "Wsman".
.PARAMETER ProtocolFallback
Укажите этот параметр, чтобы автоматически попытаться использовать другой
протокол, если с помощью предустановленного связь с машиной установить
не удается.
.EXAMPLE
Get-MachineInfo -ComputerName ONE,TWO,THREE
Этот пример опросит три машины.
.EXAMPLE
Get-ADUser -filter * | Select -Expand Name | Get-MachineInfo
Этот пример попытается опросить все машины в AD.
#>
```

Как мы уже говорили, это минимальный набор элементов. Вы можете указать намного больше.

14.3. РАСШИРЕННАЯ ПОМОЩЬ БЛАГОДАРЯ КОММЕНТАРИЯМ

Можно использовать раздел `.INPUTS` для построчного перечисления типов классов .NET, которые команда получает на входе из конвейера. Это поможет другим разработчикам понять, какой вид ввода ваша команда способна обработать:

```
.INPUTS
System.String
```

Аналогичным образом `.OUTPUTS` показывает имена типов, которые выводит ваш скрипт. Наш пока выводит только обобщенный `PSObject`, поэтому особого смысла в составлении данного списка нет.

В разделе `.NOTES` можно указывать дополнительную информацию, отображающуюся только в тот момент, когда пользователь запрашивает развернутую справку:

```
.NOTES
version     : 1.0.0
last updated: 1 February, 2023
```

Заголовок `.LINK`, сопровождаемый названием темы или URL, отражается в справке как Related Topic (связанная тема). Для каждой связанной темы следует использовать отдельное ключевое слово `.LINK`. Не размещайте под одним `.LINK` несколько тем:

```
.LINK
https://powershell.org/forums/
.LINK
Get-CimInstance
.LINK
Get-WmiObject
```

Есть и другие разделы справки, полный список которых можно найти в теме `about_comment_based_help` PowerShell. В последующих главах по мере расширения функциональности этих ключевых слов мы добавим еще несколько дополнительных вариантов ввода, так что будьте внимательны.

14.4. НЕРАБОТАЮЩАЯ СПРАВКА

PowerShell очень строг в отношении форматирования справки и ее синтаксиса. Достаточно сделать что-то одно неправильно, и она полностью перестанет работать. Причем никаких сообщений об ошибке или пояснений оболочка не предоставит. Так что если вы не получаете ожидаемой справки, то внимательно просмотрите, как написаны ключевые слова, расставлены точки, и прочие детали.

14.5. НЕ ТОЛЬКО КОММЕНТАРИИ

Справка на основе комментариев имеет множество ограничений, но очень важно понимать смысл ее существования. Изначально PowerShell поддерживал только внешнюю справку, которая хранилась в XML-файлах, написанных на диалекте Microsoft Assistance Markup Language (MAML). MAML очень сложен для понимания человеком, но справка в этом формате имеет некоторые преимущества перед справочными комментариями. Несмотря на сложность своего создания, MAML предлагает следующее.

- В нем информация отделена от кода, что позволяет выполнять независимое обновление, и при этом на ее основе в PowerShell работает команда `Update-Help`.
- Реализовать такую справку можно на разных языках, что позволяет PowerShell предоставлять ее локализованные версии для разной аудитории.
- PowerShell анализирует такую справку в виде иерархии объектов, предоставляя дополнительные возможности и функциональность, позволяющую создавать ценный контент в широком спектре ситуаций.

Итак, если MAML настолько хорош, но сложен в создании, как быть? В прошлом многие специалисты создавали инструменты для копирования содержимого в GUI, генерирующий файлы MAML. Этот подход проще, но требует очень много времени. Сегодня все опытные разработчики используют проект с открытым исходным кодом под названием PlatyPS. Он позволяет писать справку на простом языке разметки Markdown. Это нативный язык разметки для GitHub, то есть в случае размещения проекта на этом ресурсе ваши файлы справки будет легко читать и редактировать. PlatyPS может создавать на основе этих файлов рабочий файл MAML. А если вам по какой-то причине нужна веб-справка, то другие инструменты могут создавать на их основе HTML. Markdown становится исходным форматом для ваших справочных файлов (его легко читать и редактировать в любом текстовом редакторе — специальный редактор Markdown не требуется, хотя в VS Code есть

прекрасные плагины этого языка, которые вы можете попробовать), и далее все создается уже на их основе.

Если вы никогда не писали справку для своего кода, то PlatyPS может проанализировать ваш код и создать схему (или *заготовку*) для ваших справочных файлов Markdown. Эта заготовка будет содержать все параметры со всеми данными, которые PlatyPS сможет выяснить, — в том числе информацию о том, какие из этих параметров являются обязательными, какие получают ввод из конвейера и т. д. Кроме того, PlatyPS помогает *сопровождать* справочные файлы. Предположим, вы добавляете параметр, изменяете или делаете что-то еще. PlatyPS анализирует ваш код, определяет эти изменения и обновляет имеющиеся справочные файлы с помощью заготовок, которые далее вы можете полноценно заполнить, указав все изменения.

Мы любим PlatyPS и Markdown. Но это очень обширные темы, которые мы не готовы раскрыть в данной книге. Тем не менее нам хотелось хотя бы дать вам ориентиры, что стоит изучить самостоятельно.

14.6. УПРАЖНЕНИЯ

Пора добавить в вашу функцию справочные комментарии.

14.6.1. Вводная информация

В листинге 14.2 показано, на чем мы остановились в главе 13. Можете использовать в качестве отправной точки этот код или же собственные результаты из той же главы.

Листинг 14.2. Функция Set-TMServiceLogon

```
function Set-TMServiceLogon {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string]$ServiceName,
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string[]]$ComputerName,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewPassword,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewUser,
        [string]$ErrorLogFilePath
    )
```

```
BEGIN{}
PROCESS{
    ForEach ($computer in $ComputerName) {
        Write-Verbose "Connect to $computer on WS-MAN"
        $option = New-CimSessionOption -Protocol Wsman
        $session = New-CimSession -SessionOption $option `
                                  -ComputerName $Computer
        If ($PSBoundParameters.ContainsKey('NewUser')) {
            $args = @{'StartName'=$NewUser
                      'StartPassword'=$NewPassword}
        } Else {
            $args = @{'StartPassword'=$NewPassword}
            Write-Warning "Not setting a new user name"
        }
        Write-Verbose "Setting $servicename on $computer"
        $params = @{'CimSession'=$session
                    'MethodName'='Change'
                    'Query'="SELECT * FROM Win32_Service " +
                            "WHERE Name = '$ServiceName'"
                    'Arguments'=$args}
        $ret = Invoke-CimMethod @params
        switch ($ret.ReturnValue) {
            0  { $status = "Success" }
            22 { $status = "Invalid Account" }
            Default { $status = "Failed: $($ret.ReturnValue)" }
        }
        $props = @{'ComputerName'=$computer
                   'Status'=$status}
        $obj = New-Object -TypeName PSObject -Property $props
        Write-Output $obj
        Write-Verbose "Closing connection to $computer"
        $session | Remove-CimSession
    } #foreach
} #PROCESS
END{}
} #function
```

14.6.2. Ваша задача

Как минимум вам нужно добавить в инструмент следующие элементы:

- аннотацию;
- описание;
- описание параметров;
- два примера, в том числе описания.

Импортируйте ваш модуль и протестируйте справку (например, `Help Set-TMServiceLogon -ShowWindow`), чтобы убедиться в ее работоспособности.

14.6.3. Наше решение

В листинге 14.3 показана справка, которую реализовали мы. Как обычно, ее можно найти среди материалов для скачивания в папке текущей главы по адресу http://mng.bz/rjgE.

Листинг 14.3. Наше решение

```
function Set-TMServiceLogon {
<#
.SYNOPSIS
Устанавливает логин и пароль сервиса.
.DESCRIPTION
Эта команда использует для установки пароля сервиса CIM (по умолчанию) либо WMI
и необязательно имя пользователя (logon) для сервиса, который может выполняться
на одной или нескольких удаленных машинах.
Запускать эту команду нужно от имени пользователя, имеющего разрешение на
выполнение этой задачи удаленно на целевых компьютерах.
.PARAMETER ServiceName
Имя сервиса. Для проверки корректности имени запросите класс Win32_Service.
.PARAMETER ComputerName
Одно или несколько имен компьютеров. В случае CIM использование IP-адресов приведет
к сбою подключения. IP-адреса будут работать с WMI. Первой всегда пробуется CIM.
.PARAMETER NewPassword
Текстовая строка нового пароля.
.PARAMETER NewUser
Необязательно; новое имя пользователя (logon) в формате DOMAIN\USER.
.PARAMETER ErrorLogFilePath
Если указать этот параметр, то он будет содержать путь и имя текстового файла, куда
будут логироваться имена компьютеров, к которым не удалось подключиться.
#>
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string]$ServiceName,
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string[]]$ComputerName,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewPassword,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewUser,
        [string]$ErrorLogFilePath
    )
BEGIN{}
PROCESS{
    ForEach ($computer in $ComputerName) {
        Write-Verbose "Connect to $computer on WS-MAN"
        $option = New-CimSessionOption -Protocol Wsman
        $session = New-CimSession -SessionOption $option `
                                  -ComputerName $Computer
```

```
        If ($PSBoundParameters.ContainsKey('NewUser')) {
            $args = @{'StartName'=$NewUser
                      'StartPassword'=$NewPassword}
        } Else {
            $args = @{'StartPassword'=$NewPassword}
            Write-Warning "Not setting a new user name"
        }
        Write-Verbose "Setting $servicename on $computer"
        $params = @{'CimSession'=$session
                    'MethodName'='Change'
                    'Query'="SELECT * FROM Win32_Service " +
                            "WHERE Name = '$ServiceName'"
                    'Arguments'=$args}
        $ret = Invoke-CimMethod @params
        switch ($ret.ReturnValue) {
            0  { $status = "Success" }
            22 { $status = "Invalid Account" }
            Default { $status = "Failed: $($ret.ReturnValue)" }
        }
        $props = @{'ComputerName'=$computer
                   'Status'=$status}
        $obj = New-Object -TypeName PSObject -Property $props
        Write-Output $obj
        Write-Verbose "Closing connection to $computer"
        $session | Remove-CimSession
    } #foreach
} #PROCESS
END{}
} #function
```

Добавление справочных комментариев необязательно должно быть утомительным занятием. Используйте функциональность сниппетов в редакторе скриптов, чтобы создать шаблон.

Ну и в конце дадим профессиональный совет: помните, что справочные комментарии допускают лишние пустые строки. То есть вместо:

```
.SYNOPSIS
Устанавливает логин и пароль сервиса.
.DESCRIPTION
Эта команда использует для установки пароля сервиса CIM (по умолчанию) либо WMI
и необязательно имя пользователя (logon) для сервиса, который может выполняться
на одной или нескольких удаленных машинах.
Запускать эту команду нужно от имени пользователя, имеющего разрешение
на выполнение этой задачи удаленно на целевых компьютерах.
.PARAMETER ServiceName
Имя сервиса. Для проверки корректности имени запросите класс Win32_Service.
```

можно сделать так:

```
.SYNOPSIS
Устанавливает логин и пароль сервиса.
```

```
.DESCRIPTION

Эта команда использует для установки пароля сервиса CIM (по умолчанию) либо WMI
и необязательно имя пользователя для сервиса, который может выполняться на одной
или нескольких удаленных машинах.
Запускать эту команду нужно от имени пользователя, имеющего разрешение
на выполнение этой задачи удаленно на целевых компьютерах.

.PARAMETER ServiceName
Имя сервиса. Для проверки корректности имени запросите класс Win32_Service.
```

Эти дополнительные пробелы *существенно* облегчают чтение кода, не влияя на справочный файл, создаваемый из ваших комментариев.

ИТОГИ ГЛАВЫ

В этой главе мы поговорили о том, как важно внедрять вспомогательную документацию в скрипты и функции PowerShell. Для этого можно использовать справочные комментарии. Благодаря этому пользователи смогут понять, для чего служит ваш инструмент. Реализовать добавление такой документации можно двумя способами: с помощью справочных комментариев и внешних файлов справки. Справочные комментарии подразумевают внедрение в начало скриптов или функций особым образом отформатированных комментариев, которые PowerShell интерпретирует как файлы справки. Мы обозначили три специальных места, в которых PowerShell ищет эти комментарии внутри функций. Кроме того, мы продемонстрировали интеграцию справочных комментариев в функцию, взяв за пример `Get-MachineInfo`, которая содержит разделы `SYNOPSIS` (аннотация), `DESCRIPTION` (описание), `PARAMETER` (параметр) и `EXAMPLE` (пример), дающие целостное представление о назначении функции, ее параметрах и способах применения. При этом мы подчеркнули важность корректного форматирования, упомянув, что ключевые слова пишутся прописными буквами, комментарии могут быть многострочными, а для улучшения читабельности кода лучше добавлять пустые строки. Кроме того, мы рассмотрели расширенные темы, такие как разделы кода под названием `.INPUTS`, `.OUTPUTS`, `.NOTES` и `.LINK`, в которых можно размещать дополнительную информацию, и поговорили о том, как исправить неработающее форматирование справки. В финале главы мы кратко познакомили вас с инструментом PlatyPS, позволяющим генерировать содержимое справки в формате Markdown, который можно преобразовывать в файлы MAML, что позволяет увеличить гибкость и простоту использования. PlatyPS и Markdown предоставляют больше расширенных функций, однако их рассмотрение выходит за рамки темы текущей главы. И наконец, мы показали примеры разного оформления комментариев.

15 Ошибки и их обработка

Вам еще предстоит добавить в ваш инструмент много функциональности, и до текущего момента мы сильно отклонялись от этой задачи. В данной же главе мы сосредоточимся на перехвате, устранении, логировании и другой обработке ошибок, которая может потребоваться нашему инструменту.

ПРИМЕЧАНИЕ

На PowerShell.org есть бесплатная электронная книга *The Big Book of PowerShell Error Handling*, в которой эта тема раскрывается с более технической стороны (https://PowerShell.org/free-resources/). Советуем вам ознакомиться с ней после того, как закончите чтение главы.

15.1. ЧТО ТАКОЕ ОШИБКИ И ИСКЛЮЧЕНИЯ

В PowerShell возможно два общих вида нежелательных ситуаций: ошибки и исключения. Большинство команд оболочки созданы для одновременной работы с несколькими элементами, и проблема с одним из них не означает, что нужно перестать обрабатывать все остальные, в итоге PowerShell старается «продолжать работать до окончательного сбоя». Поэтому зачастую в случае неполадки оболочка просто выдает ошибку и продолжает выполнение. Например:

```
Get-Service -Name BITS, Nobody, WinRM
```

Сервиса с названием `Nobody` не существует, в связи с чем PowerShell выдаст в отношении второго элемента ошибку (обратите внимание на серый текст на рис. 15.1). Но по умолчанию оболочка продолжит обрабатывать третий элемент списка. Когда PowerShell находится в таком режиме «продолжать до последнего», вы *не можете* заставить код реагировать на проблемное состояние. Если вы хотите как-то

решить проблему, то нужно изменить предустановленную реакцию PowerShell на *-terminating error*.

```
PS C:\tools> Get-Service -Name BITS, Nobody, WinRM
Get-Service: Cannot find any service with service name 'Nobody'.

Status   Name               DisplayName
------   ----               -----------
Stopped  BITS               Background Intelligent Transfer Servi…
Running  WinRM              Windows Remote Management (WS-Managem…
```

Рис. 15.1. Сообщение об ошибке касательно сервиса Nobody

На глобальном уровне PowerShell определяет переменную `$ErrorActionPreference`, а та сообщает оболочке, что делать при возникновении ошибки, которая не ведет к прекращению выполнения. По этой переменной PowerShell понимает, что делать в случае проблемы, когда выполнение может продолжаться. По умолчанию для этой переменной установлено значение `Continue`. Ниже приводится пояснение данного значения наряду с несколькими другими вариантами.

- `Continue` — генерируется сообщение об ошибке, и выполнение продолжается. Ваш код не может понять, что произошла ошибка, поэтому ничего другого вы сделать не можете.
- `SilentlyContinue` — сообщение об ошибке не генерируется, и выполнение продолжается. Опять же, вы не можете определить проблему или самостоятельно на нее среагировать.
- `Inquire` — выводится запрос к пользователю: остановить выполнение или продолжить?
- `Stop` — непрерывающая *ошибка* превращается в *прерывающее исключение*, и выполнение команды прекращается. В этом случае код может обнаружить проблему и среагировать на нее.
- `Ignore` — не является значением для этой переменной, но может использоваться в параметре `-Error-Action`, который мы вскоре разберем. Поведение данного значения аналогично поведению `SilentlyContinue`.
- `Suspend` — применяется только в рабочем потоке PowerShell, знакомство с которым выходит за рамки этой книги.

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Выполните $ErrorActionPreference из окна PowerShell. По умолчанию данная переменная должна быть установлена на Continue, если до этого ее никто не менял.

Самым лучшим вариантом считается оставлять `$ErrorActionPreference` нетронутой. Вместо этого обычно нужно указывать поведение для каждой отдельной команды. Это можно делать с помощью стандартного параметра `-ErrorAction` или

его псевдонима (`-EA`, используется здесь в целях экономии пространства страницы), который есть у каждой команды PowerShell — даже тех, которые вы пишете сами с использованием `[CmdletBinding()]`. Например, попробуйте выполнить следующие команды и обратите внимание на различия в их поведении:

```
Get-Service -Name BITS, Nobody, WinRM -EA Continue
Get-Service -Name BITS, Nobody, WinRM -EA SilentlyContinue
Get-Service -Name BITS, Nobody, WinRM -EA Inquire
Get-Service -Name BITS, Nobody, WinRM -EA Ignore
Get-Service -Name BITS, Nobody, WinRM -EA Stop
```

Напомню, что вы *не можете* обрабатывать исключения в коде, *пока* PowerShell их не сгенерирует. И большинство команд этого не сделают, если не будут выполнены с применением действия `Stop` в случае сбоя. Одна из самых серьезных ошибок, которую обычно можно допустить, — это забыть добавить в команду `-EA Stop` в тех местах, где нужно обработать проблему.

15.2. НЕУДАЧНАЯ ОБРАБОТКА

Мы нередко видим, как разработчики применяют две фундаментально нежелательные практики. Они, конечно, не всегда являются *нежелательными*, но *чаще* это так, поэтому мы хотим привлечь к ним ваше внимание.

Первая — это глобальная установка настроечной переменной в верхней части скрипта или функции:

```
$ErrorActionPreference='SilentlyContinue'
```

В бытность работы с VBScript разработчики использовали сообщение `On Error Resume Next`. Оно означает следующее: «Если с моим кодом что-то не так, то я не хочу ничего об этом знать». Разработчики делают это по заблуждению, пытаясь проигнорировать ошибки, которые не имеют особого значения. Например, попытка удалить несуществующий файл вызовет ошибку — но вам наверняка это неважно, поскольку задача в любом случае выполнена, так ведь? Но чтобы проигнорировать эту незначительную ошибку, вам следует использовать в команде `Remove-Item` параметр `-EA SilentlyContinue`, а не глобально игнорировать *все* ошибки в вашем скрипте.

Вторая нежелательная практика менее очевидна и может возникать в той же ситуации. Предположим, вы выполняете `Remove-Item` с `-EA SilentlyContinue`, а затем пытаетесь удалить существующий файл, но не имеете на это разрешения. Вы проигнорируете ошибку и будете удивляться, почему файл не был удален.

Прежде чем начать игнорировать ошибки, тщательно все продумайте. Ничто так не раздражает, как длительная и утомительная отладка скрипта, вызванная лишь тем, что вы скрыли сообщение об ошибке, которое бы указало на источник проблемы. И мы уже потеряли счет случаям, когда подобные нюансы всплывают среди вопросов на форумах.

15.3. ДВЕ ПРИЧИНЫ ДЛЯ ОБРАБОТКИ ИСКЛЮЧЕНИЙ

Для обработки исключений в коде есть две веские причины. (Заметьте, что под исключениями мы понимаем сбои, которые можно обработать. Таким образом мы отличаем их от необрабатываемых *ошибок*, о которых говорилось ранее.)

Первая причина состоит в том, что ваш инструмент будет работать вне вашего ведения. Возможно, это запланированная задача, или вы пишете инструменты, которые будут использовать удаленные потребители. В любом случае вам нужно сделать так, чтобы любые возникающие проблемы были очевидны, что позволит упростить их отладку. В данном сценарии вы можете установить `$ErrorActionPreference` глобально, чтобы обернуть весь скрипт в единую конструкцию по обработке ошибок. Любые ошибки, даже непредсказуемые, можно перехватить и занести в журнал, чтобы позже провести диагностику. И хотя это допустимый сценарий, в данной книге мы будем рассматривать не его.

Мы сосредоточимся на второй причине: когда вы при выполнении команды можете предвидеть определенную проблему и хотите оперативно ее обработать. Это может быть безуспешное подключение к компьютеру, сбой аутентификации где-либо или какой-то иной сценарий. Разберем все это на примере создаваемого вами инструмента.

15.4. ОБРАБОТКА ИСКЛЮЧЕНИЙ В ВАШЕМ ИНСТРУМЕНТЕ

Вы можете ожидать, что в команде `New-CimSession` создаваемого нами инструмента возникнут проблемы: компьютер может оказаться не в сети, не поддерживать выбранный протокол или не существовать в принципе. Вам нужно перехватить это состояние и в зависимости от использованных параметров занести информацию об этом компьютере в текстовый файл и/или повторить попытку, используя другой протокол. Для начала нужно сосредоточиться на команде, которая вызвала проблему, и сделать так, чтобы в случае сбоя она генерировала *прерывающее исключение*. Для этого измените данный фрагмент:

```
Write-Verbose "Connecting to $computer over $protocol"
$session = New-CimSession -ComputerName $computer `
                          -SessionOption $option
```

на:

```
Write-Verbose "Connecting to $computer over $protocol"
$params = @{'ComputerName'=$Computer
            'SessionOption'=$option
            'ErrorAction'='Stop'}
$session = New-CimSession @params
```

Здесь важно отметить, что вы уже создали команду так, чтобы она пыталась подключиться только к одному компьютеру одновременно, используя цикл `ForEach`. Каждый раз, когда вы будете обрабатывать ошибки, важно создавать все так, чтобы сбой мог единовременно произойти только в одном месте. Дело в том, что мы просим PowerShell прекратить выполнение, и если пробовать подключиться одновременно к пяти компьютерам, то сбой в любом из них исключит попытку подключения к остальным. Поэтому очень важно уяснить описанный принцип проектирования.

Тем не менее недостаточно просто изменить действие при ошибке на `Stop`. Кроме того, вам нужно обернуть код в конструкцию `Try/Catch`. Если в блоке `Try` произойдет исключение, то весь его последующий код будет пропущен, и вместо этого выполнится блок `Catch`. Поэтому теперь блок `PROCESS{}` функции выглядит так:

```
PROCESS {
    ForEach ($computer in $ComputerName) {
        if ($protocol -eq 'Dcom') {
            $option = New-CimSessionOption -Protocol Dcom
        } else {
            $option = New-CimSessionOption -Protocol Wsman
        }
        Try {                                    Блок Try скрипта
            Write-Verbose "Connecting to $computer over $protocol"
            $params = @{'ComputerName'=$Computer
                        'SessionOption'=$option
                        'ErrorAction'='Stop'}
        $session = New-CimSession @params
        Write-Verbose "Querying from $computer"
        $os_params = @{'ClassName'='Win32_OperatingSystem'
                       'CimSession'=$session}
        $os = Get-CimInstance @os_params
        $cs_params = @{'ClassName'='Win32_ComputerSystem'
                      'CimSession'=$session}
        $cs = Get-CimInstance @cs_params
        $sysdrive = $os.SystemDrive
        $drive_params = @{'ClassName'='Win32_LogicalDisk'
                          'Filter'="DeviceId='$sysdrive'"
                          'CimSession'=$session}
        $drive = Get-CimInstance @drive_params
        $proc_params = @{'ClassName'='Win32_Processor'
                         'CimSession'=$session}
        $proc = Get-CimInstance @proc_params |
                Select-Object -first 1
        Write-Verbose "Closing session to $computer"
        $session | Remove-CimSession
        Write-Verbose "Outputting for $computer"
        $obj = [pscustomobject]@{'ComputerName'=$computer
                   'OSVersion'=$os.version
                   'SPVersion'=$os.servicepackmajorversion
```

```
                    'OSBuild'=$os.buildnumber
                    'Manufacturer'=$cs.manufacturer
                    'Model'=$cs.model
                    'Procs'=$cs.numberofprocessors
                    'Cores'=$cs.numberoflogicalprocessors
                    'RAM'=($cs.totalphysicalmemory / 1GB)
                    'Arch'=$proc.addresswidth
                    'SysDriveFreeSpace'=$drive.freespace}
                Write-Output $obj
            } Catch {
            } #try/catch
        } #foreach
} #PROCESS
```

Блок Catch скрипта

Идея в том, чтобы при возникновении проблемы с `New-CimSession` отменялось *все остальное*. И это разумно, поскольку без сеанса невозможно выполнять запросы, без них — генерировать результаты, а без результатов — получать вывод. Если сбой происходит в чем-то одном, то нужно останавливать все.

Теперь разберем, что мы делаем в случае ошибки — ах нет, *исключения*:

```
} Catch {
    Write-Warning "FAILED $computer on $protocol"
    # Указали ли мы резервный протокол?
    # Если да, повторить попытку.
    # Если мы указали логирование проблемы, то здесь
    # логировать ее не будем. Мы сделаем это,
    # если резервный протокол тоже не сработает.
    If ($ProtocolFallback) {
        If ($Protocol -eq 'Dcom') {
            $newprotocol = 'Wsman'
        } else {
            $newprotocol = 'Dcom'
        } #if protocol
        Write-Verbose "Trying again with $newprotocol"
        $params = @{'ComputerName'=$Computer
                    'Protocol'=$newprotocol
                    'ProtocolFallback'=$False}
        If ($PSBoundParameters.ContainsKey('LogFailuresToPath')){
            $params += @{'LogFailuresToPath'=$LogFailuresToPath}
        } #if logging
        Get-MachineInfo @params
    } #if protocolfallback
    # Если мы не указывали резервный протокол, но указали логирование,
    # тогда логируем ошибку, так как повторять попытку не будем
    If (-not $ProtocolFallback -and
        $PSBoundParameters.ContainsKey('LogFailuresToPath')){
        Write-Verbose "Logging to $LogFailuresToPath"
        $computer | Out-File $LogFailuresToPath -Append
    } # если запись в журнал
} #try/catch
```

Проверяем параметры

Записываем предупреждение

Резервный протокол не задан, выполняется логирование

Разберем, что происходит в этом коде.

1. В блоке `Catch` можно для удобства пользователей записать предупреждающее сообщение. Они смогут его заглушить, сопроводив выполнение команды параметром `-Warning-Action SilentlyContinue`.
2. Выполняется проверка, был ли указан `-ProtocolFallback`. Если да, то в `$newprotocol` устанавливается протокол, который не использовался при выполнении. После этого устанавливается хеш-таблица параметров с текущими именем компьютера и новым протоколом, а также указывается `$False` для `ProtocolFallback`. Вы *уже использовали резервный* протокол, поэтому нет смысла пробовать его снова и входить в бесконечный цикл. Если в процессе вы используете параметр `-LogFailuresToPath`, то добавьте его в хеш-таблицу и — самое интересное — *вызовите всю функцию* с этими параметрами. Ее вывод станет частью *вашего* вывода, что позволит вам легко использовать другой протокол, не дублируя большое количество кода.
3. Проводится проверка на отсутствие `-ProtocolFallback` и выполнение с `-LogFailuresToPath`, чтобы вы могли логировать имя компьютера, на котором произошел сбой. Почему бы не логировать имя компьютера изначально? Если *текущий* протокол не срабатывает, но вас просят использовать резервный, то автоматический вызов к `Get-MachineInfo` произведет логирование в случае сбоя и второго протокола.

Это непростая логика. Просмотрите ее несколько раз, чтобы как следует понять происходящее.

15.5. ПЕРЕХВАТ ИСКЛЮЧЕНИЯ

В приведенном выше примере не учитывалось, какая именно проблема произошла с `New-CimSession`, и мы одинаково реагировали на любой сбой. В некоторых же случаях вам может быть интересно, какое конкретно исключение произошло. Простой способ выяснить это — указать параметр `ErrorVariable` (или `-EV`) и имя переменной (`$` к имени не относится, поэтому здесь данный символ опускается). В результате исключение будет помещено в указанную переменную, чтобы у вас была возможность использовать его в дальнейшей работе.

15.6. ОБРАБОТКА ИСКЛЮЧЕНИЙ НЕ ТОЛЬКО ДЛЯ КОМАНД

А если вы выполняете что-то — например, метод .NET Framework — без параметра `-ErrorAction`? *Чаще всего* его можно выполнить в блоке `Try` как есть, поскольку *большинство* таких методов при возникновении проблемы будут выбрасывать перехватываемые, прерывающие выполнение исключения. Непрерывающее

исключение является достаточно уникальным случаем для таких команд PowerShell, как функции и командлеты.

Но у вас *все равно* могут возникать ситуации, где потребуется выполнить следующие действия:

```
Try {
    $ErrorActionPreference = "Stop"
    # Выполнение чего-то, что не содержит -ErrorAction
    $ErrorActionPreference = "Continue"
} Catch {
    # ...
}
```

Это крайняя мера обработки ошибки. Здесь вы изменяете `$Error-ActionPreference` на время выполнения одной команды (или чего-то иного), в процессе которого хотите перехватить исключение. Исходя из нашего опыта, можем сказать, что это довольно редкая ситуация, но мы все же решили упомянуть о ней.

15.7. БОЛЕЕ СЛОЖНАЯ ОБРАБОТКА ИСКЛЮЧЕНИЙ

После блока `Try` вполне может идти несколько блоков `Catch`, каждый из которых будет обрабатывать конкретный вид исключения. Например, если удалить файл не удалось, то вы можете по-разному отреагировать на ситуации File Not Found или Access Denied. Для этого вам нужно знать название типа каждого исключения в .NET Framework, которое вы хотите перехватывать отдельно. В электронной книге *The Big Book of PowerShell Error Handling* (https://PowerShell.org/free-resources/) приводятся названия наиболее распространенных типов исключений и рекомендации по их определению (например, намеренная генерация ошибки в качестве эксперимента и выяснение типа возникшего исключения). В общем виде синтаксис для подобного случая будет выглядеть так:

```
Try {
    # Что-то здесь генерирует исключение
} Catch [Exception.Type.One] {
    # Здесь это исключение обрабатывается
} Catch [Exception.Type.Two] {
    # Здесь обрабатывается другое исключение
} Catch {
    # Здесь обрабатывается все остальное
} Finally {
    # Выполнение чего-то другого
}
```

В этом списке показан и необязательный блок `Finally`, который вне зависимости от возникновения исключения всегда выполняется после `Try` или `Catch`.

УСТАРЕВШИЙ СПОСОБ ОБРАБОТКИ ОШИБОК

В Интернете вы можете встретить конструкцию `Trap`. Она появилась в v1, когда команда PowerShell просто не имела времени наладить работу `Try/Catch`, и `Trap` стала лучшим краткосрочным исправлением, который смогли придумать разработчики. Сегодня `Trap` является *устаревшим решением*, то есть сохраняется в оболочке для обратной совместимости, но использовать его в новом коде нежелательно. Собственно, поэтому мы не будем разбирать данную конструкцию на страницах нашей книги. В глобальном контексте есть некоторые варианты ее применения, когда вам нужно перехватывать и логировать все возможные ошибки. Но `Try/Catch` считается более структурированным и профессиональным подходом к обработке исключений, и мы советуем вам придерживаться именно его.

15.8. УПРАЖНЕНИЯ

Пришло время заняться ошибками в вашем коде.

15.8.1. Вводная информация

В листинге 15.1 показан код, на котором мы остановились в конце главы 14. Можете использовать в качестве отправной точки его или собственные результаты из той же главы.

Листинг 15.1. Set TMServiceLogon

```
function Set-TMServiceLogon {
<#
.SYNOPSIS
Устанавливает логин и пароль сервиса.
.DESCRIPTION
Эта команда использует для установки пароля сервиса CIM (по умолчанию) либо WMI
и необязательно имя пользователя (logon) для сервиса, который может выполняться
на одной или нескольких удаленных машинах.
Запускать эту команду нужно от имени пользователя, имеющего разрешение
на выполнение этой задачи удаленно на целевых компьютерах.
.PARAMETER ServiceName
Имя сервиса. Для проверки корректности имени запросите класс Win32_Service.
.PARAMETER ComputerName
Одно или несколько имен компьютеров. В случае CIM использование IP-адресов приведет
к сбою подключения. IP-адреса будут работать с WMI. Первой всегда пробуется CIM.
.PARAMETER NewPassword
Текстовая строка нового пароля.
.PARAMETER NewUser
 Необязательно; новое имя пользователя (logon) в формате DOMAIN\USER.
.PARAMETER ErrorLogFilePath
```

```
Если указать этот параметр, то он будет содержать путь и имя текстового файла, куда
будут логироваться имена компьютеров, к которым не удалось подключиться.
#>
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string]$ServiceName,
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string[]]$ComputerName,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewPassword,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewUser,
        [string]$ErrorLogFilePath
    )
BEGIN{}
PROCESS{
    ForEach ($computer in $ComputerName) {
        Write-Verbose "Connect to $computer on WS-MAN"
        $option = New-CimSessionOption -Protocol Wsman
        $session = New-CimSession -SessionOption $option `
                                  -ComputerName $Computer
        If ($PSBoundParameters.ContainsKey('NewUser')) {
            $args = @{'StartName'=$NewUser
                      'StartPassword'=$NewPassword}
        } Else {
            $args = @{'StartPassword'=$NewPassword}
            Write-Warning "Not setting a new user name"
        }
        Write-Verbose "Setting $servicename on $computer"
        $params = @{'CimSession'=$session
                    'MethodName'='Change'
                    'Query'="SELECT * FROM Win32_Service " +
                            "WHERE Name = '$ServiceName'"
                    'Arguments'=$args}
        $ret = Invoke-CimMethod @params
        switch ($ret.ReturnValue) {
            0  { $status = "Success" }
            22 { $status = "Invalid Account" }
            Default { $status = "Failed: $($ret.ReturnValue)" }
        }
        $props = @{'ComputerName'=$computer
                   'Status'=$status}
        $obj = New-Object -TypeName PSObject -Property $props
        Write-Output $obj
        Write-Verbose "Closing connection to $computer"
        $session | Remove-CimSession
```

```
    } #foreach
} #PROCESS
END{}
} #function
```

15.8.2. Ваша задача

Вам нужно добавить в свой инструмент обработку ошибок. Напомним, что в случае ошибки наша структура подразумевает автоматическое использование протокола объектной модели распределенных компонентов (DCOM), поскольку начинаете вы всегда с протокола Web Services Management (WSman). И если при соединении с компьютером произошел сбой, то вам следует логировать его имя, *только* если это действие было прописано и *только* после попытки использования обоих протоколов.

Ваша задача слегка усложняется, поскольку в схеме параметров нет параметра для данного протокола. Это означает, что вы не можете просто повторно вызвать собственную функцию с другим параметром протокола. Вместо этого вам придется написать *цикл*, который будет выполнять ваш код до двух раз. Один такой цикл может выглядеть так:

```
Do {
    # Код
} Until ($something -eq 'else')
```

Подобный цикл будет всегда выполняться не менее одного раза. Он будет продолжать выполнение, *пока* указанное в его конце условие не окажется `$True`. Посмотрите, сможете ли найти логику, которую необходимо добавить в скрипт.

15.8.3. Наше решение

У нас получилось такое решение (листинг 15.2).

Листинг 15.2. Наше решение

```
function Set-TMServiceLogon {
<#
.SYNOPSIS
Устанавливает логин и пароль сервиса.
.DESCRIPTION
Эта команда использует для установки пароля сервиса CIM (по умолчанию) либо WMI
и необязательно имя пользователя (logon) для сервиса, который может выполняться
на одной или нескольких удаленных машинах.
Запускать эту команду нужно от имени пользователя, имеющего разрешение
на выполнение этой задачи удаленно на целевых компьютерах.
.PARAMETER ServiceName
Имя сервиса. Для проверки корректности имени запросите класс Win32_Service.
.PARAMETER ComputerName
```

```
Одно или несколько имен компьютеров. В случае CIM использование IP-адресов приведет
к сбою подключения. IP-адреса будут работать с WMI. Первой всегда пробуется CIM.
.PARAMETER NewPassword
Текстовая строка нового пароля.
.PARAMETER NewUser
Необязательно; новое имя пользователя (logon) в формате DOMAIN\USER.
.PARAMETER ErrorLogFilePath
Если указать этот параметр, то он будет содержать путь и имя текстового файла,
куда будут логироваться имена компьютеров, к которым не удалось подключиться.
#>
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string]$ServiceName,
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string[]]$ComputerName,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewPassword,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewUser,
        [string]$ErrorLogFilePath
    )
BEGIN{}
PROCESS{
    ForEach ($computer in $ComputerName) {
        Do {
            Write-Verbose "Connect to $computer on WS-MAN"
            $protocol = "Wsman"
            Try {
                $option = New-CimSessionOption -Protocol $protocol
                $session = New-CimSession -SessionOption $option `
                                          -ComputerName $Computer `
                                          -ErrorAction Stop
                If ($PSBoundParameters.ContainsKey('NewUser')) {
                    $args = @{'StartName'=$NewUser
                              'StartPassword'=$NewPassword}
                } Else {
                    $args = @{'StartPassword'=$NewPassword}
                    Write-Warning "Not setting a new user name"
                }
                Write-Verbose "Setting $servicename on $computer"
                $params = @{'CimSession'=$session
                            'MethodName'='Change'
                            'Query'="SELECT * FROM Win32_Service " +
                                    "WHERE Name = '$ServiceName'"
                            'Arguments'=$args}
                $ret = Invoke-CimMethod @params
                switch ($ret.ReturnValue) {
                    0  { $status = "Success" }
                    22 { $status = "Invalid Account" }
```

```
                    Default { $status = "Failed: $($ret.ReturnValue)" }
                }
                $props = @{'ComputerName'=$computer
                           'Status'=$status}
                $obj = New-Object -TypeName PSObject -Property $props
                Write-Output $obj
                Write-Verbose "Closing connection to $computer"
                $session | Remove-CimSession
            } Catch {
                # Изменение протокола – если были опробованы оба и было указано
                # логирование, тогда логируется компьютер,
                # с которым не удалось связаться
                Switch ($protocol) {
                    'Wsman' { $protocol = 'Dcom' }
                    'Dcom' {
                        $protocol = 'Stop'
                            if ($PSBoundParameters.ContainsKey(
➥ 'ErrorLogFilePath')) {
                            Write-Warning "$computer failed; logged to
➥ $ErrorLogFilePath"
                            $computer | Out-File $ErrorLogFilePath -Append
                        } # if logging
                    }
                } #switch
            } # try/catch
        } Until ($protocol -eq 'Stop')
    } #foreach
} #PROCESS
END{}
} #function
```

Приносим извинения за сокращения слов. Полная версия кода доступна для скачивания по ссылке http://mng.bz/rjgE.

В этой версии мы изменили `New-CimSessionOption` на использование переменной для протокола. Мы вручную установили ее базовое значение `"Wsman"`, но в случае сбоя переключаемся на `"Dcom"`. Если сбой повторяется, то мы даем команду `Stop`, которая вызывает выход из цикла `Do`. Помимо этого, мы используем возможность логировать имя компьютера, если есть соответствующий запрос.

ИТОГИ ГЛАВЫ

В главе 15 мы перешли к теме обработки ошибок внутри создаваемого инструмента. Здесь вы научились перехватывать, обрабатывать и логировать ошибки, возникшие в процессе выполнения. Мы выявили различие между ошибками и исключениями в PowerShell, а также обсудили предустановленное поведение оболочки, которое заключается в том, что выполнение продолжается независимо от возникающих ошибок. Кроме того, мы подчеркнули, насколько важно понимать суть переменной `$ErrorActionPreference` и уметь вносить в нее изменения — это позволяет управлять

реакцией PowerShell на ошибки. Мы рассмотрели различные способы обработки ошибок, такие как `Stop`, `Continue` и `SilentlyContinue`, и разобрали практические примеры их применения. Помимо этого, мы познакомили вас с лучшими практиками обработки ошибок, такими как избегание глобального игнорирования и рассмотрение возможности их логирования. Затронули мы и расширенные принципы наподобие обработки исключений и указания в командах PowerShell действий на случай ошибок. Все это должно было сформировать у вас полноценное представление об обработке ошибок в скриптах PowerShell. Наконец, мы поставили перед вами задачу реализовать обработку ошибок внутри некоего кода. Эту задачу мы сопроводили решением, в котором была показана изящная обработка ошибок при выполнении команд на нескольких компьютерах. Используя все теоретические описания и примеры, вы должны были научиться повышать надежность ваших скриптов PowerShell за счет эффективной обработки ошибок.

16 Заполняем манифест

До этого момента выполнение команд внутри модуля PowerShell вы реализовывали возможностями PowerShell. И сейчас пришло время более подробно изучить их, так как они гораздо шире, чем вы можете себе представить.

16.1. ПОРЯДОК ВЫПОЛНЕНИЯ МОДУЛЕЙ

Когда PowerShell ищет модули, она сначала перебирает все папки, указанные в переменной среды `PSModulePath`. Каждая папка в каждом из этих путей считается потенциальным модулем.

Внутри папки модуля PowerShell ищет следующие элементы.

1. Файл `.psd1` с тем же именем, что и у папки модуля. Манифест этого модуля сообщает оболочке, что еще нужно загрузить.
2. Файл `.dll` с тем же именем, что и у папки модуля. Это *компилируемый* или *двоичный* модуль, обычно написанный на C#.
3. Файл `.psm1` с тем же именем, что и у папки модуля. Это *модуль скрипта*.

Вы использовали третий пункт из этого списка. Если вы создаете файл `\Documents\PowerShell\Modules\MyPSModule\MyPSModule.psm1`, то создаете модуль скрипта `MyPSModule`, и находящиеся в этом файле `.psm1` функции становятся командами, которые PowerShell сможет выполнять. Это очень быстрый и простой способ запустить модуль, но у него есть свои недостатки.

Во-первых, модуль не может легко обрабатывать такие действия, как версионирование, настройка необходимых предварительных условий (пререквизитов)

и загрузка сопутствующих файлов (например, пользовательских представлений форматов, о которых мы будем говорить позже). Постепенно по мере усложнения и периодического перебора модулей вам понадобится делать это все.

Во-вторых, модуль со временем расширяется и содержит множество команд, поэтому может замедлять работу PowerShell — даже если вы его не используете. Дело в том, что в момент запуска PowerShell нужно выяснить, какие модули у вас есть и какие команды они содержат. Для отдельного модуля скрипта это означает загрузку и парсинг (синтаксический анализ) *всего файла*, чтобы определить, какие функции в нем находятся. Парсинг занимает время, и если модули более крупные или их много, то такой анализ может затягиваться, что вызывает замедление работы PowerShell при каждом запуске нового окна оболочки.

Манифест (пункт 1 приведенного выше списка) решает эти проблемы, поскольку позволяет указывать подробную дополнительную информацию о вашем модуле. При правильном применении манифест может значительно ускорять обнаружение модулей.

16.2. СОЗДАНИЕ МАНИФЕСТА

Создать новый простой манифест легко. Нужно перейти в папку модуля и выполнить `New-ModuleManifest`. Укажите имя файла манифеста (которое будет совпадать с именем папки модуля, но иметь разрешение `.psd1`), а также задайте в качестве *корневого модуля* имеющийся модуль скрипта `.psm1`:

```
New-ModuleManifest -Path MyModule.psd1 -Root ./MyModule.psm1
```

ВНИМАНИЕ

PowerShell никак не проверяет корректность вашего ввода. Опечатка в любом из этих путей приведет к созданию нерабочего манифеста и может стать причиной, по которой весь модуль перестанет загружаться.

Этот пример предполагает, что вы находитесь в каталоге `MyModule`, в связи с чем официальным именем модуля станет `MyModule`. Результат будет примерно следующим (вам нужно воссоздать его, чтобы вы могли следить за нитью повествования). Автоматически сгенерированные комментарии для каждого раздела помогут разобраться в указанной информации:

```
#
# Манифест для модуля 'MyModule'
#
# Сгенерировал: User
#
# Сгенерирован: 7/23/2023 #
```

```
@{

# Модуль скрипта или двоичный файл модуля, связанный с этим манифестом.
RootModule = './MyModule.psm1'

# Версия этого модуля. ModuleVersion = '0.0.1'

# Поддерживаемые PSEditions
# CompatiblePSEditions = @()

# ID для уникальной идентификации этого модуля
GUID = 'ce7775f9-e168-48d6-8e8f-f4c04696d673'

# Автор модуля
Author = 'User'

# Компания или производитель модуля
CompanyName = 'Unknown'

# Заявленные авторские права на этот модуль
Copyright = '(c) User. All rights reserved.'

# Описание функциональности, предоставляемой модулем
# Description = ''

# Минимальная версия PowerShell, необходимая модулю
# PowerShellVersion = ''

# Имя хоста PowerShell, необходимого этому модулю
# PowerShellHostName = ''

# Минимальная версия PowerShell на хосте, необходимая этому модулю
# PowerShellHostVersion = ''

# Минимальная версия Microsoft .NET Framework, необходимая модулю. Это требование
актуально только для PowerShell Desktop.
# DotNetFrameworkVersion = ''

# Минимальная версия необходимой модулю общеязыковой среды выполнения (common
language runtime, CLR). Это требование актуально только для PowerShell Desktop.
# ClrVersion = ''

# Архитектуры процессоров (None, X86, Amd64), необходимые модулю
# ProcessorArchitecture = ''

# Модули, которые необходимо импортировать в глобальную переменную перед импортом
этого модуля
# RequiredModules = @()

# Сборки, которые необходимо загрузить перед импортом этого модуля
# RequiredAssemblies = @()
```

```
# Файлы скриптов (.ps1), выполняемые в среде вызывающего компонента перед
импортом модуля
# ScriptsToProcess = @()

# Файлы типов (.ps1xml), которые загружаются при импорте этого модуля
# TypesToProcess = @()

# Файлы форматов (.ps1xml), которые загружаются при импорте этого модуля
# FormatsToProcess = @()

# Модули, которые импортируются как вложенные в модуль, указанный
в RootModule/ModuleToProcess
# NestedModules = @()

# Функции для экспорта из этого модуля. Чтобы улучшить производительность,
не используйте символы подстановки и не удаляйте эту запись. Если функций
для экспорта нет, то используйте пустой массив.
FunctionsToExport = '*'

# Командлеты для экспорта из этого модуля. Чтобы улучшить производительность,
не используйте символы подстановки и не удаляйте эту запись. Если командлетов
для экспорта нет, то используйте пустой массив.
CmdletsToExport = '*'

# Переменные для экспорта из модуля
VariablesToExport = '*'

# Псевдонимы для экспорта из модуля. Чтобы улучшить производительность,
не используйте подстановку и не удаляйте эту запись. Если псевдонимов
для экспорта нет, то используйте пустой массив.
AliasesToExport = '*'

# DSC-ресурсы для экспорта из этого модуля
# DscResourcesToExport = @()

# Список всех модулей, упакованных с этим модулем
# ModuleList = @()

# Список всех файлов, упакованных с этим модулем
# FileList = @()

# Приватные данные для передачи в модуль, указанный в RootModule/ModuleToProcess.
Они также могут содержать хеш-таблицу PSData с дополнительными метаданными модуля,
используемыми PowerShell.
PrivateData = @{

    PSData = @{

        # Теги, применяемые к этому модулю. Помогают находить его в онлайн-галереях.
        # Tags = @()
```

```
        # URL лицензии этого модуля
        # LicenseUri = ''

        # URL основного сайта этого проекта
        # ProjectUri = ''

        # URL значка, представляющего этот модуль
        # IconUri = ''

        # ReleaseNotes этого модуля
        # ReleaseNotes = ''

        # Строка Prerelease этого модуля
        # Prerelease = ''

        # Флаг, указывающий, требует ли модуль подтверждения
        # для установки/обновления/сохранения
        # RequireLicenseAcceptance = $false

        # Внешние модули, зависимые от этого модуля
        # ExternalModuleDependencies = @()

    } # Конец хеш-таблицы PSData

} # Конец хеш-таблицы PrivateData

# HelpInfo URI этого модуля
# HelpInfoURI = ''

# Предустановленный префикс для команд, экспортируемых из этого модуля.
# Переопределяйте этот префикс с помощью Import-Module -Prefix.
# DefaultCommandPrefix = ''

}
```

ПРИМЕЧАНИЕ

Мы предполагаем, что вы будете выполнять описанное в системе, работающей с PowerShell. Все это применимо и к Windows PowerShell, но в ней манифест может выглядеть немного иначе, так что не пугайтесь.

16.3. ИЗУЧЕНИЕ МАНИФЕСТА

Далее мы подробнее разберем несколько критических разделов. Стоит отметить, что почти все данные в манифесте можно указать заранее, используя параметры `New-ModuleManifest`. Тем не менее зачастую мы создаем простой манифест, как показано здесь, и потом уже редактируем его в VS Code, когда хотим что-то добавить в модуль.

16.3.1. Метаданные

Вы увидите в манифесте очень много *метаданных*, то есть данных о самом модуле.

- `ModuleVersion` — это пункт, который нужно заполнять в соответствии со стандартной формой записи версии W.X.Y.Z. Если вы планируете отправлять модули в PowerShell Gallery (www.PowerShellGallery.com), то данная информация является обязательной для вашего манифеста.
- Глобально уникальный идентификатор (`GUID`) — это необходимый элемент, который генерируется автоматически. Он уникально идентифицирует ваш модуль.
- `Author` представляет ваше имя, а `CompanyName` — название организации. Если вы отправляете модуль в PowerShell Gallery, то указывать его автора обязательно.
- `Copyright` и `Description` являются необязательными, но при отправке в PowerShell Gallery вам следует добавлять `Description` (в какой-то момент эта информация может оказаться обязательной).
- `ModuleList` представляет список всех подмодулей модуля — по сути, имена всех файлов `.psm1`. Он ничего *не делает* и нужен только для документирования, причем используется достаточно редко.
- `FileList` аналогичен `ModuleList` — он просто определяет способ документирования всех файлов, добавленных в модуль.

16.3.2. Корневой модуль

Это файл `.psm1`, содержащий все функции или код для выполнения необходимых скриптов в текущем контексте. Предполагается, что файл `.psm1` находится в одном каталоге с манифестом. PowerShell не станет ругаться, если тот окажется пустым, но сам модуль не будет работать ожидаемым образом.

16.3.3. Предварительные условия

В манифесте есть несколько свойств, которые помогают PowerShell понять, можно ли выполнить модуль на конкретном компьютере.

- `CompatiblePSEditions` — сообщает движку, можно ли выполнить модуль в PowerShell или Windows PowerShell. Здесь возможны два варианта: `Core` и `Desktop`.
- `PowerShellVersion` — указывает минимальную версию PowerShell, необходимую для выполнения модуля.
- `PowerShellHostName` и `PowerShellHostVersion` — указывают приложение, используемое на выполняющем модуль хосте, и его версию. Эти параметры могут

сужать возможность выполнения модуля до конкретных хост-конфигураций, таких как `"Console-Host"` или других сред.

- `DotNetFrameworkVersion` и `CLRVersion` — указывают требование к минимальной версии .NET Framework или его общеязыковой среды выполнения (Common Language Runtime, CLR).
- `ProcessorArchitecture` — отражает зависимости от платформ, например X86 или Amd64.
- `RequiredModules` — массив имен модулей, которые должны импортироваться *до* загрузки команд модуля. PowerShell попытается их загрузить, и если по какой-то причине не сможет это сделать, то откажется загружать модуль.
- `NestedModules` — этот пункт немного отличается от `RequiredModules`. Модули, указанные в `RequiredModules`, загружаются в глобальном сеансе, то есть при выгрузке модуля выгружены не будут. Что же касается модулей в `NestedModules`, то они видимы *только* для модуля, и их не может увидеть или использовать человек, который загрузил этот модуль (если только он не импортирует их вручную).

16.3.4. Скрипты, типы и форматы

Вы можете указать ряд сопровождающих ваш модуль элементов, которые будут загружаться и выгружаться вместе с ним. Каждый из них будет представлен массивом, то есть вы сможете указывать какое-то количество соответствующих элементов для загрузки или не указывать ни одного.

- `ScriptsToProcess` — здесь перечисляются скрипты PowerShell (файлы `.ps1`), которые необходимо выполнить *до* загрузки модуля. Это немного необычно, но таким образом вы можете, например, выполнять задачи по настройке. Кроме того, эти команды настройки можно поместить в файл `.psm1`, хотя если вынести их в отдельный предварительно загружаемый скрипт, то можно сделать код более читабельным и удобным в сопровождении.
- `TypesToProcess` — это список расширений, которые предоставляет расширяемая система типов PowerShell (Extensible Type System, ETS), обычно XML-файлов `.ps1`, которые вашему модулю нужно загрузить.
- `FormatsToProcess` — список файлов представлений форматов (обычно XML-файлов `.ps1`), которые должен загрузить ваш модуль. О них мы будем говорить позднее.

Вы можете указывать полные пути ко всем этим сопутствующим элементам, однако в соответствии с общим соглашением принято добавлять каждый из них в папку модуля и обращаться к `./filename` в массиве.

16.3.5. Экспорт членов

Эта возможность позволяет вам сократить в PowerShell время загрузки. Вместо того чтобы заставлять оболочку парсить весь модуль скрипта и выяснять, какие функции в нем есть, вы можете объявить их как экспортируемые из модуля. Но здесь есть один побочный эффект: все функции, которые вы *не* экспортируете, становятся частными для модуля. Это означает, что видеть и использовать их может все содержимое модуля, но не человек, который его загрузит. Благодаря этому вы можете создавать вспомогательные функции, которые используются другими командами внутри модуля, но не раскрываются для кого-либо еще.

Экспортировать можно пять видов элементов, каждый из которых представляет массив в манифесте:

- `FunctionsToExport` — содержит функции, которые другие могут использовать как команды;
- `CmdletsToExport` (не будет использоваться в модуле скрипта) — аналогичен `FunctionsToExport` при публикации скомпилированного модуля;
- `VariablesToExport` — содержит переменные уровня модуля, которые должны быть добавлены в глобальную область видимости. Это хороший способ публикации переменных, которые настраивают такие элементы, как имена файлов журналов, строки подключений к базам данных и т. д.;
- `AliasesToExport` — содержит псевдонимы, которые вы определяете в своем модуле (используя `New-Alias`) и которые должны раскрываться при загрузке модуля;
- `DscResourcesToExport` — специальный список, связанный с созданием модулей ресурсов конфигурации желаемого состояния (Desired State Configuration, DSC). Это особый тип инструментов PowerShell, о котором мы не будем говорить в книге.

ИСКЛЮЧЕНИЯ ПРИ ЭКСПОРТЕ

Вам нужно знать о нескольких исключениях для экспорта. Если вы создаете модуль скрипта, а не двоичный компилируемый модуль (который является экспортируемым и должен экспортировать переменные и псевдонимы), то в конце файла `.psm1` нужно использовать `Export-ModuleMember`. Нет ничего плохого в том, чтобы с помощью `Export-ModuleMember` перечислять функции здесь, как в манифесте. В конце файла `.psm1` у вас может быть следующая строка:

```
Export-modulemember -function Get-Foo,Set-Foo -variable myfoo -alias gf,sf
```

Чтобы достичь согласованности, выработайте привычку использовать `Export-ModuleMember` и манифест. PowerShell — очень активный продукт, и никто не знает, когда в очередную версию может быть добавлена поддержка экспорта из манифеста переменных и псевдонимов. Поэтому лучше предусмотреть все варианты.

Отметим, что для большинства из этих элементов можно указывать *, означающую «экспортировать все». К сожалению, это не повышает быстродействие PowerShell, поскольку оболочка вынуждена открывать и парсить весь модуль скрипта, чтобы выяснить, что конкретно подразумевает «все». В качестве лучшей практики рекомендуется не использовать * и перечислять экспортируемые элементы явно.

16.4. УПРАЖНЕНИЯ

Мы предоставим вам модуль (в виде файла `.psm1`) и попросим создать соответствующий манифест. Много времени это не займет.

16.4.1. Вводная информация

В листинге 16.1 показано содержимое модуля скрипта `MyTools.psm1`.

Листинг 16.1. Модуль скрипта MyTools.psm1

```
function Get-TMIPInfo {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True)]
        [string[]]$ComputerName
    )
    BEGIN {}
    PROCESS {
        ForEach ($comp in $ComputerName) {
            Write-Verbose "Connecting to $comp"
            $s = New-CimSession -ComputerName $comp
            $adapters = Get-NetAdapter -CimSession $s |
                        Where Status -ne 'Disconnected'
            ForEach ($adapter in $adapters) {
                Write-Verbose " Interface $($adapter.interfaceindex)"
                $addresses = Get-NetIPAddress
➥ -InterfaceIndex $adapter.InterfaceIndex `
                                              -CimSession $s
                ForEach ($address in $addresses) {
                    $props = @{'ComputerName'=$Comp
                               'Index'=$adapter.interfaceindex
                               'Name'=$adapter.interfacealias
                               'MAC'=$adapter.macaddress
                               'IPAddress'=$address.ipaddress}
                    New-Object -TypeName PSObject -Property $props
                } #foreach address
            } #adapter
            $s | Remove-CimSession
        } #foreach computer
    } #process
    END {}
} #function
```

Будем считать, что вы сохранили этот файл как `\Documents\PowerShell\Modules\MyTools\MyTools.psm1`.

16.4.2. Ваша задача

Создайте манифест для модуля `MyTools` и выполните в нем следующие действия:

- укажите как минимум версию, описание и автора;
- укажите `MyTools.psm1` в качестве корневого модуля;
- экспортируйте функцию `Get-TMIPInfo`.

16.4.3. Наше решение

Мы выполнили следующую команду (здесь мы немного изменили форматирование для лучшей читаемости; по факту же ввели все одной длинной строкой):

```
New-ModuleManifest -Path MyTools.psd1
                   -RootModule ./MyTools.psm1
                   -ModuleVersion 1.0.0.0
                   -Author 'Jeff and Don'
                   -Description 'A test module'
                   -FunctionsToExport @('Get-TMIPInfo')
```

Чтобы сэкономить пространство страницы, мы сократили часть комментариев. Вот результат:

```
#
# Манифест модуля для 'MyModule'
#
# Сгенерировал: User
#
# Сгенерирован: 6/19/2017
#
@{
# Модуль скрипта или двоичный файл модуля, связанный с этим манифестом.
RootModule = 'MyModule.psm1'
# Версия этого модуля. ModuleVersion = '1.0'
# Поддерживаемые PSEditions
# CompatiblePSEditions = @()
# ID, используемый для идентификации этого модуля
GUID = 'ea4d119b-6bcf-4540-a389-67cf7d261726'
# Автор этого модуля
Author = 'User'
# Компания или поставщик этого модуля
CompanyName = 'Unknown'
# Заявление об авторских правах на этот модуль
Copyright = '(c) 2017 User. All rights reserved.'
# Описание функциональности, предоставляемой этим модулем
# Description = ''
# Минимальная версия Windows PowerShell, необходимая модулю
```

```
# PowerShellVersion = ''
# Имя хоста с Windows PowerShell, необходимого этому модулю
# PowerShellHostName = ''
# Минимальная версия хоста с Windows PowerShell, необходимая модулю
# PowerShellHostVersion = ''
# Минимальная версия Microsoft .NET Framework, необходимая модулю ...
# DotNetFrameworkVersion = ''
# Минимальная версия общеязыковой среды выполнения (CLR), необходимой этому ...
# CLRVersion = ''
# Архитектура процессора (None, X86, Amd64), необходимая модулю
# ProcessorArchitecture = ''
# Модули, которые необходимо импортировать в глобальную среду до ...
# RequiredModules = @()
# Сборки, которые должны быть загружены до импорта модуля
# RequiredAssemblies = @()
# Файлы скриптов (.ps1), выполняющиеся в среде вызывающего до ...
# ScriptsToProcess = @()
# Файлы типов (.ps1xml), загружаемые при импорте этого модуля
# TypesToProcess = @()
# Файлы форматов (.ps1xml), загружаемые при импорте модуля
# FormatsToProcess = @()
# Модули, импортируемые как вложенные в этот модуль, указанный в ...
# NestedModules = @()
# Функции, доступные для экспорта из этого модуля.
# Чтобы улучшить производительность, не ...
FunctionsToExport = @('Get-TMIIPInfo')
# Командлеты, доступные для экспорта из этого модуля.
# Чтобы улучшить производительность, не ...
CmdletsToExport = '*'
# Переменные, доступные для экспорта из этого модуля
VariablesToExport = '*'
# Псевдонимы, доступные для экспорта из этого модуля.
# Чтобы улучшить производительность, не ...
AliasesToExport = '*'
# DSC-ресурсы, доступные для экспорта из этого модуля
# DscResourcesToExport = @()
# Список всех модулей, упакованных с этим модулем
# ModuleList = @()
# Список всех файлов, упакованных с этим модулем
# FileList = @()
# Закрытые данные, передаваемые в модуль, указанный в ...
PrivateData = @{
    PSData = @{
        # Применяемые к этому модулю теги. Помогают при его поиске...
        # Tags = @()
        # URL лицензии модуля
        # LicenseUri = ''
        # URL основного сайта этого проекта
        # ProjectUri = ''
        # URL значка, представляющего этот модуль
        # IconUri = ''
        # ReleaseNotes этого модуля
        # ReleaseNotes = ''
```

```
    } # Конец хеш-таблицы PSData
} # Конец хеш-таблицы PrivateData
# HelpInfo URI этого модуля
# HelpInfoURI = ''
# Предустановленный префикс команд, экспортируемых из этого модуля. Переопределите ...
# DefaultCommandPrefix = ''
}
```

ИТОГИ ГЛАВЫ

В этой главе мы рассмотрели нюансы, связанные с управлением модулями в PowerShell, в частности создание и использование их манифестов. Манифесты модулей, представленные в виде файлов `.psd1`, обеспечивают структурированный подход к определению метаданных и зависимостей модулей, повышая их эффективность и управляемость. Вы изучили ключевые компоненты манифеста, такие как версионирование модулей, сведения об авторстве, зависимости и экспортированные элементы. Благодаря применению манифестов пользователи PowerShell могут упростить обнаружение модулей, обеспечить совместимость версий и облегчить интеграцию модулей в свой рабочий поток. Предоставив практические примеры и рекомендации, мы снабдили вас знаниями и инструментами, которые необходимы для использования всего потенциала модулей PowerShell в рамках любых задач по разработке и администрированию.

Часть III

Мы добрались до самого сложного этапа нашего путешествия. В этой части вы сосредоточитесь на продвинутых техниках и профессиональных методах скриптинга. В главе 17 вы узнаете, как перейти от традиционной модели мышления к такой, которая будет лучше соответствовать требованиям сложных сценариев написания скриптов. В главе 18 вы поднимете свои навыки до профессионального уровня, познакомившись с принципами и практиками, отличающими любительские скрипты от тех, которые отвечают общепринятым стандартам индустрии и характеризуются надежностью, масштабируемостью и удобством сопровождения. В главе 19 вы перейдете к теме управления исходным кодом, получив базовые знания о работе с Git — мощной системой контроля версий — и узнав, как она способствует совместной разработке и отслеживанию изменений в скриптах. В главе 20 вы начнете совершенствовать свои скрипты и научитесь скрупулезно улучшать код, используя тестирование и анализ, а также обеспечивая его надежность в различных сценариях. В главе 21 мы будем говорить о безопасности скриптов и о важности их подписания — решающий аспект, который позволяет гарантировать их целостность и подлинность. Глава 22 станет завершающей, и в ней вы подробно разберете процесс публикации скриптов — предоставления их более широкой аудитории.

Изучая теорию и выполняя упражнения и задания, вы освоите новые навыки и профессиональный подход к написанию скриптов. Если вы хотите заниматься скриптингом на высоком уровне, то должны владеть сложными приемами написания скриптов, уметь применять лучшие практики и делать так, чтобы ваши скрипты были функциональными и высококачественными. Надеемся, эта часть книги поможет вам повысить уровень вашего мастерства. Желаем успехов!

17 Перестройка мышления в отношении скриптинга

Давайте ненадолго отойдем от главной темы. В предыдущих главах основное внимание мы уделяли созданию инструментов, соответствующих общепринятым соглашениям и практикам PowerShell. У этого подхода есть свои достоинства. Но бывают случаи, когда лучший способ передать идею — показать ее на контрасте.

ПРИМЕЧАНИЕ

Это та самая бонусная глава. Рекомендуем не спешить и прочесть ее внимательно. Очень важно хорошо понять основную суть. Если какие-то концепции потребуют дополнительного пояснения, то обратитесь с вопросами к сообществу на PowerShell.org. Принципы, описываемые в этой главе, лежат в основе данной книги. Весь остальной материал, по сути, является средством реализации и закрепления этих фундаментальных идей. Если вы намерены развиваться и осваивать более продвинутый скриптинг, которому посвящена книга *The PowerShell Scripting & Toolmaking Book* (https://leanpub.com/powershell-scripting-toolmaking), то вам необходимо четко понимать принципы, описываемые в этой главе.

17.1. ПРИМЕР 1

Рассмотрим пост с форума PowerShell.org, который мы приводим с разрешения его автора. Цель размещенного в данном посте кода — перечисление размеров базовых папок каждого пользователя и показ всех «осиротевших» папок — тех, которые больше не относятся к пользователю Active Directory (AD). Сам код мы привели в листинге 17.1. Обратите внимание, что для его работы необходимо установить модуль AD.

Листинг 17.1. Типичный код PowerShell

```
$UserNames = Get-ADUser -Filter * -SearchBase `
"OU=NAME_OF_OU_WITH_USERS3,OU=NAME_OF_OU_WITH_USERS2,
OU=NAME_OF_OU_WITH_USERS1,DC=DOMAIN_NAME,DC=COUNTRY_CODE" |
```

```
Select -ExpandProperty samaccountname
$UserRegex = ($UserNames | ForEach{[RegEx]::Escape($_)}) -join "|"
$myArray = (Get-ChildItem -Path "\\file2\Felles\Home\*" -Directory |
Where{$_.Name -notmatch $UserRegex})
#$myArray
foreach ($mapper in $myArray) {
    #Param ($mapper = $(Throw "no folder name specified"))
    # Вычисление размера папки и рекурсия в случае необходимости
    $size = 0
    Foreach ($file in $(ls $mapper -recurse)){
    If (-not ($file.psiscontainer)) {
        $size += $file.length
        }
    }
    # Возврат значения и возвращение к вызывающему
    echo $size
}
```

17.1.1. Критика

Сразу оговоримся, что мы не хотим обидеть автора поста. Разработчики осваивают разные навыки в разное время и приходят к тому или иному решению в коде, используя разные способы. Поэтому просто рассмотрим приведенный код объективно.

- Если бы нам нужно было решить ту же задачу, то мы бы написали две функции, а не один скрипт. Одна функция суммировала бы размеры папок, что является полезным подходом во многих сценариях, а вторая выявляла бы «осиротевшие» папки.
- Кроме того, мы бы пошли более характерным для PowerShell путем, избегая таких элементов, как `echo`. Вместо этого мы бы старались выводить объекты, поскольку их можно передавать в команды, через которые они бы попадали в CSV-файлы, HTML-отчеты и многие другие места. В большинстве систем `echo` должна быть псевдонимом `Write-Output`, что подразумевает вывод объектов в конвейер. Но использование псевдонима не делает это явным, и кто-нибудь может использовать `echo` в качестве псевдонима для `Write-Host` — в результате чего мы снова лишимся возможности использовать объекты в конвейере.
- Пожалуй, мы бы более активно использовали нативные команды PowerShell, поскольку они выполняются чуть быстрее скрипта.
- Чтобы максимально повысить вероятность повторного использования, мы бы постарались сделать функции максимально обобщенными и независимыми от контекста. То есть никаких жестко прописанных имен или путей.

При этом нужно помнить, что в Windows папки *не* имеют размера, и суммировать нужно все содержащиеся в них файлы, а не просто складывать *их* размеры.

17.1.2. Наше решение

В листинге 17.2 показана реализованная нами функция. Мы не будем подробно объяснять каждую строку и рекомендуем вам попробовать выполнить этот код самостоятельно. Обратите внимание, что мы явно выводим пустой объект, если папки не существует.

Листинг 17.2. Функция Get-FolderSize

```
function Get-FolderSize {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string[]]$Path
    )
    BEGIN {}
    PROCESS {
        ForEach ($folder in $path) {
            Write-Verbose "Checking $folder"
            if (Test-Path -Path $folder) {
                Write-Verbose " + Path exists"
                $params = @{'Path'=$folder
                            'Recurse'=$true
                            'File'=$true}
                $measure = Get-ChildItem @params |
                           Measure-Object -Property Length -Sum
                [pscustomobject]@{'Path'=$folder
                                  'Files'=$measure.count
                                  'Bytes'=$measure.sum}
            } else {
                Write-Verbose " - Path does not exist"
                [pscustomobject]@{'Path'=$folder
                                  'Files'=0
                                  'Bytes'=0}
            } #если папка существует
        } #foreach
    } #PROCESS
    END {}
} #function
```

Результаты нашей первой функции:

```
Path                                              Files        Bytes
----                                              -----        -----
C:\Get-DiskInfo                                      35        44101
C:\nope                                               0            0
```

Можно передать их в `Select-Object`, чтобы превратить количество `Bytes` в другую единицу измерения, например мегабайты. Тем не менее мы считаем, что наш инструмент окажется более полезным, если будет выводить максимально низкоуровневую

информацию. Обратите внимание, что мы не проверяли эту функцию конкретно на базовых папках. Мы хотим, чтобы она была универсальным инструментом для сложения размеров папок. Позднее мы напишем скрипт-контроллер, который позволит применить эту функцию более целенаправленно, например, для суммирования размеров базовых папок пользователя.

Далее мы напишем вторую функцию, которая будет определять «осиротевшие» папки и содержать функцию `Get-FolderSize`. Мы предполагаем, что последняя уже была загружена в сеанс PowerShell. Этот инструмент предназначен для решения более предметной задачи, поскольку должен «понимать», что нас интересует обнаружение «осиротевших» базовых папок. Итак, наша вторая функция выглядит следующим образом (листинг 17.3).

Листинг 17.3. Функция Get-UserHomeFolderInfo

```
function Get-UserHomeFolderInfo {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True)]
        [string]$HomeRootPath
    )
    BEGIN {}
    PROCESS {
        Write-Verbose "Enumerating $HomeRootPath"
        $params = @{'Path'=$HomeRootPath
                    'Directory'=$True}
        ForEach ($folder in (Get-ChildItem @params)) {
            Write-Verbose "Checking $($folder.name)"
            $params = @{'Identity'=$folder.name
                        'ErrorAction'='SilentlyContinue'}
            $user = Get-ADUser @params
            if ($user) {
                Write-Verbose " + User exists"
                $result = Get-FolderSize -Path $folder.fullname
                [pscustomobject]@{'User'=$folder.name
                                  'Path'=$folder.fullname
                                  'Files'=$result.files
                                  'Bytes'=$result.bytes
                                  'Status'='OK'}
            } else {
                Write-Verbose " - User does not exist"
                [pscustomobject]@{'User'=$folder.name
                                  'Path'=$folder.fullname
                                  'Files'=0
                                  'Bytes'=0
                                  'Status'="Orphan"}
            } #если User существует
        } #foreach
    } #PROCESS
    END {}
}
```

Перебираем все дочерние папки в корневом каталоге

Проверяем наличие пользователя AD

Выполняем функцию Get-FolderSize

Здесь мы берем корневое расположение, в котором находятся базовые папки, поочередно перебираем их и проверяем, существует ли для них соответствующий пользователь AD. Если его нет, то выводим пустой объект со свойством `Orphan Status`. Кроме того, можно было использовать `Where-Object`, чтобы выделить только «осиротевшие» папки для их последующей обработки. Если же `User` существует, то мы используем функцию `Get-FolderSize` для получения информации о размере каталога и вывода того же самого объекта. Только на этот раз объект оказывается полностью заполненным и имеет состояние `OK`. В любом случае вывод объекта одного вида обеспечивает согласованность и повышает вероятность повторного использования информации. Этот код находится среди материалов, разбитых по главам и доступных для скачивания на GitHub (https://github.com/psjamesp/MOL-Scripting).

17.1.3. Мышление вне шаблонов

Наша идея состоит в том, чтобы разделить задачу на части. В оригинальном посте с форума исходными данными являлись «все пользователи AD», что создало сложности с поиском «осиротевших» папок. В нашем же подходе мы используем в качестве исходных данных список папок и выполняем проверку на существование соответствующих пользователей AD. Мы не получим сообщения о том, есть ли пользователи *без* базовых папок, но это и не требуется (да и в большинстве случаев мы ожидаем, что пользователи, у которых нет базовой папки, обратятся в службу поддержки).

Мы взяли одну обобщенную часть задачи и прописали ее в виде инструмента: `Get-FolderSize`. Мы сделали так, чтобы он был полезен сам по себе, а также получал ввод из конвейера и какой-то другой, несмотря на то, что в итоге `Get-UserHomeFolderInfo` использует его иначе. Мы добавили подробный вывод, который слегка упростит понимание и возможную отладку функции. И поскольку мы использовали функции, то каждая из них оказалась узкоспециализированной и выполняла только одну задачу. Благодаря этому они получились менее сложными сами по себе, а также более простыми для отладки и сопровождения.

17.2. ПРИМЕР 2

В листинге 17.4 приведен пример скрипта, который будет по электронной почте уведомлять пользователя об истечении срока действия его пароля. Это очень длинный скрипт, поэтому предлагаем вам скачать его с GitHub (https://github.com/psjamesp/MOL-Scripting).

Обратите внимание: для работы скрипта потребуется модуль Graph PowerShell (`Install-Module Microsoft.Graph`) и регистрация приложения в Microsoft Entra ID.

Листинг 17.4. Скрипт PasswordChangeNotification.ps1

```
<# Этот скрипт будет выполнять подключение к Azure AD через Microsoft Graph и по
электронной почте уведомлять пользователей о том, что срок действия их пароля
истекает через 7 дней.
#>
Connect-MgGraph -ClientId 'YOUR_CLINET_ID' -TenantId 'YOUR_TENANT_ID'

# Срок истечения пароля домена в днях
$PasswordValidDay = get-mgdomain -DomainId PowerShell.org | select-object
-ExpandProperty PasswordNotificationWindowInDays

# Проверка даты истечения срока действия пароля пользователя
$today = Get-Date
$allUsers = get-mguser -All -Property lastPasswordChangeDateTime,mail |
Where-Object {$_.PasswordPolicies -contains "DisablePasswordExpiration"}
foreach ($user in $allUsers) {
    $passwordExpirationDate = $user.LastPasswordChangeDateTime +
[System.TimeSpan]::FromDays($PasswordValidDay)

    # Установка порога уведомления (например, 7 дней до истечения срока
    # действия пароля)
    $notificationThreshold = 7

    # Вычисление количества дней до истечения срока действия пароля
    $daysUntilExpiration = ($passwordExpirationDate - $today).Days

    if ($daysUntilExpiration -le $notificationThreshold) {
        #Отправка письма с помощью Graph API
        $params = @{
            Message                = @{
                Subject               = "Your Password is About to Expire"
                Body                  = @{
                    ContentType = 'HTML'Content = "Your password will
expire on $passwordExpirationDate. Please update your password."
                }
                ToRecipients = @(
                    @{
                        EmailAddress = @{
                            Address = $user.mail
                        }
                    }
                )
            }
            SaveToSentItems = $true
        }
        # Отправка сообщения
        Send-MgUserMail -UserId $from -BodyParameter $params
    } else {
        Write-Host "Password is not yet expired. Days until expiration
$daysUntilExpiration"
    }

}
```

Блок вспомогательных комментариев (заметьте, что он не слишком полезен)

Подключение к Graph API

Сбор всех пользователей в массив

Несколько строк кода для вычисления даты

Перебор полученного массива

Сбор всей информации, необходимой для отправки письма

Отправка письма через Graph API

17.2.1. Разбор скрипта

Разберем основные разделы этого скрипта, чтобы вы имели представление о происходящем. Мы повторим несколько строк кода, чтобы вам не пришлось перелистывать страницы туда-сюда. Итак, рассмотрим каждую часть кода более подробно.

1. Этот скрипт хорошо работает, но не является инструментом. Для начала, в нем ничего не говорится о том, как его использовать.
2. Далее скрипт подключается к приложению Graph API в Microsoft Entra ID (формально Azure AD), указывая свои Client ID (приложения) и Tenant ID:

```
Connect-MgGraph -ClientId 'YOUR_CLINET_ID' -TenantId 'YOUR_TENANT_ID'
```

3. Затем идет строка, проверяющая дату:

```
$today = Get-Date
```

4. Следующий блок кода проверяет установленную политику использования пароля на уровне домена. После этого происходит сбор всех пользователей каталога, политика которых в отношении паролей не установлена как Never Expires:

```
$allUsers = get-mguser -All -Property ` lastPasswordChangeDateTime,mail,
passwordProfile | Where-Object
{$_.PasswordPolicies -contains "DisablePasswordExpiration"}
```

5. Следующий этап — перебор массива. В этой мешанине кода вычисляется текущая дата: вычитается дата последнего изменения пароля и определяется, меньше ли полученный результат установленного порога уведомлений (семь дней).
6. Далее собирается вся информация, необходимая для отправки пользователю письма с уведомлением о том, что его пароль скоро истечет:

```
$params = @{
            Message         = @{
                Subject       = "Your Password is About to Expire"
                Body          = @{
                    ContentType = 'HTML'
                    Content     = "Your password will expire on
$passwordExpirationDate. Please update your password."
                }
                ToRecipients = @(
                    @{
                        EmailAddress = @{
                            Address = $user.mail
                        }
                    }
                )
            }
            SaveToSentItems = $true
        }
```

7. Наконец, с помощью Graph API отправляется письмо:

```
Send-MgUserMail -UserId $from -BodyParameter $params
```

17.2.2. Наше решение

Это хороший пример того, что мы называем *монолитным скриптом*. Он выполняет несколько задач в рамках обширного процесса, но все они выполняются как последовательность, а не разбиты на отдельные инструменты. Такой скрипт требует большого объема работы и может быть сложным в отладке, поскольку очень многое происходит исключительно в памяти. Мы же в подобных случаях предпочитаем писать более маленькие самодостаточные инструменты, каждый из которых служит определенной цели. Создавая и тестируя эти инструменты по отдельности, мы существенно упрощаем написание кода и его отладку.

Сразу заметно, что мы разделили скрипт на четыре функции: `Connect-MyMgGraph`, `Get-PasswordExpirationWindow`, `Check-PasswordExpiration` и `Send-PasswordExpirationNotification`. В первой есть два обязательных параметра: Client ID для созданного вами приложения и Tenant ID для вашего аккаунта Microsoft Entra ID (официально Azure AD). Далее с помощью команды `Connect-MgGraph` мы создаем подключение к Graph API:

```
function Connect-MyMgGraph {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $true)]
        [string]$ClientId,
        [Parameter(Mandatory = $true)]
        [string]$TenantId
        [Parameter(Mandatory = $true)]
        [string]$from

    )

# Подключение к Microsoft Graph с указанными учетными данными
Connect-MgGraph -ClientId $ClientId -TenantId $TenantId -NoWelcome

Write-Host "Connected to Microsoft Graph"
```

Следующая функция — `Get-PasswordExpirationWindow`. Она достаточно мала, но нужно помнить, что мы стараемся разделить код на удобные для повторного использования части. В этой функции есть один обязательный параметр: Domain ID. С помощью Graph API данная функция определяет, какая политика по части истечения срока действия паролей установлена в организации:

```
function Get-PasswordExpirationWindow {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $true)]
        [string]$DomainId
)

    $PasswordValidDay = Get-MgDomain -DomainId $DomainId | Select-Object
-ExpandProperty PasswordNotificationWindowInDays
    return $PasswordValidDay
```

Следующая функция — `Check-PasswordExpiration`, которая представляет основную часть кода. Она собирает список (массив) всех пользователей, которые не установили в отношении паролей политику `DisablePassword-Expiration`. Затем мы смотрим, когда пароль устанавливался в последний раз, и вычитаем эту дату из сегодняшней. Если результат оказывается меньше установленного порога, то мы вызываем последнюю функцию, `Send-Password-ExpirationNotification`, уведомляя пользователя о том, что его пароль скоро перестанет действовать:

```
function Check-PasswordExpiration {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $true)]
        [int]$NotificationThreshold,
        [Parameter(Mandatory = $true)]
        [string]$DomainId
    )

    $today = Get-Date
    $allUsers = Get-MgUser -All -Property lastPasswordChangeDateTime, `
➥ mail | Where-Object {$_.PasswordPolicies -contains `
➥ "DisablePasswordExpiration"}

    foreach ($user in $allUsers) {
        $passwordExpirationDate = $user.LastPasswordChangeDateTime
+[System.TimeSpan]::FromDays((Get-PasswordExpirationWindow -DomainId $DomainId))

        $daysUntilExpiration = ($passwordExpirationDate - $today).Days

        if ($daysUntilExpiration -le $NotificationThreshold) {
            Send-PasswordExpirationNotification -User $user -ExpirationDate
$passwordExpirationDate
        }
        else {
            Write-Host "Password is not yet expired. Days until expiration:
$daysUntilExpiration"
        }
    }
}
```

Последняя функция — `Send-PasswordExpirationNotification`, которая использует Graph API для отправки письма, уведомляющего пользователя о скором истечении срока действия его пароля:

```
function Send-PasswordExpirationNotification {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $true)]
        [PSCustomObject]$User,
        [Parameter(Mandatory = $true)]
        [datetime]$ExpirationDate
    )
```

```
    $params = @{
        Message         = @{
            Subject      = "Your Password is About to Expire"
            Body         = @{
                ContentType = 'HTML'
                Content     = "Your password will expire on
$ExpirationDate. Please update your password."
            }
            ToRecipients = @(
                @{
                    EmailAddress = @{
                        Address = $User.mail
                    }
                }
            )
        }
        SaveToSentItems = $true
    }

    # Отправка письма с помощью Graph API
    Send-MgUserMail -UserId $from -BodyParameter $params
}
```

Последние две строки кода вызывают две наши функции. Первая вызывает `Connect-MyMgGraph` для подключения к Graph API:

```
Connect-MyMgGraph -ClientId 'YOUR_CLIENT_ID' -TenantId 'YOUR_TENANT_ID'
```

И второй мы вызываем `Check-PasswordExpiration`, которой отправляем установленный порог вместе с именем нашего домена:

```
Check-PasswordExpiration -NotificationThreshold 7 -DomainID "Your_Domain"
```

Теперь рассмотрим доработанный нами скрипт (листинг 17.5).

Листинг 17.5. Доработанный код уведомления об истечения срока действия пароля

```
<#
.SYNOPSIS
    Подключается к Microsoft Graph API, используя предоставленные client ID
    и tenant ID.
.DESCRIPTION
    Эта функция устанавливает соединение с Microsoft Graph API,
    используя указанные client ID и tenant ID.
.PARAMETER ClientId
    Client ID вашего приложения.
.PARAMETER TenantId
    Tenant ID, связанное с вашей организацией.
#>
function Connect-MyMgGraph {
    [CmdletBinding()]
    param (
```

```
        [Parameter(Mandatory = $true)]
        [string]$ClientId,
        [Parameter(Mandatory = $true)]
        [string]$TenantId
    )

    # Подключение к Microsoft Graph по указанным учетным данным
    Connect-MgGraph -ClientId $ClientId -TenantId $TenantId -NoWelcome

    Write-Host "Connected to Microsoft Graph"
}

<#
.SYNOPSIS
    Выяснение окна истечения срока действия пароля для конкретного домена.
.DESCRIPTION
    Эта функция получает окно истечения срока действия пароля (в днях)
    для указанного домена.
.PARAMETER DomainId
    ID домена, для которого вы хотите получить окно истечения срока
    действия пароля.
#>
function Get-PasswordExpirationWindow {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $true)]
        [string]$DomainId
    )

    $PasswordValidDay = Get-MgDomain -DomainId $DomainId |
Select-Object -ExpandProperty PasswordNotificationWindowInDays
    return $PasswordValidDay
}

<#
.SYNOPSIS
    Выполняет проверку и уведомляет пользователей об окончании срока
    действия пароля.
.DESCRIPTION
    Эта функция проверяет срок действия пароля для каждого пользователя
    и отправляет уведомление тем, чьи пароли скоро перестанут действовать.
.PARAMETER NotificationThreshold
    За сколько дней до истечения срока действия пароля должно отправляться
    уведомление.
#>
function Check-PasswordExpiration {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $true)]
        [int]$NotificationThreshold,
        [Parameter(Mandatory = $true)]
        [string]$DomainId
    )
```

```
    $today = Get-Date
    $allUsers = Get-MgUser -All -Property lastPasswordChangeDateTime, mail
| Where-Object {$_.PasswordPolicies -contains "DisablePasswordExpiration"}

    foreach ($user in $allUsers) {
        $passwordExpirationDate = $user.LastPasswordChangeDateTime +
[System.TimeSpan]::FromDays((Get-PasswordExpirationWindow -DomainId $DomainId))

        $daysUntilExpiration = ($passwordExpirationDate - $today).Days

        if ($daysUntilExpiration -le $NotificationThreshold) {
            Send-PasswordExpirationNotification -User $user -ExpirationDate
$passwordExpirationDate
        }
        else {
            Write-Host "Password is not yet expired. Days until expiration:
$daysUntilExpiration"
        }
}
}

<#
.SYNOPSIS
    Отправляет пользователю письмо с уведомлением об истечении срока
    действия пароля.
.DESCRIPTION
    Эта функция отправляет уведомление пользователю, пароль которого скоро
    перестанет действовать.
.PARAMETER User
    Объект пользователя, для которого предназначено уведомление.
.PARAMETER ExpirationDate
    Дата, когда пароль пользователя перестанет действовать.
#>
function Send-PasswordExpirationNotification {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $true)]
        [PSCustomObject]$User,
        [Parameter(Mandatory = $true)]
        [datetime]$ExpirationDate
    )

    $params = @{
        Message         = @{
            Subject      = "Your Password is About to Expire"
            Body         = @{
                ContentType = 'HTML'
                Content     = "Your password will expire on
$ExpirationDate. Please update your password."
            }
            ToRecipients = @(
                @{
                    EmailAddress = @{
```

```
                        Address = $User.mail
                    }
                }
            )
        }
        SaveToSentItems = $true
    }

    # Отправка письма с помощью Graph API
    Send-MgUserMail -UserId $from -BodyParameter $params
}

Connect-MyMgGraph -ClientId 'YOUR_CLIENT_ID' -TenantId 'YOUR_TENANT_ID'
Check-PasswordExpiration -NotificationThreshold 7 -DomainID "Your_Domain"
```

ВНИМАНИЕ

В этом примере мы в первую очередь хотели показать, как изменили бы код. Мы не тестировали это решение всесторонне и опустили некоторые моменты из оригинального скрипта, чтобы сэкономить место в книге. Вы можете закончить его, если захотите. Не забудьте поделиться результатами с автором исходного скрипта.

17.3. УПРАЖНЕНИЯ

Настало время потренироваться. В этом упражнении вы будете преобразовывать скрипт, приводя его в правильный вид.

17.3.1. Вводная информация

Рассмотрим пример (листинг 17.6). (Нам очень жаль, что пришлось разбивать строки. Это неизбежно и является частью проблемы, которую мы хотим показать.)

Листинг 17.6. Вводная информация

```
foreach ($domain in (Get-ADForest).domains) {
  Get-ADDomainController -filter * -server $domain |
  sort hostname  |
  foreach {
    Get-CimInstance -ClassName Win32_ComputerSystem -ComputerName
➥ $psitem.Hostname |
    select @{name="DomainController";Expression={$_.PSComputerName}},
➥ Manufacturer, Model,@{Name="TotalPhysicalMemory(GB)"
➥ ;Expression={ "{0:N0}"
-f  ($_.TotalPhysicalMemory / 1Gb) }}
    }
}
```

Это вовсе не *плохой* код, но он ограничен. Предположим, вам однажды захочется вывести его результат на экран — и с этим не будет никаких проблем. Но на следующий день вы захотите вывести его в CSV-файл. А еще через день ваш руководитель попросит сделать вывод в HTML-отчет. Что вы измените, чтобы все эти сценарии стали реализуемыми?

17.3.2. Ваша задача

Перепишите код в соответствии с нативными паттернами и практиками PowerShell, которые мы обсуждали выше. При этом не нужно добавлять никакие излишества наподобие обработки ошибок и прочего, хотя при желании вы вполне можете сделать и это.

17.3.3. Наше решение

Мы решили эту задачу так (листинг 17.7).

Листинг 17.7. Наше решение

```
function Get-DiskInfo {
 foreach ($domain in (Get-ADForest).domains) {
   $hosts = Get-ADDomainController -filter * -server $domain |
   Sort-Object -Prop hostname
   ForEach ($h in $hosts) {
    $cs = Get-CimInstance -ClassName Win32_ComputerSystem -ComputerName $h
    $props = @{'ComputerName' = $h
               'DomainController' = $h
               'Manufacturer' = $cs.manufacturer
               'Model' = $cs.model
               'TotalPhysicalMemory(GB)'=$cs.totalphysicalmemory / 1GB}
     New-Object -Type PSObject -Prop $props
    } #foreach $h
  } #foreach $domain
} #function
```

Обратите внимание на следующие нюансы.

- Мы переключились на скриптовую конструкцию `ForEach`, так как она выполняется чуть быстрее и легче читается.
- Вместо использования `Select-Object` мы вручную создали объект, который получился более читабельным.
- Мы добавили свойства `DomainController` и `ComputerName`. Изначальный код создавал `DomainController`, но мы предпочитаем использовать `ComputerName`, поскольку оно лучше вписывается в конвейер с параметрами `-ComputerName`.
- Самое главное: мы заключили код в функцию. Так будет проще передавать вывод в `Export-CSV`, `ConvertTo-HTML` и т. д.

Наше решение не идеально, поскольку по-прежнему выполняет две задачи: получает аккаунты компьютеров из AD и информацию о диске. Можно написать инструмент для получения аккаунтов компьютеров в подходящей производственной среде, возможно, на основе каких-то критериев. Тогда мы изменим эту функцию так, чтобы она обрабатывала только информацию о диске. Если мы правильно запланируем свойства и параметры, то гипотетически сможем использовать такие команды:

```
Get-CompanyServers | Get-DiskInfo
Get-CompanyServers | Get-DiskInfo | Convertto-html -title "DiskInfo Report"
```

Можете продолжить ваши эксперименты с этим кодом.

ИТОГИ ГЛАВЫ

В текущей главе мы работали над перестройкой модели мышления в контексте скриптинга, разбирая сложные приемы и профессиональные практики. Мы отошли от традиционного видения и призвали вас использовать несколько иной подход к скриптингу.

В оставшейся части книги мы сосредоточимся на внедрении лучших практик скриптинга и создании функциональных и высококачественных скриптов.

18 Профессиональный скриптинг

Мы *почти* готовы назвать вас *профессиональным* разработчиком инструментов на PowerShell. Прежде чем вы сможете добавить в свое резюме пункт «Разработчик инструментов PowerShell», вам следует убедиться, что вы можете продемонстрировать свои навыки как настоящий профессионал. Учитывая это, в данной главе мы привели основные рекомендации и описали лучшие практики, следуя которым вы сможете показать себя как опытного специалиста по работе с PowerShell.

18.1. ИСПОЛЬЗОВАНИЕ СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ

У профессиональных составителей скриптов PowerShell есть отличительная черта: они стремятся создавать долговечный и удобный в сопровождении код. И решающую роль в этом процессе играют системы контроля версий, подробно о которых мы поговорим в главе 19. Для некоторых людей контроль версий может быть сравним с оформлением налоговых деклараций; тем не менее современные инструменты значительно упростили работу с ним. Благодаря интеграции с такими платформами, как Visual Studio Code, контроль исходного кода теперь так же прост, как сохранение файлов и отправка изменений с помощью простого нажатия нескольких клавиш.

При этом важно понимать, что контроль версий не рутинная работа, а показатель профессионализма. Использование этой системы в проектах дает множество плюсов.

- *Сплоченность команды.* В условиях коллективной работы контроль версий исходного кода помогает избежать конфликтов, позволяя отслеживать авторов изменений и исключать случайное переписывание чужих правок.

- *История версий*. Можно легко просматривать предыдущие версии кода, чтобы исправлять ошибки или использовать прежние решения.
- *Резервные копии и восстановление*. Репозитории в системах контроля версий зачастую становятся для организации частью стратегии резервного копирования, обеспечивающей безопасность базы кода.
- *Обмен кодом*. Можно легко делиться кодом с другими, сохраняя контроль над тем, кто и когда вносит изменения, что очень важно для проектов, реализуемых сообществом.
- *Управление запросами (issue)*. Ведущие системы, такие как Team Foundation Server (TFS), Azure DevOps и GitHub, предоставляют инструменты для отслеживания и обсуждения проблем, а также последующего выпуска обновлений.

Напомним, что умение работать с системами контроля версий повышает ваш профессиональный авторитет в глазах IT-менеджеров и коллег.

18.2. ЯСНОСТЬ КОДА

Сокращения могут сэкономить время в консоли, но очень важно, чтобы скрипты можно было легко читать. Избегайте использования псевдонимов и сокращенных имен параметров. Мы наблюдаем, как создатель PowerShell Джеффри Сновер проводит демонстрации, на которых использует выражения наподобие `icm { ps } -com cl2`. Выглядит это все удивительно — и загадочно. При этом во время демонстраций кто-то должен стоять рядом со Сновером и объяснять, что именно тот написал. Так что указывайте полные названия команд и параметров, а также используйте функцию автозавершения, доступную при нажатии клавиши Tab. Это позволит повысить читабельность скрипта и сократить количество опечаток, ведь компьютер не может допустить опечатку в имени команды или параметра. В PowerShell автозавершение с помощью нажатия Tab — ключевой прием, помогающий экономить усилия при составлении инструкций. Использование его — один из признаков профессионального программирования.

Опять же, если вы работаете в консоли и исключительно для себя, то вполне можете вводить то, что помните, и экономить время. Мы все так делаем. Но скрипт — это постоянный артефакт, который вы передаете другим и проверяете в системе контроля версий. Он должен быть более надежным. *Прописывайте* имя каждой команды, каждого параметра и *используйте* имена параметров вместо того, чтобы задействовать позиционные значения. Тогда ваш скрипт будет куда более понятным для других. И, как часто говорит соавтор книги Дон Джонс, через несколько месяцев вы и будете тем самым «другим», и будущий вы оценит то, что прошлый вы не поленился расписать все подробно.

ПРИМЕЧАНИЕ

Мы не единственные, кто так серьезно относится к этому моменту. Если вы используете VS Code, то будете видеть множество красных волнистых подчеркиваний, указывающих на то, что в коде есть что-то неправильное. Дело в том, что расширение PowerShell в VS Code использует инструмент PowerShell Script Analyzer (PSScriptAnalyzer), который выполняет проверку правил на наличие псевдонимов. Возможно, он не заметит, что вы использовали позиционный параметр, но поймет, если вместо Get-Service вы введете gsv. Так что пишите код правильно с самого начала.

18.3. ПОЛЕЗНЫЕ КОММЕНТАРИИ

Комментарии очень помогают сделать код понятным, главное — не переусердствовать, добавляя их. Давайте короткие пояснения к сложным разделам скрипта. Во внутристрочных комментариях не нужно пересказывать очевидное. Используйте подробные объяснения или внутристрочные комментарии для пояснения логики кода. Помните, что эти комментарии служат для вас способом передать свои мысли другим. Мы не имеем в виду такое:

```
# Запрос объекта Win32_ComputerSystem object из WMI
Get-WMIObject -Class Win32_ComputerSystem
```

Так вот что делает `Get-WmiObject`? Ну надо же. Нет, мы не говорим, что вам нужно построчно и поэтапно пояснять действия своего кода. Достаточно кратких комментариев. Вот пример:

```
# Проверяем, был ли указан -NewUser, и изменяем аргументы
# В любом случае используем StartPassword
If ($PSBoundParameters.ContainsKey('NewUser')) {
  $args = @{'StartName'=$NewUser
            'StartPassword'=$NewPassword}
} Else {
  $args = @{'StartPassword'=$NewPassword}
  Write-Warning "Not setting a new user name"
}
```

Здесь мы использовали комментарий, чтобы в общих чертах описать происходящее и его причины. Комментарии лучше всего отражают ход *ваших мыслей*, и это будет полезно для кого-то другого. Опять же, этим кем-то можете оказаться *вы* сами несколько месяцев спустя.

Кроме того, мы обычно не возражаем против использования подробных пояснений вместо внутристрочных комментариев. Вот пример:

```
Write-Verbose "Closing connection to $computer"
$session | Remove-CimSession
```

Удаление `CimSession` очевидно, если исходить из имени команды, поэтому внутристрочный комментарий здесь не требуется. Но пояснение помогает описать ход выполнения скрипта, в результате чего подробный вывод окажется полезен и тому, кто *использует* скрипт, и тому, кто его читает.

ПРИМЕЧАНИЕ

И где же в этой книге внутристрочные комментарии? Мы опустили многие из них, поскольку хотели сэкономить место и обратить ваше внимание на команды. Примеры, которые мы приводим в книге, с точки зрения практик и паттернов не являются тем кодом, который мы развертываем в производственных средах.

18.4. ФОРМАТИРОВАНИЕ КОДА

Для запутанного кода *нет* оправданий. К сожалению, листинг 18.1 — слишком реалистичный пример того, что разработчики обычно постят на онлайн-форумах. Учитывая то, что в книге мы еще и переносим строки, вы наверняка не сможете его прочесть. Но загляните в доступный для скачивания файл с примером кода, и даже его вам будет сложно прочесть.

Листинг 18.1. Неформатированный код

```
function Set-TMServiceLogon {
[CmdletBinding()]
Param(
[Parameter(Mandatory=$True,ValueFromPipelineByPropertyName=$True)]
[string]ServiceName[Parameter(Mandatory=$True,ValueFromPipeline=$True,
ValueFromPipelineByPropertyName=$True)][string[]]$ComputerName,
[Parameter(ValueFromPipelineByPropertyName=$True)]
[string]$NewPassword,[Parameter(ValueFromPipelineByPropertyName=$True)]
[string]$NewUser,
[string]$ErrorLogFilePath
)
BEGIN{}
PROCESS{
    ForEach ($computer in $ComputerName) {
        Do {
    Write-Verbose "Connect to $computer on WS-MAN"
     $protocol = "Wsman"
           Try
{
               $option = New-CimSessionOption -Protocol $protocol
               $session = New-CimSession -SessionOption $option
-ComputerName $Computer -ErrorAction Stop
               If ($PSBoundParameters.ContainsKey('NewUser'))
```

```
{
                    $args = @{'StartName'=$NewUser
                              'StartPassword'=$NewPassword}
                }
Else {
                    $args = @{'StartPassword'=$NewPassword}
                    Write-Warning "Not setting a new user name"
                }
                Write-Verbose "Setting $servicename on $computer"
                $params = @{'CimSession'=$session
                  'MethodName'='Change'
      'Query'="SELECT * FROM Win32_Service WHERE Name = '$ServiceName'"
             'Arguments'=$args}
              $ret = Invoke-CimMethod @params
              switch ($ret.ReturnValue) {
                0 { $status = "Success" }
              22 { $status = "Invalid Account" }
                Default { $status = "Failed: $($ret.ReturnValue)" }
              }
              $props = @{'ComputerName'=$computer;'Status'=$status}
            $obj = New-Object -TypeName PSObject -Property $props
          Write-Output $obj
            Write-Verbose "Closing connection to $computer"
             $session | Remove-CimSession
   } Catch {
              # Изменение протокола: если были опробованы оба
              # и указано логирование, то логируется имя компьютера
              Switch ($protocol) {
              'Wsman' { $protocol = 'Dcom' }
                  'Dcom' {
                $protocol = 'Stop'
                        If
($PSBoundParameters.ContainsKey('ErrorLogFilePath')) {
Write-Warning "$computer failed; logged to
$ErrorLogFilePath"
                     $computer | Out-File $ErrorLogFilePath -Append
                       } }
       }
}
        } Until ($protocol -eq 'Stop')
 } }
END{}
}
```

Попробуйте понять, что написано в этом коде. Считайте это вызовом.

А теперь взгляните на листинг 18.2, в котором тот же код выполняет те же самые действия.

Листинг 18.2. Форматированный код

```
function Set-TMServiceLogon {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string]$ServiceName,
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string[]]$ComputerName,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewPassword,
        [Parameter(ValueFromPipelineByPropertyName=$True)]
        [string]$NewUser,
        [string]$ErrorLogFilePath
    )
BEGIN{}
```

← Пропуск строк для улучшения читабельности

```
PROCESS{
    ForEach ($computer in $ComputerName) {
        Do {
            Write-Verbose "Connect to $computer on WS-MAN"
            $protocol = "Wsman"
            Try {
                $option = New-CimSessionOption -Protocol $protocol
                $session = New-CimSession -SessionOption $option
-ComputerName $Computer -ErrorAction Stop
                If ($PSBoundParameters.ContainsKey('NewUser')) {
                    $args = @{'StartName'= $NewUser
                              'StartPassword' = $NewPassword}
                } Else {
                    $args = @{'StartPassword' = $NewPassword}
                    Write-Warning "Not setting a new user name"
                }
                Write-Verbose "Setting $servicename on $computer"
                $params = @{'CimSession'=$session
                            'MethodName'='Change'
                            'Query'="SELECT * FROM Win32_Service WHERE Name
= '$ServiceName'"
                            'Arguments'=$args}
                $ret = Invoke-CimMethod @params
                switch ($ret.ReturnValue) {
                    0  { $status = "Success" }
                    22 { $status = "Invalid Account" }
                    Default { $status = "Failed: $($ret.ReturnValue)" }
                }
```

← Аккуратно структурированные хеш-таблицы

```
                $props = @{'ComputerName'=$computer
                           'Status'=$status}
                $obj = New-Object -TypeName PSObject -Property $props
                Write-Output $obj
                Write-Verbose “Closing connection to $computer”
                $session | Remove-CimSession
            } Catch {
                # Изменение протокола: если были опробованы оба
                # и указано логирование, то логируется имя компьютера
                Switch ($protocol) {
                    'Wsman' { $protocol = 'Dcom' }
                    'Dcom' {
                        $protocol = 'Stop'
                        If
($PSBoundParameters.ContainsKey('ErrorLogFilePath')) {
Write-Warning "$computer failed; logged to
$ErrorLogFilePath"
                            $computer | Out-File $ErrorLogFilePath -Append
                        } # if logging
                    }
                } #switch
            } # try/catch
        } Until ($protocol -eq 'Stop')
    } #foreach
} #PROCESS
END{}
} #function
```

Комментарии для закрывающих скобок

В файле (где, стоит признать, более длинные строки все равно получаются немного сумбурными) читать этот код весьма приятно. Видно, где начинается и заканчивается каждый блок кода. Обратите внимание на следующие моменты:

- когда мы закрываем конструкцию с помощью }, то добавляем комментарий, указывающий, что именно закрывает эта скобка;
- мы используем пустые строки для отделения фрагментов кода, чтобы более отчетливо видеть конкретные функциональные единицы;
- в каждой конструкции мы делаем отступ в четыре пробела;
- хеш-таблицы создаются так, чтобы на каждой строке имелось по одной паре «ключ — значение» и все они были выровнены по общему левому краю.

В VS Code (который, повторимся, мы вам рекомендуем) есть возможность быстро отформатировать все эти нюансы, причем это делает сам редактор. Он даже будет пытаться выполнять форматирование по мере ввода кода, чтобы не допустить беспорядка с самого начала. В этом ценность хорошего редактора, которую в случае VS Code вы получаете совершенно бесплатно.

СОВЕТ

Откройте в VS Code панель команд и введите Format Document. Щелкните на ней, и редактор автоматически отформатирует ваш документ в соответствии с рекомендациями. Кроме того, вы можете добавить эту строку в файл settings.json, чтобы форматирование выполнялось при сохранении: "editor.formatOnSave": true.

18.5. ОПИСАТЕЛЬНЫЕ ИМЕНА ПЕРЕМЕННЫХ

Выбирайте для переменных осмысленные и описательные имена. Избегайте сокращений. Понятные названия переменных улучшают читабельность кода и делают его более удобным в сопровождении. Да, в этой книге мы иногда использовали варианты `$c` или `$s`, но только для того, чтобы сэкономить пространство страниц. Переменная, которая содержит множество объектов системных дисков, должна иметь имя наподобие `$drives` (множественная форма будет напоминать, что это коллекция, а не один объект). Имя пользователя должно быть представлено как `$username`, а не `$un`. Есть только одно исключение: переменные, используемые для объявления параметров, должны соблюдать соглашение об именовании параметров, в соответствии с которым существительные нужно указывать в единственном числе: `$ComputerName`, но не `$ComputerNames`.

Кроме того, для именования переменных старайтесь не использовать венгерскую нотацию, которая появилась вместе с VBScript *в 1990-х годах*. Подумайте об этом, прежде чем создавать переменные с такими названиями, как `$strComputer` и `$intCounter`. Подобное написание было актуально в VBScript, поскольку это был слаботипизированный и не объектно-ориентированный язык. В PowerShell реализована более строгая типизация, и этот язык уже объектно ориентирован. Строка в нем является объектом с типом `System.String`. Нет необходимости добавлять к имени переменной приставку `str`, которая будет об этом напоминать. В PowerShell технически все будет являться `$obj`, так что венгерская нотация становится бессмысленной, и ее использование показывает, что вы отстаете от современных тенденций.

18.6. ИЗБЕГАНИЕ ПСЕВДОНИМОВ

Псевдонимы экономят время работы в консоли, а вот в скриптах использовать их не нужно. Двусмысленные псевдонимы могут вносить путаницу. Вместо них, чтобы улучшить читабельность и понятность кода, используйте полные имена команд. Исключением могут стать разве что общепринятые псевдонимы наподобие `'Where'` вместо `'Where-Object'`. Вариант `ForEach` применять не рекомендуется, так как его легко спутать с языковой конструкцией `ForEach`; используйте `ForEach-Object`, если хотите задействовать эту команду. Избегайте непонятных псевдонимов наподобие `icm` и `gwmi`; пишите имена команд целиком и забудьте об использовании псевдонимов в скриптах.

18.7. ЛОГИКА ВМЕСТО СЛОЖНОСТИ

Поддерживайте логическую структуру скриптов. Избегайте сложных вложенных выражений и непонятных конвейеров. Ваш скрипт должен быть структурированным, постоянным артефактом, а не однострочной загадкой. Отдавайте приоритет читабельности, а не заумности. Вот пример:

```
Gwmi Win32_operatingsystem | select *,@{n='RAM';e={gwmi
 win32_computersystem | select -exp totalphysicalmemory} | % { $_ |
 Out-File temp.txt -Append ; $_.Reboot() }
```

Пожалуйста, не выполняйте этот код, если пока еще чувствуете неуверенность в своих силах по отношению к скриптингу. Просто оцените, насколько трудно его читать и понимать: здесь и вложенные выражения, и разделенные точкой с запятой команды, и прочее. Опять же, это вполне *допустимо* для ситуативной команды, чего-то одноразового, но не для скрипта.

При этом мы не избегаем автоматического использования в скрипте конвейера, так как это одна из наиболее мощных возможностей PowerShell. Мы подходим к ней иначе:

```
$os = Get-WmiObject -Class Win32_OperatingSystem
$cs = Get-WmiObject -Class Win32_ComputerSystem
$os | Add-Member -MemberType NoteProperty -Name RAM -Value `
 $cs.TotalPhysicalMemory
$os | Out-File temp.txt -Append
$os.Reboot()
```

Опять же, мы не рекомендуем выполнять этот код, пока вы не наберетесь решимости. Но этот его вариант определенно проще читать. Каждая строка делает что-то одно, основываясь на предыдущих строках. Это не единственное корректное изменение структуры изначального неуклюжего примера. Оптимизировать его можно десятком разных способов, добившись выполнения той же задачи за тот же промежуток времени. В PowerShell досконально продуманные однострочники обычно самые сложные для понимания — не перегружайте свои скрипты такими командами.

18.8. ПРЕДОСТАВЛЕНИЕ ПОМОЩИ

Мы все понимаем. Документировать код *скучно*. Но это надо делать в любом случае. Известно ли вам, как сильно расстраивает ситуация, когда ты пытаешься найти справку по команде, а она либо оказывается очень скудной, либо вообще отсутствует? Не будьте таким программистом.

Лучше научитесь работать с PlatyPS, проектом с открытым исходным кодом, который команда PowerShell использует для генерации внешних (то есть не основанных на комментариях) файлов справки.

18.9. ИЗБЕГАНИЕ WRITE-HOST И READ-HOST

По мере развития PowerShell эта проблема становилась все более запутанной, но суть осталась прежней: всякий раз, когда вы используете `Write-Host` для вывода, происходит нечто плохое. Смысл в том, что команды `-Host` будет вводить некий программист. Иными словами, они привязывают вашу команду к конкретному контексту — взаимодействию с человеком, — которого инструменты должны избегать. Конечно, есть и исключения.

Во-первых, если вы пишете скрипт-контроллер, который должен использовать инструменты в контексте взаимодействия с человеком, то команды `-Host` явно будут уместны. Еще они полезны, если вы создаете инструмент, использующий глагол `Show`, один из официальных глаголов PowerShell. Этот глагол (который вы можете использовать в командах наподобие `Show-Menu`) предполагает взаимодействие с человеком, а значит, подразумевает конкретный контекст.

Во-вторых, в PowerShell v5 и более свежих версиях конкретно `Write-Host` становится своеобразным сокращением для канала `Write-Information`, устраняя практически все проблемы с привязкой к контексту, которые раньше возникали при использовании `Write-Host`. Мы по-прежнему не считаем применение `Write-Host` хорошей идеей. Если вы хотите задействовать информационный поток, то выберите `Write-Information`. Использование `Write-Host` говорит о том, что вы не знаете о существовании `Write-Information` и выбираете `Write-Host` из неправильных соображений.

ПРИМЕЧАНИЕ

Еще один контраргумент, который мы часто слышим, звучит так: «Но с помощью Write-Host я показываю пользователю, что происходит». С одной стороны, этот аргумент оправдан. Если у вас есть скрипт или инструмент, который требует времени на обработку или проходит через сложный процесс, то предоставление обратной связи может оказаться полезным. Но в таком случае лучше научитесь использовать вместо Write-Host командлет Write-Progress.

18.10. ПРИДЕРЖИВАЙТЕСЬ ИСПОЛЬЗОВАНИЯ ОДИНАРНЫХ КАВЫЧЕК

В PowerShell мы предпочитаем разделять строки с помощью одинарных кавычек, если только не требуется интерполировать переменную или подвыражение, как в этом случае с переменной:

```
$message = "The computer name is $ComputerName"
```

Или в таком подвыражении, как это:

```
$message = "Yesterday was $( (Get-Date).AddDays(-1) )"
```

Одинарные кавычки позволяют сохранить ясность кода и препятствуют непредусмотренному расширению переменных (variable expansion). Если вы не привыкли использовать одиночные кавычки в качестве разделителей строк, то придется начинать делать это (мы не уверены, что сами следовали этому правилу на страницах данной книги), но оно того стоит.

18.11. НЕ ЗАГРЯЗНЯЙТЕ ГЛОБАЛЬНУЮ ОБЛАСТЬ ВИДИМОСТИ

Не отправляйте свои переменные в данную область. Это ужасная практика, которая сильно затрудняет отладку скриптов, а в некоторых ситуациях может привести к их ненадежному и несогласованному выполнению (как в случае с хостом, который управляет глобальной областью иначе). Модули могут свободно экспортировать переменные, которые в итоге окажутся в глобальной области видимости, но лишь те, которыми PowerShell сможет управлять в рамках жизненного цикла модуля. Ничто иное в эту область отправляться не должно.

18.12. СОХРАНЯЙТЕ ГИБКОСТЬ ИНСТРУМЕНТОВ

Мы надеемся, что это очевидно, но все же проговорим: старайтесь не прописывать фиксированные значения и ссылки. Не создавайте в коде функцию с жестко прописанным значением для сервера Exchange. Вместо этого создайте необязательный параметр и установите значение по умолчанию. Так вы сможете легко выполнять функцию со значениями по умолчанию, но при этом обрабатывать редкие ситуации, когда нужно указать другой сервер. Не пишите такие команды:

```
Function Get-ServerStuff {
$server = 'SRV01'
...
}
```

Естественно, вы можете решить, что вам никогда не придется указывать другое значение, но все может измениться буквально завтра. Профессионалы пишут инструменты, учитывая их гибкость:

```
Function Get-ServerStuff {
Param ([string]$ComputerName = 'SRV01')
...
}
```

Вам нужно продумывать не только то, как кто-либо может использовать ваш инструмент сегодня, но и то, как этот инструмент может измениться в будущем.

18.13. АКЦЕНТ НА БЕЗОПАСНОСТИ

Никогда не прописывайте в скрипте учетные данные. Для их безопасной обработки используйте в качестве параметра объект `[pscredential]`. Поддерживайте безопасность, делая так, чтобы скрипт можно было по-разному использовать в различных сценариях:

```
Function Get-Diskspace {
[cmdletbinding()]
Param([string]$ComputerName,[pscredential]$Credential)
$PSBoundParameters.Add("classname","win32_logicaldisk")
$PSBoundParameters.Add("filter","drivetype=3")
Get-WmiObject @PSBoundParameters |
Select PSComputerName,DeviceID,Size,Freespace
}
```

Пользователь сможет выполнить эту функцию так:

```
Get-diskspace -ComputerName SRV01 -credential company\administrator
```

Система попросит его ввести пароль или передать объект `credential`:

```
$cred = get-credential company\administrator
Get-diskspace -ComputerName SRV01 -credential $cred
```

Благодаря использованию объекта `pscredential` ваш код остается безопасным и гибким.

18.14. СТРЕМЛЕНИЕ К ЭЛЕГАНТНОСТИ

Стремитесь к тому, чтобы ваш код был элегантным. Упрощайте скрипты, избегая повторов. Используйте приемы, такие как сплэттинг (splatting) параметров в хеш-таблицы, чтобы создавать более чистый и читабельный код. Старайтесь постепенно вырабатывать более элегантный стиль написания. Когда будете создавать инструменты (надеемся, в соответствии с рекомендациями книги), старайтесь достичь простоты или элегантности. Мы считаем, что элегантные скрипты проще читать и отлаживать, к тому же они быстрее выполняются. Один из вспомогательных принципов — избегайте повтора кода.

Предположим, вы создаете код для получения системной информации из Windows Management Instrumentation (WMI) с помощью `Get-CimInstance` на основе значения переменной. Изначально ваш вариант может выглядеть так:

```
Switch ($value) {
"OS" {
    $data = Get-Ciminstance -class win32_operatingsystem
-ComputerName $ComputerName | Select PSComputerName,Version,Caption
}
```

```
"CS" {
    $data = Get-Ciminstance -class win32_computersystem
-ComputerName $ComputerName | Select PSComputerName,Model,Manufacturer
}
"CPU" {
    $data = Get-Ciminstance -class win32_processor
-ComputerName $ComputerName | Select PSComputerName,CPUID,Name,MaxClockSpeed
}
"Memory" {
    $data = Get-Ciminstance -class win32_physicalmemory
-ComputerName $ComputerName | SelectPSComputerName,Banklabel,Capacity,Speed
}
}
```

Этот код будет прекрасно работать, но в нем есть лишнее копирование, вставка и редактирование. Сравните его со следующим примером:

```
$ComputerName = 'localhost'
$value = "OS"
$cimparams=@{ComputerName=$ComputerName}
$props = @('PSComputerName')
Switch ($value) {
'OS' {
    $cimparams.Add('classname','win32_operatingsystem')
    $props+='Version','Caption'
}
'CS' {
    $cimparams.Add('classname','win32_computersystem')
    $props+='Model','Manufacturer'
}
'CPU' {
    $cimparams.Add('classname','win32_processor')
    $props+='CPUID','Name','MaxClockSpeed'
}
'Memory' {
    $cimparams.Add('classname','win32_physicalmemory')
    $props+='Banklabel','Capacity','Speed'
}
}
$data = Get-CimInstance @cimparams | Select-object -Property $props
```

Использует хеш-таблицу с параметрами для сплэттинга

Изменяет параметры на лету

Выполняет Get-CimInstance один раз

Обратите внимание на использование хеш-таблицы с параметрами для `Get-CimInstance`, к которой мы в итоге применим сплэттинг. Это прекрасный прием для упрощения кода при условии, что вы знакомы с хеш-таблицами, приемами сплэттинга и массивами. Тем не менее этот пример лучше читается и уже не такой громоздкий.

В данной книге мы приводим множество приемов, и вам нужно превратить владение ими в искусство. Навык написания элегантного кода придет со временем, когда вы

наберетесь опыта и мастерства. Линейная графика Пикассо очень выразительная, и со стороны кажется, что художник создавал эти рисунки, прикладывая минимум усилий. Однако прошли годы, прежде чем он достиг необходимого уровня мастерства. Возможно, сегодня ваши творения совсем просты, но мы хотим, чтобы в итоге вы создавали элегантные шедевры.

ИТОГИ ГЛАВЫ

Быть профессионалом в мире скриптинга PowerShell означает писать скрипты, которые соответствуют общепринятым стандартам индустрии. Практики, описанные в данной главе, помогут вам поднять ваши навыки разработчика на новый уровень и сделать ваш код надежным, масштабируемым и удобным в сопровождении. Используя их, вы сможете зарекомендовать себя в сообществе PowerShell как грамотного специалиста.

19 Управление версиями с помощью Git

Один из признаков профессионального разработчика инструментов — использование системы управления версиями. У многих ли из вас на системном диске C: есть папка, в которой вы храните все свои скрипты? Или, возможно, у вас на работе есть сетевой диск со всеми IT-скриптами? Мы трудимся в сфере автоматизации и DevOps, поэтому правильно обслуживать проекты PowerShell очень важно. Сегодня многие организации возлагают эту задачу на *Git* — систему управления версиями исходного кода, которая появилась и стала популярна благодаря Linux (ее разработал создатель Linux Линус Торвальдс). Мы решили, что будет полезно предоставить вам экспресс-курс по Git, чтобы вы могли начать использовать данную систему в своей работе. Вполне очевидно, что это обширная тема. Ее глубокое изучение потребует времени. Помочь в этом может книга Рика Умали (Rick Umali) *Learn Git in a Month of Lunches* (Manning, 2015, http://mng.bz/mj7P).

19.1. ЗАЧЕМ УПРАВЛЯТЬ ВЕРСИЯМИ

Управление версиями подразумевает отслеживание изменений в файле, в том числе исправление журнала или документации, где сказано, кто и почему внес то или иное изменение. Такие системы, среди прочего, упрощают поиск последней или наиболее надежной версии. В некоторых из них работа с файлом подразумевает, что на него нужно переключиться (`git checkout`). По завершении работы вы можете отправить (закоммитить) (check in, а именно `git commit`) внесенные изменения в мастер-ветку, зачастую добавив комментарий о том, что вы изменили и почему. Пока вы работаете с файлом, никто другой не может изменять его, что может быть удобно для небольших команд.

Возможно, в вашей организации уже предусмотрено какое-то решение для управления версиями, будь то Microsoft Team Foundation Services (TFS), GitHub, GitLab или одна из десятка других систем. Если это так, то придерживайтесь именно его, поскольку обслуживание еще одной программы управления версиями явно станет лишним.

19.2. ЧТО ТАКОЕ GIT

Многие традиционные системы управления версиями централизованны. Зачастую в них есть централизованный сервер или база данных, доступ к которым находится под строгим контролем. Как вы можете себе представить, такие системы подвергаются большой нагрузке. Git же, напротив, была разработана как децентрализованная система. Она создавалась для Linux в целях упрощения управления исходным кодом ядра, поэтому довольно надежна. В парадигме Git у каждого есть собственные копии исходных файлов, которые можно периодически совмещать и обновлять.

Git — это в первую очередь инструмент командной строки, и начать работать с ним вы сможете, используя всего несколько базовых команд. По мере изучения экосистемы Git вы столкнетесь с несколькими графическими фронтенд-решениями и даже модулями PowerShell, которые, по сути, являются обертками команд Git. Мы советуем вам использовать традиционные инструменты командной строки Git. Как только вы выработаете определенный навык, можете смело начинать применять какие-либо инструменты GUI, если так вам удобнее. Кроме того, мы рекомендуем учиться работать из командной строки, поскольку огромный объем онлайн-информации практически всегда обрабатывается именно через нее.

Основная причина использовать Git заключается в том, что она крайне проста и нужно только привыкнуть к ней. Причем существует много вспомогательных инструментов, позволяющих дополнительно упростить работу. Кроме того, благодаря своей структуре Git очень удобно использовать для сильно распределенного управления версиями. Это означает, что вы можете хранить локально копии файлов для работы, а основные копии держать на защищенном сервере, в веб-сервисе управления версиями наподобие GitHub.com и т. д. Есть даже инструменты Git для мобильных устройств под управлением iOS и Android, позволяющие вам всегда иметь доступ к своим рабочим файлам. И, возможно, самый главный нюанс: Git стала особенно популярна в мире PowerShell. То есть очень многие проекты сообщества — в том числе исходный код и документация для ядра PowerShell — размещаются в Git (особенно на веб-сервисе GitHub.com). Ознакомившись с этой системой, вы не только облегчите себе работу, но и сможете вносить свой вклад в проекты сообщества и PowerShell. Если вы будете размещать собственные проекты в таких местах, как GitHub, то быстрее привлечете других участников.

19.2.1. Установка Git

Для начала перейдите по адресу https://git-scm.com/downloads и скачайте последний клиент для Windows. Запустите установку — у вас должен быть доступ ко всем базовым функциям. В результате установки вы получите возможность запустить Git в Linux-подобном окне терминала. В качестве альтернативы можете использовать традиционную консоль Windows и PowerShell; именно это мы обычно и делаем.

19.2.2. Основы Git

Когда установка завершится, откройте окно PowerShell. Если во время установки у вас был открыт сеанс, то его нужно перезапустить, чтобы система обнаружила изменение в переменной среды. Для получения справочной информации введите в командной строке команду `git`:

```
PS C:\> git
usage: git [--version] [--help] [-C <path>] [-c name=value]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
...
```

По мере знакомства с Git мы рекомендуем вам возвращаться к началу и более подробно изучать раздел справки, посвященный командам. Кроме того, выполните `git help tutorial`, чтобы открыть страницу HTML-документации. (При этом вам должен быть доступен браузер.) Вдобавок на этой странице вы увидите ссылку на руководство пользователя, с которым стоит ознакомиться.

Мы будем использовать Git в качестве системы управления версиями, рассматривая вас в качестве основного пользователя. Вам нужно будет указать информацию об имени пользователя и адрес электронной почты:

```
git config --global user.email "James@globomantics.com"
git config --global user.name "James Petty"
```

Позднее мы покажем, как выполнять интеграцию с GitHub, позволяющую налаживать сотрудничество с другими разработчиками. Если у вас есть учетные данные GitHub, то используйте здесь их.

19.3. ОСНОВЫ РАБОТЫ С РЕПОЗИТОРИЯМИ

Сначала вам нужно инициализировать репозиторий Git. Сделав это, вы, по сути, попросите Git следить за созданным каталогом. Это может быть корневой каталог вашего модуля из проекта скриптинга. В качестве демонстрации мы создали каталог с названием MyPSTool и перешли в него:

```
PS C:\> mkdir MyPSTool
    Directory: C:\
Mode                 LastWriteTime         Length  Name
----                 -------------         ------  ----
d-----        6/14/2023   3:20 PM                  MyPSTool
PS C:\> cd .\MyPSTool
PS C:\MyPSTool>
```

При выполнении команды Git нужно находиться в этом репозитории. Мы обычно стараемся выполнять команды Git из корневого каталога.

19.3.1. Создание репозитория

Мы хотим, чтобы этим каталогом управляла Git, поэтому инициализируем его как репозиторий:

```
PS C:\MyPSTool> git init
Initialized empty Git repository in C:/MyPSTool/.git/
PS C:\MyPSTool> Get-ChildItem      -Hidden
    Directory: C:\MyPSTool
Mode                 LastWriteTime         Length  Name
----                 -------------         ------  ----
d--h--         6/14/2023   3:26 PM                 .git
```

Этот процесс создает скрытый каталог. У нас не должна возникать потребность обращаться к нему или изменять что-либо непосредственно в нем. Кроме того, в процессе инициализации создается основная (мастер-) ветка. Позднее мы сможем создавать дополнительные:

```
PS C:\MyPSTool> git status
On branch main
Initial commit
nothing to commit (create/copy files and use "git add" to track)
PS C:\MyPSTool>
```

Далее мы создадим несколько файлов, после чего перепроверим состояние:

```
PS C:\MyPSTool> git status
On branch main
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1.ps1
        file2.ps1
nothing added to commit but untracked files present (use "git add" to track)
```

Git поддерживает несколько виртуальных областей для отслеживания вашей работы. Как видите, система сообщает, что у нас есть неотслеживаемые файлы. Это означает, что они не являются частью системы управления версиями. Исправим этот недочет.

19.3.2. Индексация изменения

Первый шаг — *индексация* изменений путем добавления файлов. Мы можем либо добавить отдельные файлы, либо индексировать их все:

```
PS C:\MyPSTool> git add .
```

Проверим статус:

```
PS C:\MyPSTool> git status
On branch main
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.ps1
        new file:   file2.ps1
```

Файлы индексированы и готовы к отправке в репозиторий. Если мы изменим индексированный файл, то его нужно будет добавить заново:

```
PS C:\MyPSTool> git status
On branch main
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.ps1
        new file:   file2.ps1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified:   file2.ps1
PS C:\MyPSTool> git add .\file2.ps1
```

Теперь пора закоммитить изменения.

19.3.3. Коммит изменения

Коммит позволяет впоследствии выполнить откат к конкретному состоянию или отменить внесенные изменения. Если вам так будет проще, то можете рассматривать операции `git commit` как контрольные точки, хотя в действительности они представляют собой нечто большее. Теперь мы закоммитим файлы, в том числе сопутствующее сообщение:

```
PS C:\MyPSTool> git commit -m 'added basic commands'
[main (root-commit) 038b8f9] added basic commands
 2 files changed, 1 insertion(+)
 create mode 100644 file1.ps1
 create mode 100644 file2.ps1
PS C:\MyPSTool>
```

Вам нужно сопроводить коммит сообщением, длина которого ничем не ограничена. В данном случае мы создали конструкцию here-string:

```
PS C:\MyPSTool> $m=@"
>> Это чуть удлиненное сообщение коммита,
>> которое может занимать несколько строк.
>> "@
>>
PS C:\MyPSTool> git commit -m $m.
```

ПРИМЕЧАНИЕ

Подробнее об использовании конструкций here-string читайте в разделе Using Windows PowerShell 'Here-Strings' (TechNet, http://mng.bz/9r4E).

В файлах каталога мы никаких изменений не заметим — все отслеживается в скрытом каталоге `.git`. Но мы можем просмотреть, что произошло, используя доступную в Git возможность логирования:

```
PS C:\MyPSTool> git log
commit 038b8f9ca8b846e9024532e9bda4e272cd24048b
Author: James Petty <James@globomantics.com>
Date:   Wed Jun 14 16:04:11 2023 -0500
   added basic commands
```

Изменения, внесенные конкретным пользователем, можно найти по его имени.

19.3.4. Откат изменения

Коротко разберем, почему мы вообще об этом говорим. Мы создали простой текстовый файл и отправили его в репозиторий:

```
PS C:\MyPSTool> set-content -value      'james'      -Path .\data.txt
PS C:\MyPSTool> git add .
PS C:\MyPSTool> git commit -m "Added data.txt"
[main 9113535] Added data.txt
 1 file changed, 1 insertion(+)
 create mode 100644 data.txt
PS C:\MyPSTool> git log
commit 9113535942d0c35a964deda9e869a0193bb284ad
Author: James Petty <James@globomantics.com>
Date:   Wed Jun 14 16:12:31 2023 -0500
    Added data.txt
commit 038b8f9ca8b846e9024532e9bda4e272cd24048b
Author: James Petty <James@globomantics.com>
Date:   Wed Jun 14 16:04:11 2023 -0500
    added basic commands
PS C:\MyPSTool>
```

Теперь мы изменим файл `data.txt` и закоммитим это изменение:

```
PS C:\MyPSTool> set-content -value "james" -Path .\data.txt
PS C:\MyPSTool> get-content .\data.txt
james
PS C:\MyPSTool> git commit -a -m "set data.txt to james"
[main ee546b7] set data.txt to james
 1 file changed, 1 insertion(+), 1 deletion(-)
PS C:\MyPSTool>
```

На этот раз мы использовали сокращение `-a`, чтобы закоммитить все ожидающие файлы, что избавило нас от необходимости выполнять `git -add`.

Вывод логов становится длинным, поэтому получим три последние записи:

```
PS C:\MyPSTool> git log -n 3
commit ee546b73819f1ebbc8b7073c79113e0b6adb5c33
Author: James Petty <James@globomantics.com>
Date:   Wed Jun 14 16:15:48 2023 -0500
    set data.txt to james
commit 9113535942d0c35a964deda9e869a0193bb284ad
Author: James Petty <James@globomantics.com>
Date:   Wed Jun 14 16:12:31 2023 -0500
    Added data.txt
commit 038b8f9ca8b846e9024532e9bda4e272cd24048b
Author: James Petty <James@globomantics.com>
Date:   Wed Jun 14 16:04:11 2023 -0500
    added basic commands
PS C:\MyPSTool>
```

Последний введенный коммит оказался проблемным. В этой ситуации мы можем откатить Git так:

```
PS C:\MyPSTool> git reset --hard head~1
HEAD is now at 9113535 Added data.txt
PS C:\MyPSTool> get-content .\data.txt
don
```

Или предположим, что прошло какое-то время и мы выполнили несколько других коммитов: добавили файлы в наш тестовый репозиторий. Затем мы понимаем, что нужно откатить все до этого коммита:

```
commit 9113535942d0c35a964deda9e869a0193bb284ad
Author: James Petty <James@globomantics.com>
Date:   Wed Jun 14 16:12:31 2023 -0500
    Added data.txt
```

Мы можем снова использовать возможность отката, однако на этот раз указать хеш-номер коммита. Вам не нужен весь хеш — как правило, будет достаточно его короткого фрагмента из семи первых цифр.

Теперь репозиторий выглядит так:

```
PS C:\MyPSTool> dir
    Directory: C:\MyPSTool
Mode                 LastWriteTime         Length  Name
----                 -------------         ------  ----
-a----         6/14/2023   4:49 PM             13  data.txt
-a----         6/14/2023   3:47 PM             48  file1.ps1
-a----         6/14/2023   3:56 PM             66  file2.ps1
-a----         6/14/2023   4:50 PM              0  foo.txt
-a----         6/14/2023   4:46 PM            786  num.txt
PS C:\MyPSTool> get-content .\data.txt
james
jason
```

Далее нам нужно выполнить откат к коммиту `9113535942d0c35a964deda9e869a0193bb284ad`, снова используя короткое значение хеша:

```
PS C:\MyPSTool> git reset --hard 9113535
HEAD is now at 9113535 Added data.txt
```

После этого изменения репозиторий выглядит так:

```
PS C:\MyPSTool> get-content .\data.txt
don
PS C:\MyPSTool> dir
    Directory: C:\MyPSTool
Mode                 LastWriteTime          Length  Name
----                 -------------          ------  ----
-a----         6/14/2023   5:54 PM               5  data.txt
-a----         6/14/2023   3:47 PM              48  file1.ps1
-a----         6/14/2023   3:56 PM              66  file2.ps1
```

Это сложный процесс, и вряд ли вам захочется делать так постоянно, но мы хотели показать вам ценность системы управления версиями.

Есть и другие виды операций, которые может понадобиться отменить. Хорошее руководство по этой теме доступно в разделе *Git Basics — Undoing Things* по адресу http://mng.bz/p1AP.

19.3.5. Ветвление и слияние

Одним из преимуществ Git, благодаря которому можно сократить необходимость в откате изменений, является принцип *ветвления*. Ветка Git — это копия ваших файлов, возможно, из конкретного коммита. Вы можете работать с этими файлами, не затрагивая основные (производственные) копии. Затем в момент готовности можно будет объединить внесенные изменения с основной веткой.

Создадим в каталоге MyPSTool ветку `dev`:

```
PS C:\MyPSTool> git branch dev
PS C:\MyPSTool> git branch
  dev
* main
```

Звездочка указывает на ветку, которая активна в данный момент (checked out). Мы переключаемся на ветку `dev` и добавляем файл, используя командлет PowerShell `Set-Content`:

```
PS C:\MyPSTool> git checkout dev
git : Switched to branch 'dev'
    + CategoryInfo          : NotSpecified: (Switched to branch
'dev':String) [], RemoteException
    + FullyQualifiedErrorId : NativeCommandError
PS C:\MyPSTool> set-content -value '12345' -Path devdata.txt
```

```
PS C:\MyPSTool> dir
    Directory: C:\MyPSTool
Mode                 LastWriteTime         Length  Name
----                 -------------         ------  ----
-a----         6/14/2023   5:54 PM              5  data.txt
-a----         6/14/2023   6:03 PM              7  devdata.txt
-a----         6/14/2023   3:47 PM             48  file1.ps1
-a----         6/14/2023   3:56 PM             66  file2.ps1
```

PowerShell определит это изменение ветки как ошибку. Можно ее проигнорировать. Мы добавили файл, который можем видеть в этом каталоге. Теперь выполним `add` и сделаем коммит:

```
PS C:\MyPSTool> git add .
PS C:\MyPSTool> git commit -m "added devdata"
[dev 850ca50] added devdata
 1 file changed, 1 insertion(+)
 create mode 100644 devdata.txt
PS C:\MyPSTool> git status
On branch dev
nothing to commit, working tree clean
```

Но посмотрите, что происходит, если мы переключимся обратно на основную ветку (мы опустили сообщение об ошибке):

```
PS C:\MyPSTool> git checkout main
PS C:\MyPSTool> dir
    Directory: C:\MyPSTool
Mode                 LastWriteTime         Length  Name
----                 -------------         ------  ----
-a----         6/14/2023   5:54 PM              5  data.txt
-a----         6/14/2023   3:47 PM             48  file1.ps1
-a----         6/14/2023   3:56 PM             66  file2.ps1
```

Файла в ней нет. Если мы внесли изменения в файлы, то не увидим и их.

Мы снова переключились на ветку `dev` и внесли еще несколько изменений, после чего вернулись в основную. Нас интересуют различия между этими двумя ветками:

```
PS C:\MyPSTool> git diff dev
diff --git a/data.txt b/data.txt
index f71dff2..910fbb7 100644
--- a/data.txt
+++ b/data.txt
@@ -1,3 +1 @@
 don
-james
-jason
diff --git a/devdata.txt b/devdata.txt
deleted file mode 100644
```

```
index e56e15b..0000000
--- a/devdata.txt
+++ /dev/null
@@ -1 +0,0 @@
-12345
```

Не волнуйтесь, если пока вам это непонятно, — проверка различий необязательна. Но теперь мы интегрируем две эти ветки, иными словами, выполним их *слияние* (merge):

```
PS C:\MyPSTool> dir
    Directory: C:\MyPSTool
Mode                 LastWriteTime         Length  Name
----                 -------------         ------  ----
-a----         6/14/2023  6:12 PM              5  data.txt
-a----         6/14/2023  3:47 PM             48  file1.ps1
-a----         6/14/2023  3:56 PM             66  file2.ps1
PS C:\MyPSTool> git merge dev
Updating 9113535..b62af84
Fast-forward
data.txt | 2 ++
devdata.txt | 1 +
2 files changed, 3 insertions(+)
create mode 100644 devdata.txt
PS C:\MyPSTool> dir
Directory: C:\MyPSTool
Mode                 LastWriteTime         Length  Name
----                 -------------         ------  ----
-a----        6/14/2023   6:17 PM             18  data.txt
-a----        6/14/2023   6:17 PM              7  devdata.txt
-a----        6/14/2023   3:47 PM             48  file1.ps1
-a----        6/14/2023   3:56 PM             66  file2.ps1
```

Мы привели листинги до и после, чтобы вы могли увидеть изменения.

Использование веток — идеальный способ тестирования и разработки нового кода без страха напортачить в текущей версии. Если вы решите забраковать код или закончить работу с веткой, можете ее удалить:

```
PS C:\MyPSTool> git branch -d dev
Deleted branch dev (was b62af84).
```

19.4. ИСПОЛЬЗОВАНИЕ GIT С VS CODE

Поняв фундаментальные принципы Git, такие как ветвление, индексирование и коммиты, вы сможете использовать функционал этой системы в других продуктах наподобие VS Code. В этом редакторе кода есть встроенная поддержка Git, и для него существует несколько соответствующих сторонних расширений. Использовать этот функционал вы сможете, если на вашем компьютере установлена Git v2.0.0 или более свежая версия.

В VS Code вы можете открыть целый каталог, что удобно при разработке модуля. Если каталогом является репозиторий Git, то VS Code это обнаружит. На рис. 19.1 показан наш тестовый каталог, открытый в VS Code.

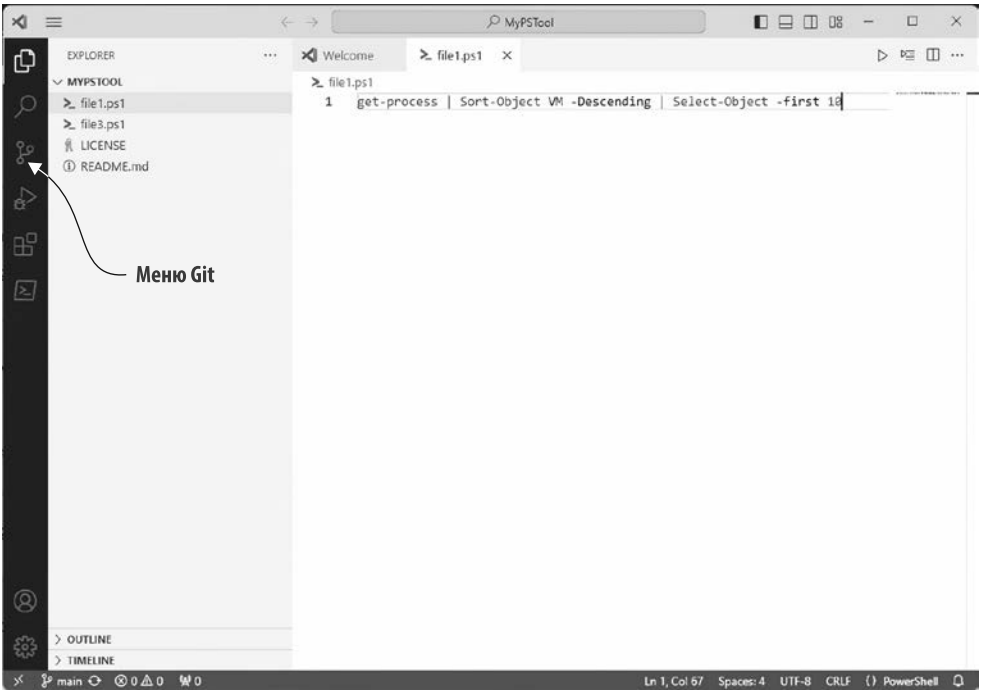


Рис. 19.1. Поддержка Git в VS Code

VS Code обнаружил текущую ветку. Здесь также есть значок для доступа к действиям, связанным с Git. С помощью редактора мы внесем некоторые изменения в файлы репозитория.

При обнаружении изменений VS Code отображает над значком Git число, обозначающее количество файлов. Щелкните на нем, чтобы увидеть изменения (рис. 19.2).

В консоли Git показывает изменения так:

```
PS C:\MyPSTool> git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified: file1.ps1
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file3.ps1
no changes added to commit (use "git add" and/or "git commit -a")
```

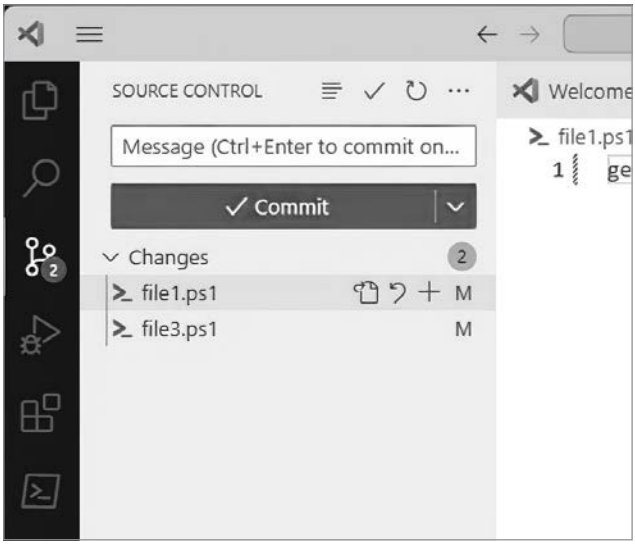


Рис. 19.2. Изменения Git в VS Code

Но вам необязательно использовать Git из командной строки. В VS Code вы можете навести указатель мыши на файл и индексировать или отклонять изменения для каждого отдельного файла либо делать это для всех файлов, наводя указатель на сами изменения. Мы индексировали все изменения (рис. 19.3). Теперь остается только закоммитить их, набрав сообщение коммита в рамке и щелкнув на значке «галочки». Кроме того, для выполнения прочих действий Git можно использовать всплывающее меню (рис. 19.4).

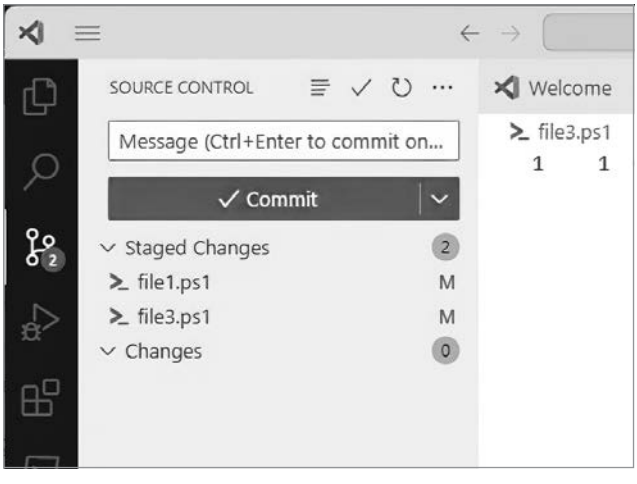


Рис. 19.3. Индексированные изменения в VS Code

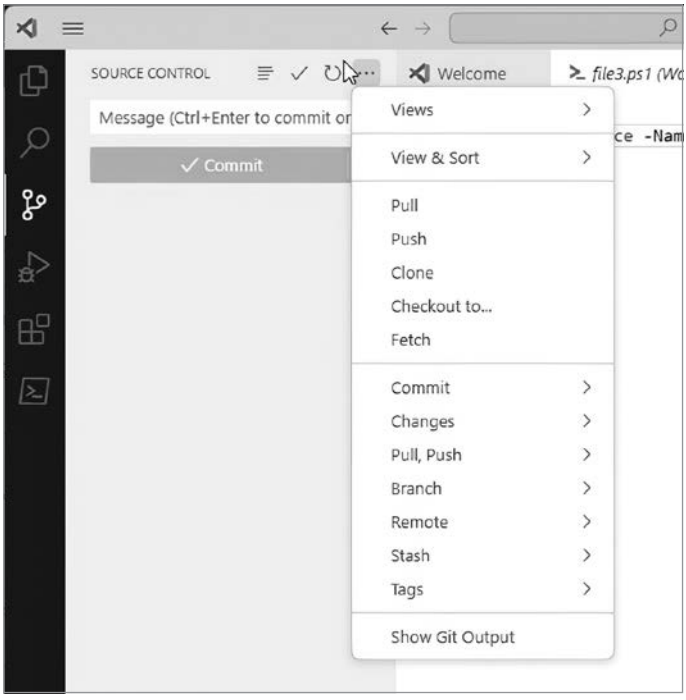


Рис. 19.4. Остальные функции Git

В редакторе можно даже переключаться на другие ветки или создавать их. Перейдите к палитре команд, нажав сочетание клавиш Ctrl+Shift+P. В поле введите `git`, и VS Code автоматически предложит доступные команды. Промотайте вниз до опции создания новой ветки и введите для нее имя. Редактор автоматически переключится на нее. Это можно определить по тому, что в левом нижнем углу будет указана активная ветка. Когда будете готовы, щелкните на ее имени, а затем в палитре команд — на имени ветки, на которую хотите переключиться.

VS Code позволяет просматривать, отменять и сравнивать изменения. Уделите время изучению и других связанных с Git значков в приложении.

Но VS Code — в первую очередь редактор, а не графический инструмент Git, поэтому для выполнения некоторых операций все же потребуется командная строка. Один из примеров таких операций — слияние. Да, вы можете создать новую ветку, изменять и коммитить файлы, но в той версии VS Code, которая доступна на момент написания книги, нет возможности выполнять слияние. К счастью, для выполнения команд Git можно использовать терминал, встроенный в редактор (рис. 19.5).

СОВЕТ

Более подробную информацию о VS Code и интеграции исходного кода вы можете получить, перейдя по ссылке http://mng.bz/OP5P.

Рис. 19.5. Команды Git из терминала VS Code

19.5. ИНТЕГРАЦИЯ С GITHUB

Еще одним отличным инструментом, интегрированным в Git, является GitHub. Это веб-репозиторий Git, на котором размещается сервис с собственным набором возможностей. На данном ресурсе вам доступны множество уровней на выбор. Технически в этом случае вам необязательно иметь на компьютере Git, но многие все же устанавливают данную систему. Это позволяет клонировать онлайн-репозитории, вносить в них изменения локально, а затем передавать их обратно на GitHub. К тому же этот веб-репозиторий дает возможность разработчикам сотрудничать. Если вам интересно, то изучите эти страницы:

- https://github.com/psjamesp;
- https://github.com/powershellorg;
- https://github.com/devops-collective-inc;
- https://github.com/powershell.

Интеграция Git с GitHub (особенно когда вы начинаете клонировать другие репозитории и вносить изменения с помощью соответствующих запросов) может выглядеть непонятной и даже раздражать. Но мы хотели дать вам некое базовое представление того, как GitHub можно использовать в работе.

Предположим, что в этом веб-репозитории вы хотите создать копию проекта MyPSTool, над которым работали локально. В GitHub можно хранить основной код в процессе его локальной разработки и корректировки, и если кому-то другому потребуется поработать над проектом, то он сможет клонировать копию репозитория на свой локальный компьютер.

В целях упрощения мы будем использовать репозиторий Джеймса (https://github.com/psjamesp), что также даст вам возможность клонировать его и выполнять все самостоятельно. Кроме того, это означает, что мы изменили в нашей конфигурации Git имя пользователя и адрес почты, чтобы они соответствовали учетным данным Джеймса. Мы предполагаем, что когда вы зарегистрируетесь на GitHub (это можно сделать бесплатно), то сможете взять те же данные, которые используете локально, или наоборот.

Есть два способа интеграции локального проекта Git и GitHub. Выбор зависит от того, с чего вы начинаете. В нашем случае у нас уже есть локальный репозиторий, который мы хотим отправить на GitHub. В самом же GitHub мы создадим новый публичный репозиторий (рис. 19.6).

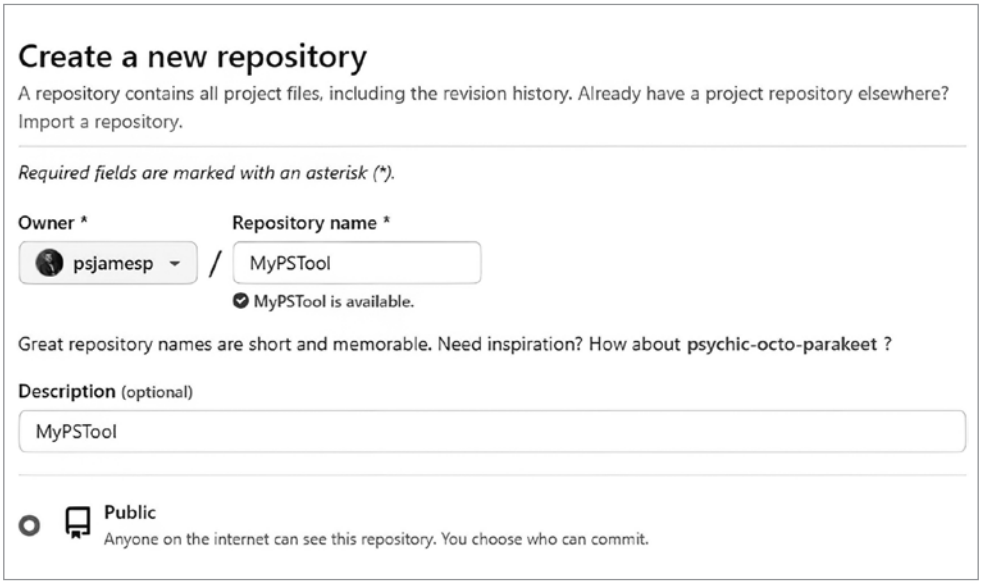


Рис. 19.6. Создание репозитория GitHub

Это необязательно, но мы советуем использовать то же имя, которое носит локальный каталог. Можете добавить описание. В этом случае вам не потребуется добавлять файл readme или что-то еще, поскольку вы будете использовать существующий локальный репозиторий.

Далее GitHub в соответствии с ситуацией предоставляет нужный вам код. В нашем случае мы хотим передать имеющийся репозиторий через командную строку. Для этого мы используем следующие команды из корня локального каталога:

```
PS C:\MyPSTool> git remote add origin
https://github.com/psjamesp/MyPSTool.git
PS C:\MyPSTool> git push -u origin main
Counting objects: 17, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (17/17), 1.46 KiB | 0 bytes/s, done.
Total 17 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/psjamesp/MyPSTool.git
 * [new branch]      main -> main
Branch main set up to track remote branch main from origin.
```

Добавляет удаленную ссылку на GitHub

Отправляет основную ветку в удаленный репозиторий

Проверить конфигурацию удаленного сервера можно так:

```
PS C:\MyPSTool> git remote
Origin
```

В качестве альтернативы можно получить более подробные сведения:

```
PS C:\MyPSTool> git remote -v
origin https://github.com/psjamesp/MyPSTool.git (fetch)
origin https://github.com/psjamesp/MyPSTool.git (push)
```

Теперь вы можете видеть на GitHub репозиторий с текущими файлами (рис. 19.7).

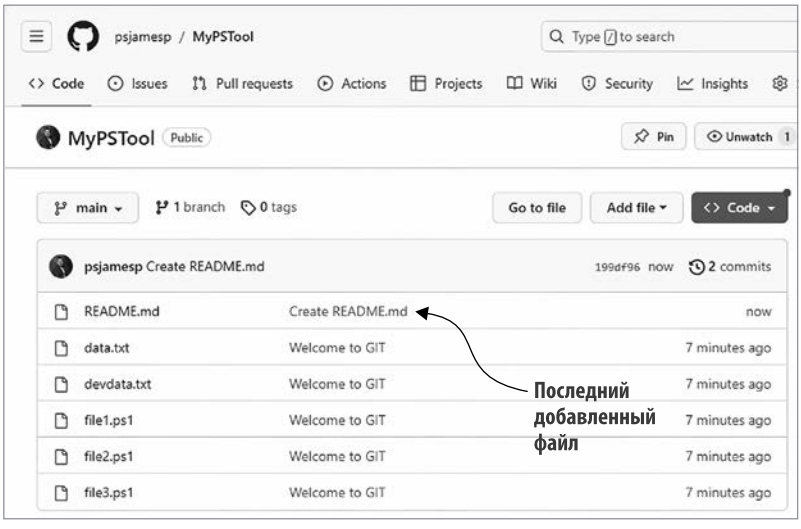


Рис. 19.7. Теперь локальный репозиторий находится на GitHub

Вы могли бы внести изменения с помощью редактора на GitHub, но мы предполагаем, что вы будете делать это локально. При этом можете использовать локальные команды Git как обычно, например, коммитить файлы:

```
PS C:\MyPSTool> git commit -m 'new changes'
[main 737445d] new changes
 3 files changed, 9 insertions(+), 1 deletion(-)
 create mode 100644 file4.ps1
```

Но теперь при проверке статуса Git сообщит вам, что локальный репозиторий не синхронизирован с репозиторием GitHub:

```
PS C:\MyPSTool> git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)
nothing to commit, working tree clean
```

Ресурс даже предоставит вам инструкции, сообщив о доступных возможностях.

```
PS C:\MyPSTool> git push
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (5/5), 600 bytes | 0 bytes/s, done.
Total 5 (delta 0), reused 0 (delta 0)
To https://github.com/psjamesp/MyPSTool.git
   abeeecd..737445d  main -> main
```

Вернитесь к браузеру, обновите вкладку — и увидите изменения.

Если вы или ваш коллега измените файлы на GitHub, то должны вручную проверить и подтянуть эти изменения. Выполнение `git status` не покажет вам, что удаленные файлы изменились:

```
PS C:\MyPSTool> git status
On branch main
Your branch is up-to-date with 'origin/main'.
nothing to commit, working tree clean
```

Вам потребуется получить их (`git fetch`) или подтянуть (`git pull`):

```
PS C:\MyPSTool> git fetch
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.
From https://github.com/psjamesp/MyPSTool
   737445d..01f65d7  main     -> origin/main
```

Команда `fetch` получает удаленные изменения. Если система выдаст сообщение об отсутствии изменений, как показано выше, тут все ясно. Если же при получении

изменений что-то возвращается, то потребуется подтянуть (`git pull`) измененные файлы из удаленного репозитория:

```
PS C:\MyPSTool> git pull
Updating 737445d..01f65d7
Fast-forward
 data.txt  | 1 -
 file1.ps1 | 2 +-
 2 files changed, 1 insertion(+), 2 deletions(-)
PS C:\MyPSTool>
```

Это изменения, которые мы внесли на GitHub. Опять же, локальный и удаленный репозитории синхронизированы.

Еще один способ — создать свой проект на GitHub и затем клонировать его локально. Выполните все те же действия, с помощью которых можно добавить репозиторий на GitHub. Мы добавили свой с файлом readme и лицензией, пропускающий страницу с командами. Затем нажмите кнопку Clone или Download и скопируйте ссылку в буфер обмена.

В PowerShell установите в качестве расположения родительский каталог, в котором вы хотите создать репозиторий. В целях демонстрации мы создали репозиторий GitHub для набора инструментов SharePoint, который собираемся написать (как будто). Мы хотели, чтобы локальный репозиторий находился в каталоге `C:\scripts`, поэтому перед выполнением команды `git clone` переключились на него:

```
PS C:\scripts> git clone https://github.com/psjamesp/sharepointtools.git
Cloning into 'sharepointtools'...
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), done.
```

После этого мы перешли в созданный репозиторий, чтобы просмотреть новые файлы:

```
PS C:\scripts> cd .\sharepointtools\
PS C:\scripts\sharepointtools> dir
    Directory: C:\scripts\sharepointtools
Mode                 LastWriteTime         Length  Name
----                 -------------         ------  ----
-a----        6/19/2023   2:59 PM           1088  LICENSE
-a----        6/19/2023   2:59 PM             44  README.md
```

С этого момента мы делали те же действия, которые показывали вам выше.

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Для этой темы у нас нет упражнений. Работа с Git — это навык, который вам нужно вырабатывать самостоятельно. Мы рекомендуем вам установить Git на тестовый компьютер, создать каталог и начать экспериментировать с различными командами. Опыт — лучший учитель. Если вы столкнетесь с проблемой, то сможете найти очень много справочной информации и советов в Интернете.

ИТОГИ ГЛАВЫ

Нам неважно, какую систему управления версиями вы используете, но мы очень рекомендуем применять *хоть какую-то*. Git — отличный выбор, поскольку эта система широко известна, в Интернете есть очень много вспомогательных материалов по ней и обычно все профессиональные разработчики используют именно Git. Эта технология сродни иностранному языку: вы не сможете овладеть ею профессионально, если не будете использовать постоянно.

Вам необязательно задействовать GitHub, но это очень удобный инструмент для совместной работы над проектами. К тому же вы можете размещать в данном веб-репозитории свой код. У вашей компании уже, возможно, есть доступный корпоративный аккаунт GitHub, который вы можете использовать, или частный сервер репозиториев, который предоставляет ту же функциональность.

20 Тестирование скрипта

По мере перехода в мир, ориентированный на DevOps, обеспечение надежности скриптов становится крайне важной задачей. Никто не хочет, чтобы в производственной среде работал скрипт с ошибками. Даже если вы изначально протестируете свой скрипт, могут возникнуть изменения или некие уникальные условия, из-за которых тестирование потребуется повторить. В этой главе мы рассмотрим автоматизированное модульное тестирование для скриптов PowerShell, выполняемое с помощью инструмента Pester.

20.1. ЦЕЛЬ

Мы хотим, чтобы вы использовали следующий алгоритм.

1. Вы пишете новый код или изменяете старый.
2. Проверяете свой код в репозитории системы управления версиями.
3. Репозиторий запускает конвейер непрерывной интеграции. Этот конвейер, обычно включающий сторонние инструменты, создает виртуальную машину для тестирования вашего скрипта, копирует скрипт на нее и выполняет несколько автоматизированных тестов. Если они завершаются неудачей, то вы получаете письмо с описанием произошедшего.
4. Если тесты проходят успешно, то ваш код развертывается в репозитории (возможно, на PowerShellGallery.com или в закрытом репозитории) и становится доступным для производственной среды.

Этап 3 в данном списке, называемый «Чудо», полностью автоматизирован. Чтобы сделать это чудо возможным, вам нужно добавить в код механизм автоматизированного тестирования. Это позволит вам быстро тестировать любые изменения кода, подтверждая его работоспособность перед развертыванием.

20.2. ПРОБЛЕМЫ С РУЧНЫМ ТЕСТИРОВАНИЕМ

Мы уверены, что вы уже тестировали скрипты вручную — возможно, даже когда писали их, выполняя задания, приведенные в книге. И это прекрасно, ведь вам нужно тестировать код по мере его создания. Но ручное тестирование сопряжено с некоторыми проблемами.

- Вы наверняка ленивы, да и мы тоже. Поэтому вряд ли вы планируете всякий раз выполнять каждый тест. Но по закону подлости часто будет оказываться, что именно пропущенный тест должен был обнаружить ошибку, которую вы допустили в коде.
- Даже если вы не ленитесь, ручное тестирование требует времени и сил, которые можно более эффективно потратить на что-то еще.
- Ручное тестирование не заучиваемый процесс. Вряд ли у вас есть огромный список тестов, которые, как вы знаете, нужно выполнить. Вы наверняка делаете то же, что и мы, думая: «Я выполню данный тест один раз с параметрами и еще раз, передав в него что-нибудь. Пожалуй, этого будет достаточно». Если вы исправите проблему, то можете тут же протестировать результат, но станете ли вы снова тестировать ее в будущем — вам это неизвестно.
- Это ручной процесс. Вы не можете достичь «Чуда» с помощью ручного тестирования. Напомним, что PowerShell ориентирован на автоматизацию — почему тестирование должно быть исключением?

20.3. ПРЕИМУЩЕСТВО АВТОМАТИЗИРОВАННОГО ТЕСТИРОВАНИЯ

Автоматическое тестирование, напротив, во многом выигрывает — в основном потому, что оно *автоматизированное* и *«обучаемое»*. Если вы однажды сталкиваетесь с нештатным состоянием, из-за которого работа вашего кода нарушается, то добавляете его в тестовый скрипт и уже не забываете протестировать его снова. Поэтому автоматические тесты выступают своего рода задокументированной и организованной памятью. Даже если ваш скрипт изменит кто-то другой, не зная об этом нештатном состоянии, автоматизированный тест подстрахует его и обработает возникшее состояние.

Автоматизированные тесты даже могут перенести вас в мир разработки на основе тестирования (test-driven development, TDD). Предположим, вы решили добавить в команду некую функциональность. И вместо того чтобы вскрывать и изменять скрипт команды, вы пишете несколько тестов, с помощью которых сможете проверить эту новую функциональность. В тестах вы описываете, *как* она должна работать, чтобы они служили в качестве функциональной спецификации. Поначалу эти тесты будут завершаться неудачно, поскольку вы еще не прописали новую

функциональность. Но затем вы постепенно будете ее создавать, пока все тесты не пройдут успешно. Если в тестах вы все сделаете правильно, то и функциональность будет работать корректно.

20.4. PESTER

Pester (PowerShell Tester) — это проект с открытым исходным кодом, который поставляется в комплекте с Windows 10 и старше (обновленные версии можно найти в PowerShell Gallery). Это фреймворк автоматизированного модульного тестирования для PowerShell. Иными словами, вы пишете тесты в Pester, и он их автоматически выполняет. Базовая документация Pester находится в репозитории GitHub по адресу https://github.com/pester/Pester/wiki.

ПРИМЕЧАНИЕ

В этой главе мы дадим лишь общее описание Pester, чтобы подогреть ваш интерес. Лучше всего, если вы самостоятельно прочитаете документацию, в которой описаны все возможности этого фреймворка.

Интересно отметить, что в Microsoft используют Pester для автоматизации тестирования собственных ресурсов PowerShell. Все виды тестов Pester добавлены в различные компоненты с открытым исходным кодом, написанные на базе PowerShell ее разработчиками. И эти тесты исчисляются тысячами. Это как минимум должно показать вам, насколько важен Pester для сообщества PowerShell.

20.5. КОД ДЛЯ ТЕСТИРОВАНИЯ

Чтобы успешно использовать Pester, вам нужно начать писать команды и скрипты, поддающиеся тестированию. Следуйте при этом всем нашим рекомендациям. В частности, старайтесь создавать автономные инструменты, предназначенные для решения одной задачи. Инструменты, которые выполняют восемь разных задач, будет трудно тестировать, поскольку придется проверить каждую из них во всех возможных комбинациях. Если же инструмент выполняет одну задачу, то писать и тестировать его намного проще.

Кроме того, важно учитывать, что Pester является фреймворком тестирования для PowerShell — не .NET, SQL Server или чего-либо еще. Поэтому работает он лучше всего именно применительно к командам PowerShell. Если вы последуете нашим рекомендациям (которые мы дадим дальше), то будете писать команды PowerShell для обертывания любого кода, не имеющего отношения к PowerShell, который вам может понадобиться. То есть в итоге вы работаете *именно* с командами PowerShell. В этом сценарии ваше взаимодействие с Pester будет продуктивным.

20.6. ЧТО ВЫ ТЕСТИРУЕТЕ?

В этой главе мы даем лишь общее описание Pester, поэтому для более качественной привязки к контексту PowerShell введем пару терминов, играющих серьезную роль в сфере автоматизированного тестирования. В частности, мы используем термины *«модульное тестирование»* и *«интеграционное тестирование»*, чтобы сформировать два сценария, поясняющих, для чего вообще пишутся тесты.

20.6.1. Интеграционные тесты

Интеграционные тесты проверяют *конечное состояние* вашей команды. Если вы написали команду для создания базы данных SQL Server, то интеграционный тест выполнит ее и проверит, была ли эта база создана. Иными словами, он тестирует итоговый эффект работы вашего кода. Тест рассматривает ваш код как черный ящик, то есть необязательно знает, что происходит *внутри* кода. Он не проверяет, работают ли указанные вами имя пользователя и пароль. Тест просто проверяет результат. Интеграционные тесты позволяют проверить, выполняет ли ваш набор инструментов конкретную задачу в различных ситуациях.

Интеграционные тесты хороши, но существуют и другие.

20.6.2. Модульные тесты

Модульные тесты более детализированы, и их сложнее представить. Они не выясняют, *реализует* ли ваш код какую-либо задачу, а лишь проверяют, *выполняется ли* этот код.

Например, у вас может быть команда, которая меняет режим запуска сервиса и пароль авторизации или же выполняет что-то одно из этого на основе предоставляемых параметров. Модульный тест будет выполняться всеми тремя способами, проверяя корректную работу внутренних логических решений и путей кода. При этом он не выясняет, менялся ли конкретный сервис.

Зачастую вы будете писать и модульные, и интеграционные тесты. Могут возникать ситуации, когда вы пишете исключительно модульные, поскольку вас волнует только то, правильны ли пути выполнения кода и принимает ли он корректные логические решения. Причиной выбора модульного теста может быть и то, что *выполнение чего-либо*, необходимого для интеграционного теста, навредит вашей рабочей среде или скажется на ней негативно. И усвоить этот принцип может быть сложно. Например, если вы написали команду, перезагружающую компьютер, то могли ли не проверить, *перезагружается* ли он? Это зависит от обстоятельств. Если бы вы вызвали команду наподобие `Restart-Computer`, то тестировать бы ее не пришлось: вы бы тестировали код, который *ведет к* вызову `Restart-Computer`, что подводит нас к следующему моменту.

20.6.3. Не тестируйте чужой код

Конкретно в случае модульных тестов вашей целью является тестирование *своего кода*. Команда `Restart-Computer` *не является вашим кодом*. Это код Microsoft. Если он с багами, то это не ваша проблема. Ваш модульный тест проверяет корректность работы кода, который принадлежит вам. Возьмем этот конкретный сценарий и превратим его в пример Pester.

20.7. НАПИСАНИЕ ПРОСТОГО ТЕСТА В PESTER

Начнем с команды, приведенной в листинге 20.1. Она намеренно проста, чтобы можно было сосредоточить внимание на аспекте модульного тестирования. Эта команда позволяет перезагрузить или выключить компьютер.

Листинг 20.1. Тестирование команды

```
function Set-ComputerState {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True)]
        [string[]]$ComputerName,
        [Parameter(Mandatory=$True)]
        [ValidateSet('Restart','Shutdown')]
        [string]$Action,
        [switch]$Force
    )
    BEGIN {}
    PROCESS {
        ForEach ($comp in $ComputerName) {
            $params = @{'ComputerName' = $comp}
            # force?
            if ($force) {
                $params.Add('Force',$true)
            }
            # Какое действие?
            If ($Action -eq 'Restart') {
                Write-Verbose "Restarting $comp (Force: $force)"
                Restart-Computer @params
            } else {
                Write-Verbose "Stopping $comp (Force: $force)"
                Stop-Computer @params
            }
        }
    } #PROCESS
END {}
}
```

ПРОЧТИТЕ СЕЙЧАС

Внимательно прочитайте эту команду и сформируйте ожидание в отношении того, что и как она делает. Вы можете придумать и другие, в том числе более эффективные способы ее реализации. Мы проделали все это, чтобы продемонстрировать тестирование с помощью Pester.

Когда дело доходит до модульного тестирования, мы сразу понимаем, что будем проверять два момента: работоспособность `Restart-Computer` и `Stop-Computer` — несмотря на то что это единственные команды, которые что-либо делают. Напомним: если бы мы писали интеграционный тест, это бы имело значение. Разница в том, что модульные тесты не ориентированы на конечный результат. Их задача — проверить *корректность выполнения кода*. Мы не будем проводить модульный тест, поскольку эти команды не относятся к нашему коду.

ВНУТРИ ИЛИ СНАРУЖИ?

Модульные и интеграционные тесты можно рассматривать и с позиции того, *о каком количестве вашего кода знает тест.*

В случае интеграционного теста код — своего рода черный ящик, как мы ранее и предполагали. Тест не знает, как вы реализуете перезапуск или выключение. Он отслеживает лишь то, произошли ли эти события. Интеграционный тест *ничего не знает* о содержимом команды. Он не будет стараться проверить каждый возможный путь выполнения кода, использование каждого возможного параметра и т. д.

Иначе обстоят дела в случае модульного теста. Его интересует не результат, а лишь то, *весь* ли код выполнился. Каждый ли параметр использовался? Каждый ли путь кода был выполнен? Было ли выполнено каждое логическое решение во всех комбинациях? Весь вопрос в *коде*, а не в его *результате*.

Повторимся, оба вида тестов важны, но мы пока сосредоточимся на модульных.

20.7.1. Создание фикстуры

Начнем с загрузки модуля Pester и создания новой фикстуры теста:

```
PS C:> Import-Module Pester
PS C:> Mkdir example
PS C:> New-Fixture -Path example -Name Set-ComputerState
```

Эта новая *фикстура* представляет два пустых файла: один для нашего кода (`Set-ComputerState.ps1`) и второй для наших тестов (`Set-ComputerState.Tests.ps1`). Рассматривайте эту фикстуру как скелет. Мы откроем оба файла в Visual Studio Code, после чего вставим нашу функцию в `Set-ComputerState.ps1` в качестве отправной точки, заменив пустую функцию `Set-ComputerState`.

УСТАНОВКА И ОБНОВЛЕНИЕ PESTER

Мы предполагаем, что модуль Pester установлен у вас в системе. В Windows 10 и более свежих версиях ОС он встроен по умолчанию. Если же модуля у вас нет, то сначала установите его из PowerShell Gallery, выполнив `Install-Module Pester`.

Если вы работаете на компьютере с Windows 10 или более свежей версии, то встроенный в ОС модуль Pester будет устаревшим. К сожалению, обновлять его из PowerShell Gallery проблематично. Вы не можете деинсталлировать встроенную версию (по крайне мере, это нелегко), и у вас могут быть проблемы с получением последней версии. Более подробно тема раскрывается в статье *Power-Shell Package Management and PowerShellGet Module Changes in Windows 10 Version 1511, 1607, and 1703*, написанной ведущим сотрудником Microsoft Майком Роббинсом (Mike Robbins) (3 августа 2017 года, http://mng.bz/40c7). В крайнем случае вы можете установить последнюю версию модуля Pester и запустить ее параллельно с поставляемой версией, используя следующую команду:

```
Install-module pester -Repository psgallery -force -SkipPublisherCheck
```

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Повторяйте за нами — настройте фикстуру и вставьте в скрипт кода листинг 20.1.

Скрипт теста (для которого мы не будем приводить скачиваемый образец, поскольку вы должны создать его сами, выполнив предыдущие команды) будет выглядеть так:

```
BeforeAll {
    . $PSCommandPath.Replace('.Tests.ps1', '.ps1')
}

Describe "Set-ComputerState" {
    It "Returns expected output" {
        Set-ComputerState | Should -Be "YOUR_EXPECTED_VALUE"
    }
}
```

Помимо первых трех команд вверху, которые связывают код теста с кодом скрипта, здесь есть два раздела.

- Блок `Describe` содержит набор тестов. Все они выполняются внутри одной и той же области. Деление на области в Pester — непростая, но мощная возможность. И по мере погружения во все более сложные тесты вы зачастую будете определять несколько блоков `Describe`. Мы же пока обойдемся этим.
- Блок `It` представляет один тест, который наш код пройдет успешно или провально. Блок `Describe` зачастую содержит множество блоков `It`, каждый из которых тестирует конкретное отдельное состояние.

20.7.2. Написание первого теста

Теперь изменим предоставленный блок `It` и что-нибудь протестируем:

```
Describe "Set-ComputerState" {
    It "accepts one computer name" {
        Set-ComputerState -ComputerName SERVER1 -Action restart |
        Should Be $true
    }
}
```

Это базовая модель блока `It`: вы выполняете что-либо, после чего сообщаете Pester, каким должен быть результат. Тем не менее написанный нами здесь пример работать не будет, поскольку функция `Set-ComputerState` никогда ничего не выводит в конвейер. Поэтому она ничего не выводит в `Should` и не должна (`Should`) проверять значение `$true`. И это создает серьезную проблему: когда у нас есть функция, которая не создает никакого вывода, и мы не пытаемся протестировать, выполняет ли она что-либо, то как вообще можно ее протестировать?

Если говорить конкретнее, то нам нужно увидеть, *сколько раз вызывается* `Restart-Computer`, *фактически ее не вызывая*. Это непростая задача. И решить ее поможет ключевой элемент Pester — макет.

20.7.3. Создание макета

В тестировании вам зачастую нужно иметь команды, которые *вроде бы* выполняются, но по факту *нет*. Например, вам может потребоваться `Import-CSV` для импорта конкретного CSV-файла, но создавать этот файл вы не хотите. Или, как в нашем случае, мы хотим, чтобы `Restart-Computer` *как будто* выполнилась, чтобы мы, не перезагружая компьютер, могли выяснить, пытался ли код выполнить ее. Именно для решения таких задач и можно создавать макеты Pester. В этом случае вы создаете для существующей команды фиктивную подмену, которая может делать все, что вам нужно:

```
Describe "Set-ComputerState" {
    BeforeAll {
        Mock Restart-Computer { return 1}
        Mock Stop-Computer { return 1}
    }

    It "accepts one computer name" {
        Set-ComputerState -computerName Localhost -action Restart |
        Should -Be 1
    }
}
```

Наша фиктивная версия `Restart-Computer` теперь будет выводить `1`. Она не будет перезапускать никакие компьютеры — просто выводить `1`. Итак, если ее вызвать один раз, то результатом `Set-ComputerState` должна быть `1`.

Об этом мы сообщили Pester с помощью блока `It`. Попробуем выполнить этот простой тест, чтобы понять, работает ли он. Для этого нам нужно из каталога с примером, в котором находится наш тестовый скрипт, выполнить `Invoke-Pester`:

```
Describing Set-ComputerState
 [+] accepts one computer name 678ms
Tests completed in 678ms
Passed: 1 Failed: 0 Skipped: 0 Pending: 0 Inconclusive: 0
```

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Лучше всего наблюдать полный результат, поэтому попробуйте получить аналогичный вывод, скопировав весь прописанный ранее пример.

Символ `[+]` говорит о том, что наш единственный тест пройден.

20.7.4. Добавление дополнительных тестов

Добавим еще несколько тестов:

```
Describe "Set-ComputerState" {
    Mock Restart-Computer { return 1 }
    Mock Stop-Computer { return 1 }
    It "accepts and restarts one computer name" {
        Set-ComputerState -ComputerName SERVER1 -Action Restart |
        Should -Be 1
    }
    It "accepts and restarts many names" {
        $names = @('SERVER1','SERVER2','SERVER3')
        $result = Set-ComputerState -ComputerName $names -Action Restart
        $result.Count | Should -Be 3
    }
    It "accepts and restarts from the pipeline" {
        $names = @('SERVER1','SERVER2','SERVER3')
        $result = $names | Set-ComputerState -Action Restart
        $result.count | Should -Be 3
    }
}
```

Сохраняет результаты в переменную

Проверяет количество результатов

В этих двух тестах мы пошли иным путем. Напомню, что при каждом выполнении макета `Restart-Computer` команда выдает `1`. Это означает, что ее трехкратное выполнение не приведет к выводу `3`. Она три раза выведет `1`. Мы добавляем эту коллекцию целых чисел в `$result`. Затем на новой строке передаем `$result.Count` в `Should`, проверяя, содержит ли массив три элемента. Так мы узнаем, что имитированная команда была вызвана три раза. Наши результаты выглядят следующим образом:

```
Describing Set-ComputerState
 [+] accepts and restarts one computer named 252ms
 [+] accepts and restarts many names 374ms
 [+] accepts and restarts from the pipeline 332ms
Tests completed in 959ms
Passed: 3 Failed: 0 Skipped: 0 Pending: 0 Inconclusive: 0
```

Прекрасно! Но создавать эти тесты можно и более эффективным способом. Когда вы имитируете команду в Pester, то «за кадром» фреймворк автоматически отслеживает, сколько раз использовался этот макет. Наша цель — подсчитать, сколько раз выполнилась фейковая команда, так что мы можем попросить Pester проделать это все за нас. Мы используем команду `Should -Invoke`:

```
Describe "Set-ComputerState" {
    Mock Restart-Computer { return 1 }
    Mock Stop-Computer { return 1 }
    It "accepts and restarts one computer name" {
        Set-ComputerState -ComputerName SERVER1 -Action Restart
        Should -Invoke Restart-Computer -Exactly 1
    }
    It "accepts and restarts many names" {
        $names = @('SERVER1','SERVER2','SERVER3')
        Set-ComputerState -ComputerName $names -Action Restart
        Should -Invoke Restart-Computer -Exactly 3
    }
    It "accepts and restarts from the pipeline" {
        $names = @('SERVER1','SERVER2','SERVER3')
        $names | Set-ComputerState -Action Restart
        Should -Invoke Restart-Computer -Exactly 3
    }
}
```

Тестирует, сколько раз вызывался макет

Попробуем:

```
Describing Set-ComputerState
 [+] accepts and restarts one computer named 740ms
 [-] accepts and restarts many names 144ms
  Expected Restart-Computer to be called 3 times exactly but was
    called 4 times
  18:         Should -Invoke Restart-Computer -Exactly 3
  at <ScriptBlock>, \\vmware-host\Shared Folders\Documents\example\Set-
ComputerState.Tests.ps1: line 18
 [-] accepts and restarts from the pipeline 409ms
  Expected Restart-Computer to be called 3 times exactly but was
    called 7 times
  24:         Should -Invoke Restart-Computer -Exactly 3
  at <ScriptBlock>, \\vmware-host\Shared Folders\Documents\example\Set-
ComputerState.Tests.ps1: line 24
Tests completed in 1.29s
Passed: 1 Failed: 2 Skipped: 0 Pending: 0 Inconclusive: 0
```

Нехорошо. Судя по неудачному выводу, похоже, будто по умолчанию счетчик не выполнял сброс для каждого блока `It`. Нужно изменить команду, уточнив, что она должна подсчитывать каждый блок `It`, а не суммировать все, что произошло в родительском блоке `Describe`:

```
Describe "Set-ComputerState" {
    Mock Restart-Computer { return 1 }
    Mock Stop-Computer { return 1 }
```

```
    It "accepts and restarts one computer name" {
        Set-ComputerState -ComputerName SERVER1 -Action Restart
        Should -Invoke Restart-Computer -Exactly 1 -Scope It
    }
    It "accepts and restarts many names" {
        $names = @('SERVER1','SERVER2','SERVER3')
        Set-ComputerState -ComputerName $names -Action Restart
        Should -Invoke Restart-Computer -Exactly 3 -Scope It
    }
    It "accepts and restarts from the pipeline" {
        $names = @('SERVER1','SERVER2','SERVER3')
        $names | Set-ComputerState -Action Restart
        Should -Invoke Restart-Computer -Exactly 3 -Scope It
    }
}
```

Отслеживает утвержденные в области макеты

И попробуем снова:

```
Describing Set-ComputerState
 [+] accepts and restarts one computer name 430ms
 [+] accepts and restarts many names 335ms
 [+] accepts and restarts from the pipeline 283ms
Tests completed in 1.05s
Passed: 3 Failed: 0 Skipped: 0 Pending: 0 Inconclusive: 0
```

Мы получили именно то, что искали.

20.7.5. Покрытие кода тестами

Если одна из целей модульного тестирования — убедиться, что выполняется весь ваш код, то вам нужно знать, достигли ли вы ее. И в этом вам поможет Pester. Выполнение `Invoke-Pester-Code-Coverage ./Set-ComputerState.ps1` приведет к созданию *отчета о покрытии кода* для этого скрипта. Отчет может выглядеть примерно так:

```
Describing Set-ComputerState
 [+] accepts and restarts one computer name 1.64s
 [+] accepts and restarts many names 68ms
 [+] accepts and restarts from the pipeline 1.55s
Tests completed in 3.26s
Passed: 3 Failed: 0 Skipped: 0 Pending: 0 Inconclusive: 0
Code coverage report:
Covered 70.00 % of 10 analyzed commands in 1 file.
Missed commands:
File                   Function          Line  Command
----                   --------          ----  -------
Set-ComputerState.ps1  Set-ComputerState   24  $params.Add('Force',$true)
Set-ComputerState.ps1  Set-ComputerState   32  Write-Verbose "Stopping $comp
(Force: $force)"
Set-ComputerState.ps1  Set-ComputerState   33  Stop-Computer @params
```

Отчет поможет понять, чего не хватает. Стопроцентное покрытие кода означает, что *выполнялась каждая его строка*. Этот показатель необязательно говорит о том, что тестирование закончено, поскольку иногда вам нужно протестировать различные вариации с одним и тем же кодом. Но покрытие кода помогает выявить пути выполнения, которые вы могли упустить. В нашем случае мы видим, что никогда не выполняли код, относящийся к параметру `-Force`, и никогда не выполняли тест, в котором пытались бы выключить компьютер, а не перезапустить. Добавим еще немного тестов:

```
Describe "Set-ComputerState" {
    Mock Restart-Computer { return 1 }
    Mock Stop-Computer { return 1 }
    It "accepts and restarts one computer name" {
        Set-ComputerState -ComputerName SERVER1 -Action Restart
        Should -Invoke Restart-Computer -Exactly 1 -Scope It
    }
    It "accepts and restarts many names" {
        $names = @('SERVER1','SERVER2','SERVER3')
        Set-ComputerState -ComputerName $names -Action Restart
        Should -Invoke Restart-Computer -Exactly 3 -Scope It
    }
    It "accepts and restarts from the pipeline" {
        $names = @('SERVER1','SERVER2','SERVER3')
        $names | Set-ComputerState -Action Restart
        Should -Invoke Restart-Computer -Exactly 3 -Scope It
    }                                                          Дополнительные
    It "accepts and force-restarts one computer name" {        тесты
        Set-ComputerState -ComputerName SERVER1 -Action Restart -Force
        Should -Invoke Restart-Computer -Exactly 1 -Scope It
    }                                                          Дополнительные
    It "accepts and shuts down one computer name" {            тесты
        Set-ComputerState -ComputerName SERVER1 -Action Shutdown
        Should -Invoke Stop-Computer -Exactly 1 -Scope It
    }
}
```

И выполним этот код:

```
Describing Set-ComputerState
 [+] accepts and restarts one computer name 552ms
 [+] accepts and restarts many names 64ms
 [+] accepts and restarts from the pipeline 86ms
 [+] accepts and force-restarts one computer name 277ms
 [+] accepts and shuts down one computer name 115ms
Tests completed in 1.1s
Passed: 5 Failed: 0 Skipped: 0 Pending: 0 Inconclusive: 0
Code coverage report:
Covered 100.00 % of 10 analyzed commands in 1 file.
```

Теперь мы более уверены в том, что тестируем все пути выполнения нашего кода и он реагирует должным образом.

ИТОГИ ГЛАВЫ

Завершая текущую главу, в листинге 20.2 мы собрали весь скрипт тестов, чтобы вы могли использовать его в качестве справки.

Листинг 20.2. Весь тест Pester

```
$here = Split-Path -Parent $MyInvocation.MyCommand.Path
$sut = (Split-Path -Leaf $MyInvocation.MyCommand.Path) `
➥ -replace '\.Tests\.', '.'
. "$here\$sut"
Describe "Set-ComputerState" {
    Mock Restart-Computer { return 1 }
    Mock Stop-Computer { return 1 }
    It "accepts and restarts one computer name" {
        Set-ComputerState -ComputerName SERVER1 -Action Restart
        Should -Invoke Restart-Computer -Exactly 1 -Scope It
    }
    It "accepts and restarts many names" {
        $names = @('SERVER1','SERVER2','SERVER3')
        Set-ComputerState -ComputerName $names -Action Restart
        Should -Invoke Restart-Computer -Exactly 3 -Scope It
    }
    It "accepts and restarts from the pipeline" {
        $names = @('SERVER1','SERVER2','SERVER3')
        $names | Set-ComputerState -Action Restart
        Should -Invoke Restart-Computer -Exactly 3 -Scope It
    }
    It "accepts and force-restarts one computer name" {
        Set-ComputerState -ComputerName SERVER1 -Action Restart -Force
        Should -Invoke Restart-Computer -Exactly 1 -Scope It
    }
    It "accepts and shuts down one computer name" {
        Set-ComputerState -ComputerName SERVER1 -Action Shutdown
        Should -Invoke Stop-Computer -Exactly 1 -Scope It
    }
}
```

Естественно, это может быть не *полноценный* тест. Например, мы еще не добавили интеграционные тесты и не проверяли, принимаются ли для параметра `-Action` только значения `Restart` и `Shutdown`. Этот тест определенно может усложниться — и мы предлагаем вам расширить его, чтобы вы могли лучше понять, как Pester помогает автоматизировать тестирование. Ознакомиться с данной темой вы можете в справочных материалах `about_pester`.

21 Подписание скрипта

В мире профессионального создания инструментов PowerShell подписание скриптов является привычкой, которой нельзя пренебречь. И хотя некоторые разработчики могут найти этот процесс утомительным или необязательным, мы твердо верим в его важность, в связи с чем и написали эту главу. Независимо от того, планируете вы или нет, что ваши скрипты останутся внутри компании, их подписание вкупе с управлением версиями должно стать этапом рабочего процесса скриптинга.

21.1. ЗНАЧИМОСТЬ ПОДПИСАНИЯ СКРИПТОВ

Зачем вам вкладывать свое время и силы в подписание скриптов? Подписанный скрипт служит двум важным целям. Во-первых, он аутентифицирует идентичность автора скрипта. И это необязательно гарантирует безопасность или качество скрипта, но если он подписан, то вы можете хотя бы отследить, кто его создал. Во-вторых, подписание гарантирует целостность скрипта, позволяя проверить, производилось ли вмешательство в код с момента подписания. Защита кода имеет первостепенную важность, особенно когда он выходит за пределы вашей организации. Предположим, вы получаете скрипт от Салли и скачиваете его из блога. Вы уверены, что каждая строка соответствует тому, что написала Салли? Если она подписала файл, то вы можете быть уверены, что каждый содержащийся в нем символ достоверно отражает именно ее работу. И если в скрипте будет обнаружен вредоносный код, то вы сможете призвать Салли к ответу.

Внутри организации случайные изменения в скрипте вполне возможны. Неопытный интерн или ничего не подозревающий руководитель могут неумышленно изменить ваш скрипт. Если ваш код не подписан, то вы можете осознать проблему, только когда скрипт начнет выдавать результаты, далекие от оптимальных. В случае же подписанных скриптов PowerShell выступает в роли стража, мгновенно уведомляя

вас о любых проблемах и позволяя тут же их изучить. PowerShell — действительно мощный инструмент. Даже небольшой фрагмент кода может нанести серьезный ущерб. Защитите код, а значит, и себя, освоив подписание скриптов.

21.2. НЕСКОЛЬКО СЛОВ О СЕРТИФИКАТАХ

Прежде чем начать осваивать подписание скриптов, вам потребуется сертификат. Сертификаты, выступая в качестве цифровых ID-карт, необходимы для верификации идентификатора. За выдачу сертификата отвечают коммерческие центры сертификации (certificate authorities, CA), которые делают это, добавляя в сертификат идентификационные данные его держателя. Сертификаты основаны на доверии. Если вы доверяете CA, то можете доверять и выдаваемым им сертификатам.

Подписание кода — большая ответственность для CA, поскольку код способен причинить значительный вред. Для получения от CA сертификата класса 3 обычно требуется тщательный процесс верификации личности, и обычно такие сертификаты выдаются организациям, а не отдельным людям. Тем не менее организации устанавливают собственную внутреннюю инфраструктуру публичных ключей (public key infrastructure, PKI), запускают собственный CA и утверждают правила для выдачи сертификатов. Если ваш код остается внутри организации, то это становится экономически оправданной альтернативой приобретению коммерческого сертификата.

В случаях, когда вы являетесь единственным пользователем и мейнтейнером собственного кода на своем ПК, вы можете создать самозаверенный сертификат. И хотя такие сертификаты удобны для разработчиков, при развертывании кода для других людей, даже внутри одной организации, их нужно заменять реальными сертификатами.

СЕРТИФИКАТЫ, ДОВЕРИЕ И НЕОБХОДИМЫЕ УСИЛИЯ

Далеко не секрет, что управление сертификатами сопряжено со сложностями. Они утрачивают срок действия, требуют обновления, к тому же настройка внутренней PKI— довольно трудная задача. Тем не менее для нас, как для IT-профессионалов, это часть работы, и в управлении аспектом безопасности мы должны быть экспертами.

С традиционной моделью CA конкурирует такой передовой подход, как заверение, которое позволяет создавать самозаверенные сертификаты под надзором доверенных лиц. И хотя эта децентрализованная система доверия выходит за рамки текущей главы, стоит изучить ее в качестве альтернативы централизованным моделям CA.

Когда у вас есть сертификат, вы можете установить его и начать использовать, о чем мы вскоре поговорим. Данная глава посвящена PKI, поэтому мы напомним, что сертификаты состоят из *пар ключей*. В частности, у вашего будет *закрытый ключ*, который вы должны хранить в строгой безопасности, даже защищая паролем, выданным магазином сертификатов Windows, чтобы никто не мог его использовать без вашего разрешения. Этот закрытый ключ будет использоваться для генерации *подписей* скрипта. Подпись — копия вашего скрипта (или, чаще, хеша, который тоже уникален для скрипта, но занимает меньше места), которая зашифрована с помощью закрытого ключа и сопровождается информацией о сертификате (но не закрытым ключом).

Любой, кто доверяет источнику вашего сертификата, может расшифровать эту подпись, используя соответствующий *публичный ключ* из той же пары. Возможность расшифровать подпись с помощью этого ключа означает, что некто может убедиться в вашей личности, так как зашифровать ваш скрипт мог только ваш же тщательно оберегаемый закрытый ключ. Затем этот некто сможет сравнить ранее зашифрованный скрипт с его расшифрованной текстовой версией. Если эти версии совпадут, то станет ясно, что код в точности такой, каким вы его задумывали.

21.3. НАСТРОЙКА ПОЛИТИКИ ПОДПИСАНИЯ СКРИПТОВ

Для эффективного подписания скриптов вам нужно настроить свою среду так, чтобы она требовала наличия у скриптов подписи. Одного только подписания недостаточно, если PowerShell не получила указание требовать его. В сеансе PowerShell с повышенными привилегиями выполните следующую команду:

```
Set-Executionpolicy AllSigned -force
```

Параметр `-Force` исключит запрос системы о подтверждении. Лучше всего, если вы сделаете это один раз на каждой машине, на которой собираетесь запускать скрипты. Предположительно это может быть настольная система или централизованный управляющий сервер. На удаленном сервере редко приходится выполнять интерактивный скрипт, поэтому можете оставить соответствующие политики выполнения как Restricted, какими они и установлены по умолчанию.

Даже если вы используете `Invoke-Command` для выполнения локального скрипта на удаленном сервере, PowerShell выполняет *содержимое* команды удаленно. Прежде чем выполнять скрипт удаленно, желательно верифицировать его локально. Вскоре мы вам покажем, как все это сделать.

Если вы находитесь в домене Active Directory (AD), то можете использовать для настройки политик выполнения скриптов Group Policy. Напоминаем: эти политики не являются границами безопасности, а скорее играют роль крышечек над кнопками, отвечающими за пуск ядерных ракет. Мы подробно обсуждали все это в главе 7.

21.4. ОСНОВЫ ПОДПИСАНИЯ КОДА

Детальный разбор всех основ сертификатов и PKI выходит за рамки текущей главы, но мы дадим краткое пояснение. Сертификат — это криптографическое средство верификации вашей личности. Когда вы подписываете скрипт в PowerShell, указанная в сертификате информация о вашей личности добавляется в блок его подписи, подтверждая, что вы являетесь автором скрипта. Кроме того, PowerShell на основе содержимого скрипта вычисляет хеш-значение, которое встраивает в подпись. Если в файл вносятся какие-либо изменения, то подпись тоже меняется, и PowerShell обозначает этот скрипт как измененный.

21.4.1. Получение сертификата для подписания кода

Не все сертификаты можно использовать для подписания скриптов. Это должны быть сертификаты класса 3 с поддержкой расширения Microsoft Authenticode.

ПРИМЕЧАНИЕ

Класс 3 — это термин, который когда-то использовался в компании VeriSign. Сегодня он встречается редко. Большинство разработчиков просто называют такие сертификаты сертификатами для подписания кода.

CA, которому доверяет ваш компьютер, тоже должен выдать сертификат. Предположим, вы хотите распространять подписанные инструменты за пределами организации. В этом случае вам наверняка потребуется сертификат от стороннего поставщика, такого как VeriSign или DigiCert, поскольку любой человек, который скачает ваш код, будет верить, что именно они выдали вам сертификат. Но мы ожидаем, что большинство из вас имеют домен AD, в идеале с инфраструктурой сертификатов (Active Directory Certificate Services [AD CS]). С его помощью вы можете оперативно запросить сертификат в соответствии с политиками вашей организации, используя веб-интерфейс. После этого вы можете настроить Group Policy, чтобы участники домена доверяли вашему сертификату (обычно при корректно настроенной PKI эта политика уже будет установлена). Подробности темы выходят за рамки этой книги, но если вы столкнетесь с трудностями, то участники форумов PowerShell.org наверняка вам помогут.

ПРИМЕЧАНИЕ

Если говорить в общем, то первый этап состоит в поиске CA — коммерческого или внешнего. Помните, что сертификаты для подписания кода дороги, а дешевый не будет стоить тех цифровых чернил, которыми он написан. Как правило, сертификаты выдаются только компаниям, а не частным лицам, и при их получении на коммерческой основе обычно требуется достаточно длительная процедура верификации личности.

Еще один вариант тестирования, который также актуален, если вы не планируете отправлять создаваемые скрипты и инструменты PowerShell за пределы своего

ПК, — использование самозаверенного сертификата. Когда-то давно это означало освоение загадочной утилиты командной строки `makecert.exe`. Но модуль PowerShell PKI, который должен у вас появиться при установке Remote Server Administration Tools, включает в себя команду, которая все это упрощает. Если вы хотите попробовать подписание скриптов, то выполните следующее:

```
PS Cert:\> New-SelfSignedCertificate -type CodeSigningCert -Subject
"CN=James Petty" -CertStoreLocation Cert:\CurrentUser\My\ -testroot
   PSParentPath: Microsoft.PowerShell.Security\Certificate::CurrentUser\My

Thumbprint                                Subject     EnhancedKeyUsageList
----------                                -------              -----------
F33E9122D73BE220117339E9647F0037F3F875A6  CN=James Petty      Code Signing
```

Естественно, в элемент `CN=` нужно вставить свое имя. Поскольку это самозаверенный сертификат, то добавьте параметр `-TestRoot`. Вы получите сертификат, который сможете использовать, но PowerShell выдаст сообщение `unknown error`, так как не может верифицировать конвейер сертификатов. То есть ваш компьютер не *доверяет себе* как источнику сертификатов.

Мы попросили PowerShell сохранить сертификат для текущего пользователя. Это можно легко проверить с помощью параметра `-codesigningcert` в `Get-ChildItem`. Мы используем псевдоним `dir`:

```
PS C:\> dir Cert:\CurrentUser\My\ -CodeSigningCert
   PSParentPath: Microsoft.PowerShell.Security\Certificate::CurrentUser\My
Thumbprint                                Subject
----------                                -------
F33E9122D73BE220117339E9647F0037F3F875A6CN=James Petty
```

Вы можете установить несколько сертификатов для подписания кода, но использовать для подписей можно только один. Если у вас установлено их несколько, то потребуется с помощью PowerShell отфильтровывать все лишние, оставив нужный.

СОВЕТ

В мире сертификатов отпечаток (thumbprint) представляет официальное, уникальное имя сертификата. Вы будете встречать множество ссылок на него и теперь знаете, как его найти.

21.4.2. Доверие самозаверенным сертификатам

Прежде чем использовать самозаверенный сертификат, вам может потребоваться совершить несколько дополнительных действий вне PowerShell. Выполните в командной строке следующую команду, чтобы открыть модуль управления сертификатами:

```
Certmgr.msc
```

Перейдите туда, где сохранили сертификат (рис. 21.1). Вы увидите, что его выдает CertReq Test Root. При этом вы столкнетесь с проблемой: сертификат для

этого корня не является полностью доверенным. А почему он должен быть таким? Естественно, вы не можете использовать свое водительское удостоверение, которое вы же и подписали, поскольку кроме вас доверять ему никто не будет. Аналогичная ситуация с самозаверенным сертификатом. Вы можете установить этот корневой сертификат, перетащив его из контейнера Intermediate Certification Authority в Trusted Root Certification Authority (рис. 21.2).

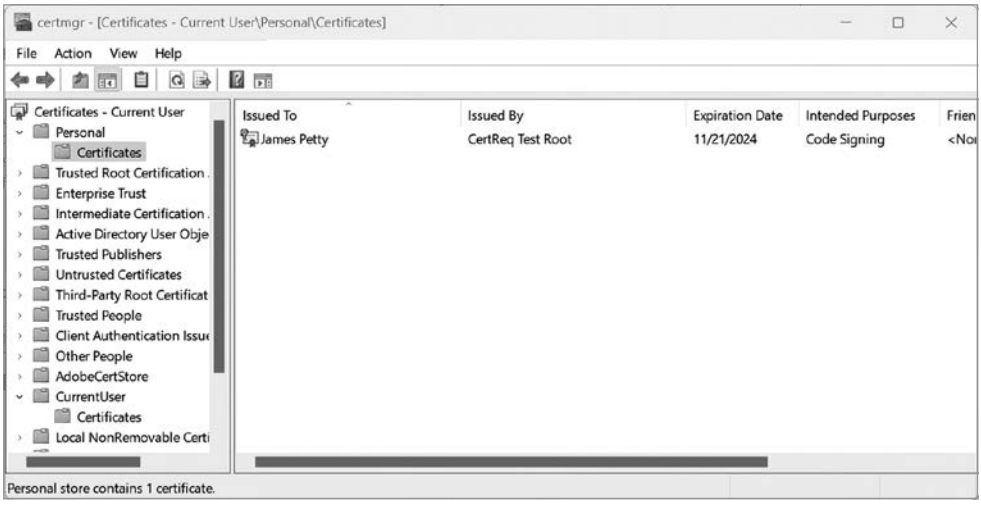


Рис. 21.1. Выбор самозаверенного сертификата

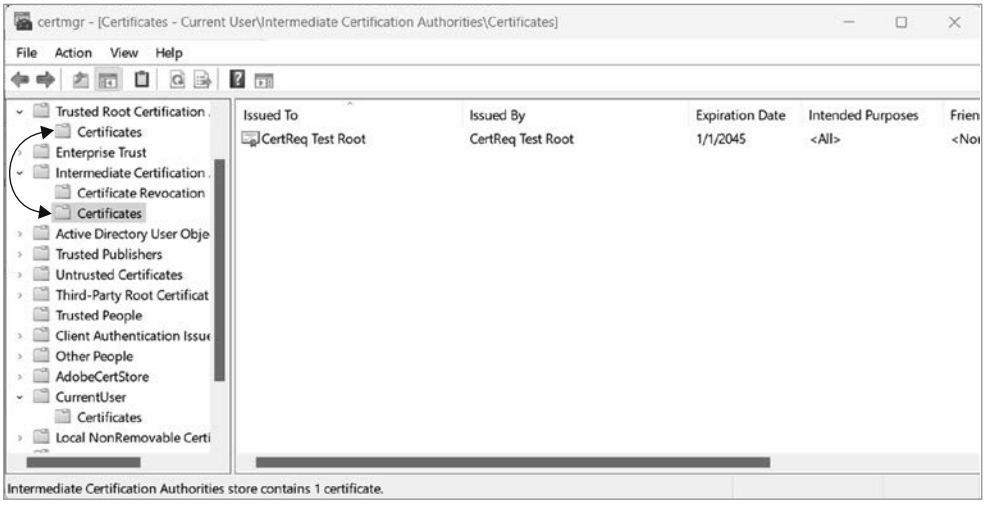


Рис. 21.2. Перемещение самозаверенного корневого сертификата

Появится диалоговое окно с предупреждением. Дайте согласие и установите этот сертификат. Теперь при использовании сертификата, созданного вашим компьютером, вы не будете получать от PowerShell сообщения об ошибке, вызванной недоверенным корнем (Root).

ПРИМЕЧАНИЕ

Эта процедура не скомпрометирует ваш компьютер. Она лишь заставит его доверять созданным им же сертификатам. При этом сертификаты, созданные где-то еще, по-прежнему не будут считаться доверенными.

21.4.3. Подписание скриптов

Чтобы подписать скрипт PowerShell, вам нужно создать на него ссылку. Сохранить сертификат для подписания кода в переменную несложно:

```
PS C:\> $cert = dir Cert:\CurrentUser\My\ -CodeSigningCert
```

Вы можете решить добавить такую строку в скрипт из профиля PowerShell, чтобы он всегда был доступен. В нашем каталоге `scripts` есть крайне простой скрипт PowerShell:

```
PS C:\scripts> get-content psvm.ps1
get-process | sort vm -desc | select -first 5
```

Мы используем командлет `Set-AuthenticodeSignature`. Вводить его непросто, поэтому здесь уместно использовать автозаполнение нажатием клавиши Tab. Но поскольку вы наверняка будете подписывать скрипты интерактивно, мы предлагаем создать в профиле PowerShell псевдоним:

```
Set-Alias -Name sign -Value Set-AuthenticodeSignature
```

Этот псевдоним мы будем использовать только для того, чтобы сокращать примеры:

```
PS C:\scripts> sign .\psvm.ps1 -Certificate $cert
    Directory: C:\scripts
SignerCertificate                         Status                    Path
-----------------                         ------                    ----
9D16AF2573AC6C01A33752CA5135F3700A6FE9CFCN Valid                   psvm.ps1
```

Вот как этот файл выглядит теперь:

```
PS C:\scripts> get-content .\psvm.ps1
get-process | sort vm -desc | select -first 5
# SIG # Начало блока подписи
# MIIFWAYJKoZIhvcNAQcCoIIFSTCCBUUCAQExCzAJBgUrDgMCGgUAMGkGCisGAQQB
# gjcCAQSgWzBZMDQGCisGAQQBgjcCAR4wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR
# AgEAAgEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQUWlS7aTI+/TUJU7Izf4mzM8b1
# HmWgggL6MIIC9jCCAd6gAwIBAgIQYcqwRS2cF6ZKK2DMJNsC6DANBgkqhkiG9w0B
```

```
# AQsFADATMREwDwYDVQQDDAhBcnQgRGVjbzAeFw0xNzA2MTkxNDQ5NDZaFw0xODA2
# MTkxNTA5NDZaMBMxETAPBgNVBAMMCEFydCBEZWNvMIIBIjANBgkqhkiG9w0BAQEF
# AAOCAQ8AMIIBCgKCAQEAotwzL7nKq3uG1oZ/uMAwSELAeVaoIqFHr+zW1hWwW+UG
# h/dftEaGsAmETjPnYRkABkGLqloiXXhmLQjY+QKtn51cue78B85mrSF5dqrfuuK6
# XIVm7rjvMGwqyU6mpCs2RA3c+eObqgQZMJeOd/U9BnawlUijTcYGXptxc7M7ewWp
# oVGSm2C385hB09pZJ5UpmonW81iZZ+nkoos1oMC2jdhdETR2JC/cfpjU1sP406Et
# s2gR5jIiZuBBzTMgAlU4IRU38gXiS8q2UA3oyysyd2/+svRgDx/SrO+HV5ZmEqiF
# epsY8DpaWn86MLYn+rjPSLgPbW6SNkwvHg58trEsIwIDAQABo0YwRDAOBgNVHQ8B
# Af8EBAMCB4AwEwYDVR0lBAwwCgYIKwYBBQUHAwMwHQYDVR0OBBYEFH1ccCLNFjh0
# ZqYdX2NvAASUku2PMA0GCSqGSIb3DQEBCwUAA4IBAQCXxfRfgI4KbsvXk0HKVI65
# fJ4CAXDJaZyx2WtuaH4HF1WjhPMh9JjupA2244p/vH1FWERZ5llwR9AcwA8kK8EM
# 6aPD5Nu0MGis7gFvzK1K/dnxmgv+7ICS9j92GM4qIa8bcfIwBTTPehQKaJS2Q+bg
# cm3eipPI4nxPPhSXLdg3FcglNfwU3aqQznHfmWj5cVgiqtMbe/CBh9hDcCFeW+y1
# X6aAY1q+ADrMjILnhOETFpIn3eHmdHiC/q0PpKGJzn+uhwLncaVnahRaSXhIbApc
# /9VqkPEg4kJFYVbewIeOjPWB+2IVtdtgag9X9HwTTP4nEIQ7KEz4jKMM9hPGacnV
# MYIByDCCAcQCAQEwJzATMREwDwYDVQQDDAhBcnQgRGVjbwIQYcqwRS2cF6ZKK2DM
# JNsC6DAJBgUrDgMCGgUAoHgwGAYKKwYBBAGCNwIBDDEKMAigAoAAoQKAADAZBgkq
# hkiG9w0BCQMxDAYKKwYBBAGCNwIBBDAcBgorBgEEAYI3AgELMQ4wDAYKKwYBBAGC
# NwIBFTAjBgkqhkiG9w0BCQQxFgQUsoYetaVPGXeBkFV4ddJTInDikFwwDQYJKoZI
# hvcNAQEBBQAEggEARmE9VVlQ+HMYTFnOQ+lJGLvOcm7RKi5+pEVFhxTwoahbu6Zb
# oZLEB6zUKx2RxLWkO1+FWiOJWGAAARPnNWCCxBKqAnedtqPNc0UVQ0J5gxuVzfO6
# J5Q+3Uu7YbrbgeErC/hYOMmu9hY8a7H7ttxD0p0qHscV7R1kOSxrUGehU3+KLKFU
# heKQlOL26DVGdk3KRayZTGzpDXHavkGAtcjcyiQPSPyRdmFcagdZ4VzrKzTT4m1w
# i+uHap5xQ80EQBxfgHZT3yXKRA1tl9Mgnmi9XNcUro25i0tiKZTjkZe0voPJ7MX1
# ePgJFLinSiRvIvzoqpOgN51CfQ/yWWdCsH+v4w==
# SIG # Конец блока подписи
```

Вам не следует вмешиваться в блок подписи, если только вы не хотите его полностью удалить. Это единственный способ снять с файла подпись.

Кроме того, можно подписать целый каталог скриптов:

```
PS C:\scripts> dir *.ps1 | sign -Certificate $cert -WhatIf
What if: Performing the operation "Set-AuthenticodeSignature" on target
"C:\scripts\DirReport.ps1".
What if: Performing the operation "Set-AuthenticodeSignature" on target
"C:\scripts\psvm.ps1".
What if: Performing the operation "Set-AuthenticodeSignature" on target
"C:\scripts\lastdayofwork.ps1".
What if: Performing the operation "Set-AuthenticodeSignature" on target
"C:\scripts\newhire.ps1".
```

Вы можете подписывать файлы `.ps1`, `.psm1` и `.ps1xml`.

СОВЕТ

Обратите внимание: вы не можете подписывать файлы .psd1, представляющие манифест для модуля скрипта. Если вы разрешите выполнение в системе неподписанных скриптов, то теоретически какой-нибудь вирус может найти файл .psd1 и изменить его так, чтобы загружался вредоносный скрипт при загрузке вроде бы подписанного модуля. Это риск, но, честно говоря, тот же вирус может атаковать вас и десятками других способов. Помните о такой возможности и сохраняйте бдительность.

21.4.4. Тестирование подписей скриптов

Для тестирования подписи скрипта используйте `Get-AuthenticodeSignature`:

```
PS C:\scripts> Get-AuthenticodeSignature .\psvm.ps1
Directory: C:\scripts
SignerCertificate                         Status                Path
-----------------                         ------                ----
9D16AF2573AC6C01A33752CA5135F3700A6FE9CF   Valid             psvm.ps1
```

Выводом `Get-AuthenticodeSignature` будет другой вид объекта. Свойства этого объекта говорят сами за себя:

```
PS C:\scripts> Get-AuthenticodeSignature .\psvm.ps1 | select *
SignerCertificate      : [Subject]
                           CN=James Petty
                         [Issuer]
                           CN=CertReq Test Root, OU=For Test Purposes Only
                         [Serial Number]
                           5B0A36A612E5A78F400FEE5F02F930BB
                         [Not Before]
                           6/19/2017 10:07:53 AM
                         [Not After]
                           6/19/2018 10:27:53 AM
                         [Thumbprint]
                           9D16AF2573AC6C01A33752CA5135F3700A6FE9CF
TimeStamperCertificate :
Status                 : Valid
StatusMessage          : Signature verified
Path                   : C:\scripts\psvm.ps1
SignatureType          : Authenticode
IsOSBinary             : False
```

Если вы не последовали нашей рекомендации установить самозаверенный корневой сертификат, то увидите статус `unknown error`. Это нормально, но скрипт вы выполнить не сможете.

Если у вас установлена политика выполнения `AllSigned`, то запустить скрипт все же получится:

```
PS C:\scripts> set-executionpolicy allsigned -force
PS C:\scripts> .\psvm.ps1
Do you want to run software from this untrusted publisher?
File C:\scripts\psvm.ps1 is published by CN=James Petty and is not trusted on your
system. Only run scripts from trusted publishers.
[V] Never run  [D] Do not run  [R] Run once  [A] Always run  [?] Help
(default is "D"): a
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI  ProcessName
-------  ------   -----      -----     ------      --  --  -----------
   1179      80   73368      472         9.31    376   2  SearchUI
    873      44   67712      43888     579.25    472   0  svchost
   3395     194   97616      27868     446.81   1116   0  svchost
    948      29   56096      22904       2.19   4920   2  powershell
    876      37   99696      47176       4.33   6080   0  powershell
```

При первом выполнении скрипта PowerShell поинтересуется, можно ли ему доверять. Мы скажем, что ему можно доверять всегда, и далее сможем запускать этот скрипт уже без всплывающих вопросов.

Теперь мы внесем в скрипт небольшое изменение, но повторно подписывать не станем и попробуем выполнить:

```
PS C:\scripts> .\psvm.ps1
File C:\scripts\psvm.ps1 cannot be loaded. The contents of file C:\scripts\psvm.ps1
might have been changed by an unauthorized user or process, because the hash of the
file does not match the hash stored in the digital signature.
The script cannot run on the specified system. For more information, run Get-Help
about_Signing..
    + CategoryInfo          : SecurityError: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
```

Мы получаем довольно серьезную ошибку, и скрипт не выполняется. Этого мы и хотели! Если бы мы не вносили изменения, то решили бы выяснить, что произошло. Завершив устранение неполадок, мы можем повторно подписать скрипт, и он заработает.

ИТОГИ ГЛАВЫ

Реализовать подписание скриптов не слишком сложно, особенно если у вас есть AD PKI (которая дешевле и проще коммерческого CA) или другой бренд PKI. Пожалуй, вы даже можете настроить свой редактор скриптов на их автоматическое подписание — многие программы предлагают такую возможность при сохранении файла. Помимо всего прочего, это позволяет подписывать все скрипты разом. Как мы уже говорили, реализация цифровых подписей или требование их использования не относятся к границам безопасности. Тем не менее этот прием позволяет добавить важную проверку безопасности, чтобы убедиться в том, что скрипт, который вы или кто-то другой собираетесь выполнить, написали *именно* вы.

22 Публикация скрипта

Надеемся, что по ходу чтения книги или вскоре после этого вы разработаете отличный инструмент PowerShell, который будет решать актуальную задачу. Будет еще приятнее, если это приведет к ощутимому повышению вашей зарплаты. Тем не менее мы рассчитываем, что, помимо личных выгод, вы подумаете и о том, чтобы поделиться своим творением с сообществом PowerShell. В последние годы это весьма просто реализовать с помощью PowerShell Gallery, где размещаются тысячи модулей и скриптов.

22.1. ВАЖНОСТЬ ПУБЛИКАЦИИ

Публикация скрипта дает несколько преимуществ. Начнем с того, что внесение положительного вклада в сообщество PowerShell — акт щедрости. Мы заранее выражаем свою благодарность за ваше желание пойти на такой шаг. Более того, публикация позволяет обмениваться инструментами с коллегами. Вы можете публиковать текущую версию в PowerShell Gallery (часто называемой PSGallery) и устанавливать или обновлять ее по мере необходимости. Если у вас есть новая версия, то вы легко можете опубликовать ее. При этом более старые версии останутся доступны, и вы сможете тестировать их или ссылаться на них при необходимости.

22.2. ЗНАКОМСТВО С POWERSHELL GALLERY

PowerShell Gallery — бесплатный сайт, поддерживаемый Microsoft и доступный по ссылке www.powershellgallery.com. Вы можете пользоваться им напрямую, но для большинства взаимодействий наверняка будете использовать набор командлетов PowerShell, таких как `Find-Module` и `Install-Module`. В Microsoft реализовали строгие проверки в процессе загрузки скриптов, чтобы пропускать только те, которые

соответствуют лучшим практикам. Кроме того, компания с помощью команд PowerShell Script Analyzer тщательно анализирует ваш код. Если он не пройдет конкретные тесты, то вы сразу можете получить отказ. Чтобы избежать лишних проблем, вы можете использовать для разработки редактор Visual Studio Code, так как он поможет пройти эти тесты. Важно помнить: Microsoft не может гарантировать эффективность модуля, поэтому все скачиваемые материалы вы запускаете на свой страх и риск. Именно поэтому важно организовать надежную среду для тестирования.

22.3. ДРУГИЕ ВАРИАНТЫ ПУБЛИКАЦИИ

Мы подчеркиваем доступность PowerShell Gallery, но обратите внимание вот на что: по сути, это сайт специализированного вида, работающий как репозиторий на базе NuGet. Подобные репозитории уже давно считаются надежными средствами для публикации и распространения материалов. Такой репозиторий может организовать кто угодно, в том числе ваша организация. И хотя мы не станем углубляться в тонкости настройки и управления этими серверами, процесс их публикации из PowerShell должен совпадать с тем, который мы описали для PowerShell Gallery.

22.4. ПЕРЕД ПУБЛИКАЦИЕЙ

Прежде чем публиковать что-либо, вы должны убедиться, что ваш проект завершен, протестирован и должным образом задокументирован. Это значит, что он содержит как минимум справочную информацию в виде комментариев. Ваш проект говорит о вас, поэтому с его помощью нужно произвести наилучшее впечатление. Но есть и другие предварительные формальности, с которыми нужно разобраться.

22.4.1. Не изобретаете ли вы колесо?

Нет правила, запрещающего публикацию чего-то, что уже существует, однако желательно перепроверить этот факт. Нет ли модуля, который предлагает ту же функциональность, что и ваш? Чем отличается ваш? Используйте `Find-Module`, чтобы понять, какой из существующих модулей может соперничать с вашим.

Предположим, у вас есть модуль с командами Active Directory (AD).

Вы можете выполнить:

```
find-module *activedirectory* | Select Version,Name,Author, `
 Description,PublishedDate
```

или искать по тегам:

```
find-module -tag ad,activedirectory
```

Вы можете использовать и свой браузер, зайдя на www.powershellgallery.com и выполнив поиск там (рис. 22.1). Кроме того, вы даже можете уточнить при поиске конкретные типы и категории.

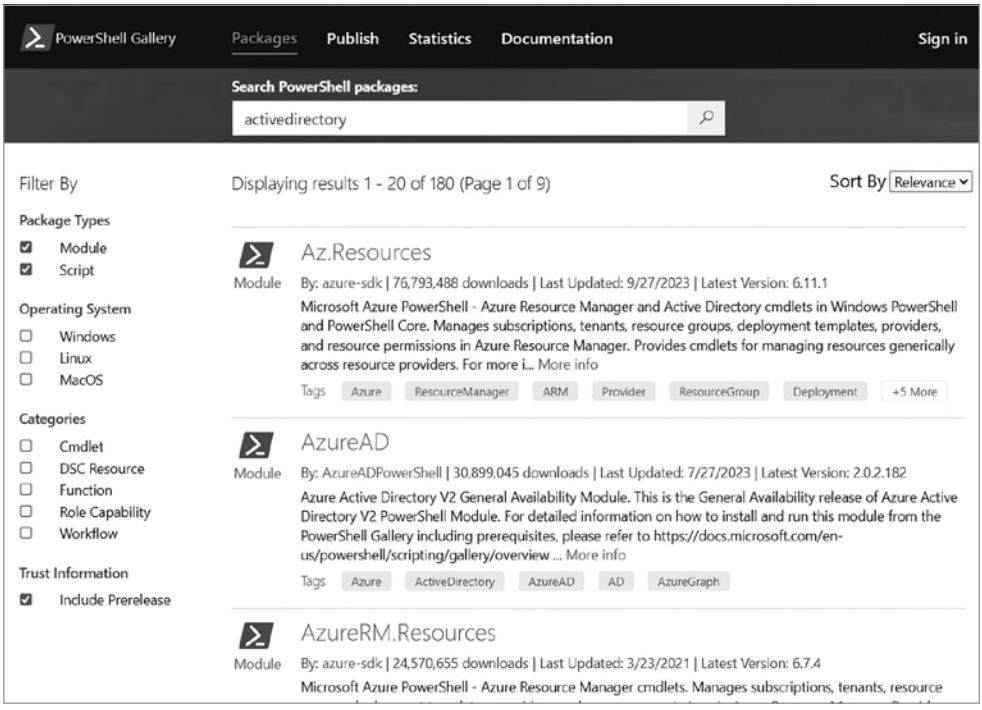


Рис. 22.1. Поиск Active Directory в PowerShell Gallery

Можно искать модули, характерные для операционной системы или являющиеся кросс-платформенными. Это легко определить с помощью тегов. Рассмотрим в качестве примера модуль DBATools (рис. 22.2). Если развернуть раздел Package Details, то можно увидеть теги для Mac и Linux. DBATools можно запускать из PowerShell, установленной на любой из этих операционных систем.

Рис. 22.2. DBATools с тегами Mac и Linux

22.4.2. Обновление файла манифеста

Вам потребуется подходящий манифест, сгенерированный `New-ModuleManifest`. В нем нужно настроить следующие разделы.

- `ModuleVersion` — общепринятым стандартом считается *семантическое версионирование*. Технически значение версии должно указываться в формате *a.b.c.*, например 1.0.0. Но можно обойтись и формой 1.0.
- `Author` — здесь наверняка будет ваше имя. Старайтесь использовать во всех своих проектах одно значение. Так разработчики смогут определить, что принадлежит вам. Не используйте Don Jones в одном и Donald Jones в другом. Выберите одно и придерживайтесь его. Единственный вариант изменить имя — это опубликовать новую версию.
- `Description` — это масштабный раздел. Здесь вам нужно предоставить всю информацию о том, почему ваш модуль существует, какие задачи решает и чем отличается от связанных проектов.
- `PrivateData` — еще один крупный раздел, поскольку Microsoft будет брать значения из манифеста, чтобы заполнять метаданные вашего проекта в галерее.
  - `Tags` — вам следует ввести не менее одного тега. При этом можно ввести столько, сколько нужно, разделив их запятыми. Посмотрите на существующие модули, чтобы понять, какие теги используют разработчики.
  - `LicenseUri` — в идеале ваш проект должен быть публично доступен в системе управления версиями наподобие GitHub. Вставьте ее адрес в свой файл лицензии, который у вас, естественно, есть.
  - `ProjectUri` — это может быть URL вашего репозитория GitHub или любого онлайн-ресурса, в котором хранится проект. Некоторые разработчики любят, когда у них есть возможность проверить ваш исходный код или, в случае GitHub, использовать возможность создания «заявки» (issue), чтобы сообщить о багах или задать вопросы.

Мы предполагаем, что вы уже установили ожидаемые значения для таких элементов, как `RootModule`, `Guid` и `FunctionsToExport`.

22.4.3. Получение ключа API

Для публикации в PowerShell Gallery вам нужно быть зарегистрированным пользователем с ключом API. На сайте этого ресурса (www.PowerShellGallery.com) щелкните по ссылке Register и следуйте инструкциям. (Оформление сайтов время от времени меняется, поэтому мы не станем приводить скриншоты.) На определенной стадии процесса вы получите ключ API. Найти его вы можете так: авторизуйтесь и щелкните на своем имени, чтобы просмотреть свой профиль. Вы должны увидеть раздел Credentials. Щелкните на ссылке Show Key, чтобы открыть все данные. Если

вы работаете за защищенным компьютером, то можете подумать о добавлении ключа API в менеджер паролей или хранилище и использовать для его извлечения PowerShell Secrets Module. Кроме того, вы можете скопировать его значение и создать в профиле PowerShell переменную:

```
$PSGalleryKey = 2XXXX7bd-771d-9999-8XXa-da41XXXX1abc
```

Но мы все же предпочитаем первый вариант. Он будет удобен, когда придет время публиковать ваш модуль.

22.5. ПРОЦЕСС ПУБЛИКАЦИИ

Когда вы публикуете модуль в PowerShell Gallery, командлет `Publish-Module` создает из каталога вашего модуля пакет NuGet. Человек, который установит модуль, по сути, получит копию вашего каталога. Помните, как ваша мама просила вас убраться в комнате, поскольку должны были прийти гости? Здесь нечто похожее. На удаление временных или ненужных файлов из каталога требуется всего несколько минут. Если вы используете Git, то скрытый каталог `.git` будет проигнорирован. Если вам нужно сохранить для разработки файлы, которые не являются частью финального проекта, то можете создать отдельный пустой каталог исключительно под файлы модулей.

Если у вас есть грамотно заполненный манифест, вы сможете выполнить такую команду:

```
Publish-Module -path c:\scripts\MyTools -repository PSGallery
➥ -nugetapikey $psgallerykey
```

В этом примере мы используем сохраненный ключ API, который установили ранее. Если вы не заполните манифест в соответствии с нашими рекомендациями, то должны выполнить команду и указать дополнительные параметры, такие как `-Tags` и `-FormatVersion`.

К сожалению, вы не можете опубликовать модуль, не отправив его в PowerShell Gallery. Мы бы хотели иметь возможность публиковать или сохранять пакеты локально, чтобы проверять их содержимое перед тем, как делиться ими с другими людьми. Лучшее, что здесь можно сделать, — это сохранить модуль из PowerShell Gallery и просмотреть скачанные файлы. Если вам что-то не понравится, то обновите модуль, увеличьте номер его версии и опубликуйте заново.

22.5.1. Управление ревизиями

В какой-то момент вы можете решить доработать модуль или исправить баги. После этого вы можете повторно его опубликовать в PowerShell Gallery, действуя по той же схеме. Самое главное — не забыть обновить номер версии в манифесте модуля. Пользователи могут получать последние версии, выполняя `Update-Module`.

После публикации модуля уже нет возможности управлять им из PowerShell — потребуется использовать веб-страницу PowerShell Gallery. Для этого выполните следующие действия.

1. Авторизуйтесь в аккаунте.
2. Щелкните на ссылке в его названии.
3. Нажмите Manage My Items.
4. Выберите модуль из списка.
5. Промотайте вниз страницы до раздела Version History. Вы не можете ничего удалить, но можете скрыть версию. Щелкните на ссылке под Listed. Все указанные здесь версии наверняка обозначены как Yes.
6. Снимите флажок на следующей странице, чтобы отключить выдачу этой конкретной версии в результатах поиска.
7. Щелкните Save.

Теперь при использовании `Find-Module` эту версию никто не увидит.

Кроме того, на странице модуля вы увидите ссылку Edit Module. Перейдя по ней, вы можете изменить некоторые элементы, такие как описание модуля и его сводные данные, которые отражаются в Additional Metadata при использовании `Find-Module`. Эти же элементы вы можете настроить в манифесте модуля — вносить подобные изменения лучше в нем.

22.6. ПУБЛИКАЦИЯ СКРИПТОВ

Ваш модуль должен иметь все необходимые вам функции и инструменты. То, как вы их используете, будет определяться скриптом-контроллером. Он автоматизирует процесс, поэтому, вместо того чтобы вводить конкретную последовательность действий с помощью команд из опубликованного модуля, вам достаточно выполнить скрипт. Вы можете решить поделиться своими контроллерами. В Microsoft недавно запустили онлайн-репозиторий для скриптов, который может стать подходящим вариантом.

22.6.1. Использование репозитория скриптов Microsoft

Находить скрипты можно с помощью командлета `Find-Script`. Вы можете выполнить его без параметров или же искать что-то в имени скрипта:

```
PS C:\> find-script *weather*
Version    Name                Repository     Description
-------    ----                ----------     -----------
1.0        Get-Weather         PSGallery      Shows Weather information....
```

Для нашего примера вы можете увидеть одно или два сообщения об ошибке. Это связано с проблемами скриптов в репозитории, а не с тем, что вы сделали что-то не так локально или при выполнении `Find-Script`.

Если вы видите нужный скрипт, то можете сохранить его в каталог, чтобы просмотреть:

```
PS C:\> save-script get-weather -path c:\dltemp
```

Теперь можете открыть файл — в нашем случае это `c:\dltemp\get-weather.ps1` — и решить, что с ним делать. Если он вас устраивает, то скопируйте его в каталог скриптов или же используйте предоставляемую PowerShell возможность установки:

```
PS C:\> install-script get-weather
PATH Environment Variable Change
Your system has not been configured with a default script installation path
yet, which means you can only run a script by specifying the full path to
the script file. This action places the script into the folder 'C:\Program
Files\PowerShell\Scripts', and adds that folder to your PATH
environment variable. Do you want to add the script installation path
'C:\Program Files\PowerShell\Scripts' to the PATH environment
variable?
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"): y
```

Как вы поймете из запросов системы, скрипты устанавливаются в конкретный каталог, который добавляется в путь:

```
PS C:\> get-command get-weather
CommandType     Name            Version  Source
-----------     ----             -------  ------
ExternalScript Get-Weather.ps1           C:\Program Files\WindowsPowerShell\...
PS C:\> get-command get-weather  |   Select path
Path
----
C:\Program Files\WindowsPowerShell\Scripts\Get-Weather.ps1
```

Очень удобно, что вам не нужно указывать весь путь к файлу скрипта. Достаточно ввести его имя, и он выполнится:

```
PS C:\> get-weather "Austin"
Weather report: Austin
     \   /     Sunny
      .-.      100-102 °F
   ? (   ) ?   ? 0 mph
      `-'      9 mi
     /   \     0.0 in
```

Как и в случае модулей, скрипты можно обновлять и удалять. Более того, вы можете публиковать собственные скрипты. Весь мир увидит ваш код, так что проследите, чтобы он был чистым, хорошо документированным и содержал все, о чем мы говорили, когда перечисляли признаки профессионального создания инструментов.

22.6.2. Создание ScriptFileInfo

Прежде чем публиковать скрипт, вам нужно создать особый вид заголовка, содержащий все необходимые метаданные, такие как теги, версии и требования. Это делается с помощью командлета `New-ScriptFileInfo`. Вы можете либо внести код скрипта в этот файл, либо переместить блок комментариев в свой файл скриптов. Мы продемонстрируем публикацию одного из скриптов, который проверяет обновления модуля в PowerShell Gallery (автор скрипта — Джефф Хикс):

```
PS C:\> New-ScriptFileInfo -Path C:\Work\PSReleaseTools.ps1 -Version 1.0.0
-Author 'Jeff Hicks' -Description 'Check for module updates from the
PowerShell gallery and create a comparison object' -Copyright 2017 -Tags
PowerShellget,Module,PSGallery
```

Файл должен иметь расширение `.ps1`. Ниже вы видите результат — заголовки должны быть описательными и соответствовать тому, что прописано в манифесте модуля:

```
<#PSScriptInfo
.VERSION 1.0.0
.GUID 7da2acc6-30d8-4cc9-a3d9-ba645fceebb2
.AUTHOR Jeff Hicks
.COMPANYNAME
.COPYRIGHT 2023
.TAGS PowerShellget Module PSGallery
.LICENSEURI
.PROJECTURI
.ICONURI
.EXTERNALMODULEDEPENDENCIES
.REQUIREDSCRIPTS
.EXTERNALSCRIPTDEPENDENCIES
.RELEASENOTES
#>
<#
.DESCRIPTION
 Проверка обновлений модуля из PowerShell Gallery и создание объекта для сравнения.
#>
Param()
```

Возьмите все, кроме строки `Param()`, и переместите в начало файла скрипта. Мы его немного почистим и убедимся, что ничего не перепутали:

```
PS C:\> Test-ScriptFileInfo -Path C:\scripts\Check-ModuleUpdate.ps1 |
 select *
Name                       : Check-ModuleUpdate
Version                    : 1.0.0
Guid                       : 7da2acc6-30d8-4cc9-a3d9-ba645fceebb2
Path                       : C:\scripts\Check-ModuleUpdate.ps1
ScriptBase                 : C:\scripts
Description                : Check for module updates from the PowerShell
                             Gallery and create a comparison object
Author                     : Jeff Hicks
```

```
CompanyName                :
Copyright                  : 2023
Tags                       : {PowerShellget, Module, PSGallery}
ReleaseNotes               : {This code is described at
http://jdhitsolutions.com/blog/powershell/5…}
RequiredModules            :
ExternalModuleDependencies :
RequiredScripts            :
ExternalScriptDependencies :
LicenseUri                 :
ProjectUri                 : https://gist.github.com/jdhitsolutions/8a49...
IconUri                    :
DefinedCommands            :
DefinedFunctions           :
DefinedWorkflows           :
```

Мы не получили никаких ошибок, поэтому предположим, что все хорошо. Как только вы создадите что-то подобное, все это можно хранить в виде фрагмента или файла, который вы можете копировать, вставлять и изменять по мере необходимости. Главное, не забывайте генерировать новый GUID, используя командлет `New-Guid`, для каждого скрипта, который собираетесь публиковать.

22.6.3. Публикация скрипта

Публикация скрипта в PowerShell Gallery тоже требует ключа API. Обновив файл скрипта необходимыми метаданными, вы можете легко опубликовать его:

```
Publish-Script -Path C:\scripts\Check-ModuleUpdate.ps1 -NuGetApiKey
 $psgallerykey -Repository PSGallery
```

В течение минуты скрипт станет доступен для скачивания и установки:

```
PS C:\> find-script check-moduleupdate
WARNING: Unable to resolve package source ''.
WARNING: Cannot bind argument to parameter 'Path' because it is an empty
string.
Version  Name                Repository  Description
-------  ----                ----------  -----------
1.0.0    Check-ModuleUpdate  PSGallery   Check for module updates from ...
```

Вы можете увидеть другую версию, так как Джефф наверняка будет вносить изменения и публиковать скрипт повторно.

22.6.4. Управление опубликованными скриптами

Как и в случае с опубликованными модулями, в PowerShell нет команд для управления опубликованными скриптами. Если вам нужно изменить скрипт, то сделайте это, после чего отредактируйте информационный заголовок его файла, указав в нем новую версию. У вас должна быть возможность, как и ранее, выполнить `Publish-Script`.

Кроме того, для модулей вы можете использовать страницу Manage My Items на сайте PowerShell Gallery, как мы показывали ранее. Промотайте список вниз до нужного скрипта. Вы увидите, что его тип — `Script`. Как и в случае модулей, вы не можете удалять опубликованные элементы, но можете скрывать предыдущие версии. Для этого вам нужно выполнить действия, описанные выше.

ИТОГИ ГЛАВЫ

Нет обязательного требования, чтобы вы публиковали свои модули и скрипты или делились ими. Но это относительно безболезненный процесс, позволяющий сделать ваш прекрасный код доступным для всех, кому он нужен. Мы считаем, что в долгосрочной перспективе Microsoft предоставит дополнительные руководства и инструменты для IT-профессионалов, позволяющие настраивать внутренние репозитории, что вполне разумно. Ну а пока вы можете ознакомиться с процессом, публикуя свои проекты в PowerShell Gallery.

Часть IV

Мы добрались до заключительной части книги. Здесь вы углубитесь в тонкости написания скриптов, расширив границы ваших навыков и изучив передовые техники, которые определяют мастерство в области скриптинга.

Глава 23 посвящена неизбежным сложностям, связанным с багами. В ней вы освоите сложные приемы, позволяющие выявлять, диагностировать и устранять баги, и улучшите свои навыки создания функциональных и устойчивых скриптов. Прочитав главу 24, вы научитесь визуализировать вывод ваших скриптов и изучите продвинутые методы, благодаря которым сможете повысить привлекательность и юзабилити их результатов, помогая пользователям получать четкую и убедительную информацию. В главе 25 описывается .NET Framework. Вы изучите сложные приемы скриптинга, в которых используется функциональность этой платформы, благодаря чему ваши скрипты получат новые возможности. Затем в главе 26 мы углубимся в способы работы с хранилищами данных и изучим альтернативы Microsoft Excel. Вы познакомитесь с расширенными методами хранения и управления данными, используя которые сможете создать масштабируемые решения для своих проектов. Наконец, в главе 27 мы поразмышляем на тему безграничных возможностей скриптинга, динамической природы и постоянной эволюции данной сферы, а также поговорим о том, должно ли стремление к мастерству когда-нибудь прекращаться. Итак, начинается последний этап нашего путешествия.

23 Устранение багов

Ни одно подробное руководство по скриптингу не будет полноценным без разбора критической темы отладки. Скажем честно, отладка может быть весьма неприятным процессом. Но не волнуйтесь: мы поделимся ценной информацией и практическими советами, с помощью которых вы сможете упростить эту работу.

23.1. ТРИ ВИДА БАГОВ

В сфере скриптинга баги обычно делятся на три категории: синтаксические, логические и баги в результатах. Последние — относительно новая категория. Она была введена для обработки определенных сценариев, которые озадачивали авторов скриптов. Рассмотрим эти три категории багов в порядке возрастания сложности.

- *Синтаксические баги* возникают, когда вы что-то неверно ввели. Возможно, вместо `Foreach Object` вы написали `ForEach`. Или забыли поставить закрывающую фигурную скобку `}`. PowerShell попытается предупредить вас о множестве синтаксических багов, используя красное волнистое подчеркивание. Но есть и более серьезные синтаксические баги, выявить которые PowerShell уже не поможет. Так, неверный ввод имени переменной в скрипте — например, `$CompuerName` вместо `$ComputerName` — приведет к нежелательным результатам. Если вы работаете в Visual Studio Code, то можете увидеть, что переменная подчеркнута красным цветом, пока не используете ее где-то еще в скрипте.
- *Баги в результатах* возникают, когда выполнение команды приводит к неожиданному выводу. Например, если вы ожидаете, что `Test-Connection SERVER1` вернет `$True`, когда SERVER1 находится онлайн, то будете расстроены, если этого не произойдет и соответствующий код будет работать не так, как предполагалось.

- *Логические баги* — самые сложные, поскольку не ведут к очевидным ошибкам. Такие баги возникают, когда команды вашего скрипта выполняются без ошибок, но есть проблема со структурой кода или тем, как он написан. Основную часть этой главы мы посвятим способам исправления именно этих багов.

ПОЧТИ ЧЕТВЕРТАЯ КАТЕГОРИЯ: «ПОДВОХ» POWERSHELL

Есть уникальный случай, который можно рассмотреть как смесь синтаксического бага и бага в результатах. Взгляните на следующую команду:

```
Get-CimInstance -ClassName Win32_OperatingSystem |
Select-Object -Prop PSHostName,Version,BuildNumber
```

Она выполняется без ошибок, но создает один пустой столбец. Внимание привлекает свойство `PSHostName`, запрашиваемое с помощью `Select-Object`. Извлеченный нами класс общей информационной модели (CIM) не содержит свойства `PSHostName`. PowerShell же может создавать новые свойства динамически, что бывает полезно во многих ситуациях. В данном случае оболочка создала новое свойство `PSHostName`, не имеющее какого-либо содержимого. Вы не получите ожидаемых результатов, если впоследствии будете считать, что `PSHostName` содержит какие-то значения. Пока классифицируем эту ситуацию как баг в результатах и рекомендуем прочесть раздел 23.3.

23.2. ОБРАБОТКА СИНТАКСИЧЕСКИХ БАГОВ

Самый простой способ обработки синтаксических багов выглядит так: избегать опечаток и обращать внимание на красные волнистые подчеркивания в PowerShell, указывающие на возможные проблемы. Кроме того, вы можете повысить надежность скриптов, добавляя в их начало команду `Set-StrictMode-Version 2.0`. Она изменяет поведение оболочки следующим образом.

- Предполагается, что вы будете вызывать функции PowerShell, используя конкретный синтаксис. Например, функцию с тремя входными параметрами можно вызвать, выполнив `My-Function 1 2 3` и передав в качестве этих параметров значения `1`, `2` и `3` по порядку. Новички иногда используют синтаксис в стиле методов наподобие `My-Function(1,2,3)`, когда в первый параметр передается один массив из трех элементов. Строгий режим это запрещает и в таких случаях выводит ошибку. Вы сможете избежать данной проблемы, всегда используя при вызове функции именованные параметры, как в `My-Function -Param 1 -OtherParam 2 -ThirdParam 3`.
- Обращение к несуществующим свойствам объектов обычно ведет к возвращению значения `$null`; в строгом режиме такое действие вызовет ошибку.

Это *не* решит проблему с `Select-Object`, о которой мы упомянули во врезке выше, — то состояние, как мы отметили, является особым *свойством* команды.

- Обращение к переменной, которой не было присвоено значение в текущей области, обычно заставляет PowerShell подняться по дереву области в попытке найти эту переменную. Например, обращение к `$ErrorActionPreference` в скрипте сработает, поскольку глобальная область, в отличие от локальной, содержит эту предопределенную переменную. В строгом режиме это поведение меняется. Обращение к переменным, которым в текущей области еще не было присвоено значение, вызовет ошибку. Это помогает избежать синтаксических багов, когда допускается ошибка при вводе имени переменной.

Мы рекомендуем использовать во всех скриптах строгий режим. Да, мы во всех своих примерах кода так не делаем, но лишь потому, что эти файлы не предназначены для производственной среды.

ИСПОЛЬЗОВАНИЕ ПОСЛЕДНЕЙ ВЕРСИИ

Вы могли заметить, что параметр `-Version` для `Set-StrictMode` предлагает опцию `"Latest"`. И хотя ее использование кажется удобным, важно учесть возможные последствия. На данный момент 2.0 — последняя задокументированная версия, поэтому более безопасно выбирать ее. Вы можете быть уверены, что в случае ее использования ваш код будет работать ожидаемым образом. Если Microsoft выпустит версию 3.0, то вам наверняка потребуется пересмотреть свой код, поскольку в PowerShell могут быть внесены значительные изменения.

23.3. ОБРАБОТКА БАГОВ В РЕЗУЛЬТАТАХ

Чтобы не допускать возникновения этой категории багов, следуйте процессу скриптинга, который мы описывали в книге. Всегда начинайте с выполнения отдельных команд непосредственно в консоли, прежде чем встраивать их в скрипты. Таким образом вы сможете наблюдать вывод. И хотя это может показаться простым и очевидным, авторы скриптов зачастую пренебрегают этим моментом, стремясь создать код как можно скорее.

23.4. ОБРАБОТКА ЛОГИЧЕСКИХ БАГОВ

А теперь займемся самой непростой категорией. Мы выработали простое правило, которое упрощает процесс выявления и исправления логических багов: логические ошибки возникают, когда в свойстве или переменной есть то, что отличается от предполагаемого содержимого. Чтобы увидеть этот принцип, рассмотрим скрипт

в листинге 23.1. Он содержит функцию, которая должна получать информацию о диске, используя команду `Get-CimInstance`.

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Возьмите этот скрипт из доступных для скачивания примеров кода (http://mng.bz/rjgE) и выполните его. Это ничему не навредит, но и код корректно не заработает. Если вы в какой-то момент столкнулись с данной проблемой, то причина будет очевидна, но мы используем ее в качестве примера процедуры, которую проделываем для отладки подобных проблем. Кроме того, мы не добавляем к коду примечания, поскольку хотим, чтобы вы повторили процесс отладки.

Листинг 23.1. Скрипт с багом

```
function Get-DiskInfo {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True)]
        [string[]]$ComputerName
    )
    BEGIN {
        Set-StrictMode -Version 2.0
    }
    PROCESS {
        ForEach ($comp in $ComputerName) {
            $params = @{'ComputerName' = $comp
                        'ClassName' = 'Win32_LogicalDisk'}
            $disks = Get-CimInstance @params
            ForEach ($disk in $disks) {
                $props = @{'ComputerName' = $comp
                           'Size' = $disk.size
                           'DriveType' = $disk.drivetype}
                if ($disk.drivetype -eq 'fixed') {
                    $props.Add('FreeSpace',$disk.FreeSpace)
                } else {
                    $props.Add('FreeSpace','N/A')
                }
                New-Object -TypeName PSObject -Property $props
            } #foreach disk
        } #foreach computer
    } #PROCESS
    END {}
}
Get-DiskInfo -ComputerName localhost
```

Проблема с этим скриптом заключается в том, что, как и в случае логических багов, у нас есть либо переменная, либо свойство, которое содержит *не то, что мы думали*. Конкретно в этом примере, который мы намеренно сделали простым, это баг в результатах. Мы бы не оказались в такой ситуации, если бы выполнили команду в консоли и посмотрели на ее вывод. Но в некоторых скриптах вы заполняете

переменные и свойства значениями, которые вычислили или создали, так что это сложнее, чем просто выполнить команду ради просмотра ее результатов. В данном примере мы будем рассматривать этот баг как чисто логический и следовать процедуре его выявления.

Предположим, основной проблемой является свойство или переменная, не содержащая то, что вы ожидаете. В этом случае исправление подразумевает, что мы определим, какое свойство или переменную содержит код, и выясним, что именно содержит это свойство или переменная. Для этого мы разберем несколько отдельных методов.

ПРИМЕЧАНИЕ

После появления VS Code и встроенной в него поддержки PowerShell мы изменили свой подход к отладке. Мы больше не используем Write-Debug и в большинстве случаев интерактивной отладки, как здесь, не так часто задействуем Set-PSBreakpoint. Все эти инструменты по-прежнему полезны, и в более сложных книгах наподобие *The PowerShell Scripting & Toolmaking Book* (https://leanpub.com/powershell-scripting-toolmaking) мы подробно разбираем нюансы их использования. Тем не менее сейчас для начала отладки мы используем возможности VS Code.

23.4.1. Установка точек останова

Точка останова позволяет вам выполнить скрипт до конкретного места. Встретив такую точку, скрипт приостановится. Благодаря этой паузе вы можете изучить скрипт, проверить содержимое переменных и свойств, выполнить скрипт построчно или вернуться к штатному выполнению. Точки останова — это инструмент отладки вашего кода, и он невероятно полезен.

Мы любим устанавливать такие точки сразу же после определения содержимого переменной или свойства либо перед тем как использовать это содержимое. На рис. 23.1 показан наш скрипт в VS Code. Мы перешли к строке 17 и нажали клавишу F9, чтобы активировать точку останова. Она отображается в виде красной точки слева от номера строки.

Установив точку останова, можно нажать клавишу F5, чтобы выполнить скрипт и начать отладку. На рис. 23.2 показано, что происходит, когда выполнение достигает строки 17: в левой части окна VS Code открывается панель Debug, и в окне терминала PowerShell можно увидеть, что мы достигли точки останова. Скрипт приостановлен, и выделена строка 17.

И пока точка останова активна, мы можем использовать панель Terminal для анализа происходящего. Например, мы выполняем `$disk`, чтобы увидеть текущее содержимое переменной. Результат показан на рис. 23.3.

Внимательные читатели наверняка заметили проблему: свойство `DriveType` содержит `3`, но наш код явно ожидал, что оно будет содержать строковое значение `'fixed'`. Представим на секунду, что вы невнимательны, — на такой случай у нас есть и другие приемы отладки.

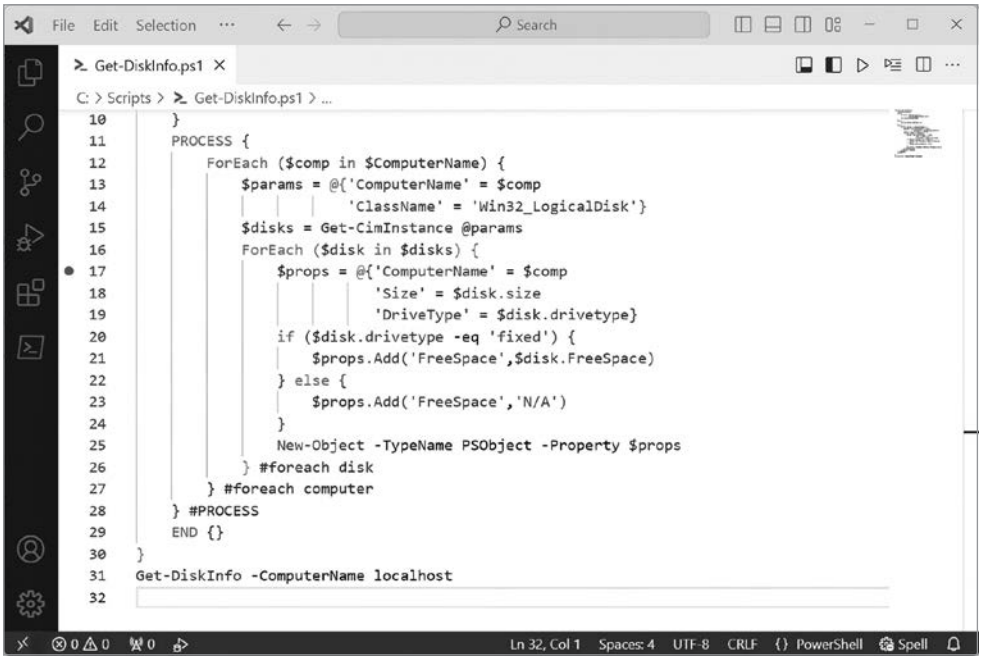


Рис. 23.1. Установка точки останова в VS Code

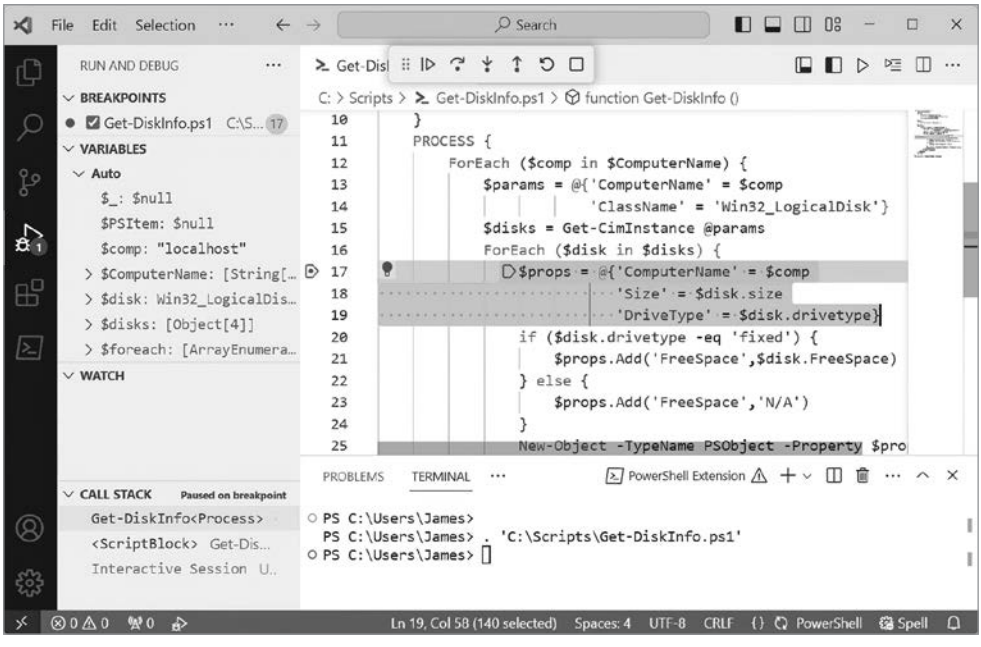


Рис. 23.2. Точка останова в VS Code

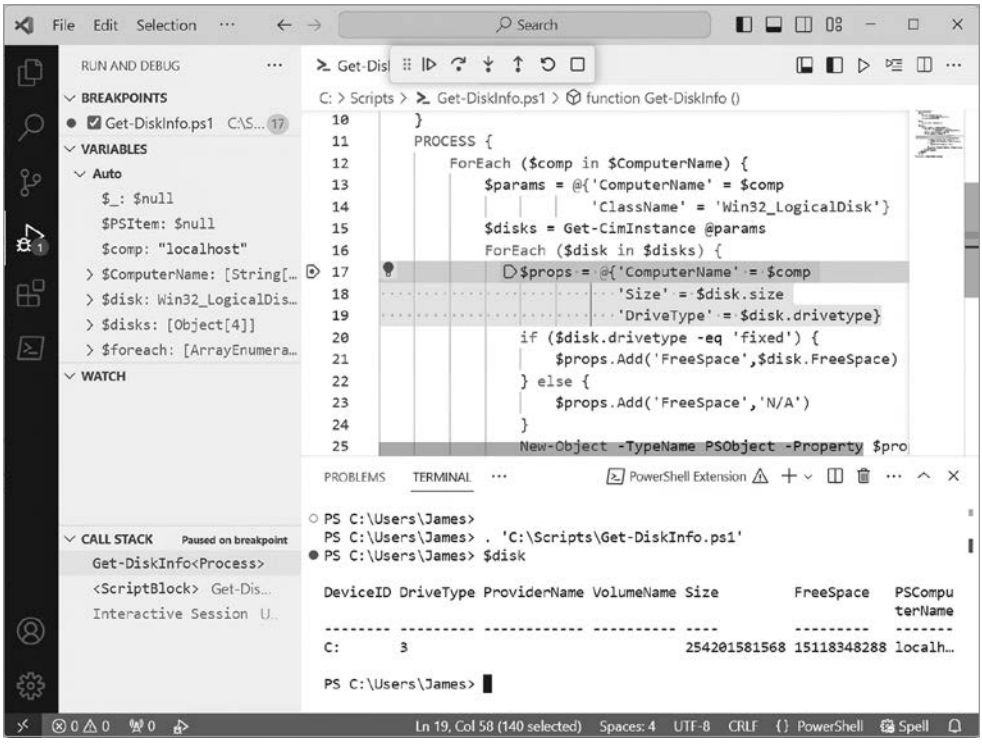


Рис. 23.3. Проверка содержимого переменной в режиме отладки

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Следующую часть интереснее наблюдать вживую, поэтому мы рекомендуем вам запустить VS Code с активным расширением PowerShell и создать новый файл. Сохраните его с расширением .ps1 (так VS Code будет понимать, что файл относится к PowerShell) и вставьте в него содержимое листинга 23.1. Установите точку останова на строке 17, как сделали мы, и выполните скрипт.

На строке 17 скрипт собирается войти в конструкцию `If`, где примет логическое решение. Нередко логические баги проявляются как раз в этих решениях. Скрипт решит, создавать ли ему свойство `FreeSpace` со значением, отражающим свободное пространство диска, или же вставить в его качестве значение `'N/A'`. Нажмите клавишу F11, выберите команду Step Into, и скрипт, продвинувшись на одну строку, снова остановится (рис. 23.4). Вы собираетесь выполнить логическую конструкцию.

Еще раз нажмите клавишу F11. Скрипт перейдет к строке 32 — вы можете видеть результаты предыдущего логического шага. Это означает, что `$disk.drivetype` не содержит `'fixed'`. Вы ожидали иного, а значит, нашли место бага. В этот момент вы можете нажать Shift+F5, чтобы остановить отладку и начать исправлять проблему.

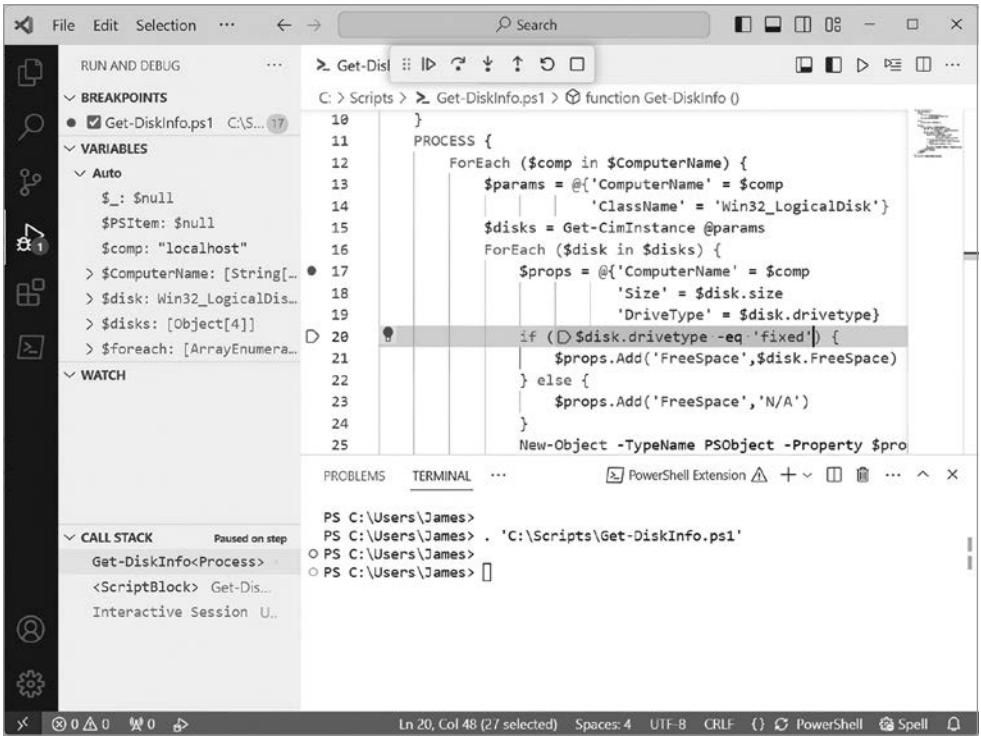


Рис. 23.4. Переход в следующую строку кода во время отладки

ВСЕ ОСНОВАНО НА ОЖИДАНИЯХ

Мы пропустили один ценный урок — или сохранили его для этого особого момента. Отладка связана с поиском мест, в которых ваши предположения и ожидания расходятся с реальностью. И суть в том, что вы имеете некие ожидания. Иными словами, у вас *должно быть* представление о том, какие действия будет выполнять ваш скрипт. Отладка же позволяет вам пронаблюдать, делает он это или нет.

В этом и заключаются ваши ожидания. Когда наступает время отладки, вы просто сравниваете реальность с этими ожиданиями. Там, где они различаются, и находится баг.

Если у вас нет ожиданий относительно того, что скрипт будет делать на каждом своем шаге и что будет содержать каждая переменная и свойство, то вы не сможете выполнять отладку.

23.4.2. Установка наблюдения

Поскольку содержимое переменных и свойств — столь важная часть отладки, VS Code предлагает функцию *наблюдения* (watch), которая сосредотачивается конкретно на этой части. В VS Code вы можете выбрать в меню Debug пункт Remove All Breakpoints, чтобы получить чистое состояние. Тем не менее панель Debug по-прежнему будет открыта — если вы случайно ее закрыли, нажмите Ctrl+Shift+D. В разделе WATCH щелкните на значке + (он не виден, пока вы не наведете указатель мыши на заголовок раздела WATCH). В появившемся текстовом окне введите `$disk` и нажмите клавишу Enter. Хорошо, если вы увидите нечто подобное изображению на рис. 23.5.

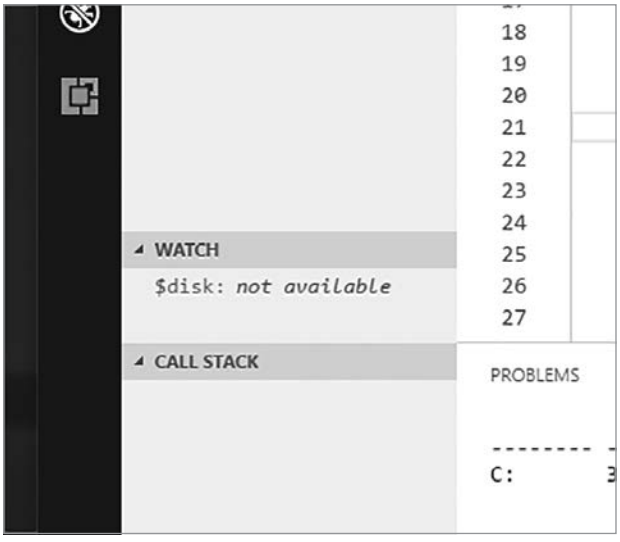


Рис. 23.5. Добавление наблюдения для $disk

Мы видим, что переменная сейчас `'unavailable'`, что вполне естественно, поскольку скрипт не выполняется. Мы повторно активируем точку останова на строке 17, которая идет сразу за `$disk` в цикле `ForEach`, и затем выполним скрипт.

23.4.3. Многое другое

Для реализации этих возможностей отладки VS Code использует команды PowerShell PSBreakpoint. Есть и много других функций, о которых мы не говорили в текущей главе. Читайте документацию VS Code, чтобы узнать о дополнительных возможностях, которые этот инструмент предлагает в плане отладки. И хотя

представленные нами приемы являются базовыми, они позволяют заложить прочную основу устранения багов в ваших скриптах.

Но вы *знали*, что это был `Win32_LogicalDisk`. Последующее *использование* переменной `$disk` заключается в проверке свойств `Size` и `DriveType` на строке 17 (рис. 23.6). Дважды щелкните на созданной функции «наблюдения», чтобы отредактировать ее. Добавьте `.drivetype` в конец `$disk`. Как показано на рис. 23.7, в нашей системе мы видим, что `DriveType` равен `3`.

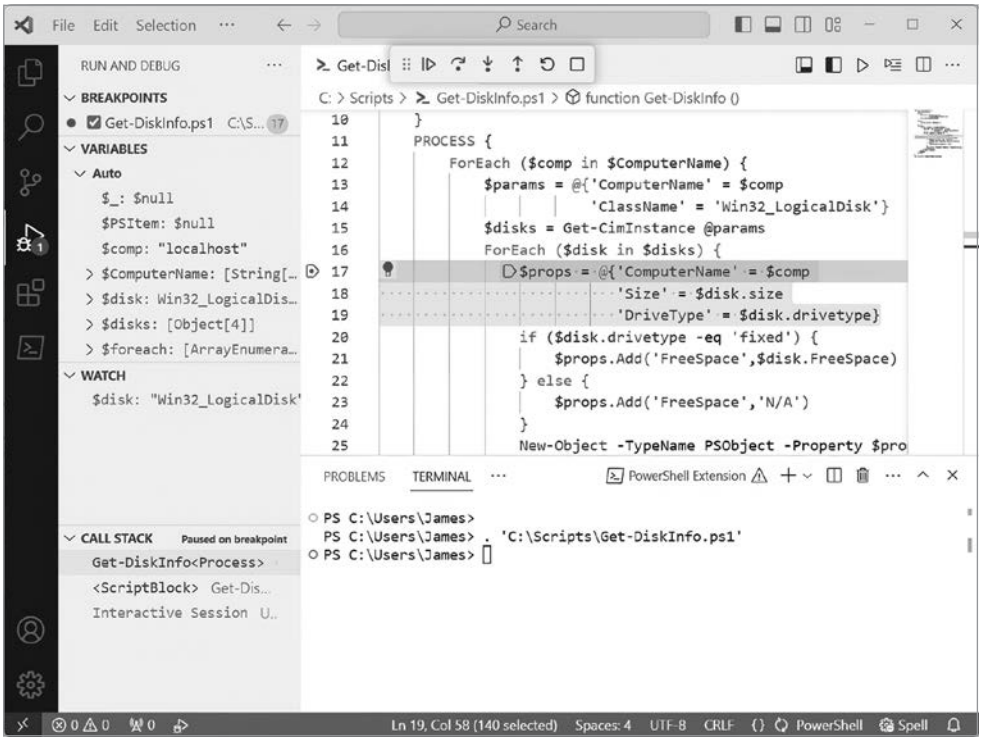


Рис. 23.6. Наблюдение того, как $disk раскрывает содержимое переменной на текущий момент

Польза подобных наблюдений в том, что вы можете снова нажать клавишу F5, чтобы выполнить скрипт до момента, пока он в очередной раз не встретит точку останова на строке 17. На нашем компьютере мы увидели изменение `DriveType` на `5`. У вас наверняка будет иначе, в зависимости от конфигурации. И вместо того чтобы каждый раз вводить `$disk.drivetype`, вы можете быстро обращаться к «наблюдениям» и видеть, что делают все ваши переменные.

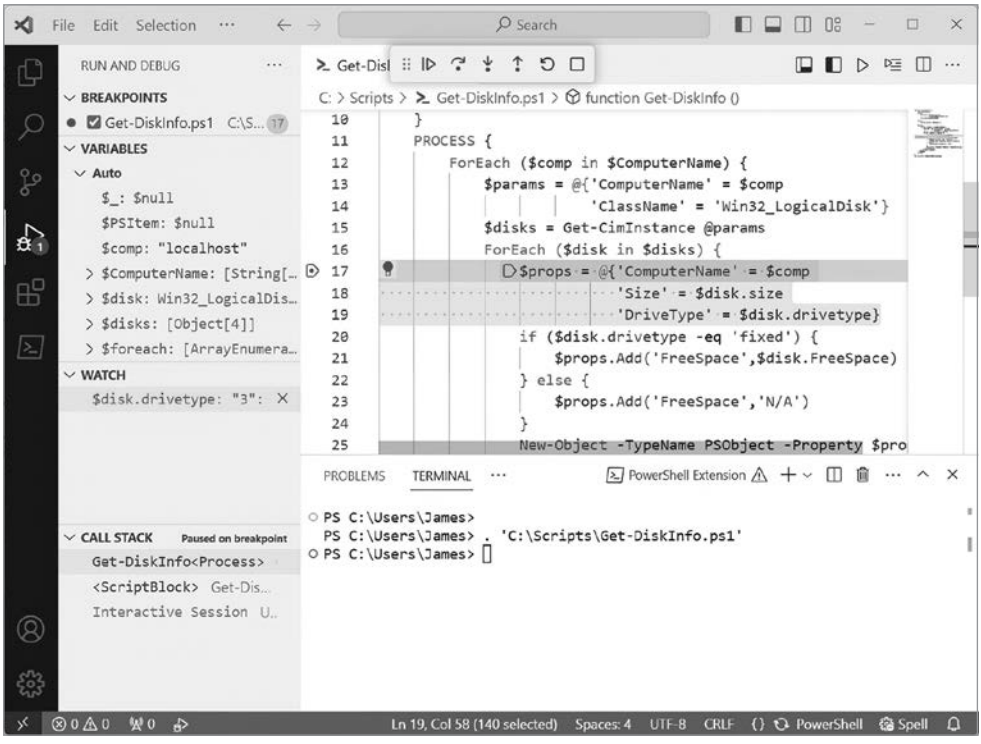


Рис. 23.7. Изменение функции «наблюдения», чтобы она сосредоточилась на конкретном свойстве

23.4.4. Не ленитесь

После всего этого вы можете думать: «Как много всего нужно делать». Но еще больший объем работы возникает, если вы будете пытаться выявить проблему, читая код или делая случайные догадки о том, что могло пойти не так. И мы видели, как разработчики действительно так поступают. Уделите время изучению процесса и отладочных возможностей вашего редактора. На это может уйти очень много времени, но все равно так будет быстрее, чем заниматься отладкой, строя догадки.

И обязательно вносите изменения по одному. Не стоит добавлять сразу пять изменений, поскольку вы не будете знать, какое из них решает проблему, к тому же рискуете создать еще больше багов. Меняйте что-то одно, тестируйте. Если проблема решена — отлично. Если нет, то откатывайте код назад и пробуйте что-то еще. Кроме того, будьте готовы к тому, что у вас может быть несколько багов, но некоторые из них вы обнаружите только после того, как устраните один или два.

23.5. УПРАЖНЕНИЯ

Освоив эти несколько приемов, вы уже можете обрабатывать большинство логических багов, которые будут возникать в процессе написания скрипта PowerShell. Но не верьте нам на слово — используйте навыки отладки.

23.5.1. Вводная информация

В листинге 23.2 показан скрипт с багом. Все верно, в своей текущей форме он *не выполнится*. Нам это известно, поскольку в этом и суть упражнения. Мы не хотим, чтобы вы выполняли данный скрипт, — пока что зайдите в VS Code, где вы можете его просмотреть. Сохраните скрипт в виде файла с расширением `.ps1`, иначе не сможете воспользоваться возможностями PowerShell, доступными в VS Code.

Листинг 23.2. Скрипт с багом, на котором можно отточить навыки отладки

```
Function Get-DiskCheck {
    [cmdletbinding(DefaultParameterSetName = "name")]
    Param(
        [Parameter(Position = 0, Mandatory,
            HelpMessage = "Enter a computer name to check",
            ParameterSetName = "name",
            ValueFromPipeline)]
    [Alias("cn")]
    [ValidateNotNullorEmpty()]
    [string[]]$ComputerName,
    [Parameter(Mandatory,
        HelpMessage = "Enter the path to a text file of the computer names",
        ParameterSetName = "file"
    )]
    [ValidateScript( {
            if (Test-Path $_) {
                $True
            }
            else {
                Throw "Cannot validate path $_"
            }
        })]
    [ValidatePattern("\.txt$")]
    [string]$Path,
    [ValidateRange(10, 50)]
    [int]$Threshhold = 25,
    [ValidateSet("C:", "D:", "E:", "F:")]
    [string]$Drive = "C:",
    [switch]$Test
)
Begin {
    Write-Verbose "[BEGIN ] Starting: $($MyInvocation.Mycommand)"
    $cimParam = @{
```

```
        Classname    = "Win32_LogicalDisk"
        Filter       = "DeviceID='$Drive'"
        Computername = $Null
        ErrorAction  = "Stop"
    }
    } #begin
    Process {
        if ($PSCmdlet.ParameterSetName = 'name') {
            $names = $Computernme
        }
        else {
            #получаем список имен и обрезаем лишние пробелы
            Write-Verbose "[PROCESS] Importing names from $path"
            $names = Get-Content -Path $path | Where {$_ -match "\w+"} |
foreach {$_.Trim()}
        }
        if ($test) {
            Write-Verbose "[PROCESS] Testing connectivity"
            #игнорируем ошибки компьютеров, которые не находятся в сети
            $names = $names | Where {Test-WSMan $_ -ErrorAction
SilentlyContinue}
        }
        foreach ($computer in $names) {
            $cimParam.ComputerName = $Computer
            Write-Verbose "[PROCESS] Querying $($computer.toUpper())"
            Try {
                $data = Get-Ciminstance @cimParam
                #записываем полученные результаты в конвейер
                $data | Select ComputerName,
                DeviceID, Size, Freespace,
                @{Name = "PctFree"; Expression =
{[math]:Round(($_.freespace / $_.size) * 100, 2)}},
                @{Name = "OK"; Expression = {
                        [int]$p = ($_.freespace / $_.size) * 100
                        if ($p -ge $Threshhold) {
                            $True
                        }
                        else {
                            $false
                        }
                    }
                }, @{Name = "Date"; Expression = {(Get-Date)}}
            }
            Catch {
                Write-Warning "[$($computer.toUpper())]
Failed.$($_.Exception.message)"
            }
        } #foreach computer
    } #process
    End {
        Write-Verbose "[END ] Ending: $($MyInvocation.Mycommand)"
    } #end
}
```

23.5.2. Ваша задача

Начните с *чтения* скрипта. Что он делает? Что будет содержать каждая переменная по ходу его выполнения? Что будут содержать свойства? Вы можете увидеть в нашем примере несколько багов — мы добавили логические и синтаксические, чтобы вы могли заняться отладкой как следует. Не ожидайте, что при чтении скрипта найдете все баги.

Когда закончите читать скрипт, займитесь его *отладкой*. Используйте приемы, которые освоили в этой главе, и посмотрите, сможете ли создать безошибочную версию, которая выполнится идеально.

23.5.3. Наше решение

Эта глава больше посвящена самой процедуре, чем коду, но мы хотим убедиться, что вы нашли все баги, поэтому в листинге 23.3 привели исправленный скрипт. Хотите выполнить дополнительное задание? Мы не добавляли примечания к данному листингу. Вместо этого попробуйте использовать `Compare-Object`, чтобы сравнить листинги 23.2 и 23.3, или сравните с одним из них свой исправленный скрипт, чтобы увидеть различия.

Листинг 23.3. Полностью отлаженный скрипт

```
Function Get-DiskCheck {
    [cmdletbinding(DefaultParameterSetName = "name")]
    Param(
        [Parameter(Position = 0, Mandatory,
            HelpMessage = "Enter a computer name to check",
            ParameterSetName = "name",
            ValueFromPipeline)]
        [Alias("cn")]
        [ValidateNotNullorEmpty()]
        [string[]]$ComputerName,
        [Parameter(Mandatory,
            HelpMessage = "Enter the path to a text file of the computer names",
            ParameterSetName = "file"
        )]
        [ValidateScript( {
                if (Test-Path $_) {
                    $True
                }
                else {
                    Throw "Cannot validate path $_"
                }
            })]
        [ValidatePattern("\.txt$")]
        [string]$Path,
        [ValidateRange(10, 50)]
        [int]$Threshhold = 25,
        [ValidateSet("C:", "D:", "E:", "F:")]
```

```
        [string]$Drive = "C:",
        [switch]$Test
    )
    Begin {
        Write-Verbose "[BEGIN ] Starting: $($MyInvocation.Mycommand)"
        $cimParam = @{
            Classname    = "Win32_LogicalDisk"
            Filter       = "DeviceID='$Drive'"
            ComputerName = $Null
            ErrorAction  = "Stop"
        }
    } #begin
    Process {
        if ($PSCmdlet.ParameterSetName -eq 'name') {
            $names = $ComputerName
        }
        else {
            #получаем список имен и обрезаем все лишние пробелы
            Write-Verbose "[PROCESS] Importing names from $path"
            $names = Get-Content -Path $path | Where {$_ -match "\w+"} |
foreach {$_.Trim()}
        }
        if ($test) {
            Write-Verbose "[PROCESS] Testing connectivity"
            #игнорируем ошибки компьютеров, которые не находятся в сети
            $names = $names | Where {Test-WSMan $_ -ErrorAction
SilentlyContinue}
    }
    foreach ($computer in $names) {
        $cimParam.ComputerName = $Computer
        Write-Verbose "[PROCESS] Querying $($computer.toUpper())"
        Try {
            $data = Get-Ciminstance @cimParam
            #записываем полученные результаты в конвейер
            $data | Select PSComputerName,
            DeviceID, Size, Freespace,
            @{Name = "PctFree"; Expression =
{[math]::Round(($_.freespace / $_.size) * 100, 2)}},
                @{Name = "OK"; Expression = {
                        [int]$p = ($_.freespace / $_.size) * 100
                        if ($p -ge $Threshhold) {
                            $True
                        }
                        else {
                            $false
                        }
                    }
                }, @{Name = "Date"; Expression = {(Get-Date)}}
            }
            Catch {
                Write-Warning "[$($computer.toUpper())]
Failed.$($_.Exception.message)"
            }
```

```
        } #foreach computer
    } #process
    End {
        Write-Verbose "[END ] Ending: $($MyInvocation.Mycommand)"
    } #end
}
```

ИТОГИ ГЛАВЫ

В этой главе вы получили ценные знания по отладке скриптов PowerShell, а это очень важный навык для любого автора скриптов. Мы разобрали три основные категории багов: синтаксические, логические и баги в результатах. При этом особое внимание уделили логическим, устранять которые обычно сложнее всего. Используя в выбранной интегрированной среде разработки (например, VS Code) такие приемы, как добавление точек останова и наблюдение, вы можете эффективно выявлять баги в своих скриптах и устранять их. Напоминаем, что отладка основывается на сравнении ваших ожиданий с реальностью и, постоянно практикуясь, вы станете профессионалом по части устранения багов в скриптах PowerShell. Сейчас же примените свои новообретенные навыки, исправив скрипт, представленный в этой главе. Посмотрите, как его отладить с помощью приемов, которые вы освоили.

24 Визуализация вывода скрипта

В предыдущих главах мы стремились помочь вам в создании инструментов, которые превосходно выполняют какую-то исключительно одну задачу. Для этих инструментов безразличен источник получаемого ввода до тех пор, пока этот ввод может быть передан им в качестве параметра. Их также не волнует, какова цель или место назначения создаваемых ими результатов, в связи с чем они не ориентированы на создание красивого вывода. Вы можете использовать встроенные командлеты `Format-` или `Select-Object` для улучшения эстетического облика выходных данных или удовлетворения предпочтений руководства. Тем не менее в этой главе мы разберем два сложных приема, благодаря которым можно повысить визуальную привлекательность вывода и выйти за рамки возможностей команд `Format-`.

24.1. ОТПРАВНАЯ ТОЧКА

Мы начнем с кода, который скопировали из конца главы 17 (листинг 24.1).

Листинг 24.1. Отправная точка для текущей главы

```
function Get-DiskInfo {
 foreach ($domain in (Get-ADForest).domains) {
   $hosts = Get-ADDomainController -filter * -server $domain |
   Sort-Object -Prop hostname
   ForEach ($h in $hosts) {
    $cs = Get-CimInstance -ClassName Win32_ComputerSystem `
                          # -ComputerName $h
    $props = @{'ComputerName' = $h
               'DomainController' = $h
               'Manufacturer' = $cs.manufacturer
               'Model' = $cs.model
               'TotalPhysicalMemory(GB)'=$cs.totalphysicalmemory / 1GB
               }
```

```
    New-Object -Type PSObject -Prop $props
   } #foreach $h
  } #foreach $domain
} #function
Export-ModuleMember -function Get-DiskInfo
```

Сохраните этот код в виде нового модуля с именем `Test.psm1`, которое будет означать, что он находится в каталоге `Test`, расположенном в `Documents/PowerShell/Modules`. В итоге его полное имя будет `Documents/PowerShell/Modules/Test/Test.psm1`. Все понятно?

В своем текущем виде вывод выглядит не очень хорошо. У кода есть пять свойств, что превышает допустимые четыре, на основе которых PowerShell может создать таблицу по умолчанию. Это означает, что вывод по умолчанию возвращается в виде списка:

```
PS C:\> get-diskinfo
DomainController          :  SRV1
ComputerName              :  SRV1
Model                     :  Virtual Machine
Manufacturer              :  Microsoft Corporation
TotalPhysicalMemory(GB)   :  31.5475044250488
```

Нам это не нравится. Возможно, вам нужна таблица или конкретные предустановленные свойства. Но вы знаете, что не нужно встраивать форматирование *в* саму команду, поскольку это действие нарушит важные правила, которые мы озвучивали в книге, ведь так?

24.2. СОЗДАЕМ ПРЕДСТАВЛЕНИЕ ПО УМОЛЧАНИЮ

Вместо этого мы используем встроенную в PowerShell систему форматирования. Нам нужно добиться, чтобы вывод вашей команды всегда отображался в виде таблицы и при этом вам не приходилось использовать дополнительные вспомогательные команды (например, передачи в `Format-Table`). Вы создадите *базовое представление*, которое подсистема форматирования PowerShell будет автоматически применять для отрисовки вывода команды. Вы измените только *визуальное представление* вывода команды, никак не изменяя фактически выводимых объектов.

24.2.1. Изучение представлений Microsoft

Почти любая нативная основная команда, которую вы выполняете в PowerShell, имеет вид по умолчанию, или базовое представление. Выполните в PowerShell `cd $pshome`, чтобы перейти в базовую папку, после чего выполните `Dir`. Вы увидите несколько файлов с расширением `.format.ps1xml`. Нас интересуют именно они, поскольку в них Microsoft определяет базовые представления для основных команд оболочки.

ЛОЖЬ И ЗАГАДКИ

Надеемся, вам известно, что эти базовые представления создают впечатление, будто PowerShell иногда врет. Например, выполнение `Get-EventLog system -newest 10` выведет красиво форматированную таблицу (попробуйте!), но некоторые имена столбцов будут отличаться от внутренних имен свойств. При рассмотрении предопределенного списка или таблицы заголовки определяются в представлении и *необязательно отображают внутренние объекты*. Когда вы выполняете `Get-Process`, числа, которые вы видите, вычисляются базовым представлением. Внутренние данные отображаются в байтах, а не килобайтах, мегабайтах или иных единицах. Представления визуальны, и вам нужно быть внимательными при их использовании в качестве дескрипторов фактических данных.

Аналогичный обман вы можете реализовать при создании представлений. Не хотите, чтобы заголовок таблицы читался как `ComputerName`? Не проблема — при желании вы можете сделать так, что он будет выводиться как `Mandolin`. И это способно вызвать большое недоумение у любого пользователя вашей команды, поскольку он может попробовать выполнить что-то вроде `Get-Whatever | Select-Object mandolin`, получив в ответ лишь пустой столбец, так как фактически никакого свойства `mandolin` нет. И это лишний раз напоминает об известной «обманчивости» PowerShell.

Стоит отметить и тот факт, что мы будем работать с XML-файлами, не имеющими формального определения или *объявления типа документа* (document type declaration, DTD). Якобы потому, что Microsoft хочет свободно вносить корректировки в систему в будущем (хотя за 15 с лишним лет так ни разу этого и не сделала). Если Microsoft не задокументирует форматы файлов, то вы не сможете пожаловаться, когда однажды они изменятся (теоретически). Честно говоря, мы видели код подсистемы форматирования (напомним, что сейчас код PowerShell уже в открытом доступе) и очень хотели бы верить, что компания несколько смущена всем этим и не хочет ее документировать, поскольку это вызывает болезненные воспоминания. Вся существующая документация доступна по адресу http://mng.bz/QBX0, и мы можем лишь пожелать удачи в ее изучении, так как она изложена очень сжато.

Если вам интересна эта тема, то прочтите книгу *PowerShell in Depth* (Manning, 2014; http://mng.bz/xjzq), где мы описываем ее более подробно. В текущей главе мы дадим скорее некое руководство, чем подробный разбор того, что вы можете реализовать.

В Блокноте, Visual Studio Code или любом другом редакторе вам нужно открыть файл `dotnettypes.format.ps1xml`. Существуют и другие файлы форматов, но этот гигант содержит представления для большинства основных объектов типов, с которыми

работает PowerShell. Немного разберем его, поскольку вы будете копировать отсюда фрагменты. Этот файл начинается так:

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- ****************************************************************
These sample files contain formatting information used by the Windows
PowerShell engine. Do not edit or change the contents of this file
directly. Please see the PowerShell documentation or type
Get-Help Update-FormatData for more information.
Copyright (c) Microsoft Corporation. All rights reserved.
THIS SAMPLE CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY
OF ANY KIND,WHETHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR
PURPOSE. IF THIS CODE AND INFORMATION IS MODIFIED, THE ENTIRE RISK OF USE
OR RESULTS IN CONNECTION WITH THE USE OF THIS CODE AND INFORMATION
REMAINS WITH THE USER.
**************************************************************** -->
<Configuration>
<ViewDefinitions>
```

Первая и две последние строки необходимы для создания файла. Создайте в VS Code новый файл и скопируйте эти три строки в его начало.

Сохраните созданный файл в той же папке, в которой хранится файл `.psm1` (если повторяете за нами). Назовите его `TestViews.format.ps1xml`. После его сохранения VS Code будет обеспечивать правильную раскраску синтаксиса для XML, что от него и требуется. Завершите файл, закрыв эти два открывающих тега:

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
<ViewDefinitions>
</ViewDefinitions>
</Configuration>
```

Все в XML заключается в парные *теги*, и в каждую пару вкладывается другая. Открывающая часть `<?xml ?>` не является тегом. Это определение документа, поэтому оно здесь только одно.

Все остальное в файле состоит из разделов `<View></View>`. Каждый из них является *представлением* — что понятно по имени тега — и определяет один способ отображения одного вида объекта. Вот пример:

```
<View>
<Name>System.CodeDom.Compiler.CompilerError</Name>      ← Имя представления
<ViewSelectedBy>                                        ← Дополнительные
<TypeName>System.CodeDom.Compiler.CompilerError</TypeName>   критерии выбора
</ViewSelectedBy>
<ListControl>      ← Тип представления
<ListEntries>
<ListEntry>
<ListItems>        ← Определение списка
```

```
<ListItem>
<PropertyName>ErrorText</PropertyName>
</ListItem>
<ListItem>
<PropertyName>Line</PropertyName>
</ListItem>
<ListItem>
<PropertyName>Column</PropertyName>
</ListItem>
<ListItem>
<PropertyName>ErrorNumber</PropertyName>
</ListItem>
<ListItem>
<PropertyName>LineSource</PropertyName>
</ListItem>
</ListItems>
</ListEntry>
</ListEntries>
</ListControl>
</View>
```

Разберем, что мы здесь видим.

- У представления есть имя. Обычно это названия типа объекта, но необязательно. Честно говоря, идея именования представлений, чтобы к ним можно было обращаться, так и не была реализована. Смысл состоял в том, чтобы один тип объекта мог иметь несколько вариантов представления и что с помощью команд `Format-` вы могли бы дать PowerShell указание, какой из них использовать. Но при этом нет способа вывести их все, поэтому замысел так и не развился.
- Представление выбирается по конкретному имени типа. Это важно! Сейчас команда создает объекты типа `System.PSCustomObject`. Это распространенный тип, и он не уникален для этой команды, из-за чего возникает проблема. Вы можете создать представление, только если ваша команда производит объект уникального типа. Вам потребуется исправить это в своей команде.
- Этот пример в противоположность представлению-таблице показывает представление-список.
- Представление-список состоит из записей, каждая из которых содержит элемент списка. В этом примере они указывают имена свойств, которые должны отображаться в списке.

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Прокрутите файл и обратите внимание на другие типы представлений и элементы (кроме имен свойств), которые они содержат. Заметьте, что элементы управления таблицей стали более сложными и в них есть весь раздел исключительно для заголовков столбцов, к которому, в свою очередь, прилагаются разделы для содержимого этих столбцов.

24.2.2. Добавление к выводимым объектам пользовательского имени типа

Вы знаете, что вам нужно изменить код, и это изменение представлено в листинге 24.2.

Листинг 24.2. Добавление объекту пользовательского имени типа

```
function Get-DiskInfo {
 foreach ($domain in (Get-ADForest).domains) {
   $hosts = Get-ADDomainController -filter * -server $domain |
   Sort-Object -Prop hostname
   ForEach ($h in $hosts) {
    $cs = Get-CimInstance -ClassName Win32_ComputerSystem -ComputerName $h
    $props = @{'ComputerName' = $h
               'DomainController' = $h
               'Manufacturer' = $cs.manufacturer
               'Model' = $cs.model
               'TotalPhysicalMemory(GB)'=$cs.totalphysicalmemory / 1GB
               }
     $obj = New-Object -Type PSObject -Prop $props
     $obj.psobject.typenames.insert(0,'Toolmaking.DiskInfo')
     Write-Output $obj
   } #foreach $h
  } #foreach $domain
} #function
Export-ModuleMember -function Get-DiskInfo
```

Сохраняет объект в переменную

Вставляет новое имя типа

Это незначительное изменение: вы сохранили вывод объекта в переменную — `$obj` — вместо того, чтобы непосредственно отправить его в конвейер. После этого вы вставляете имя типа, `Toolmaking.DiskInfo`, и помещаете объект в конвейер. Новое имя типа заменит его изначальное обобщенное имя.

ВЫБОР ИМЕНИ ТИПА

Соглашения по именованию .NET Framework созданы так, чтобы делать каждое имя типа уникальным. Вам не нужно добавлять собственное имя типа наподобие `System.DiskInfo`, поскольку, как вам известно, оно либо уже существует, либо может появиться в будущем. `System` считается *пространством имен* и «принадлежит» Microsoft. Все, что начинается с `System`, находится под контролем Microsoft, и вам не следует вторгаться на территорию компании.

По сути, мы определили новое пространство имен `Toolmaking`, в котором у нас есть свобода создавать все, что мы захотим, — и вы должны поступить так же, возможно используя в качестве пространства имен верхнего уровня форму имени вашей организации. Если вы работаете в IT-отделе и конкретно в команде Storage, то в этом примере можете выбрать вариант `MyCompany.ITOps.Storage.DiskInfo`. Смысл в том, чтобы создать иерархию, которая позволяет отдельным группам получать полный контроль над их собственными пространствами имен, не пересекаясь с другими.

24.2.3. Создание нового файла представлений

В листинге 24.3 показано начало нового файла представлений. Обратите внимание: мы нашли представление-таблицу, которое нам нравится, и будем использовать его в качестве отправной точки.

Листинг 24.3. Начало нового файла представлений

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
<ViewDefinitions>
<View>
<Name>System.Reflection.Assembly</Name>
<ViewSelectedBy>
<TypeName>System.Reflection.Assembly</TypeName>
</ViewSelectedBy>
<TableControl>          <-- Определение таблицы
<TableHeaders>
<TableColumnHeader>     <-- Определяем заголовки таблицы
<Label>GAC</Label>
<Width>6</Width>
</TableColumnHeader>
<TableColumnHeader>
<Label>Version</Label>
<Width>14</Width>
</TableColumnHeader>
<TableColumnHeader/>    <-- Одиночный тег
</TableHeaders>
<TableRowEntries>       <-- Соответствующие значения
<TableRowEntry>
<TableColumnItems>
<TableColumnItem>
<PropertyName>GlobalAssemblyCache</PropertyName>
</TableColumnItem>
<TableColumnItem>
<PropertyName>ImageRuntimeVersion</PropertyName>
</TableColumnItem>
<TableColumnItem>
<PropertyName>Location</PropertyName>
</TableColumnItem>
</TableColumnItems>
</TableRowEntry>
</TableRowEntries>
</TableControl>
</View>
</ViewDefinitions>
</Configuration>
```

Вам предстоит выполнить кое-какую работу, например, добавить пользовательское имя типа и упорядочить таблицу нужным образом. Но мы хотим привлечь ваше внимание к конкретной строке:

```
<TableColumnHeader/>
```

Это коварный XML-элемент, который часто используют в Microsoft, и он будет сбивать вас с толку. Помните, мы говорили о том, что XML-элементы идут парами? Так вот это не всегда так. Этот *одиночный* тег одновременно и открывает, и закрывает себя — что и означает слеш в его конце. По сути, это то же самое, что:

```
<TableColumnHeader>
</TableColumnHeader>
```

Пересчитайте количество заголовков столбцов таблицы в файле прямо сейчас. Хорошо, если у вас получится три. Количество записей столбцов таблицы *должно* совпадать. Если это не так, то представление не загрузится в оболочке. Тем не менее есть риск упустить одиночные теги при копировании и вставке, что приведет к повреждению файла форматирования. Поэтому будьте внимательны. Такие теги означают: «Хочу столбец здесь, но не хочу указывать ничего в качестве заголовка — просто используйте внутреннее имя свойства и самостоятельно выясните ширину». В листинге 24.4 показан итоговый файл.

Листинг 24.4. Итоговый файл представления

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
<ViewDefinitions>
<View>
<Name>DiskInfo</Name>
<ViewSelectedBy>
<TypeName>Toolmaking.DiskInfo</TypeName>
</ViewSelectedBy>
<TableControl>
<TableHeaders>
<TableColumnHeader>
<Label>Host</Label>
<Width>16</Width>
</TableColumnHeader>
<TableColumnHeader>
<Label>DC</Label>
<Width>16</Width>
</TableColumnHeader>
<TableColumnHeader>
<Label>Model</Label>
</TableColumnHeader>
<TableColumnHeader>
<Label>RAM</Label>
<Alignment>Right</Alignment>
</TableColumnHeader>
</TableHeaders>
<TableRowEntries>
<TableRowEntry>
<TableColumnItems>
<TableColumnItem>
<PropertyName>ComputerName</PropertyName>
</TableColumnItem>
<TableColumnItem>
```

Имя представления

Использует пользовательское имя типа

Заголовки таблицы

Обеспечивает правое выравнивание этого столбца

Значения таблицы

```
<PropertyName>DomainController</PropertyName>
</TableColumnItem>
<TableColumnItem>
<PropertyName>Model</PropertyName>
</TableColumnItem>
<TableColumnItem>
<PropertyName>TotalPhysicalMemory(GB)</PropertyName>
</TableColumnItem>
</TableColumnItems>
</TableRowEntry>
</TableRowEntries>
</TableControl>
</View>
</ViewDefinitions>
</Configuration>
```

Предлагаем открыть XML-файл в текстовом редакторе или VS Code и просмотреть его еще раз. Вам нужно обратить внимание на следующие нюансы.

- Вы указываете имя (которое должно быть уникальным только *для каждого имени типа*; вполне допустимо, если есть представление с таким же именем *для другого типа*) и пользовательское имя типа.
- Вы можете сделать так, чтобы количество заголовков столбцов и записей было одинаковым, отказавшись от использования раздражающих одиночных тегов.
- Вы указываете правое выравнивание для числового столбца RAM.
- Заголовки столбцов не совпадают по количеству с внутренними именами свойств. Дело в том, что имена свойств слишком длинные, — вы никак не можете добиться красивого вывода при таких длинных именах.

Важным выводом здесь будет то, что мы проектировали инструмент без особых стараний. Взгляните на это свойство — `TotalPhysicalMemory(GB)`, — его же явно не назовешь красивым. Мы лишь сделали так, чтобы *красиво выглядел* базовый вывод инструмента, — остальное вторично. Мы получили несуразное и неудобное для обращения свойство, которое *всегда* будет сложно вводить.

Изменим код. В листинге 24.5 представлена его обновленная версия, а в листинге 24.6 — соответствующий файл представления. Этот код был написан для того, чтобы показать, почему нет нужды беспокоиться о красивом облике изнутри инструмента, и указать на важность исправления подобных ошибок, когда вы их за собой замечаете.

Листинг 24.5. Измененный код инструмента

```
function Get-DiskInfo {
 foreach ($domain in (Get-ADForest).domains) {
   $hosts = Get-ADDomainController -filter * -server $domain |
   Sort-Object -Prop hostname
   ForEach ($h in $hosts) {
    $cs = Get-CimInstance -ClassName Win32_ComputerSystem -ComputerName $h
```

```
    $props = @{'ComputerName' = $h
               'DomainController' = $h
               'Manufacturer' = $cs.manufacturer
               'Model' = $cs.model
               'TotalPhysicalMemory'=$cs.totalphysicalmemory / 1GB
               }
     $obj = New-Object -Type PSObject -Prop $props
     $obj.psobject.typenames.insert(0,'Toolmaking.DiskInfo')
     Write-Output $obj
   } #foreach $h
  } #foreach $domain
} #function
Export-ModuleMember -function Get-DiskInfo
```

Листинг 24.6. Измененное представление

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
<ViewDefinitions>
<View>
<Name>DiskInfo</Name>
<ViewSelectedBy>
<TypeName>Toolmaking.DiskInfo</TypeName>
</ViewSelectedBy>
<TableControl>
<TableHeaders>
<TableColumnHeader>
<Label>Host</Label>
<Width>16</Width>
</TableColumnHeader>
<TableColumnHeader>
<Label>DC</Label>
<Width>16</Width>
</TableColumnHeader>
<TableColumnHeader>
<Label>Model</Label>
</TableColumnHeader>
<TableColumnHeader>
<Label>RAM</Label>
<Alignment>Right</Alignment>
</TableColumnHeader>
</TableHeaders>
<TableRowEntries>
<TableRowEntry>
<TableColumnItems>
<TableColumnItem>
<PropertyName>ComputerName</PropertyName>
</TableColumnItem>
<TableColumnItem>
<PropertyName>DomainController</PropertyName>
</TableColumnItem>
<TableColumnItem>
<PropertyName>Model</PropertyName>
</TableColumnItem>
```

```
<TableColumnItem>
<PropertyName>TotalPhysicalMemory</PropertyName>
</TableColumnItem>
</TableColumnItems>
</TableRowEntry>
</TableRowEntries>
</TableControl>
</View>
</ViewDefinitions>
</Configuration>
```

Так намного лучше!

24.2.4. Добавление файла представления в модуль

Вы уже сохранили файл представления в одной папке с файлом модуля `.psm1`. Но таким образом PowerShell не получит указание о том, что нужно этот файл представления *использовать*. Вместо этого вам нужно создать манифест модуля, как вы делали ранее, и сохранить его в виде `Test.psd1` (поскольку `Test` — имя модуля). При создании манифеста вам нужно указать представление формата. Или, если вы уже создали манифест, можете добавить представление форматов в него. Мы возьмем второй вариант, чтобы вы могли увидеть, как это делается. Выполните следующую команду:

```
new-modulemanifest -Path test.psd1 -RootModule test.psm1
```

Она создаст файл `.psd1`, но не укажет представление. Откройте его и отредактируйте в соответствии с листингом 24.7.

Листинг 24.7. Завершенный манифест модуля

```
#
# Манифест модуля для модуля'test'
#
# Сгенерировал: User
#
# Сгенерирован: 09/24/2023
#
@{
# Модуль скрипта или двоичный файл модуля, связанный с этим манифестом
RootModule = 'test.psm1'
# Версия этого модуля
ModuleVersion = '1.0'
# Поддерживаемые PSEditions
# CompatiblePSEditions = @()
# ID, используемый для идентификации этого модуля
GUID = 'e2baeaab-4dc7-4eda-a8a8-ad38298e3af0'
# Автор модуля
Author = 'User'
# Компания или поставщик модуля
CompanyName = 'Unknown'
# Заявление об авторских правах на модуль
```

```
Copyright = '(c) 2023 User. All rights reserved.'
# Описание функциональности модуля
# Description = ''
# Минимальная версия PowerShell, необходимая модулю
# PowerShellVersion = ''
# Имя хоста PowerShell, необходимого модулю
# PowerShellHostName = ''
# Минимальная версия хоста PowerShell, необходимого модулю
# PowerShellHostVersion = ''
# Минимальная версия Microsoft .NET Framework, необходимого модулю. Актуально только
для PowerShell Desktop.
# DotNetFrameworkVersion = ''
# Минимальная версия общеязыковой среды common language runtime (CLR), необходимой
модулю. Актуально только для PowerShell Desktop.
# CLRVersion = ''
# Архитектура процессора (None, X86, Amd64), необходимая модулю
# ProcessorArchitecture = ''
# Модули, которые необходимо импортировать в глобальную среду, прежде чем
импортировать этот модуль
# RequiredModules = @()
# Сборки, которые необходимо загрузить перед импортом этого модуля
# RequiredAssemblies = @()
# Файлы скриптов (.ps1), выполняемые в среде вызывающего перед импортом этого модуля
# ScriptsToProcess = @()
# Фалы типов (.ps1xml), загружаемые при импорте этого модуля
# TypesToProcess = @()
# Файлы форматов (.ps1xml), загружаемые при импорте этого модуля
FormatsToProcess = @('./TestView.format.ps1xml')
# Модули, импортируемые в качестве вложенных в модуль, указанный в RootModule/
ModuleToProcess
# NestedModules = @()
# Функции, экспортируемые из этого модуля. Чтобы улучшить производительность,
не используйте подстановку и не удаляйте запись. Если функции для экспорта нет,
то используйте пустой массив.
FunctionsToExport = '*'
# Командлеты для экспорта из этого модуля. Чтобы улучшить производительность,
не используйте подстановку и не удаляйте запись. Если для экспорта командлетов нет,
то используйте пустой массив.
CmdletsToExport = '*'
# Переменные для экспорта из этого модуля
VariablesToExport = '*'
# Псевдонимы для экспорта из этого модуля. Чтобы улучшить производительность,
не используйте подстановку и не удаляйте запись. Если для экспорта псевдонимов нет,
то используйте пустой массив.
AliasesToExport = '*'
# DSC-ресурсы для экспорта из модуля
# DscResourcesToExport = @()
# Список всех модулей, упакованных с этим модулем
# ModuleList = @()
# Список всех файлов, упакованных с этим модулем
# FileList = @()
# Приватные данные для передачи в модуль, указанный в RootModule/ModuleToProcess.
Они могут содержать хеш-таблицу PSData с дополнительными метаданными модуля,
используемыми PowerShell.
```

```
PrivateData = @{
    PSData = @{
        # Теги, применяемые к модулю. Они помогают находить его в онлайн-галереях.
        # Tags = @()
        # URL лицензии этого модуля
        # LicenseUri = ''
        # URL основного сайта этого проекта
        # ProjectUri = ''
        # URL значка, представляющего модуль
        # IconUri = ''
        # ReleaseNotes этого модуля
        # ReleaseNotes = ''
    } # Конец хеш-таблицы PSData
} # Конец хеш-таблицы PrivateData
# HelpInfo URI этого модуля
# HelpInfoURI = ''
# Предустановленный префикс для команд, экспортируемых из этого модуля.
Переопределите этот префикс, используя Import-Module -Prefix.
# DefaultCommandPrefix = ''
}
```

Если вы не можете найти внесенные нами изменения, то вот они:

```
# Файлы форматов (.ps1xml), загружаемые при импорте этого модуля
FormatsToProcess = @('./TestView.format.ps1xml')
```

Мы раскомментировали строку `FormatsToProcess` и добавили файл `TestView.format.ps1xml`, который находится в одной папке с файлами `.psd1` и `.psm1`. Наведя порядок, вы сможете выполнить команду и увидеть в качестве ее базового вывода новое представление:

```
PS C:\> get-diskinfo
Host            DC           Model                                      RAM
----            --           -----                                      ---
SRV1           SERV          Virtual Machine               1.99906539916992
```

24.3. УПРАЖНЕНИЯ

Мы хотим дать вам возможность выполнить все это самостоятельно. Мы предоставим вам инструмент и попросим сделать для него собственное представление.

24.3.1. Вводная информация

В листинге 24.8 приведен инструмент PowerShell. Он должен отлично работать (и выглядеть знакомым, поскольку мы его уже использовали). Вам нужно создать для него пользовательское представление. Это подразумевает, что его нужно сохранить в качестве модуля.

Листинг 24.8. Начальный скрипт

```
function Get-MachineInfo {
    [CmdletBinding()]
    Param(
        [Parameter(ValueFromPipeline=$True,
                   Mandatory=$True)]
        [Alias('CN','MachineName','Name')]
        [string[]]$ComputerName
    )
BEGIN {}
PROCESS {
    foreach ($computer in $ComputerName) {

        $session = New-CimSession -ComputerName $computer `
                                  -SessionOption $option
        # Запрос данных
        $os_params = @{'ClassName'='Win32_OperatingSystem'
                       'CimSession'=$session}
        $os = Get-CimInstance @os_params
        $cs_params = @{'ClassName'='Win32_ComputerSystem'
                       'CimSession'=$session}
        $cs = Get-CimInstance @cs_params
        $sysdrive = $os.SystemDrive
        $drive_params = @{'ClassName'='Win32_LogicalDisk'
                          'Filter'="DeviceId='$sysdrive'"
                          'CimSession'=$session}
        $drive = Get-CimInstance @drive_params
        $proc_params = @{'ClassName'='Win32_Processor'
                         'CimSession'=$session}
        $proc = Get-CimInstance @proc_params |
                Select-Object -first 1
        # Закрытие сеанса
        $session | Remove-CimSession
        # Вывод данных
        $props = @{'ComputerName'=$computer
                   'OSVersion'=$os.version
                   'SPVersion'=$os.servicepackmajorversion
                   'OSBuild'=$os.buildnumber
                   'Manufacturer'=$cs.manufacturer
                   'Model'=$cs.model
                   'Procs'=$cs.numberofprocessors
                   'Cores'=$cs.numberoflogicalprocessors
                   'RAM'=($cs.totalphysicalmemory / 1GB)
                   'Arch'=$proc.addresswidth
                   'SysDriveFreeSpace'=$drive.freespace}
        $obj = New-Object -TypeName PSObject -Property $props
        Write-Output $obj
    } #foreach
} #PROCESS
END {}
} #function
```

24.3.2. Ваша задача

Мы хотим, чтобы созданное вами представление содержало пять столбцов: `ComputerName`, `OSVersion`, `Model`, `Cores` и `RAM`. Используйте для них исходные имена свойств, вместо того чтобы создавать другие заголовки.

24.3.3. Наше решение

В листинге 24.9 показан измененный инструмент — нам нужно добавить собственное имя типа.

Листинг 24.9. Измененный файл .psm1

```
function Get-MachineInfo {
    [CmdletBinding()]
    Param(
        [Parameter(ValueFromPipeline=$True,
                   Mandatory=$True)]
        [Alias('CN','MachineName','Name')]
        [string[]]$ComputerName
    )
BEGIN {}
PROCESS {
    foreach ($computer in $ComputerName) {

        # Сеанс подключения
        $session = New-CimSession -ComputerName $computer `
                                  -SessionOption $option
        # Запрос данных
        $os_params = @{'ClassName'='Win32_OperatingSystem'
                       'CimSession'=$session}
        $os = Get-CimInstance @os_params
        $cs_params = @{'ClassName'='Win32_ComputerSystem'
                       'CimSession'=$session}
        $cs = Get-CimInstance @cs_params
        $sysdrive = $os.SystemDrive
        $drive_params = @{'ClassName'='Win32_LogicalDisk'
                          'Filter'="DeviceId='$sysdrive'"
                          'CimSession'=$session}
        $drive = Get-CimInstance @drive_params
        $proc_params = @{'ClassName'='Win32_Processor'
                         'CimSession'=$session}
        $proc = Get-CimInstance @proc_params |
                Select-Object -first 1
        # Закрытие сеанса
        $session | Remove-CimSession
        # Вывод данных
        $props = @{'ComputerName'=$computer
                   'OSVersion'=$os.version
                   'SPVersion'=$os.servicepackmajorversion
                   'OSBuild'=$os.buildnumber
                   'Manufacturer'=$cs.manufacturer
```

```
                    'Model'=$cs.model
                    'Procs'=$cs.numberofprocessors
                    'Cores'=$cs.numberoflogicalprocessors
                    'RAM'=($cs.totalphysicalmemory / 1GB)
                    'Arch'=$proc.addresswidth
                    'SysDriveFreeSpace'=$drive.freespace}
        $obj = New-Object -TypeName PSObject -Property $props
        $obj.psobject.typenames.insert('Toolmaking.MachineInfo')
        Write-Output $obj
    } #foreach
} #PROCESS
END {}
} #function
```

Вставляет пользовательское имя типа

В листинге 24.10 показан наш файл представления. Мы хотели использовать в качестве заголовков столбцов имена свойств, поэтому могли применить для большинства из них одиночный тег (мы хотели, чтобы столбцы `Cores` и `RAM` были выровнены по правому краю, поэтому были вынуждены применить полноценные теги). Но эти одиночные теги так часто создавали путаницу, что мы решили указать заголовок каждого столбца, используя полноценную пару тегов.

Листинг 24.10. Наш новый файл .format.ps1xml

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
<ViewDefinitions>
<View>
<Name>MachineInfo</Name>
<ViewSelectedBy>
<TypeName>Toolmaking.MachineInfo</TypeName>
</ViewSelectedBy>
<TableControl>
<TableHeaders>
<TableColumnHeader>
<Label>ComputerName</Label>
</TableColumnHeader>
<TableColumnHeader>
<Label>OSVersion</Label>
</TableColumnHeader>
<TableColumnHeader>
<Label>Model</Label>
</TableColumnHeader>
<TableColumnHeader>
<Label>Cores</Label>
<Alignment>Right</Alignment>
</TableColumnHeader>
<TableColumnHeader>
<Label>RAM</Label>
<Alignment>Right</Alignment>
</TableColumnHeader>
</TableHeaders>
<TableRowEntries>
```

```
<TableRowEntry>
<TableColumnItems>
<TableColumnItem>
<PropertyName>ComputerName</PropertyName>
</TableColumnItem>
<TableColumnItem>
<PropertyName>OSVersion</PropertyName>
</TableColumnItem>
<TableColumnItem>
<PropertyName>Model</PropertyName>
</TableColumnItem>
<TableColumnItem>
<PropertyName>Cores</PropertyName>
</TableColumnItem>
<TableColumnItem>
<PropertyName>RAM</PropertyName>
</TableColumnItem>
</TableColumnItems>
</TableRowEntry>
</TableRowEntries>
</TableControl>
</View>
</ViewDefinitions>
</Configuration>
```

В листинге 24.11 показан наш манифест для этого модуля.

Листинг 24.11. Наш новый файл .psd1

```
#
# Манифест для модуля 'test'
#
# Сгенерировал: User
#
# Сгенерирован: 09/24/2023
#
@{
# Модуль скрипта или двоичный файл модуля, связанные с этим манифестом
RootModule = 'test.psm1'
# Номер версии модуля
ModuleVersion = '1.0'
# Поддерживаемые PSEditions
# CompatiblePSEditions = @()
# ID, используемый для идентификации модуля
GUID = 'e2baeaab-4dc7-4eda-a8a8-ad38298e3af0'
# Автор модуля
Author = 'User'
# Компания или поставщик модуля
CompanyName = 'Unknown'
# Заявление об авторских правах на модуль
Copyright = '(c) 2023 User. All rights reserved.'
# Описание функциональности модуля
# Description = ''
# Минимальная версия PowerShell, необходимая модулю
```

```
# PowerShellVersion = ''
# Имя хоста PowerShell, необходимого модулю
# PowerShellHostName = ''
# Минимальная версия хоста PowerShell, необходимая модулю
# PowerShellHostVersion = ''
# Минимальная версия Microsoft .NET Framework, необходимого модулю. Актуально только
для PowerShell Desktop.
# DotNetFrameworkVersion = ''
# Минимальная версия общеязыковой среды (common language runtime, CLR), необходимой
модулю. Актуально только для PowerShell Desktop.
# CLRVersion = ''
# Архитектура процессора, необходимая модулю (None, X86, Amd64)
# ProcessorArchitecture = ''
# Модули, которые необходимо импортировать в глобальную среду до импорта этого
модуля
# RequiredModules = @()
# Сборки, которые необходимо загрузить до импорта этого модуля
# RequiredAssemblies = @()
# Файлы скриптов (.ps1), которые выполняются в среде вызывающего до импорта этого
модуля
# ScriptsToProcess = @()
# Файлы типов (.ps1xml), загружаемые при импорте модуля
# TypesToProcess = @()
# Файлы форматов (.ps1xml), загружаемые при импорте модуля
FormatsToProcess = @('./TestView.format.ps1xml')
# Модули, импортируемые в качестве вложенных в модуль, указанные в RootModule/
ModuleToProcess
# NestedModules = @()
# Функции, экпортируемые из этого модуля. Чтобы улучшить производительность,
не используйте подстановку и не удаляйте запись. Если функции для экспорта нет,
то используйте пустой массив.
FunctionsToExport = '*'
# Командлеты для экспорта из этого модуля. Чтобы улучшить производительность,
не используйте подстановку и не удаляйте запись. Если командлетов для экспорта нет,
то используйте пустой массив.
CmdletsToExport = '*'
# Переменные для экспорта из этого модуля. VariablesToExport = '*'
# Псевдонимы для экспорта из этого модуля. Чтобы улучшить производительность,
не используйте подстановку и не удаляйте запись. Если псевдонимов для экспорта нет,
то используйте пустой массив.
AliasesToExport = '*'
# Ресурсы DSC для экспорта из этого модуля
# DscResourcesToExport = @()
# Список всех модулей, упакованных с этим модулем
# ModuleList = @()
# Список всех файлов, упакованных с этим модулем
# FileList = @()
# Приватные данные для передачи в модуль, указанный в RootModule/ModuleToProcess.
Они могут содержать хеш-таблицу PSData с дополнительными метаданными модуля,
используемыми PowerShell.
PrivateData = @{
    PSData = @{
        # Теги, применяемые к этому модулю. Помогают находить его в онлайн-галереях.
        # Tags = @()
```

```
        # URL лицензии модуля
        # LicenseUri = ''
        # URL основного сайта проекта
        # ProjectUri = ''
        # URL значка, представляющего модуль
        # IconUri = ''
        # ReleaseNotes модуля
        # ReleaseNotes = ''
    } # Конец хеш-таблицы PSData
} # Конец хеш-таблицы PrivateData
# HelpInfo URI этого модуля
# HelpInfoURI = ''
# Предустановленный префикс для команд, экспортируемых из этого модуля.
Переопределите его, используя Import-Module -Prefix.
# DefaultCommandPrefix = ''
}
```

ИТОГИ ГЛАВЫ

Эта глава была посвящена оптимизации представления вывода скрипта в PowerShell. Главным принципом остается создание инструментов, которые качественно выполняют конкретные задачи вне зависимости от источников ввода или точки вывода. Мы представили продвинутые техники, которые позволяют улучшить визуальную привлекательность вывода и выходят за пределы возможностей встроенных командлетов `Format-`.

В начале главы мы привели фрагмент кода из предыдущей главы, указав на необходимость улучшить его вид по умолчанию. Затем вы узнали, как создать базовое представление с помощью системы форматирования PowerShell на основе существующих определений представлений для нативных команд. В процесс входили трактовка и изменение XML-файлов для определения пользовательских представлений.

Вот ключевые этапы:

- мы описали базовые представления для нативных команд PowerShell;
- изменили файлы форматов, чтобы определить пользовательские представления конкретных типов объектов;
- создали манифест модуля, в котором можно указать представление формата пользовательского модуля PowerShell;
- показали весь процесс с помощью упражнения, в котором был приведен скрипт PowerShell для извлечения информации о машине.

Таким образом, в главе была представлена ценная информация по улучшению визуального представления вывода скрипта в PowerShell. Изучив ее, вы познакомились с возможностями создания более качественных и профессиональных инструментов.

25 Использование .NET Framework

По мере изучения возможностей, предоставляемых PowerShell, вы неизбежно столкнетесь с ситуациями, когда не будет встроенного командлета, который необходим для решения задачи. Во многих случаях вы можете обнаружить, что решить ее можно с помощью платформы .NET Framework, внешней команды, старого объекта объектной модели компонентов (component object model, COM) или чего-то другого. Можете ли вы использовать в скриптах необработанные компоненты .NET? Как бы это ни прозвучало, но ответом будет: и да, и нет.

25.1. ПОЧЕМУ ПОЯВИЛСЯ POWERSHELL

Чтобы лучше это понять, мы порассуждаем на тему того, почему вообще существует PowerShell. Microsoft Windows предлагает множество инструментов, предназначенных для упрощения автоматизации. Это свойственно самой природе компьютеров. Проблема Windows всегда заключалась в том, что эти возможности автоматизации предназначены для профессиональных разработчиков ПО и могут быть сложными для администраторов, у которых нет богатого опыта программирования или достаточного количества времени.

Вы можете эффективно автоматизировать Windows, если хорошо знаете C++, C# и прочие основные языки программирования этой ОС. Тем не менее если у вас нет знаний, касающихся указанных низкоуровневых языков либо их API, или времени для их освоения, то возникают трудности.

PowerShell появился не для того, чтобы добавить в Windows новые возможности автоматизации. Его задача — предоставить удобные для администраторов средства использования того, что уже существует. Выполняя командлет PowerShell, такой как `Get-Process`, вы выполняете не новый код, изобретенный кем-то в Microsoft.

Внутри этого командлета вы найдете фундаментальные ссылки на .NET Framework, написанные на C#. По сути, разработчик на C# выступил для вас в качестве переводчика. Вы выполняете командлет PowerShell, который переводится в понятный для Windows код C# и .NET Framework.

Иными словами, PowerShell является своеобразным переводчиком — оберткой. Его командлеты выполняются поверх .NET Framework, общей информационной модели (CIM), объектной модели компонентов (COM) и прочих Windows API. Такой подход обеспечивает более согласованный пользовательский опыт: имена командлетов следуют единому соглашению об именовании, получают ввод через параметры и т. д. Вам не нужно знать тысячи Windows API или несколько языков программирования, необходимых для доступа к ним. PowerShell выполняет перевод за вас, благодаря разработчикам, которые прописали его командлеты.

Так допустимо ли использование необработанных компонентов .NET Framework в скриптах? В их исходной форме — нет. Тем не менее разработчик вполне может создать обертки для этой функциональности .NET Framework. Вместо того чтобы интегрировать в скрипт произвольный C#-подобный код .NET, вы будете создавать командлеты, которые представят .NET в виде стандартной команды PowerShell. Об этом мы и поговорим в текущей главе.

ПРИМЕЧАНИЕ

Эта тема в последнее время стала менее актуальной, так как в Microsoft постоянно создают все новые командлеты. В прошлом мы часто использовали в качестве примера сервер доменных имен (domain name server, DNS), но сегодня у нас в PowerShell есть множество командлетов, предназначенных для этого сервера. Просим проявить терпение, если наш пример покажется вам слишком легким или недостаточно реалистичным. Наша задача — на пальцах показать вам, как создаются пользовательские командлеты PowerShell. А этот навык будет актуальным всегда.

25.1.1. Экспресс-знакомство с .NET

Если вы собираетесь использовать .NET, то вам нужно знать некоторые термины, иначе будет сложно разобраться в документации.

- *Тип* — это определение программного элемента. Данное слово встречается в PowerShell повсеместно — например, при выполнении `Get-Member` вы видите *имя типа* или нечто иное, что передали в `Get-Member`.
- *Класс* — это разновидность типа. Он представляет собой определение элемента функционирующего программного обеспечения и описывает, как нужно взаимодействовать с ПО. Но при этом является просто определением. Например, `System.Diagnostics.Process` — имя типа класса, которое описывает выполняющиеся в Windows процессы.

- *Экземпляр* — это конкретная реализация класса. Например, локальный процесс Local Security Authority Subsystem Service (LSASS) представляется экземпляром `System.Diagnostics.Process`. В большинстве случаев для взаимодействия с ним нужно иметь *экземпляр* класса. Например, вы не можете закрыть процесс, если не будет его конкретного экземпляра.
- Некоторые классы *абстрактны*, то есть для взаимодействия не требуют конкретного экземпляра. Например, класс `Math` в .NET абстрактен, поэтому для вычисления тангенсов и косинусов вам не нужно создавать его экземпляры.
- Классы состоят из *членов*. Это элементы, которые формируют определение класса, и именно отсюда берет свое имя `Get-Member`. Существует ряд распространенных членов.
  - *Свойства* описывают, что представляет класс, например имя процесса или статус сервиса. В одних случаях свойства допускают только чтение, в других вы можете их изменять. Например, вам может быть доступно изменение имени сервиса, но вы не можете скорректировать свойство `Status`, чтобы изменить информацию о том, выполняется ли этот сервис.
  - *Методы* выполняют действия. Метод может завершить процесс или запустить сервис. Иногда методы получают *аргументы*, подобные параметрам команд. Перезапуск компьютера, например, может позволить вам указать принудительную перезагрузку или отключение.
  - *События* срабатывают, когда что-то происходит с экземпляром, например, сервис завершает запуск. И хотя PowerShell не слишком хорошо подходит для событийно-управляемого программирования, вы можете в каком-то смысле подписываться на эти события, что позволяет выполнять код при их возникновении.

Первый серьезный вопрос по работе с .NET обычно звучит так: «Как найти часть .NET, которая выполнит нужную задачу?» Мы не знаем. Мы активно используем Google. Платформа .NET огромна. Вы можете назвать огромным расстояние до какого-нибудь гипермаркета, но это ничто в сравнении с масштабом .NET. И половина продуктов, которые продает Microsoft, прибавляется к .NET. Так что да, используйте Google.

Как только вы решите, что нашли нужную часть .NET, вам нужно найти ее документацию. Обычно она доступна на сайте Microsoft. Запросите в Google имя класса. Например, введите в строке поиска `System.Diagnostics.Process` — и найдете страницу наподобие http://mng.bz/G9PN. Эти страницы зависят от версии, поэтому вам нужно убедиться, что вы выбираете (из выпадающего меню в верхней части страницы) правильную версию .NET Framework. Кроме того, к моменту выхода книги URL, который мы указали чуть выше, может перестать существовать — в Microsoft это обычное дело. Поэтому мы и гуглим.

25.2. ИЗУЧЕНИЕ КЛАССА

Универсальность PowerShell проявляется в способности этой оболочки функционировать как окно непосредственной отладки для .NET, позволяя вам экспериментировать с кодом .NET в реальном времени. Разберем некоторые практические примеры.

Например, вы можете использовать класс `Math` из .NET для математических операций:

```
[Math]::Abs(-5)
```

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Попробуйте проделать это на своей рабочей станции.

В этом примере используется доступный в .NET класс `Math`, который состоит полностью из статических членов.

- Квадратные скобки `[]` используются в PowerShell для идентификации типов. Помещая имя типа в эти скобки, вы просите PowerShell искать соответствующий тип (в данном случае класс) в .NET. Это равнозначно объявлению переменной как `[string]` — в этом случае вы обращаетесь к классу `System.String`.
- Двойное двоеточие `::` обозначает статические члены класса. Оно всегда используется с `[classname]`, поскольку вы не создаете экземпляр класса. Иными словами, вы бы не указывали двойное двоеточие с экземпляром, сохраненным в переменной (как в `$myobject::method`).
- `Abs()` — статический метод класса `Math`, который мы нашли в Microsoft Developer Network (MSDN). Он возвращает абсолютное значение предоставляемого вами ввода.

Сделаем кое-что более сложное и при этом веселое: заставим ваш компьютер говорить (благодарим за это предложение Марка Минаси). Проверьте, включен ли звук, задайте уровень громкости 11 % и следуйте инструкциям.

Мы обратились к поиску Google с запросом `.NET speech synthesis` (синтез речи .NET) и нашли такой ресурс: http://mng.bz/z0yZ. Здесь задокументировано *пространство имен* `System.Speech.Synthesis`. Иначе говоря, `System.Speech.Synthesis` не является именем типа (то есть это не имя класса). Это верхний уровень имени нескольких типов (в том числе классов). В начале страницы документации перечисляются классы, которые подпадают под это пространство имен. К прочим типам относятся *перечисления*, являющиеся структурами, которые определяют различные доступные входные аргументы (и присваивают этим аргументам более простые для запоминания имена, а не числа). Благодаря примечаниям, приведенным ближе к концу страницы, вы можете получить базовое представление о том, как использовать классы в этом пространстве имен.

Судя по этим примечаниям, нужным нам классом является `System.Speech.Synthesis.SpeechSynthesizer`, поэтому мы перейдем на его страницу документации: http://mng.bz/0lPz.

ПРИМЕЧАНИЕ

В Microsoft иногда реорганизуют документацию. Так что, если приведенные URL не работают, не волнуйтесь. Ищите имя класса в Google — и найдете нужную документацию, где бы она ни находилась.

В частности, здесь интересно то, что ни один из этих методов (напомним: методы выполняют действия, а нас интересует именно это, в связи с чем мы их и рассматриваем) не статичен. Это можно понять по тому, что ни один из них не имеет значка *S*, который в Microsoft используют для обозначения статических членов. Статических методов нет, поэтому потребуется создать конкретный экземпляр класса, предоставляющий доступ к этим методам. Экземпляр класса можно создать с помощью специального метода под названием *«конструктор»*. У многих классов есть множество конструкторов, зачастую получающих входные аргументы, сообщающие новому экземпляру, как себя создавать. В этом случае класс приведен со всего одним конструктором, который не содержит входных аргументов, поэтому все должно быть просто:

```
PS C:\> $talk = new-object system.speech.synthesis.speechsynthesizer
new-object : Cannot find type
[system.speech.synthesis.speechsynthesizer]: verify that the
assembly containing this type is loaded.
At line:1 char:9
+ $talk = new-object system.speech.synthesis.speechsynthesizer
+         ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidType: (:) [New-Object], PSArgum
   entException
    + FullyQualifiedErrorId : TypeNotFound,Microsoft.PowerShell.Comm
   ands.NewObjectCommand
```

Что ж, управление данными — непростое занятие. Мы догадываемся, что PowerShell наверняка не загружает автоматически часть `Speech` пространства имен `System`. Почему вообще она должна это делать? Чтобы получить нужную часть .NET, нам потребуется загрузить эту сборку вручную. В начале документации говорится, что эта сборка находится в `System.Speech.dll`:

```
PS C:\> Add-Type -AssemblyName System.Speech
PS C:\> $talk = new-object system.speech.synthesis.speechsynthesizer
```

Важно указать параметр `-AssemblyName` и опустить расширение файла `.dll`. Это должно сработать для любой основной части .NET, которая является частью глобального кэша сборок (Global Assembly Cache, GAC); .NET знает, как найти корректный физический файл. И, как видите, теперь у нас есть переменная `$talk` с нашим экземпляром `SpeechSynthesizer` и мы можем заставить ее говорить.

В документации приводится несколько методов `Speak()`, каждый из которых получает свой тип входного аргумента. Они называются *перегрузками*. В .NET у вас может быть несколько методов с одним именем при условии, что каждый получает уникальную комбинацию входных аргументов. Похоже, одна перегрузка получает строку, значит, у нас должно получиться выполнить следующий код:

```
PS C:\> $talk.speak('PowerShell to the rescue!')
```

Сработало! С этого момента мы можем начать экспериментировать с другими методами и свойствами экземпляра, чтобы понять, как они действуют.

25.3. СОЗДАНИЕ ОБЕРТКИ

Мы еще не закончили. Напомним, что приведенный выше код .NET выглядит некрасиво и мы хотим сделать его более аккуратным, как должен выглядеть любой код PowerShell. Для этого напишем обертку. Ознакомьтесь с листингом 25.1, в котором содержится вызов к новой функции, что позволяет тестировать ее.

Листинг 25.1. Обертка для синтезатора речи

```
function Invoke-Speech {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$true,
                   ValueFromPipeline=$true)]
        [string[]]$Text
    )
    BEGIN {
        Add-Type -AssemblyName System.Speech
        $speech = New-Object -TypeName
System.Speech.Synthesis.SpeechSynthesizer
    }
    PROCESS {
        foreach ($phrase in $text) {
            $speech.speak($phrase)
        }
    }
    END {}
}
"One","Two","Three" | Invoke-Speech
```

Команда, получающая ввод из конвейера

Один раз загружает сборку

Обсудим несколько моментов.

- Мы пытались максимально придерживаться нативных паттернов PowerShell. Например, эта функция получает ввод из конвейера, и мы используем данный прием в тестовом вызове.
- В режиме конвейера нет причины раз за разом добавлять сборку и создавать экземпляр синтезатора, так что это действие выполняется в блоке `Begin`.
- Когда `$speech` выходит из области видимости, синтезатор автоматически прекращает свое существование, поэтому удалять объект в блоке `End` необязательно. Аналогичным образом мы не считаем необходимым выгружать сборку (она никому не мешает и не занимает память), так что этого мы тоже не делаем.

Но это не идеальное решение. Экспериментируя с объектом `speech`, мы заметили, что он содержит метод `Speak()` для реализации *синхронной* речи (то есть на время произнесения речи скрипт будет приостанавливаться) и метод `SpeakAsync()`, который запускает речь и позволяет скрипту продолжиться. Мы видим, что используются обе модели, поэтому хотим добавить их в качестве опций для пользователей нашей команды-обертки. В листинге 25.2 представлен новый код.

Листинг 25.2. Добавление поддержки SpeakAsync()

```
function Invoke-Speech {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$true,
                   ValueFromPipeline=$true)]
        [string[]]$Text,
        [switch]$Asynchronous
    )
    BEGIN {
        Add-Type -AssemblyName System.Speech
        $speech = New-Object -TypeName
  System.Speech.Synthesis.SpeechSynthesizer
    }
    PROCESS {
        foreach ($phrase in $text) {
            if ($Asynchronous) {
                $speech.SpeakAsync($phrase)
            } else {
                $speech.speak($phrase)
            }
        }
    }
    END {}
}
1..10 | Invoke-Speech -Asynchronous
Write-Host "This appears"
```

Добавляет новый параметр

Вызывает SpeakAsync(), если используется новый параметр

В противном случае использует метод Speak()

`This appears` будет отображаться до любого другого вывода:

```
This appears
IsCompleted
-----------
      False
      False
      False
      False
      False
      False
      False
      False
      False
      False
```

Выглядит не очень. Если снова почитать документацию, то оказывается, что `SpeakAsync()` возвращает объект, указывающий, завершена ли речь. Нас это не волнует, поэтому нужно заглушить вывод. В листинге 25.3 представлена наша заключительная попытка.

Листинг 25.3. Заглушаем вывод SpeakAsync()

```
function Invoke-Speech {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$true,
                   ValueFromPipeline=$true)]
        [string[]]$Text,
        [switch]$Asynchronous
    )
    BEGIN {
        Add-Type -AssemblyName System.Speech
        $speech = New-Object -TypeName
        System.Speech.Synthesis.SpeechSynthesizer
    }
    PROCESS {
        foreach ($phrase in $text) {
            if ($Asynchronous) {
                $null = $speech.SpeakAsync($phrase)
            } else {
                $speech.speak($phrase)
            }
        }
    }
    END {}
}
1..10 | Invoke-Speech -Asynchronous
Write-Host "This appears"
```

$null слева, чтобы заглушить вывод

СДЕЛАЙТЕ ЭТО СЕЙЧАС

Попробуйте выполнить все эти действия. Это весело! Затем посетите http://mng.bz/YRa7, где показана более сложная версия нашей оболочки, с которой вам наверняка понравится экспериментировать.

Обертывание этого небольшого фрагмента кода может показаться пустой тратой времени, но по факту это вложение. Вы получаете несколько выгод.

- Никакому другому участнику вашей команды не потребуется снова «препарировать» этот объект — он сможет использовать вашу простую, PowerShell-совместимую команду. Мы, естественно, добавим справочную информацию, чтобы эта команда стала еще более нативной для PowerShell.
- Если вы начнете заниматься модульным тестированием, то *не* сможете имитировать код .NET — но поскольку вы написали обертку, то при необходимости *сможете* имитировать вызовы к `Invoke-Speech`.

- Документация (если вы уделите время добавлению справочной информации хотя бы в виде комментариев) встроена, поэтому не требуется искать что-то в Google или MSDN.

25.4. БОЛЕЕ ПРИКЛАДНОЙ ПРИМЕР

Приведем пример, который вы можете использовать в контроллере. Предположим, вы хотите предоставить для скрипта графический блок ввода. Мы создавали его в VBScript, и эта функциональность по-прежнему доступна в той части .NET Framework, которая посвящена Visual Basic. Для начала вам нужно добавить сборку:

```
Add-Type -AssemblyName "microsoft.visualbasic"
```

Класс `[microsoft.visualbasic.interaction]` содержит статический метод `InputBox()`, который получает три аргумента в таком порядке: запрос к пользователю, заголовок и предустановленный выбор. Выполните этот код, чтобы создать окно ввода, аналогичное показанному на рис. 25.1:

```
[microsoft.visualbasic.interaction]::inputbox ("Enter a server name",
"PSServer Management",$null)
```

Рис. 25.1. Окно ввода

Пользователь вводит значение и нажимает кнопку OK, после чего это значение записывается в конвейер. Вам понадобится добавить обработку ошибок и валидацию на случай, если пользователь не введет ничего или нажмет кнопку Cancel. Если вы предполагаете, что такое будет происходить часто, то можете создать специальную функцию. Например, в листинге 25.4 показан наш вариант короткой функции.

Листинг 25.4. Быстрая и простая версия обертки InputBox

```
function Invoke-InputBox {
  [CmdletBinding()]
  Param(
    [Parameter(Mandatory=$True)]
    [string]$Prompt,
    [Parameter(Mandatory=$True)]
```

```
    [string]$Title,
    [Parameter()]
    [string]$Default = ''
  )
  Add-Type -Assembly Microsoft.VisualBasic
  [microsoft.visualbasic.interaction]::inputbox($prompt,$title,$default)
} #function
```

Данный код показывает, насколько маленькой может быть обертка, как легко ее создавать и насколько сильно обертки могут упрощать использование .NET.

25.5. УПРАЖНЕНИЯ

Это настолько важная задача, что вы обязательно должны попробовать реализовать ее.

25.5.1. Вводная информация

Класс `System.Net.Dns` содержит статический метод `GetHostByAddress()`. Он запрограммирован искать имя хоста по его IP-адресу. Найдите этот класс онлайн и поэкспериментируйте с ним в оболочке.

25.5.2. Ваша задача

Напишите функцию-обертку `Get-DnsHostByAddress`. Она должна получать один или несколько IP-адресов и для каждого генерировать объект, содержащий IP-адрес и соответствующее имя хоста. Если же доступных имен хоста нет, вместо него должен возвращаться `null`.

25.5.3. Наше решение

Поэкспериментировав в командной строке, мы поняли, что метод возвращает объект с тремя свойствами: `HostName` (что хорошо), `Aliases` (что может быть интересно) и `AddressList`, который, похоже, является массивом. Мы решили не усложнять обертку и сосредоточиться в ней только на `HostName` (листинг 25.5).

Листинг 25.5. Наша обертка для поиска имен хостов DNS

```
function Get-DnsHostByAddress {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$true,
                   ValueFromPipeline=$true)]
        [string[]]$Address
    )
    BEGIN {}
```

```
    PROCESS {
        ForEach ($Addr in $Address) {
            $props = @{'Address'=$addr}
            Try {
                $result = [System.Net.Dns]::GetHostByAddress($addr)
                $props.Add('HostName',$result.HostName)
            } Catch {
                $props.Add('HostName',$null)
            }
            New-Object -TypeName PSObject -Property $props
        } #foreach
    } #PROCESS
    END {}
} #function
Get-DnsHostByAddress -Address '204.79.197.200','192.168.254.254',
 '35.166.24.88'
```

Здесь нужно отметить несколько моментов.

- Мы протестировали оба легитимных IP-адреса (привет, Bing.com!), а также один нерабочий, поскольку в каждой ситуации возвращаем другой вывод.
- При нормальной обработке ошибок в командах нам бы потребовалось указать `-ErrorAction`, чтобы реализовать перехватываемое исключение. Но методы .NET работают иначе: в случае ошибки они всегда выдают перехватываемое исключение, поэтому наш блок `Try` работает прекрасно.
- Если хосты дают сбой, то вы можете выбрать не `$null`, а `Unknown` или другое значение. Нам нравится `$null`, поэтому мы использовали его.
- Мы сразу завели хеш-таблицу для конечных свойств выводимого объекта. Затем в зависимости от результата запроса добавили свойство `HostName`. Нам нравится этот прием — он позволяет динамически создавать вывод по одному элементу и затем передавать его весь в конвейер в виде объекта.

ИТОГИ ГЛАВЫ

В этой главе была представлена информация по интеграции PowerShell с .NET Framework. Мы начали с обсуждения причин появления PowerShell, которая создана для предоставления удобного пользовательского интерфейса, позволяющего задействовать доступные в Windows возможности автоматизации. Реализует все это PowerShell, выступая переводчиком или оберткой вокруг различных Windows API, таких как .NET Framework, COM и CIM.

Затем мы разобрали основы .NET, дав определение ее ключевых концепций, таких как типы, классы, экземпляры и члены. Благодаря этому вы увидели масштаб экосистемы .NET и поняли, насколько важно для эффективной навигации использовать онлайн-ресурсы наподобие документации Microsoft.

Далее мы показали, как PowerShell может взаимодействовать с классами .NET в реальном времени, позволяя вам выполнять такие задачи, как математические операции и синтез текста в речь. Мы разобрали процесс использования классов .NET внутри скриптов PowerShell и обратили ваше внимание на то, что создание оберток PowerShell для функциональности .NET позволяет повысить юзабилити .NET и сделать его более удобным в сопровождении.

Завершилась глава практическими примерами по созданию оберток PowerShell для функциональности .NET: обертки для синтеза речи и функции окна для ввода. Мы порекомендовали вам самостоятельно поэкспериментировать с интеграцией .NET в скрипты PowerShell и дали упражнение для закрепления этих знаний.

Таким образом, глава 25 послужила ценным руководством по использованию всех возможностей .NET Framework в скриптах PowerShell, позволив расширить функционал PowerShell и эффективно упростить задачи по автоматизации.

26 Хранение данных: только не в Excel!

PowerShell предлагает возможность динамически генерировать и изменять документы Excel. Но сам факт наличия такой возможности еще не говорит о том, что это правильный подход. Программа Excel не ориентирована на работу в качестве базы данных, и очень печально видеть, как некоторые разработчики все же используют ее для этих целей. Разработка скриптов, которые взаимодействуют с Excel через PowerShell, ведет к необходимости использовать компоненты Microsoft Office Programmability, которые интегрируются в .NET при установке Office. Эти компоненты, в свою очередь, используют интерфейс объектной модели компонентов (COM), которую в Microsoft давно не обновляли. Печально видеть, как администраторы создают скрипты, которые содержат много связанного с Excel кода, что приводит к большим затратам времени и непродуктивному опыту. Мы настоятельно советуем избегать этого подхода. Тем не менее потребность в хранении данных неизбежно возникнет. В подобных случаях есть более подходящее альтернативное решение.

26.1. ОБЗОР SQL SERVER

Вы наверняка слышали о Microsoft SQL Server. Если у вас в среде он установлен, то посмотрите, удастся ли вам настроить небольшую базу данных для его использования. Его не нужно будет загружать параллельно с работой, и это не будет вам ничего стоить. Либо на крайний случай установите бесплатный инструмент SQL Server Express (релиз от 2022 года можно найти по ссылке http://mng.bz/K9Pn, но вы можете использовать любую версию). Мы рекомендуем скачать его вместе с Advanced Services (хотя название в зависимости от версии может различаться), который содержит Reporting Services. Кроме того, мы рекомендуем скачать SQL Server Management Studio (SSMS). Честно говоря, вряд ли есть смысл давать вам прямую ссылку для скачивания, поскольку в Microsoft этот адрес часто

меняется. Вам будет проще найти актуальный ресурс в Google по запросу SQL Server Management Studio download.

ПРИМЕЧАНИЕ

Мы не хотим в этой главе слишком подробно рассказывать о том, что такое SQL Server или как с ним работать. Если вам нужен стартовый материал, то рекомендуем прочитать книгу *Learn SQL Server Administration in a Month of Lunches* (Manning, 2014; http://mng.bz/9QP8). Вдобавок у Manning есть «видеокнига» *SQL in Motion* Бена Брумма (Ben Brumm) (2017, www.manning.com/livevideo/sql-in-motion). Еще одна книга — *Learn dbatools in a Month of Lunches* (Manning, 2022, http://mng.bz/VRJ5).

Ниже мы описали некоторые из преимуществ использования SQL Server (или, честно говоря, любой системы управления реляционной базой данных — если вы предпочтете использовать такую вместо SQL Server, то все основные приемы, представленные в текущей главе, тоже будут работать).

- Базы данных упрощают добавление, удаление, обновление и запрос данных, делая все эти задачи *очень* простыми.
- SQL Server Reporting Services (SSRS) может создавать красивые отчеты, которые вы создаете в дружественной среде по принципу перетаскивания. Non-Express Reporting Services может запускаться и доставлять за вас эти отчеты по расписанию.
- PowerShell *прекрасно* работает с SQL Server (и прочими базами данных).

Вам потребуется лишь освоить некоторые термины и несколько принципов.

- Естественно, вы подключаетесь не только к серверу, но и к конкретной базе данных (БД). Существует определенная база под названием *мастер*, к которой вы подключаетесь, когда хотите создать новую базу данных.
- Подключение реализуется путем указания *строки подключения*, по сути, точки связи с базой данных. Эта строка содержит и аутентификационную информацию.
- База данных состоит из *таблиц*, каждая из которых примерно аналогична таблице Excel. Следовательно, вы можете рассматривать БД как рабочий журнал Excel.
- Таблица в БД, как и в Excel, состоит из *строк* и *столбцов*. Профессионалы по базам данных порой называют их *сущностями* и *доменами*.

26.2. НАСТРОЙКА СЕРВЕРА И БАЗЫ ДАННЫХ

Честно говоря, настройку одноразового сервера и базы данных дольше объяснять и реализовывать, чем в итоге использовать эти инструменты. Во-первых, как уже говорилось, мы предполагаем, что вы установили SQL Server Express. Мы используем версию 2016 года и выполнили базовую установку (которая не предполагает ничего дополнительного). Последующие выпуски устанавливаются практически

так же, и в случае возникающих запросов вы можете принять все сопутствующие предустановки. Если вы используете SQL Server в сети, то попросите, чтобы администратор выдал вам *имя сервера* и, если таковой имеется, *имя экземпляра*.

ПРИМЕЧАНИЕ

Какой бы пользовательский аккаунт вы ни использовали для установки, обычно SQL Server Express будет настроен как Administrator экземпляра SQL Server Express. Это верно независимо от того, находитесь вы в среде домена или нет.

Во-вторых, вам нужна база данных. Если вы используете SQL Server в сети, то администратор должен создать базу данных (будет достаточно объема в 2–3 Гбайт; скажите, что пока будет достаточно модели Simple Recovery). Администратор должен будет сообщить вам имя этой базы данных и дать знать, можете ли вы подключаться к ней со своими учетными данными Windows или же нужно использовать другие имя пользователя и пароль.

Если вы установили SQL Server Express локально и использовали все базовые настройки, то у вас установился экземпляр SQLEXPRESS. Выполните скрипт PowerShell из листинга 26.1, чтобы создать базу данных Scripting. Для подключения к ней используйте свои учетные данные Windows. Эта база будет иметь минимальный предустановленный размер (обычно около 2 Гбайт). Обратите внимание, что этот вариант не подходит для производственной среды, поскольку есть несколько настроек БД, которые обычно требуется установить, и вам не нужно организовывать резервные копии. Подробнее об этих задачах читайте в книге *Learn SQL Server Administration in a Month of Lunches* (http://mng.bz/9QP8).

Листинг 26.1. Настройка базы данных в экземпляре SQL Server Express

```
$conn_string =
"Server=localhost\SQLEXPRESS;Database=master;Trusted_Connection=True;"
$conn = New-Object System.Data.SqlClient.SqlConnection
$conn.ConnectionString = $conn_string
$conn.Open()
$sql = @"
CREATE DATABASE Scripting;
"@
$cmd = New-Object System.Data.SqlClient.SqlCommand
$cmd.CommandText = $sql
$cmd.Connection = $conn
$cmd.ExecuteNonQuery()
$conn.close()
```

Определяет строку подключения

Создает объект подключения

Настраивает объект подключения

Определяет SQL-запрос

Создает SQL-объект Command

Настраивает команду для использования запроса

Настраивает команду для использования подключения

Выполняет команду

Закрывает соединение с базой данных

Далее вам потребуется строка подключения, но у вас уже должно быть все необходимое. Вот наша:

```
Server=localhost\SQLEXPRESS;Database=master;Trusted_Connection=True;
```

Как видите, мы использовали ее в коде для создания новой базы данных. Когда мы будем готовы к ее использованию, то изменим в ней `master` на `Scripting`. Та же строка подключения работает для любой базы данных, в которой вы можете использовать для авторизации свои учетные данные Windows. Если же вам нужно указать имя пользователя и пароль, то строка будет выглядеть так:

```
Server=localhost\SQLEXPRESS;Database=master;un=xxxxx;pw=yyyyyy;
```

Здесь `xxxxx` и `yyyyy` — это ваши имя пользователя и пароль для SQL Server соответственно.

СОВЕТ

Мы создаем строки подключения с помощью очень полезного ресурса ConnectionStrings.com. Зачем помнить все это, когда можно просто найти?

26.3. ИСПОЛЬЗОВАНИЕ БАЗЫ ДАННЫХ: СОЗДАНИЕ ТАБЛИЦЫ

Сначала вам нужно решить, что вы будете помещать в базу данных. И это не разовое решение. Как и в случае с Excel, вы можете в любое время добавлять и удалять таблицы, а также изменять их столбцы. Начнем с команды в листинге 26.2. Как и большинство из написанных нами команд, эта производит на выходе объект, поэтому является отличной отправной точкой (и да, мы использовали эту команду ранее).

СОВЕТ

В целях разработки и тестирования сохраните этот скрипт в виде модуля. Вы также будете добавлять в этот файл .psm1 дополнительные команды, сохраняя всю информацию в опрятном сгруппированном виде.

Листинг 26.2. Начнем с команды, которая на выходе создает объект

```
function Get-DiskInfo {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True)]
        [string[]]$ComputerName
    )
    BEGIN {
        Set-StrictMode -Version 2.0
    }
    PROCESS {
        ForEach ($comp in $ComputerName) {
            $params = @{'ComputerName' = $comp
                        'ClassName' = 'Win32_LogicalDisk'}
```

```
            $disks = Get-CimInstance @params
            ForEach ($disk in $disks) {
                $props = @{'ComputerName' = $comp
                           'Size' = $disk.size
                           'Drive' = $disk.deviceid
                           'FreeSpace' = $disk.freespace
                           'DriveType' = $disk.drivetype}
                New-Object -TypeName PSObject -Property $props
            } #foreach disk
        } #foreach computer
    } #PROCESS
    END {}
}
```

При проверке команды выводится следующая информация:

- *имя компьютера* — строка;
- *размер диска* — большое целое число;
- *тип диска* — небольшое число (из одной цифры);
- *свободное место на диске* — большое целое число;
- *ID диска* — строка.

В связи с этим вам нужно создать таблицу, которая может содержать эту информацию. Кроме того, вы добавите поле для отслеживания даты добавления в нее каждой строки. Таким образом вы сможете периодически просматривать информацию о дисках и выстраивать линию изменения свободного пространства. (Мы бы использовали для составления отчета об изменении SSRS; эта тема выходит за рамки книги, но на сайте PowerShell.org есть бесплатная электронная публикация, в которой данную тему можно изучить самостоятельно.) В листинге 26.3 показано, что мы добавляем в наш файл `.psm1` (полноценная версия этого листинга доступна для скачивания по ссылке http://mng.bz/rjgE; здесь мы экономим место в книге, показывая только дополнительный код). Значительная часть кода должна уже казаться вам знакомой, поскольку мы использовали его ранее.

Листинг 26.3. Добавление кода для создания таблицы

```
function New-DiskInfoSQLTable {
    [CmdletBinding()]
    param()
    $conn = New-Object System.Data.SqlClient.SqlConnection
    $conn.ConnectionString = $DiskInfoSqlConnection
    $conn.Open()
    $sql = @"
        IF NOT EXISTS (SELECT * FROM sysobjects WHERE name='diskinfo' AND xtype='U')
            CREATE TABLE diskinfo (
                ComputerName VARCHAR(64),
                DiskSize BIGINT,
                DriveType TINYINT,
```

```
                FreeSpace BIGINT,
                DriveID CHAR(2),
DateAdded DATETIME2
            )
"@
    $cmd = New-Object System.Data.SqlClient.SqlCommand
    $cmd.Connection = $conn
    $cmd.CommandText = $sql
    $cmd.ExecuteNonQuery() | Out-Null
    $conn.Close()
}
$DiskInfoSqlConnection =
"Server=localhost\SQLEXPRESS;Database=Scripting;Trusted_Connection=True;"
Export-ModuleMember -Function Get-DiskInfo
Export-ModuleMember -Variable DiskInfoSqlConnection
```

Отметим, что мы добавили вне каждой функции переменную уровня модуля, которая будет содержать строку подключения к базе данных. Это упрощает повторное использование данной информации во множестве функций. Вы явно экспортируете эту переменную вместе с первой функцией, чтобы при загрузке модуля все добавлялось в глобальную область видимости оболочки. Аналогичным образом если модуль выгружается, то данные неизбежно удаляются из этой области. Почему бы вам не экспортировать эту функцию создания таблицы? Нет никаких причин кому-то выполнять ее вне модуля, поэтому, отказываясь от экспорта функции, вы делаете ее приватной для него.

Эта новая команда делает то, что мы считаем интересным приемом: сначала она проверяет, существует ли таблица. Если нет, то команда ее создает. Таким образом вы можете раз за разом вызывать эту команду, и она всегда будет гарантировать существование таблицы.

Теперь разберем процесс, используемый данным кодом, поскольку вы увидите его еще не раз.

1. Создайте объект `System.Data.SqlClient.SqlConnection`, представляющий подключение к SQL Server. Установите его свойство на строку подключения и вызовите метод `Open()`. Если строка подключения окажется ошибочной, то сгенерируется ошибка. Кроме того, в конце команды вы заполняете вызов к методу `Close()`.
2. Создайте запрос в конструкции here-string, в основном для того, чтобы его можно было красиво отформатировать. Здесь для этой строки применяются двойные кавычки, поскольку одинарные SQL Server использует для разграничения строк. Двойные кавычки упрощают использование одиночных кавычек внутри here-string, а также вставку переменных и подвыражений. Указание запроса в переменной позволяет выводить его с помощью `Write-Verbose`, чтобы можно было легко перепроверить синтаксис запроса в случае ошибки.

3. Создайте новый `System.Data.SqlClient.SqlCommand` и установите его свойства `Connection` на открытый объект `Connection`. Установите свойство `CommandText` на запрос и попросите выполнить `ExecuteNonQuery()`. Этот метод используется, когда вы знаете, что ваш запрос не вернет никаких результатов. Он *будет* возвращать `-1` в случае успешного запроса, поэтому вы передаете его в `Out-Null`, чтобы заглушить.

Аналогичным образом те же два объекта вы будете использовать в последующих командах.

ПРИМЕЧАНИЕ

Если вы работаете не с SQL Server, то .NET содержит равнозначное пространство имен System.Data.OleDbClient вместе с классами OleDbConnection и OleDbCommand, которые позволяют подключиться к другим базам данных.

Вам может быть интересно, как мы придумали все эти типы данных для оператора `CREATE TABLE`. Все просто: мы их нашли. Поиск в Google по запросу SQL Server data types привел нас на ресурс http://mng.bz/j1l9, который оказался довольно полезным. В реальности мы используем всего несколько типов данных:

- `VARCHAR()` — позволяет указывать максимальную длину поля и занимает меньше места, если вы используете меньше этого максимума. `VARCHAR(MAX)` позволяет сохранить любое количество текста;
- `CHAR()` — создает текстовые столбцы фиксированной длины;
- `TINYINT` — содержит целые числа от 0 до 255;
- `BIGINT` — содержит целое число практически любого размера;
- `DATETIME2` — содержит значения даты/времени.

Кроме того, вам могут пригодиться типы `FLOAT` или `INT`, которые подробно описаны в документации SQL.

26.4. СОХРАНЕНИЕ ДАННЫХ НА SQL SERVER

Теперь вы готовы к созданию третьей команды, показанной в листинге 26.4. Она будет получать вывод команды информации о диске и экспортировать его в вашу таблицу SQL Server. Напомним, что доступная для скачивания версия этого кода содержит *весь* модуль скрипта.

Листинг 26.4. Добавление команды для экспорта данных в SQL Server

```
function Export-DiskInfoToSQL {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True,
```

```
                    ValueFromPipeline=$True)]
        [object[]]$DiskInfo
    )
    BEGIN {
        New-DiskInfoSQLTable
        $conn = New-Object System.Data.SqlClient.SqlConnection
        $conn.ConnectionString = $DiskInfoSqlConnection
        $conn.Open()
        $cmd = New-Object System.Data.SqlClient.SqlCommand
        $cmd.Connection = $conn
    }
    PROCESS {
        ForEach ($object in $DiskInfo) {
            if ($object.size -eq $null) {
                $size = 0
            } else {
                $size = $object.size
            }
            if ($object.freespace -eq $null) {
                $freespace = 0
            } else {
                $freespace = $object.freespace
            }
            $sql = @"
                INSERT INTO DiskInfo (ComputerName,
                    DiskSize,DriveType,FreeSpace,DriveID,DateAdded)
                VALUES('$($object.ComputerName)',
                       $size,
                       $($object.DriveType),
                       $freespace,
                       '$($object.Drive)',
                       '$(Get-Date)')
"@
            $cmd.CommandText = $sql
            Write-Verbose "EXECUTING QUERY `n $sql"
            $cmd.ExecuteNonQuery() | Out-Null
        } #ForEach
    } #PROCESS
    END {
        $conn.Close()
    }
}
```

ПРИМЕЧАНИЕ

Обратите внимание, что мы проверяем, не хранят ли Size и FreeSpace значение null. Это может произойти при использовании, например, оптических дисков. В таких ситуациях мы устанавливаем их равными 0, чтобы иметь корректное значение, которое можно добавить в базу данных.

Здесь следует обозначить один важный момент. Параметр `-DiskInfo` новой команды получает ввод из конвейера — но вы заметите, что получает он *что-либо*, поскольку

его типом данных является `System.Object`. Следовательно, можно передать в параметр служебный объект или нечто другое, а тот не поймет, что с ним делать. И этого не избежать. Да, вы можете изменить функцию `Get-DiskInfo`, чтобы добавить пользовательское имя типа, но это не позволит вам указать это имя как единственный допустимый ввод для `Export-DiskInfoToSQL`. К сожалению, PowerShell так не работает. Если вы хотите тесно связать эти две команды и сделать так, чтобы `Export-DiskInfoToSQL` была способна получать только объекты, созданные `Get-DiskInfo`, то вам потребуется создать собственный класс. В PowerShell v5 и более новых версиях такое возможно, но это уже более сложная тема, которая выходит за рамки нашей книги. (Она разбирается в книге *The PowerShell Scripting & Toolmaking Book* и может обновляться, поскольку эта книга доступна только онлайн [https://leanpub.com/powershell-scripting-toolmaking]. Ситуация с классами в PowerShell постоянно меняется.) На данный момент вам следует знать, что вы должны сохранять осторожность при использовании `Export-DiskInfoToSQL`. Взглянем на наш код в листинге 26.5.

Листинг 26.5. Добавление проверок членов для входных объектов

```
function Export-DiskInfoToSQL {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True)]
        [object[]]$DiskInfo
    )
    BEGIN {
        New-DiskInfoSQLTable
        $conn = New-Object System.Data.SqlClient.SqlConnection
        $conn.ConnectionString = $DiskInfoSqlConnection
        $conn.Open()
        $cmd = New-Object System.Data.SqlClient.SqlCommand
        $cmd.Connection = $conn
        $checks = 0
    }
    PROCESS {
        if ($checks -eq 0) {
            $checks++
            $props = $DiskInfo[0] |
                     Get-Member -MemberType Properties |
                     Select-Object -Expand name
            if ($props -contains 'ComputerName' -and
                $props -contains 'Drive' -and
                $props -contains 'DriveType' -and
                $props -contains 'FreeSpace' -and
                $props -contains 'Size') {
                    Write-Verbose "Input object passes check"
                } else {
                    Write-Error "Illegal input object"
                    Break
                }
        }
```

Проверяет первый входной объект

```
        ForEach ($object in $DiskInfo) {
             if ($object.size -eq $null) {
                 $size = 0
             } else {
                 $size = $object.size
             }
             if ($object.freespace -eq $null) {
                 $freespace = 0
             } else {
                 $freespace = $object.freespace
             }
             $sql = @"
                 INSERT INTO DiskInfo (ComputerName,
                     DiskSize,DriveType,FreeSpace,DriveID,DateAdded)
                 VALUES('$($object.ComputerName)',
                         $size,
                         $($object.DriveType),
                         $freespace,
                         '$($object.Drive)',
                         '$(Get-Date)')
"@
             $cmd.CommandText = $sql
             Write-Verbose "EXECUTING QUERY `n $sql"
             $cmd.ExecuteNonQuery() | Out-Null
         } #ForEach
    } #PROCESS
    END {
        $conn.Close()
    }
}
```

Попробуйте передать какие-нибудь данные в БД:

```
get-diskinfo $env:ComputerName | Export-DiskInfoToSQL
```

ИЗБЕГАНИЕ АТАК ПУТЕМ SQL-ВНЕДРЕНИЯ

В листинге 26.5 мы не стали трогать то, что является табу для большинства публичных приложений: динамическое создание запроса путем внедрения содержимого переменных в строку. В производственных приложениях это чревато атаками под названием *SQL-внедрение*. Мы же от этого защищены, поскольку являемся единственными, кто использует эту базу данных. Но если вы отправляете данные другим людям, то вам стоит иметь в виду эту возможность и изучить соответствующую документацию.

Что вы *можете сделать*, так это создать проверки ввода команд. Код нашего листинга делает именно это. Мы решили убедиться, что передаваемые нами объекты имеют ожидаемые свойства. Это немного замедлит процесс на время проверки, поэтому мы проверяем только первый передаваемый объект и предполагаем, что все остальные аналогичны.

26.5. ЗАПРОС ДАННЫХ С SQL SERVER

Мы не думаем, что это действие будет иметь место в реальности, но все же будем планировать загрузить данные в SQL Server и оставить их там, чтобы серверная система создания отчетов (SQL Server Reporting Services, SSRS) генерировала на основе этих данных отчеты, — мы хотим показать вам пример запроса данных. В листинге 26.6 содержится заключительный фрагмент кода, который нужно добавить в модуль. Повторимся: мы рекомендуем использовать доступную для скачивания версию, поскольку в ней есть весь код целиком.

Листинг 26.6. Добавление команды для извлечения данных из SQL Server

```
function Import-DiskInfoFromSQL {
    [CmdletBinding()]
    Param()
        $conn = New-Object System.Data.SqlClient.SqlConnection
        $conn.ConnectionString = $DiskInfoSqlConnection
        $conn.Open()
        $cmd = New-Object System.Data.SqlClient.SqlCommand
        $cmd.Connection = $conn
        $sql = @"
            SELECT ComputerName,DiskSize,DriveType,FreeSpace,
            DriveID,DateAdded
            FROM DiskInfo
            ORDER BY DateAdded ASC
"@
        $cmd.CommandText = $sql
        $reader = $cmd.ExecuteReader()
        # прокрутите результаты
        while ($reader.read()) {
            $props = @{'ComputerName' = $reader['ComputerName']
                       'Size' = $reader['DiskSize']
                       'DriveType' = $reader['DriveType']
                       'FreeSpace' = $reader['FreeSpace']
                       'Drive' = $reader['DriveId']
                       'DateAdded' = $reader['DateAdded']}
            New-Object -TypeName PSObject -Property $props
        }
        $conn.Close()
}
```

Перебирает результаты и создает пользовательский объект

Снова обратите внимание на то, что вы следуете паттернам, которым мы учили вас в этой книге: создаете команду, которая использует параметры для своего ввода (и в данном случае переменную уровня модуля), создает в качестве вывода объект и т. д. Единственное, что мы опустили, исключительно из соображений экономии места в книге, — это справочные комментарии, которые обычно добавляем везде.

Кроме того, мы хотим признать, что не каждый напишет эту команду, как мы. Некоторые разработчики предпочитают использовать вместо `DataReader` объект `DataTable`, и мы не спорим, что `DataTable` конкретно в этом сценарии сработает быстрее. Мы же выбрали такой подход, поскольку он более обучающий

и процедурный. Данный объект считывает набор результатов по одной строке и выстраивает выводимые объекты по одному, подкрепляя паттерн, который мы рекомендовали в данной книге.

Наконец, если вы были внимательны, то заметите расхождения. Исходная `Get-DiskInfo` выводит объект со свойствами `Size` и `Drive`, а `Import- DiskInfoFromSQL` отображает их имена. Но в таблице SQL Server в качестве имен столбцов используются `DiskSize` и `DriveID`. Как возникает это несоответствие? С его помощью мы подчеркнули, что табличная структура не должна *в точности* соответствовать структуре объекта. В данном случае функции `Import` и `Export` переводят имена свойств в такие, которые используются в таблице. Это полезный прием, когда вам не нужно контролировать объект или структуру таблицы и вы хотите переключать все при сохранении и извлечении данных.

И, завершая круг, мы получим информацию, которую только что добавили:

```
PS C:\> Import-DiskInfoFromSQL
DateAdded    : 9/23/2023 5:24:01 PM
Drive        : C:
FreeSpace    : 27722903552
ComputerName : WIN11
DriveType    : 3
Size         : 206266429440
DateAdded    : 9/23/2023 5:24:01 PM
Drive        : D:
FreeSpace    : 16025034752
ComputerName : WIN11
DriveType    : 3
Size: 26843541504
```

ИТОГИ ГЛАВЫ

Надеемся, что, прочитав эту главу, вы увидели, насколько просто использовать в качестве базы данных SQL Server, а не что-то вроде Excel. Вы опробовали практики создания инструментов и создали набор команд, которые работают с информацией о диске. Кроме того, вы реализовали возможность автоматического создания отчетов с помощью SSRS на случай, если решите поручить составление отчетов ей. Используя запланированную задачу для выполнения инвентаризации и SSRS для периодического автоматического создания отчетов, вы можете полностью автоматизировать сбор данных и процесс составления отчетов, исключив себя из этого цикла и освободив время для работы в других направлениях. Помимо прочего, Microsoft предлагает замечательный модуль SQL Server. И не стоит забывать о DBATools.

27 Этому нет конца

Итак, наше путешествие в мире скриптинга подошло к концу. Или нет? Естественно, нет: вы только начинаете, хотя уже и можете проявить себя. Теперь пора начать думать о следующих шагах.

27.1. ДОБРО ПОЖАЛОВАТЬ В МИР СОЗДАНИЯ ИНСТРУМЕНТОВ

Мы рассчитываем, что к этому моменту вы хорошо поняли значение выражения *«создание инструментов»*. Оно подразумевает не просто скриптинг, а создание по правилам PowerShell небольших рабочих элементов, которые можно объединять. Кроме того, это понятие подразумевает создание *контроллеров*, которые помещают эти инструменты в конкретную ситуацию и контекст, давая им определенную цель на тот момент времени, но оставляя сами инструменты свободными, чтобы в другой раз их можно было использовать для другой цели. Надеемся, вы уже увидели, в чем ценность понимания того, как PowerShell выполняет действия нативно, а также почему вам нужно повторять эти подходы в своей работе.

Самый лучший отзыв, который мы получаем, когда объясняем этот материал (будь то в классе, на конференции или в такой книге, как эта), звучит примерно так: «Что ж, большое спасибо — теперь мне нужно переписать *все* свои скрипты!» И он нам нравится, поскольку показывает, что мы хорошо обучили человека и проделали прекрасную работу, показав ему всю ценность данного подхода. Естественно, это не означает, что теперь этот человек обязан переписать всю свою готовую работу. Если какие-то ваши скрипты работают, то пусть их код остается прежним. Но если вам потребуется исправить баг или добавить

функциональность, то начинайте встраивать в код изменения, беря за основу рекомендации, приведенные в этой книге.

Да, в одной книге мы можем дать лишь ограниченный объем материала. И вам потребуется продолжать обучение, причем скоро. Думаем, почти сразу после того, как вы закончите читать эту главу: пока вы не начнете писать реальный код, ваш мозг не сосредоточится полноценно на изученных принципах и приемах. Вы уже забываете материал из главы 2, поэтому очень важно сразу применять знания на практике.

27.2. ВАШ СЛЕДУЮЩИЙ ШАГ

Наша лучшая рекомендация — *ненадолго прекратить учиться* и *начать делать*. Вам доступно множество фактов и приемов, благодаря которым вы можете создать первый инструмент и контроллер. Как только вы это сделаете, сразу поймете, что забыли некоторые вещи, — и это отличные новости! Ведь осознав, что забыли что-то, вы вернетесь к нужной главе и освежите ее содержимое в памяти. Это действие укрепляет связи нейронов мозга, отвечающих за память о материале, и в следующий раз вам уже будет легче вспомнить ту же самую информацию. Но вы не поймете, что что-то забыли, и не предпримете очередные действия для освежения памяти, пока не *начнете делать*.

С учетом всего этого мы дадим несколько рекомендаций о том, какими могут быть ваши следующие действия.

- *Не пытайтесь решить самую сложную задачу из актуальных*. Ищите задачу поменьше, которую уже хорошо понимаете и знаете, как ее решить. Таким образом вы сможете сфокусироваться на новых подходах и приемах, которые освоили. По мере обретения уверенности вы сможете создавать более сложные инструменты и контроллеры.
- *Помните о целесообразности*. Описанные нами подходы и приемы не слишком удлиняют процесс программирования, но все же занимают время. Например, вам придется уделить внимание проектированию параметров и прописать получение ввода из конвейера. Но эти вложения того стоят, так как вы быстро начнете делать все это практически на автомате. Альтернатива «Я наскоро сделаю это сейчас и потом доведу до ума» станет плохой идеей. Позже у вас может не хватать времени, чтобы сделать все как следует, и тогда вы получите инструмент, который работает не так, как должен.
- *Трудности*. К худшему или лучшему, но человеческий мозг лучше усваивает информацию, когда решает задачу, чем когда пассивно ее поглощает. Исходя из этого, вам следует погружаться в какой-то процесс, сталкиваться со сложностями и решать их. В качестве ценных ресурсов рекомендуем такие форумы,

как ServerFault.com (http://serverfault.com/) и PowerShell.org (http://powershell.org/). Опишите свою проблему, ваши действия и предоставьте какие-нибудь подробности (например, сообщения об ошибках) о том, что не сработало. *Не* просите разработчиков писать ваш скрипт за вас — четко говорите, что вам нужен лишь толчок в правильном направлении.

- *Делитесь*. Каждый раз, когда сталкиваетесь с проблемой, пишите об этом в блоге. Сам факт вспоминания этой проблемы и ее решения хорошо укрепляет нейронные связи мозга. Записывание выполненных действий (даже просто для внутреннего блога компании, который не прочтет никто, кроме вашей команды) помогает учиться. Помните: из-за повышения скорости рождаемости новые люди постоянно сталкиваются с проблемами, которые вы уже решали. Помогите им.
- *Производите расчеты*. Всякий раз, когда вы автоматизируете некое действие, начните с выяснения того, сколько времени ваша организация тратит, выполняя его вручную. Если можете, то посчитайте в часах — возможно открыв ответ справочной службы, содержащий решение для поступившего запроса. Высчитайте среднюю зарплату разработчиков, которые проводят время, решая эту задачу вручную. Умножьте эту сумму на 1,14 (примерный способ вычисления зарплаты со всеми бонусами, по крайней мере, в основной части Северной Америки) и затем разделите на 2000 (среднее количество рабочих часов в году). Результатом станет стоимость часа работы для данного человека. И эту цифру вы можете умножить на количество часов, потраченных на решение задачи вручную. В итоге вы узнаете, сколько денег ваша организация тратит на эту проблему. Становится легко вычислить окупаемость вложений, когда вы знаете, сколько было потрачено, сколько потребовалось на автоматизацию задачи и сколько времени нужно оплатить теперь, когда она автоматизирована.
- *Не ищите скрипты в Google*. В начале нового проекта не нужно первым делом открывать браузер и искать существующий скрипт. Допустим, вы найдете какой-то код. Как вы узнаете, будет ли он работать в вашей среде? Есть ли у вас необходимые навыки работы с PowerShell, чтобы определить, является ли этот код хорошим? Кроме того, вы наверняка потратите немало времени на анализ жестко прописанных переменных и прочего. Лучше будет начать со справочной системы PowerShell и сделать все самостоятельно. Да, на это может уйти много времени, но вы научитесь и в итоге получите инструмент, который точно будет работать в вашей среде. Вполне нормально искать примеры использования конкретного командлета или параметра, но вы никогда не преуспеете, если будете просто копировать чужие скрипты.

Все эти пункты связаны с повышением уровня профессионализма в разработке инструментов.

27.3. ЧТО ЖЕ ДАЛЬШЕ?

Итак, что в перспективе? Какие возможности PowerShell вам следует изучить? Помните: это непрерывно меняющееся пространство, которое требует постоянного внимания, если вы хотите быть в тренде. Ниже мы указали некоторые из рекомендуемых областей для изучения.

- PowerShell Core — проект с открытым исходным кодом, доступный по адресу https://GitHub.com/powershell, который будет работать в macOS, различных дистрибутивах Linux и, естественно, в Windows. Уделите время его изучению.
- Такие проекты с открытым исходным кодом, как PlatyPS, Pester и PowerShell Script Analyzer (PSScriptAnalyzer), представляют отличные инструменты. Познакомьтесь с ними и начните учиться их использовать в повседневной разработке инструментов. Более того, подключайтесь к дискуссии, обсуждая возникающие проблемы и, возможно, отправляя код.
- События на уровне сообщества, такие как PowerShell Saturdays, ежегодный саммит PowerShell + DevOps Global Summit (powershellsummit.org) и региональные конференции PowerShell (например, PowerShell Conference Europe и PowerShell Conference Asia), стоят вашего времени, как и десятки локальных групп пользователей по всему миру.
- Наконец, всегда ищите новые источники информации. У Manning есть несколько книг, которые помогут в этом, и постоянно выходят новые. Кроме того, мы отвечаем за солидную часть контента на Pluralsight (www.pluralsight.com). PowerShell (и создание инструментов) — большая удивительная вселенная, в которой много интересного. Выделяйте понемногу времени еженедельно, чтобы следить за последними тенденциями и изучать что-то новое. И конечно же, продолжайте создавать инструменты в своей компании.

ИТОГИ ГЛАВЫ

В данной главе мы говорили о том, что такой процесс, как освоение навыков создания инструментов с помощью PowerShell, не имеет конца. Для вас все только начинается.

Мы призвали вас задуматься о значимости создания инструментов в PowerShell, подчеркнув роль оболочки в создании рабочих элементов, пригодных для повторного использования. Вдобавок мы указали на важность практического применения полученных знаний и порекомендовали сразу начать реализовывать то, что вы усвоили, поскольку это помогает укреплять понимание.

Кроме того, мы дали практические советы о том, какими могут быть ваши следующие действия: начинайте с малого, продолжайте учиться, ищите помощь на

форумах, делитесь знаниями, вычисляйте окупаемость вложений и старайтесь не использовать в скриптинге прием копипаста. Мы дали рекомендации и о том, какие еще области PowerShell могут быть вам полезны: PowerShell Core, проекты с открытым исходным кодом, события сообщества и дополнительные обучающие ресурсы.

Процесс обучения скриптингу и создания инструментов с помощью PowerShell является непрерывным. Мы призываем вас продолжать его изучение, развивать профессиональные навыки и вносить свой вклад в сообщество PowerShell.

Read IT Club

Комьюнити рецензентов
и переводчиков ИТ-литературы

Миссия участников клуба — обеспечить высокое качество профессиональной переводной литературы в России. «Книжные дебагеры» проверяют корректность терминологии и подписей на схемах и иллюстрациях, чтобы сделать книги более понятными русскоязычному читателю. Стать участником Read IT Club может любой ИТ-специалист, готовый поделиться опытом с сообществом.

присоединиться к нам

Джеймс Петти, Дон Джонс, Джеффри Хикс

Изучаем скриптинг PowerShell за месяц, занимаясь один час в день. 2-е изд.

Перевел с английского Д. Брайт

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*О. Андриевич, Н. Терех*
Верстка	*Г. Блинов*

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: 08.2025.

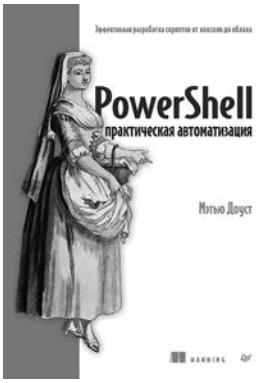
Наименование: книжная продукция.

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 18.07.25. Формат 70x100/16. Бумага офсетная. Усл. п. л. 28,380. Тираж 700. Заказ 0000.



Мэтью Доуст

PowerShell: практическая автоматизация

PowerShell — это язык для написания скриптов, инструмент, позволяющий программно управлять всем центром обработки данных. С его помощью можно создавать высокоэффективные и надежные системы автоматизации, пригодные для многократного использования и значительно повышающие производительность специалистов. Из этой книги вы узнаете, как проектировать, разрабатывать, организовывать и развертывать скрипты для автоматизации задач любого масштаба: от локальных серверов до корпоративных облачных платформ.

Вы узнаете, как создавать скрипты PowerShell для автоматизации локальных и облачных систем. Найдете советы по определению задач, которые стоит автоматизировать, по организации структуры скриптов и управлению ими, а также множество примеров кода с подробными пояснениями. Научитесь адаптировать уже готовые скрипты к новым условиям применения и упрощать работу специалистов нетехнического профиля при помощи простых и понятных интерфейсов SharePoint.