bhv®

Андрей Попов

Современный PowerShell

Работа с оболочкой Windows PowerShell в Windows Terminal

Продвинутая настройка командной строки

Описание языка PowerShell

Работа с файловой системой, структурированными данными и веб-ресурсами

Управление процессами, службами и серверами автоматизации

Построение GUI для сценариев Windows PowerShell

Кросс-платформенные возможности PowerShell в macOS и Linux

СИСАДМИН
СИСТЕМНЫЙ АДМИНИСТРАТОР

Андрей Попов

Современный PowerShell

Санкт-Петербург
«БХВ-Петербург»
2022

УДК 004.451
ББК 32.973-018.2
П58

Попов А. В.

П58 Современный PowerShell. — СПб.: БХВ-Петербург, 2022. — 368 с.: ил. — (Системный администратор)

ISBN 978-5-9775-6874-6

Рассматривается объектно-ориентированная оболочка командной строки Microsoft PowerShell и ее возможности для автоматизации повседневных задач пользователей и администраторов. Описываются основные элементы и конструкции языка PowerShell, инструменты для написания и отладки сценариев. Приведена информация о работе с файловой системой и структурированными данными (CSV, JSON). Рассмотрены приемы управления процессами, службами и серверами автоматизации. Обсуждаются вопросы взаимодействия с веб-ресурсами при помощи HTTP-запросов. Даны примеры построения GUI для сценариев PowerShell с помощью Windows Forms и Windows Presentation Foundation. Представлены кроссплатформенные возможности PowerShell в macOS и Linux.

Для начинающих системных администраторов и опытных пользователей

УДК 004.451
ББК 32.973-018.2

Группа подготовки издания:

Руководитель проекта	*Павел Шалин*
Зав. редакцией	*Людмила Гауль*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн серии	*Марины Дамбиевой*
Оформление обложки	*Зои Канторович*

Подписано в печать 02 03 22
Формат 70×100 1/16 Печать офсетная Усл печ л 29,67
Тираж 1000 экз Заказ № 3680.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул , 20

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М О , г Чехов, ул Полиграфистов, д 1

ISBN 978-5-9775-6874-6

© ООО "БХВ", 2022
© Оформление ООО "БХВ-Петербург", 2022

Оглавление

Введение **10**
Для кого предназначена эта книга 11
Структура книги 11
Принятые в книге соглашения 13

ЧАСТЬ I. ЗНАКОМИМСЯ С POWERSHELL **15**

Глава 1. Командная строка и автоматизация работы **16**
Зачем нужна командная строка и скрипты 17
Особенности языков сценариев для операционной системы 17
Инструменты автоматизации в UNIX-подобных системах 19
Особенности автоматизации в Windows 21
Командный интерпретатор cmd.exe 22
Сервер сценариев Windows Script Host 24
Оболочка и среда выполнения сценариев PowerShell 28
Итоги 31

Глава 2. Терминал, консоль и командная оболочка **32**
Терминалы в Windows 34
Стандартная консоль Windows 35
Windows Terminal 36
Установка и запуск 37
Работа с вкладками 38
Разделение окна на несколько панелей 39
Использование палитры команд 40
Запуск терминала с аргументами командной строки 40
Итоги 41

Глава 3. Первые шаги в PowerShell. Основные понятия **42**
Запуск оболочки PowerShell 42
Работают ли знакомые команды 43
Вычисление выражений 45
Типы команд PowerShell 46
Командлеты 47
Имена и структура командлетов 47

Общие параметры командлетов 50
Поиск командлетов 51
Функции 52
Сценарии 53
Внешние исполняемые файлы 53
Псевдонимы команд 53
Диски PowerShell 57
Провайдеры PowerShell 58
Навигация по дискам PowerShell 59
Просмотр содержимого дисков и каталогов 60
Создание дисков 62
Итоги 63

Глава 4. Работа в оболочке PowerShell 65
Редактирование в командной строке PowerShell 65
Автоматическое завершение команд 68
Ввод команды в несколько строках 70
Справочная система PowerShell 70
Получение справки о командлетах 71
Справочная информация, не связанная с командлетами 76
История команд в сеансе работы 78
Протоколирование действий в сеансе работы 80
Настройка оформления командной строки PowerShell 82
Заголовок командного окна 83
Приглашение командной строки 85
Настройка пользовательских профилей 86
Политики выполнения сценариев 88
Итоги 90

Глава 5. Работа с объектами 91
Конвейеризация объектов в PowerShell 91
Просмотр структуры объектов (командлет *Get-Member*) 93
Фильтрация объектов (командлет *Where-Object*) 95
Использование блока кода 95
Использование оператора сравнения 97
Сортировка объектов (командлет *Sort-Object*) 98
Выделение объектов и свойств (командлет *Select-Object*) 100
Выполнение произвольных действий над объектами в конвейере (командлет *ForEach-Object*) 103
Группировка объектов (командлет *Group-Object*) 104
Измерение характеристик объектов (командлет *Measure-Object*) 105
Обращение к статическим методам и полям 106
Итоги 108

Глава 6. Управление выводом команд 109
Форматирование выводимой информации 110
Перенаправление выводимой информации 112
Сохранение данных в файл 113
Печать данных 114

Подавление вывода ... 115
Табличный вывод данных в графическое окно ... 115
Вывод в формате HTML ... 117
Дополнительные потоки в PowerShell ... 120
Перенаправление в файл ... 121
Перенаправление в выходной поток Output ... 122
Итоги ... 123

ЧАСТЬ II. POWERSHELL КАК ЯЗЫК ПРОГРАММИРОВАНИЯ ... 125

Глава 7. Переменные, массивы и хэш-таблицы ... 126
Числовые и символьные литералы ... 126
Числовые литералы ... 126
Символьные строки ... 127
Строки в одинарных и двойных кавычках ... 127
Строки типа here-string ... 129
Переменные PowerShell ... 130
Переменные оболочки PowerShell ... 131
Пользовательские переменные ... 133
Типы переменных ... 133
Приведение типов ... 135
Дополнительные атрибуты переменных ... 136
Константы ... 136
Переменные среды Windows ... 137
Массивы в PowerShell ... 138
Обращение к элементам массива ... 139
Операции с массивом ... 140
Увеличение длины массива. Объединение массивов ... 141
Удаление элементов ... 142
Действие оператора присваивания ... 142
Сохранение в массиве вывода командлетов ... 143
Удаление массива ... 143
Хэш-таблицы (ассоциативные массивы) ... 144
Операции с хэш-таблицей ... 145
Итоги ... 147

Глава 8. Операторы и управляющие инструкции ... 149
Арифметические операторы ... 149
Оператор сложения ... 150
Оператор умножения ... 152
Операторы вычитания, деления и остатка от деления ... 153
Операторы присваивания ... 154
Операторы сравнения ... 155
Сравнения с использованием массивов ... 156
Операторы проверки на соответствие шаблону ... 157
Шаблоны с подстановочными символами ... 157
Шаблоны с регулярными выражениями ... 158
Логические операторы ... 159

Управляющие инструкции языка PowerShell ... 160
Инструкция *If ... Elseif ... Else* ... 160
Цикл *While* ... 161
Цикл *Do ... While* ... 162
Цикл *For* ... 162
Цикл *Foreach* ... 163
Инструкция *Foreach* вне конвейера команд ... 163
Инструкция *Foreach* внутри конвейера команд ... 164
Вопросы производительности ... 165
Метки циклов, инструкции *Break* и *Continue* ... 165
Инструкция *Switch* ... 166
Виды проверок внутри *Switch* ... 166
Проверка массива значений ... 169
Итоги ... 171

Глава 9. Функции, фильтры, сценарии и модули ... 172
Функции в PowerShell ... 172
Обработка аргументов с помощью переменной *$args* ... 173
Формальные параметры функций ... 175
Позиционные и именованные параметры ... 175
Ограничение параметров по типу ... 177
Значения по умолчанию для параметров ... 178
Дополнительные атрибуты и валидация параметров ... 179
Параметры-переключатели ... 181
Описание параметров в операторе *Param()* ... 182
Передача параметров с помощью сплаттинга переменных ... 183
Возвращаемые значения ... 184
Функции внутри конвейера команд ... 186
Функции в качестве командлетов. Расширенные функции ... 187
Три фазы работы функции в конвейере ... 187
Доступ к общим параметрам и дополнительным потокам. Расширенные функции ... 189
Сценарии PowerShell ... 191
Создание сценария ... 191
Запуск сценария из PowerShell ... 192
Запуск сценария из внешней программы ... 193
Передача аргументов в сценарии ... 194
Выход из сценариев. Код возврата ... 195
Области видимости функций ... 195
Глобальная область видимости ... 196
Оператор *Dot-Source* ... 196
Области видимости переменных ... 197
Модули PowerShell ... 199
Модули-сценарии ... 199
Репозиторий сценариев PowerShell Gallery ... 201
Итоги ... 204

Глава 10. Обработка ошибок при выполнении команд ... 206
Объект *ErrorRecord* и поток ошибок ... 207
Сохранение объектов, соответствующих ошибкам ... 210

Мониторинг возникновения ошибок 213
Режимы обработок ошибок 214
Обработка критических ошибок (исключений) 215
Инструкция *Trap* 216
Инструкция *Try/Catch/Finally* 218
Итоги 218

ЧАСТЬ III. АВТОМАТИЗИРУЕМ РУТИНУ 221

Глава 11. Работа с файловой системой и оболочкой Windows 222
Навигация в файловой системе 222
Получение списка файлов и каталогов 222
Определение размера каталогов 226
Создание файлов и каталогов 226
Создание нескольких файлов 227
Пересоздание файла 228
Создание файла в несуществующем каталоге 228
Чтение содержимого файлов 229
Запись файлов 230
Копирование файлов и каталогов 231
Копирование каталога с файлами 232
Копирование вложенных каталогов 232
Копирование файлов по маске 233
Конкатенация файлов 234
Переименование и перемещение файлов и каталогов 235
Переименование группы файлов 235
Перемещение файлов 235
Удаление файлов и каталогов 236
Поиск текста в файлах 237
Замена текста в файлах 239
Работа с файлами-ярлыками 240
Доступ к COM-объектам из PowerShell 240
Объект *WScript.Shell* 241
Создание ярлыка на рабочем столе 242
Удаление некорректных ярлыков 243
Итоги 244

Глава 12. Обработка структурированных данных 245
Работа с данными в формате CSV 245
Чтение из CSV-файла 245
Запись в CSV-файл 248
Обработка данных без обращения к файлу 249
Обработка данных в JSON-формате 250
Итоги 252

Глава 13. Управление процессами, службами и серверами автоматизации 253
Управление процессами 253
Просмотр списка процессов 254
Определение библиотек, используемых процессом 257

Остановка процессов ... 258
Запуск процессов ... 259
Изменение приоритетов выполнения процесса ... 261
Завершение неотвечающих процессов ... 261
Управление службами ... 261
Просмотр списка служб ... 262
Остановка и приостановка служб ... 263
Запуск и перезапуск служб ... 264
Изменение параметров службы ... 265
Работа с серверами автоматизации ... 266
Объектные модели Microsoft Word и Excel ... 266
Взаимодействие с Microsoft Word ... 268
Взаимодействие с Microsoft Excel ... 268
Итоги ... 269

Глава 14. HTTP-запросы к веб-ресурсам ... 270
Командлет *Invoke-WebRequest* ... 270
Анализ HTML-страниц ... 270
Содержимое ответа от сервера и HTTP-заголовки ... 272
Сохранение веб-ресурсов ... 274
Поиск HTML-элементов на странице ... 275
Выполнение POST-запросов ... 277
Командлет *Invoke-RestMethod* ... 279
Итоги ... 281

ЧАСТЬ IV. ПИШЕМ СЦЕНАРИИ ... 283

Глава 15. Разработка сценариев PowerShell ... 284
Переход от команд к сценариям ... 284
Среды для разработки сценариев ... 285
PowerShell ISE ... 285
Запуск сценариев и фрагментов ... 285
Справочная система ... 287
Редактирование текста ... 289
Отладка сценариев ... 292
Visual Studio Code ... 294
Другие редакторы и среды разработки ... 299
Рекомендации по разработке сценариев ... 300
Общая структура сценария ... 300
Имена и псевдонимы команд и параметров ... 301
Расширенные и базовые функции ... 303
Комментарии ... 303
Справка, основанная на комментариях ... 304
Расположение и форматирование кода ... 306
Регистр символов в именах ... 306
Скобки в коде ... 307
Отступы, пробелы и пустые строки ... 307
Точка с запятой как разделитель строк и значений ... 309
Обратный апостроф для многострочных команд ... 309

Производительность сценариев и продуктивность разработчика 310
Пример. Статистика по объектам файловой системы (cmd и PowerShell) 311
Итоги 312

Глава 16. Отладка функций и сценариев 313
Вывод диагностических сообщений 313
Командлет *Set-PSDebug* 315
Трассировка выполнения команд 316
Пошаговое выполнение команд 318
Вложенная командная строка 319
Управление точками останова (командлеты **-PSBreakPoint*) 321
Создание точки останова для сценария 322
Создание точки останова для команды 326
Создание точки останова для переменной 326
Просмотр точек останова 327
Удаление точек останова 329
Итоги 330

Глава 17. Графический интерфейс для сценариев 331
Построение GUI с помощью Windows Forms 331
Построение GUI с помощью Windows Presentation Foundation 335
Итоги 341

Что дальше? PowerShell для профессионалов 342

ПРИЛОЖЕНИЯ 343

Приложение 1. Что значат эти символы 344

Приложение 2. PowerShell в macOS и Linux 348
Установка и запуск оболочки 348
Отличия от Windows PowerShell 350

Приложение 3. Дополнительная настройка командной строки 352
Модуль PSReadLine 352
Интеграция с Git. Модуль posh-git 355
Оформление приглашения командной строки 358
Установка шрифтов Powerline 358
Модуль Oh My Posh 359
Оформление списков файлов и каталогов. Модуль Terminal-Icons 362

Предметный указатель 364

Введение

Первая версия Windows PowerShell была выпущена компанией Microsoft в уже далеком 2006 г. Главной задачей, которую ставили перед собой разработчики, было создание командной оболочки и среды выполнения сценариев, которая наилучшим образом подходила бы для Windows и была бы более функциональной, расширяемой и простой в использовании, чем любой аналогичный продукт для других операционных систем. В первую очередь эта среда должна была помогать в решении задач, стоящих перед системными администраторами, а также удовлетворять требованиям разработчиков программного обеспечения, предоставляя им средства для быстрой реализации интерфейсов управления к создаваемым приложениям.

Спустя пятнадцать лет можно уверенно сказать, что разработчики Microsoft справились со своей задачей — продукт оказался продуманным и удачным, впитал в себя лучшие решения из других оболочек и скриптовых языков.

Microsoft уже давно позиционирует данную оболочку в качестве основного инструмента управления операционной системой и своими приложениями — PowerShell входит в качестве стандартного компонента в Windows, начиная с Windows Server 2008 и Windows 7, а также используется во всех ключевых продуктах Microsoft (Azure, SQL Server, Exchange Server, System Center и т. д.). Технология PowerShell Desired State Configuration (DSC) реализует подход "инфраструктура как код", позволяя с помощью сценариев PowerShell настраивать среды выполнения приложений на множестве распределенных серверов.

PowerShell постоянно развивается и расширяется. Сегодня это кросс-платформенная оболочка и язык написания сценариев с открытым исходным кодом, с которыми можно работать не только в Windows, но и в macOS и Linux.

Таким образом, PowerShell — это отличный помощник для профессиональных системных администраторов и DevOps-инженеров. С другой стороны, действительно мощные возможности PowerShell сочетаются с удивительной простотой использования интуитивно понятных команд и удобной оболочки. Написать сценарии на PowerShell тоже совсем не сложно (даже проще, чем на Python), для их создания не нужны навыки профессионального программиста или опыт системного администратора. PowerShell может быть полезен любому ИТ-специалисту в качестве инструмента для автоматизации своей повседневной работы: манипуляции файлами,

управления процессами и службами, создания офисных документов, обращения с HTTP-запросами к сетевым ресурсам, обработки структурированных данных и т. д.

Целью настоящей книги является решение нескольких задач.

- Разобрать основные приемы для эффективной и удобной работы с PowerShell в интерактивном режиме оболочки командной строки.
- Пояснить лежащие в основе PowerShell базовые механизмы работы с объектами и описать основные конструкции и элементы языка PowerShell, которые отличают его от традиционных скриптовых языков типа Python, Ruby, JavaScript или VBScript. Эти отличия связаны с тем, что PowerShell, с одной стороны, является командной оболочкой, а с другой — высокоуровневым .NET-языком. Уверен, что, поняв такую двойную природу PowerShell, вы с удовольствием будете пользоваться этим уникальным инструментом и при необходимости сможете быстро разобраться в любом серверном продукте, который управляется с помощью PowerShell.
- Рассмотреть возможности и особенности сценариев PowerShell, привести рекомендации для их грамотного написания. Это поможет лучше понимать чужие скрипты для решения конкретных администраторских задач, модифицировать их под свои требования, а при необходимости самостоятельно создавать собственные команды и сценарии.

Для кого предназначена эта книга

Книга ориентирована на системных администраторов и ИТ-специалистов, которые устали от ежедневного выполнения повторяющихся задач и хотят быстро и с минимальными усилиями автоматизировать свою работу с помощью современного стандартного и мощного инструмента, который всегда под рукой любого пользователя Windows.

Структура книги

Книга состоит из четырех частей. В первой части (*"Знакомимся с PowerShell"*) мы начнем работать в терминале с командной строкой, рассмотрим основные концепции и конструкции оболочки PowerShell.

В *главе 1* обсуждается влияние архитектуры операционной системы на задачи автоматизации работы в Windows, рассматриваются особенности стандартных инструментов автоматизации (командная оболочка cmd.exe, сервер сценариев Windows Script Host, оболочка Windows PowerShell). Приводятся причины и цели создания Windows PowerShell, обсуждается основное отличие PowerShell от всех других оболочек командной строки — ориентация на работу с объектами, а не с потоком текста.

В *главе 2* уточняются термины, применяемые при работе с командной строкой (терминал, консоль, командная оболочка), обсуждаются особенности реализации

консольных приложений в Windows. Рассматриваются возможности нового усовершенствованного терминала Windows Terminal.

В *главе 3* приводятся инструкции по установке и запуску различных версий PowerShell, описываются типы команд, используемые в данной оболочке. Обсуждаются понятия псевдонимов команд и дисков PowerShell.

В *главе 4* изучаются приемы интерактивной работы в оболочке PowerShell и способы обращения к справочной системе PowerShell. Рассматриваются вопросы настройки интерфейса оболочки, пользовательских профилей и политик выполнения сценариев PowerShell.

В *главе 5* обсуждается основной механизм конвейеризации объектов, описываются базовые манипуляции с объектами, которые можно выполнять в оболочке PowerShell (фильтрация, сортировка, группировка и т. д.).

В *главе 6* разбираются механизмы управления выводом команд PowerShell, обсуждаются механизмы форматирования и перенаправления результирующей информации. Рассматриваются выходные потоки различных типов, доступные в PowerShell.

Во второй части книги (*"PowerShell как язык программирования"*) мы разберемся с особенностями языка PowerShell, благодаря которым он сильно отличается от традиционных языков сценариев.

Глава 7 посвящена изучению основных структур данных, использующихся в PowerShell (константы, переменные, массивы и хэш-таблицы).

В *главе 8* рассматриваются основные операторы и управляющие инструкции языка PowerShell.

В *главе 9* обсуждаются вопросы создания и использования функций, сценариев и модулей на языке PowerShell.

Глава 10 посвящена имеющимся в PowerShell средствам обработки ошибок.

В третьей части книги (*"Автоматизируем рутину"*) показаны примеры применения интерактивных команд и сценариев PowerShell для решения практических задач.

В *главе 11* приводятся примеры выполнения с помощью команд PowerShell основных операций с объектами файловой системы и оболочки Windows.

В *главе 12* изучаются команды PowerShell для работы со структурированными данными в форматах CSV и JSON.

В *главе 13* рассматриваются команды, позволяющие управлять процессами, службами и внешними серверами автоматизации (например, приложениями пакета Microsoft Office).

В *главе 14* приводятся примеры обращения с помощью команд PowerShell к веб-ресурсам по протоколу HTTP.

В четвертой части книги (*"Пишем сценарии"*) мы познакомимся с инструментами для разработки и отладки сценариев PowerShell, обсудим рекомендации и лучшие практики по созданию и оформлению сценариев.

В *главе 15* рассматриваются две среды разработки (PowerShell ISE и Visual Studio Code), которые Microsoft рекомендует использовать для написания и отладки сценариев PowerShell. Приводятся рекомендации по разработке сценариев PowerShell для получения понятного и простого кода, который можно будет легко дорабатывать и изменять впоследствии.

Глава 16 посвящена вопросам отладки сценариев PowerShell.

В *главе 17* рассматриваются варианты создания графического интерфейса для сценариев Windows PowerShell.

В *приложении 1* рассматриваются варианты применения различных специальных символов в синтаксисе языка PowerShell.

Приложение 2 посвящено кросс-платформенной версии PowerShell, которую можно устанавливать и использовать в операционных системах macOS и Linux.

В *приложении 3* описаны несколько модулей PowerShell, позволяющих дополнительно настроить командную строку, сделать ее более информативной, визуально привлекательной и удобной для работы.

Принятые в книге соглашения

Оболочка PowerShell — это интерактивная среда, поэтому во многих примерах показаны как команды, вводимые пользователем, так и ответ на них, генерируемый системой. Перед командой указывается строка-приглашение PowerShell, обычно состоящая из префикса `PS` и пути к текущему каталогу. Сама вводимая команда выделяется полужирным шрифтом (например, **`Get-Process`**). На следующих нескольких строках приводится текст, возвращаемый системой в ответ на введенную команду. Например:

```
PS C:\Users\andrv> Get-Process

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  ProcessName
-------  ------    -----      ----- -----   ------     --  ----------
     99       5     1116        692    32     0.07    232 alg
     39       1      364        500    17     0.14   1636 ati2evxx
     57       3     1028       1408    30     0.38    376 atiptaxx
    412       6     2128       3600    26     6.50    808 csrss
     64       3      812       1484    29     0.19    316 ctfmon
    386      13    13748      14448    77    16.11   1848 explorer
    171       5     4512        584    44     0.20    428 GoogleT...
      0       0        0         16     0                0 Idle
    151       4     2908        992    41     1.05    412 kav
. . .
```

Многоточие здесь указывает на то, что для экономии места приведены не все строки, возвращаемые командой `Get-Process`.

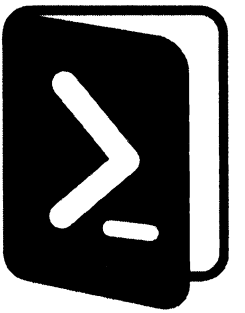
Иногда вводимые команды могут разбиваться на несколько строк. В этих случаях перед каждой строкой команды будут указываться символы >>, например:

```
PS C:\Users\andrv> dir *.tmp | ForEach-Object {
>> $arr=$_.Name.split(".");
>> $newname=$arr[0]+".new";
>> ren $_.fullname $newname -passthru;
>> }
>>
    Каталог: C:\Users\andrv

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        09.10.2021     10:05              0 3.new
```

При описании операторов, функций и методов объектов мы использовали стандартные соглашения. Названия подставляемых параметров и аргументов набраны курсивом, необязательные параметры заключены в квадратные скобки `[]`, например:

CreateObject(*strProgID* [,*strPrefix*])



ЧАСТЬ I

Знакомимся с PowerShell

Глава 1. Командная строка и автоматизация работы

Глава 2. Терминал, консоль и командная оболочка

Глава 3. Первые шаги в PowerShell. Основные понятия

Глава 4. Работа в оболочке PowerShell

Глава 5. Работа с объектами

Глава 6. Управление выводом команд

ГЛАВА 1

Командная строка и автоматизация работы

Для общения человека с компьютером, ноутбуком или мобильным устройством необходим посредник — операционная система. Любую операционную систему можно представить в виде совокупности *ядра системы*, которое управляет аппаратурой и оперирует файлами и процессами, и *интерфейса*, обеспечивающего пользователю доступ к функциональности ядра. Интерфейс может быть различных видов.

- *Командно-текстовый*, когда команды для управления системой вводятся с клавиатуры или берутся из заранее подготовленных текстовых файлов-сценариев (скриптов). В ранних операционных системах командный интерфейс был единственным.
- *Графический*, когда работа в системе происходит посредством различных визуальных элементов управления (кнопки, раскрывающиеся списки, диалоговые окна и т. д.). Графический интерфейс — основное средство взаимодействия с пользователем на современных персональных компьютерах и мобильных устройствах.
- *Голосовой*, когда человек просто произносит команды для управления системой. В последнее время широкое распространение получили несколько систем подобного типа (Siri компании Apple, Cortana компании Microsoft, Google Now), которые работают на различных мобильных платформах. Популярность этих голосовых помощников растет, технологии распознания голоса развиваются, однако сказать, что в ближайшее время управление голосом полностью заменит графический и текстовый интерфейсы, пока нельзя.

Конечно, обычному пользователю сегодня проще, удобнее и привычнее иметь дело с графическим интерфейсом, здесь не нужно изучать и запоминать специальные команды. Почему же даже в самых современных операционных системах для персональных компьютеров и серверов продолжает поддерживаться командно-текстовый интерфейс в виде оболочек командной строки и различных сред выполнения программ на специальных языках сценариев (скриптовых языках)?

Зачем нужна командная строка и скрипты

Дело в том, что графический интерфейс плохо приспособлен для *автоматизации работы*, когда однажды выполненные действия нужно воспроизводить снова и снова. Такая автоматизация может понадобиться, если мы регулярно делаем одни и те же однотипные операции (например, ежедневно копируем измененные за день документы) или, наоборот, нам нужно выполнить одну операцию, но на множестве компьютеров в сети (например, изменить путь к файлу в ярлыке на рабочих столах сотни компьютеров).

Если подобные действия выполнялись с помощью текстовых команд, то их легко можно повторить без участия человека путем создания пакетного файла (сценария) с этими командами. Это значительно увеличивает производительность работы с рутинными задачами.

Таким образом, основное преимущество командно-текстового интерфейса — *возможность автоматизации работы* пользователей и администраторов с помощью программ-сценариев, выполняющихся в операционной системе.

Кроме того, в сценариях четко определяется порядок выполнения задачи, поэтому их можно использовать в качестве документации. Сохранив свои действия в сценарии, вы не забудете, что именно делали и зачем.

Наконец, сценарии позволяют в автоматическом режиме использовать сервисы, предоставляемые внутренними объектами операционной системы и другим программным обеспечением (например, офисными приложениями).

Изучение языков сценариев для операционной системы может быть полезно разным людям:

- администраторам серверов, баз данных или локальной сети просто необходимо владеть навыками создания и использования сценариев, без этого их работа будет очень неэффективной;
- для обычного пользователя сценарии — полезный дополнительный инструмент для автоматизации повседневных действий: манипуляций файлами, работы с офисными приложениями и интернет-ресурсами;
- разработчики приложений и DevOps-специалисты могут использовать сценарии для компиляции и запуска написанных программ, выполнения HTTP-запросов, работы с командным интерфейсом системы управления версиями Git, пакетными менеджерами различных языков программирования (например, npm для JavaScript или Composer для PHP) или консольными утилитами типа wget, curl.

Особенности языков сценариев для операционной системы

Командно-сценарные языки — это один из видов обширного множества языков сценариев. Главная задача таких языков — "склеивание" имеющихся в операционной системе готовых компонентов (исполняемых файлов, динамических библиотек,

внутренних объектов системы и т. п.), связывание их друг с другом. Именно этим языки сценариев отличаются от традиционных "системных" языков программирования (C, C++, Java и т. п.), которые проектировались с расчетом на построение структур данных и алгоритмов.

Какими хотелось бы видеть идеальную командную оболочку и язык сценариев для автоматизации работы в операционной системе? Перечислим некоторые требования к таким инструментам.

- Совместимость со всеми используемыми в настоящее время версиями операционной системы и доступ к максимально широкому кругу ее возможностей.
- Простота и понятность языка — он должен быть доступен для изучения обычным пользователям, а не только профессиональным программистам.
- Возможность запуска из сценария внешних исполняемых файлов в режиме команд без использования дополнительных конструкций типа функций `exec()` или `run()`.
- Поддержка как пакетного режима работы (написание и запуск сценариев), так и удобной интерактивной командной строки (с короткими псевдонимами для команд, автодополнением команд и путей к файлам и каталогам, возможностью повтора ранее введенных команд и т. п.).
- Наличие встроенной справочной системы и централизованного репозитория сценариев для решения стандартных задач.

Давайте теперь разберемся, что мы имеем в реальности, какие языки и оболочки можно использовать для автоматизации работы в современных операционных системах.

Перефразируя известное выражение Карла Маркса о бытии, определяющем сознание, можно сказать, что операционная система определяет свою оболочку. Командные языки и оболочки создавались не на пустом месте, они развивались вместе с операционными системами. Появляются новые возможности в системе — возникает потребность в средстве для доступа к этим возможностям в автоматическом режиме. В результате создается командно-сценарный язык программирования и среда выполнения команд и сценариев на этом языке, которые естественным образом подходят для определенной операционной системы.

Практически все современные компьютеры, ноутбуки и серверы работают под управлением либо Windows, либо одной из UNIX-подобных операционных системы (macOS или Linux). Внешне операционные системы этих двух семейств могут быть более-менее похожи друг на друга (например, графический интерфейс Linux Mint напоминает Windows 7). Однако внутреннее устройство Windows и UNIX имеет принципиальное архитектурное различие, которое определило разные пути развития командных оболочек и языков сценариев для этих операционных систем и в конечном итоге привело к созданию действительно уникального и мощного инструмента автоматизации — Microsoft PowerShell.

Инструменты автоматизации в UNIX-подобных системах

В UNIX-подобных операционных системах (macOS или Linux) в качестве стандартного инструмента автоматизации выступает терминал, в котором запускаются команды и скрипты той или иной модификации оригинальной командной оболочки UNIX (чаще всего это bash или zsh).

Именно в UNIX в далеких 1970-х годах зародились и развивались принципы программирования, основанные на инструментальных средствах (software tools).

- Для решения некоторой задачи разрабатываются небольшие утилиты, каждая из которых выполняет одну функцию решаемой задачи.
- Поставленная задача решается путем взаимодействия утилит (команд) за счет последовательной обработки данных каждой из них.
- При разработке этих утилит ориентируются на их максимально независимое использование, что позволяет применять их для решения других задач. Таким образом, постепенно создаются инструментальные средства для дальнейшего универсального применения.
- Большинство утилит представляют собой фильтры, которые читают входную информацию из стандартного потока STDIN (по умолчанию это клавиатура), обрабатывают ее и передают результат в виде потока текста через стандартный поток STDOUT на другое устройство (по умолчанию — на экран).
- Утилиты-команды соединяются друг с другом в сценариях операционной системы посредством перенаправления ввода/вывода и создания программных конвейеров (направление выходного потока одной программы на вход другой).

В UNIX-системах различные системные параметры, настройки и конфигурации приложений хранятся в обычных текстовых файлах. Вообще, в мире UNIX текстовые данные принято использовать в качестве универсального представления информации (за исключением, конечно, исполняемых файлов в машинном коде, зашифрованных файлов и архивов, графических или мультимедийных файлов). Поэтому в UNIX-системах имеется множество мощных стандартных утилит для обработки текста (`grep`, `sed`, `awk`, `sort`, `cut`, `tail` и т. д.).

Еще один базовый принцип UNIX выражается фразой "все есть файл". В виде файлов (виртуальных) представляются каналы связи программ друг с другом, периферийные устройства и виртуальные устройства, эмулирующие ядро операционной системы. Например, чтобы узнать параметры процессора на компьютере с Linux, достаточно вывести с помощью команды `cat` содержимое виртуального файла cpuinfo в каталоге /proc:

```
cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 58
```

```
model name      : Intel(R) Core(TM) i7-3770S CPU @ 3.10GHz
stepping        : 9
microcode       : 0xffffffff
cpu MHz         : 3092.986
cache size      : 8192 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
                  pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx rdtscp
                  lm constant_tsc rep_good nopl xtopology cpuid pni pclmulqdq
                  ssse3 cx16 pcid sse4_1 sse4_2 popcnt xsave avx f16c rdrand
                  hypervisor lahf_lm pti ssbd ibrs ibpb stibp fsgsbase smep
                  erms xsaveopt flush_l1d arch_capabilities
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds
                  swapgs itlb_multihit srbds
bogomips        : 6185.97
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

Таким образом, несложные задачи автоматизации в UNIX-подобных системах в конечном итоге сводятся к работе с файловой системой и обработке текста. Такие задачи удобно решать путем последовательного применения нескольких команд, объединенных текстовыми потоками в конвейер (рис. 1.1).

Рис. 1.1. Преобразование текстовых данных в конвейере команд

В более сложных случаях, когда парой-тройкой конвейеров не обойтись, пишут программы-сценарии на поддерживаемом оболочкой командно-скриптовом языке (эти языки родом из 1970-х годов, их никак нельзя назвать удобными или интуитивно понятными) или на одном из универсальных интерпретируемых языков, самым популярным из которых сейчас является Python.

Особенности автоматизации в Windows

В Microsoft Windows, которая в течение уже нескольких десятилетий остается самой распространенной операционной системой для персональных компьютеров и ноутбуков, имеется несколько стандартных инструментов автоматизации, которые сильно различаются между собой. Попробуем разобраться в этом многообразии, понять причины возникновения этих инструментов и в конечном итоге определиться, какие средства автоматизации подойдут именно нам.

Начнем с того, что Windows, в отличие от UNIX, имеет *API-ориентированную архитектуру*: операционная система состоит из огромного числа подсистем и компонентов, доступ к которым осуществляется посредством специфических API (Application Program Interface, интерфейс прикладного программирования), т. е. для управления ими необходимы специальные приложения. Для упрощения работы с такой сложной системой управляемые элементы группируются в *структурированные объекты*.

В этом состоит принципиальное отличие Windows от UNIX-подобных систем, которые ориентированы на использование текста для представления всего, что только можно. Если UNIX-системы можно администрировать, просто работая с текстовыми конфигурационными файлами, то в Windows приходится использовать различные API, обращаясь к свойствам и методам множества внутренних объектных моделей Windows (.NET Framework, WMI, WSH, ADSI, CDO и т. д.).

Таким образом, для эффективной автоматизации работы в Windows недостаточно только овладеть инструментами для работы с файловой системой и обработки текстовых файлов, как в UNIX-подобных системах. Дополнительно к этому необходимо иметь возможность быстро, просто и единообразно обращаться из командной строки или сценариев к различным внутренним объектам системы.

В последней на момент написания данной книги версии Windows 10 есть три стандартных (т. е. не требующих дополнительной установки) инструмента автоматизации работы в операционной системе:

1. Командная строка интерпретатора cmd.exe, поддерживающего простой (хотя и довольно специфический) язык пакетных файлов.
2. Сервер сценариев Windows Script Host (WSH), позволяющий запускать сценарии на языках VBScript и JScript.
3. Командная оболочка и среда выполнения сценариев Windows PowerShell.

Замечание

Также в Windows 10 с помощью подсистемы WSL (Windows Subsystem for Linux) можно установить из Microsoft Store дистрибутив Linux-системы (например, Debian и Ubuntu) и пользоваться ее командной оболочкой для работы с файловой системой Windows или запуска Windows-утилит. Аналогично в этом случае можно будет запускать утилиты Linux в командной строке Windows. Такой смешанный подход дает новые интересные возможности для автоматизации, однако он выходит за рамки данной книги, здесь мы рассматривать его не будем.

Эти инструменты появились не одновременно (командные файлы поддерживаются в операционных системах Microsoft еще с 1980-х годов, а PowerShell появился на четверть века позже), они имеют разные возможности и в разной степени удовлетворяют требованиям к идеальному инструменту автоматизации в Windows, которые мы обсуждали ранее.

Замечание

При желании и соответствующем опыте можно писать сценарии для автоматизации работы в операционной системе на универсальных скриптовых языках (например, на Python, Ruby или JavaScript для Node js). Однако в этом случае платформы для поддержки этих языков и библиотеки для работы с внутренними объектами Windows придется устанавливать и настраивать дополнительно, по умолчанию в Windows их нет Преимущества стандартных инструментов автоматизации Windows (особенно PowerShell) — гарантированное наличие в любой версии Windows (это очень важно для системных администраторов, управляющих множеством рабочих станций), простота изучения и возможность работать с внутренними объектами операционной системы наиболее естественным образом.

Командный интерпретатор cmd.exe

В стандартной командной строке Windows, использующей интерпретатор cmd.exe, можно выполнять два типа команд:

- внутренние, которые выполняются непосредственно самим интерпретатором;
- внешние, представленные исполняемыми утилитами.

В качестве внутренних команд реализованы операции с файловой системой и основные алгоритмические операторы (`if`, `for`), позволяющие использовать командные файлы с расширением bat или cmd в качестве несложных программ-сценариев.

Интерпретатор cmd.exe поддерживает заимствованную из UNIX концепцию стандартных текстовых потоков и механизм переназначения устройств ввода/вывода (рис. 1.2).

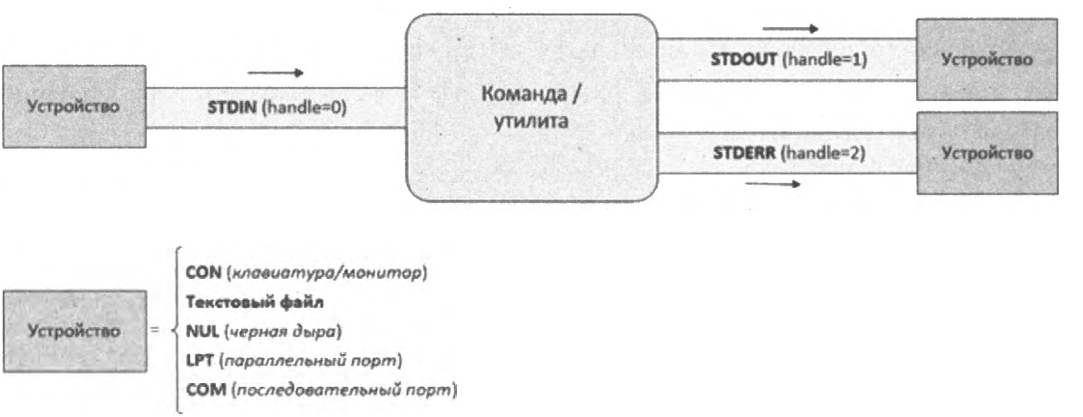


Рис. 1.2. Стандартные текстовые потоки и устройства в Windows

Таким образом, в Windows, как и в UNIX, текстовая информация (поток текста) может передаваться по конвейеру между несколькими командами (рис. 1.3).

Оболочка командной строки cmd.exe и командные файлы, поддерживаемые этим интерпретатором, — это самые простые и универсальные инструменты для автоматизации несложных задач, которые будут надежно работать в любой конфигурации операционной системы.

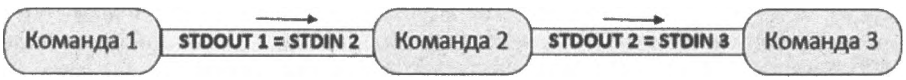


Рис. 1.3. Конвейеризация команд интерпретатора cmd exe

Для создания командных файлов нужно знать совсем немного: особенности синтаксиса интерпретатора cmd.exe и возможности нескольких десятков внутренних команд и стандартных утилит командной строки. Конкретные ключи, с которыми нужно запускать эти команды и утилиты, можно посмотреть во встроенной в них справке.

Листинг 1.1. Пример командного файла (интерпретатор cmd.exe)

```
@echo off
rem ****************************************************************
rem * Имя: sort_reps.bat
rem * Язык: cmd.exe
rem * Описание: Формирование списка каталогов с отчетами
rem ****************************************************************
if exist temp.txt del temp.txt
if exist odb.txt del odb.txt

for /D %%f in (d:\odb\202101*) do (
   dir /s/b %%f\od*.htm  >> temp.txt)

for /f "delims=\ tokens=4" %%i in ('sort /+17 temp.txt') do (
  echo %%i >> odb.txt)
del temp.txt
```

Однако командные файлы cmd.exe нельзя признать полноценными сценариями, позволяющими задействовать все возможности Windows. Основная проблема состоит в том, что в Windows нет стандартных удобных утилит для работы с внутренними объектными моделями. Кроме того, сам язык командных файлов недостаточно развит: не поддерживается работа с массивами или структурами, нет логических операций в условиях, нельзя напрямую объявить функцию.

Объяснение этим недостаткам одно — историческое. Дело в том, что командный интерпретатор cmd.exe появился еще в начале 1990-х годов, это одно из самых первых приложений для операционной системы Windows NT, на базе которой строятся все последующие версии Windows. Основной задачей тогда было обеспечение об-

ратной совместимости с командным интерпретатором command.com предыдущей операционной системы MS-DOS, в которой не было никаких внутренних объектных систем. В то время компания Microsoft ориентировалась на максимально широкую аудиторию неискушенных пользователей, не желающих вникать в технические детали работы системы. Основные усилия разработчиков направлялись на улучшение графической оболочки системы для более комфортной работы непрофессионалов, а не на создание рабочей среды для специалистов (системных администраторов и разработчиков) или опытных пользователей. Поэтому командная строка и консольные утилиты Windows долгое время отставали от аналогичных средств в UNIX-системах.

Сервер сценариев Windows Script Host

Для решения перечисленных выше проблем с функциональностью и удобством традиционных командных файлов Windows были возможны два варианта:

- Расширить количество стандартных утилит командной строки и существенно расширить язык интерпретатора cmd.exe, превратив его в полноценный алгоритмический язык с поддержкой объектно-ориентированного программирования (при этом доработанный командный интерпретатор должен был поддерживать и прежний синтаксис языка командных файлов).
- Выбрать для сценариев Windows другой простой язык программирования, который был бы привычен пользователям этой операционной системы, поддерживал все стандартные алгоритмические возможности и конструкции, а также позволял работать с внутренними объектами операционной системы.

Разработчики Microsoft пошли по второму пути и в начале 2000-х годов предложили использовать в качестве стандартного языка сценариев Windows язык VBScript (Visual Basic Script Edititon). Этот язык был выбран по следующим критериям.

- *Развитость.* VBScript — полноценный алгоритмический язык, имеющий встроенные функции и методы для анализа символьных строк, выполнения математических операций, обработки исключительных ситуаций и т. д.
- *Простота для изучения.* VBScript базируется на языке BASIC (сокращение от *англ.* Begginner's All-purpose Symbolic Instruction Code), который был разработан в 1964 г. в Дартмутском колледже как специальный инструмент для написания программ непрофессионалами. Одно из основных требований, которое учитывалось при создании этого языка, — простота в использовании для начинающих.
- *Распространенность.* BASIC — стандартный язык программирования для домашних персональных компьютеров в 1970–1990-е гг. Первый интерпретатор этого языка компании Microsoft (Altair BASIC) был разработан еще в 1975 г., вплоть до конца 1980-х годов BASIC устанавливался практически на всех персональных компьютерах (как на IBM PC-совместимых, так и на компьютерах Apple). В операционных системах Windows 95/98 интерпретатор и среда разработки Microsoft QBasic были установлены в качестве стандартного компонента системы.

- *Универсальность.* Модификации языка BASIC в то время использовались в разных областях операционной системы Windows.
  - Стандартные автономные приложения Windows (язык Visual Basic является частью среды разработки Microsoft Visual Studio).
  - Макросы для автоматизации работы в офисных приложениях Microsoft (язык Visual Basic for Applications (VBA)).
  - Сценарии в динамических клиентских HTML-страницах (язык VBScript поддерживался браузером Internet Explorer).
  - Сценарии в серверных ASP-страницах для веб-сервера Internet Information Server.
- *Расширяемость за счет внешних объектов.* В языке VBScript имеется функция `CreateObject`, позволяющая из сценариев обращаться ко многим внешним объектам, зарегистрированным в операционной системе, и использовать их функциональность.
- *Возможность добавления графического интерфейса (HTML-формы).* Сценарии на языке VBScript можно встраивать в HTML-формы, создавая так называемые приложения HTA (HTML Applications).

Языки VBScript и JScript (реализация стандарта ECMAScript компании Microsoft, аналогичен языку JavaScript) уже применялись в сценариях, встроенных в HTML-страницы, а интерпретаторы этих языков (динамические библиотеки vbscript.dll и jscript.dll) являлись стандартными компонентами Windows. Однако напрямую использовать сценарии внутри HTML-страниц в качестве сценариев операционной системы было нельзя, т. к. все вложенные сценарии выполняются с учетом действующих на веб-браузер политик безопасности, запрещающих доступ к локальной файловой системе компьютера.

Для запуска сценариев на языках VBScript и JScript непосредственно в операционной системе, без браузера и HTML-оболочки, был создан специальный сервер сценариев Windows Script Host (WSH), который включается по умолчанию во все версии Windows начиная с 2000 г.

Сценарии WSH представляют собой текстовые файлы с расширением vbs (язык VBScript) или js (язык JScript). Они могут запускаться в текстовом (консольном) режиме с помощью приложения cscript.exe или в графическом режиме с помощью файла wscript.exe.

Листинг 1.2. Пример сценария на языке VBScript

```
'*****************************************************************
'* Имя: MakeShortcut.vbs
'* Язык: VBScript
'* Описание: Создание ярлыков из сценария
'*****************************************************************
Dim WshShell,oUrlLink
```

```
' Создаем объект WshShell
Set WshShell=WScript.CreateObject("WScript.Shell")
' Создаем ярлык на сетевой ресурс
Set oUrlLink = WshShell.CreateShortcut("Microsoft Web Site.URL")
' Устанавливаем URL
oUrlLink.TargetPath = "http://www.microsoft.com"
' Сохраняем ярлык
oUrlLink.Save
```

WSH решает проблему отсутствия в языках сценариев VBScript и JScript специальных функций для управления операционной системой (например, в этих языках нет функций для работы с файловой системой). Всю фактическую работу в сценарии делают внешние объекты, а WSH служит посредником между ними и интерпретатором языка (рис. 1.4).

Рис. 1.4. Использование внешних объектов из сценариев WSH

Отметим, что кроме стандартных для Microsoft языков VBScript и JScript для написания сценариев WSH можно использовать и другие языки (например, Perl или Python), для которых нужно дополнительно устанавливать соответствующий модуль поддержки.

Таким образом, сценарии WSH складываются из двух составляющих (рис. 1.5):

- элементы выбранного языка сценариев;
- внешние объекты, не зависящие от этого языка.

Возможность использования полноценных языков сценариев и различных внешних объектов, имеющихся в операционной системе, сделала Windows Script Host мощным инструментом автоматизации. Однако через несколько лет стало ясно, что для пользователей среднего уровня и начинающих системных администраторов написание нетривиальных практических сценариев оказывается непростой задачей. Если для создания командных файлов достаточно было знать нехитрый синтаксис языка, поддерживаемого cmd.exe, и возможности пары десятков команд, то для

грамотного составления сценариев WSH приходилось как изучать выбранный язык сценариев (хотя в случае VBScript это совсем не сложно), так и разбираться с нюансами подключения к объектным моделям для управления операционной системой разных типов (WMI, ADSI и т. п.) и искать справочную информацию об этих объектах во внешних источниках.

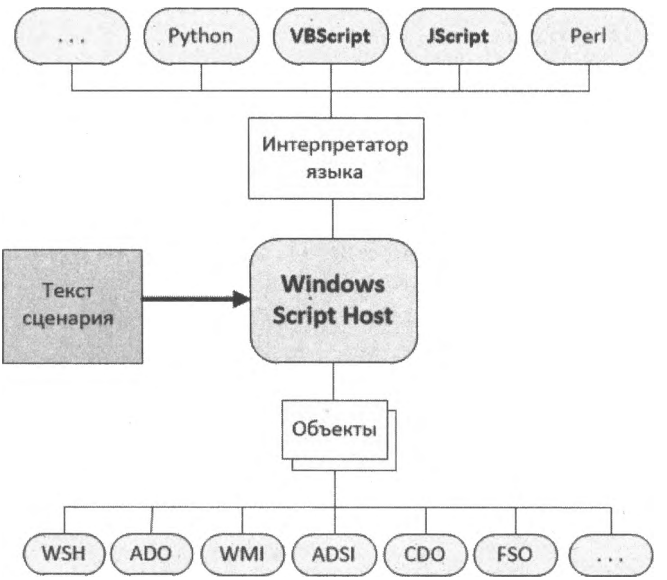


Рис. 1.5. Сценарии WSH интерпретаторы языков и объектные модели Windows

Кроме того, Windows Script Host — это не командная оболочка, а только среда выполнения сценариев, поэтому в WSH отсутствуют некоторые удобные возможности интерпретатора cmd.exe. Например, для запуска внешней исполняемой утилиты недостаточно просто указать соответствующее имя файла, необходимо пользоваться дополнительными методами `run()` и `exec()` объекта `WshShell`.

Строки сценариев WSH нельзя выполнять построчно в интерактивном режиме. Если в cmd.exe можно на лету в командной строке запускать отдельные команды или конвейеры из команд и направлять их вывод в текстовый файл, то в WSH для этого потребуется создать отдельный файл со сценарием, записать туда нужные команды и запустить сценарий из операционной системы.

Наконец, сценарии WSH представляют собой довольно серьезную потенциальную угрозу с точки зрения безопасности, известно большое количество вирусов, использующих WSH для выполнения деструктивных действий.

Хотя WSH остается стандартным компонентом Windows, языки VBScript и JScript в настоящее время утратили былую популярность, они не развиваются (последняя версия сервера сценариев WSH вышла в 2006 г.).

Оболочка и среда выполнения сценариев PowerShell

Осознавая проблемные места в командных файлах cmd.exe и сценариях Windows Script Host, компания Microsoft приняла решение о разработке совершенно новой мощной среды для написания сценариев и работы в командной строке. Этот инструмент впервые публично был представлен в 2003 г. под кодовым названием Monad, а затем был переименован в PowerShell и с 2006 г. используется в Windows. В 2017 г. Microsoft открыла исходный код PowerShell, теперь это проект Open Source (репозиторий проекта размещен на GitHub: **https://github.com/PowerShell/PowerShell**). Оболочка стала кросс-платформенной, с ней можно работать не только в Windows, но и в macOS и Linux.

Изначально главной и наиболее амбициозной целью разработчиков PowerShell было создание среды для написания и выполнения сценариев, которая наилучшим образом подходила бы для операционной системы Windows и была бы более функциональной, расширяемой и простой в использовании, чем какой-либо аналогичный продукт для любой другой операционной системы.

В первую очередь эта среда должна была подходить для решения задач, стоящих перед системными администраторами (тем самым Windows получила бы дополнительное преимущество в борьбе за сектор корпоративных платформ), а также удовлетворять требованиям разработчиков программного обеспечения, предоставляя им средства для быстрой реализации интерфейсов управления к создаваемым приложениям.

Для достижения этих целей были решены следующие задачи:

- *обеспечение прямого доступа из командной строки к объектам COM, WMI и .NET Framework.* В PowerShell присутствуют команды, позволяющие в интерактивном режиме работать с COM-объектами, а также с экземплярами классов, определенных в информационных схемах WMI и .NET Framework;
- *организация работы с произвольными источниками данных в командной строке по принципу файловой системы.* Например, навигация по системному реестру или хранилищу цифровых сертификатов выполняется из командной строки с помощью аналога команды `CD` интерпретатора cmd.exe;
- *разработка интуитивно понятной унифицированной структуры встроенных команд, основанной на их функциональном назначении.* В PowerShell имена всех внутренних команд (они называются *командлетами*) соответствуют шаблону "глагол — существительное", например `Get-Process` (получить информацию о процессе), `Stop-Service` (остановить службу), `Clear-Host` (очистить экран консоли) и т. д. Для одинаковых параметров внутренних команд используются стандартные имена, структура параметров во всех командах идентична, все команды обрабатываются одним синтаксическим анализатором. В результате облегчается запоминание и изучение команд;
- *обеспечение возможности расширения встроенного набора команд.* Внутренние команды PowerShell могут дополняться командами, которые создаются пользователями и полностью интегрируются в оболочку;

- *организация поддержки знакомых команд из других оболочек.* В PowerShell на уровне псевдонимов собственных внутренних команд поддерживаются наиболее часто используемые стандартные команды из оболочки cmd.exe и UNIX-оболочек. Например, если пользователь, привыкший работать с UNIX-оболочкой, выполнит команду `ls`, то он получит ожидаемый результат: список файлов в текущем каталоге (то же самое относится к команде `dir`);
- *реализация автоматического завершения при вводе с клавиатуры имен команд, их параметров, а также имен файлов и папок.* Данная возможность значительно упрощает и ускоряет ввод команд с клавиатуры.

Интерактивный сеанс в PowerShell аналогичен работе в других оболочках командной строки. Язык PowerShell несложен для изучения, писать на нем сценарии, обращающиеся к внешним объектам, проще, чем на VBScript или JScript. Отдельное внимание было уделено вопросам безопасности при работе со сценариями (например, запустить сценарий можно только с указанием полного пути к нему, а по умолчанию запуск сценариев PowerShell в системе вообще запрещен).

Главной особенностью среды PowerShell, отличающей ее от всех других оболочек командной строки, является ее *ориентация на объекты*. Единицей обработки и передачи информации в PowerShell является *объект* (данные вместе со свойственными им методами), а не строки текста, как в других оболочках. В силу этого работать в PowerShell становится проще, чем в традиционных оболочках, т. к. не нужно выполнять дополнительных манипуляций по выделению нужной информации из символьного потока.

Для поддержки объектов разработчики PowerShell решили не изобретать ничего нового и воспользоваться унифицированной объектной моделью .NET Framework. Таким образом, язык PowerShell является одним из .NET-языков программирования (рис. 1.6). При этом PowerShell не похож на другие .NET-языки типа VB.NET или C#, это уникальный язык, позволяющий совмещать в сценариях императивный и декларативный стили программирования.

Решение воспользоваться .NET Framework было принято по нескольким причинам. Во-первых, платформа .NET Framework повсеместно используется при разработке программного обеспечения для Windows и представляет, в частности, общую информационную схему, с помощью которой разные компоненты операционной системы могут обмениваться данными друг с другом.

Во-вторых, объектная модель .NET Framework является *самодокументируемой*: каждый .NET-объект содержит информацию о своей структуре. При интерактивной работе это очень полезно, т. к. появляется возможность непосредственно из командной строки выполнить запрос к определенному объекту и увидеть описание его свойств и методов, т. е. понять, какие именно манипуляции можно проделать с данным объектом, не изучая дополнительной документации с его описанием.

В-третьих, работая в оболочке с объектами, можно с помощью их свойств и методов легко получать нужные данные, не занимаясь разбором и анализом символьной информации, как это происходит во всех традиционных оболочках командной

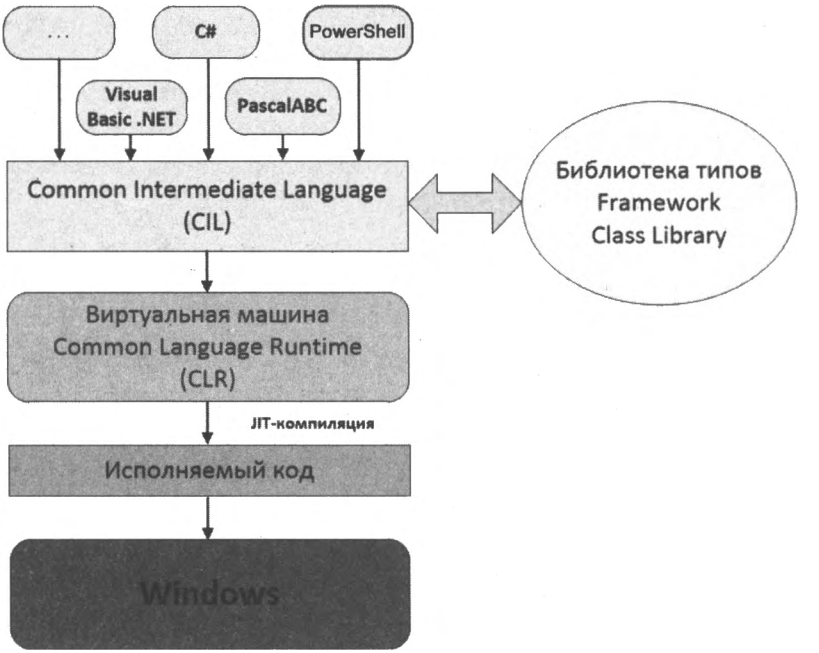


Рис. 1.6. Схема работы программ на NET-языках

строки, ориентированных на текст. Рассмотрим пример. В Windows есть утилита tasklist, которая выдает информацию о процессах, запущенных в системе:

```
C:\>tasklist
Имя образа                PID   Имя сессии        № сеанса   Память
========================= ====== ================ ======== ========
System Idle Process           0                          0       16 КБ
System                        4                          0       32 КБ
smss.exe                    560                          0       68 КБ
csrss.exe                   628                          0    4 336 КБ
winlogon.exe                652                          0    3 780 КБ
services.exe                696                          0    1 380 КБ
lsass.exe                   708                          0    1 696 КБ
svchost.exe                 876                          0    1 164 КБ
svchost.exe                 944                          0    1 260 КБ
svchost.exe                1040                          0   10 144 КБ
svchost.exe                1076                          0      744 КБ
svchost.exe                1204                          0      800 КБ
spoolsv.exe                1296                          0    1 996 КБ
kavsvc.exe                 1516                          0    9 952 КБ
klnagent.exe               1660                          0    5 304 КБ
klswd.exe                  1684                          0       64 КБ
```

Предположим, что мы в командном файле интерпретатора cmd.exe с помощью этой утилиты хотим определить, сколько оперативной памяти тратит процесс `kavsvc.exe`. Для этого нужно выделить из выходного потока команды `tasklist` соответствую-

щую строку, извлечь из нее подстроку, содержащую нужное число, и убрать пробелы между разрядами (при этом следует учесть, что в зависимости от настроек операционной системы разделителем разрядов может быть не пробел, а другой символ).

В PowerShell аналогичная задача решается с помощью команды `Get-Process`, которая возвращает коллекцию объектов, каждый из которых соответствует одному запущенному процессу. Для определения памяти, затрачиваемой процессом `kavsvc.exe`, нет необходимости в дополнительных манипуляциях с текстом, достаточно просто взять значение свойства `WS` объекта, соответствующего данному процессу.

Наконец, объектная модель .NET Framework позволяет PowerShell напрямую использовать функциональность различных своих библиотек, являющихся частью данной платформы. Например, чтобы узнать, каким днем недели было 9 ноября 1974 г., в PowerShell можно выполнить следующую команду:

```
(Get-Date "09.11.1974").DayOfWeek
```

В этом случае команда `Get-Date` возвращает .NET-объект типа `DateTime`, имеющий свойство `DayOfWeek`, при обращении к которому вычисляется день недели для заданной даты. Таким образом, разработчикам PowerShell не нужно создавать специальную библиотеку для работы с датами и временем — они просто берут готовое решение в .NET Framework.

Итоги

- Командная строка и сценарии позволяют автоматизировать работу в операционной системе, тратить меньше времени на рутинные операции.
- Операционные системы UNIX и Windows базируются на принципиально разных подходах. Принципы UNIX — "все есть файл" и "все есть текст", принцип Windows — "все есть объект". Поэтому в UNIX-подобных системах (macOS, Linux) задачи автоматизации сводятся к манипуляциям с файловой системой и обработке текста, а для управления Windows дополнительно необходимо уметь работать с внутренними объектными моделями.
- В Windows есть три стандартных инструмента автоматизации: командная строка и пакетные файлы интерпретатора cmd.exe, сценарии Windows Script Host (WSH) на языках VBScript и JScript, командная строка и среда выполнения сценариев Windows PowerShell.
- Компания Microsoft прекратила развитие cmd и WSH, они поддерживаются для работы разработанных ранее скриптов. Основной инструмент в настоящем и будущем — PowerShell.
- В отличие от всех других оболочек, команды PowerShell взаимодействуют друг с другом не с помощью символьных строк, а через объекты. Язык PowerShell является одним из .NET-языков и позволяет совмещать в сценариях императивный и декларативный стили программирования.
- Исходный код PowerShell открыт, оболочка является кросс-платформенной, ее можно установить в macOS и Linux.

ГЛАВА 2

Терминал, консоль и командная оболочка

В предыдущей главе мы говорили о режиме командной строки в Windows и о двух разных командных оболочках этой операционной системы, упоминали работу в терминале и консольные приложения. Для эффективной и удобной работы в командной строке важно выбрать правильный инструмент, а для этого необходимо четко понимать, что означают эти термины.

Само слово *терминал* происходит от глагола terminate (завершить, положить конец) и означает "оконечное устройство", т. е. устройство, находящееся на одном конце в процессе коммуникации с другим устройством (сервером). Задача терминала — отправлять вводимый с клавиатуры текст на сервер и отображать текстовые ответы от сервера.

Первые терминалы в 1950–1960-х годах подключались по телефонным линиям к большим компьютерам. Они представляли собой электрические печатные машинки — телетайпы (teletypewriters, TTY). Вводимые команды и ответы сервера телетайпы построчно печатали на рулоне бумаги (рис. 2.1).

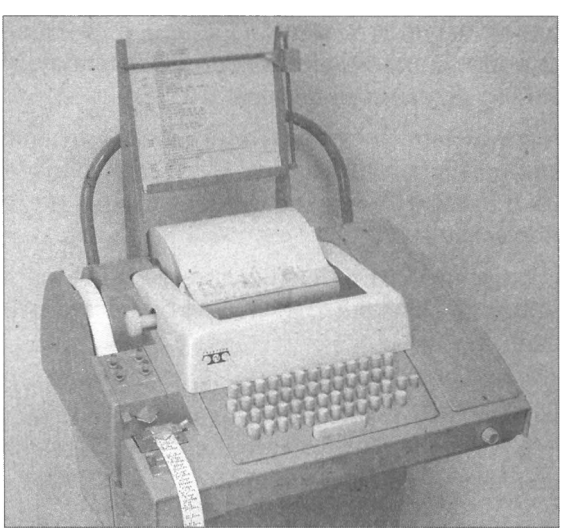


Рис. 2.1. Терминал в виде электромеханического телетайпа

В дальнейшем вместо принтера в компьютерных терминалах стали использовать CRT-дисплеи. Само устройство со встроенной клавиатурой и монитором получило название *консоль*.

Замечание

Слово *консоль* появилось задолго до изобретения компьютеров и означало кронштейн или подставку под что-либо Затем консолями стали называть пульты с кнопками и переключателями для управления электрическими устройствами (рис 2.2).

Рис. 2.2. Терминал (консоль) DEC VT100 (1978 г)

Таким образом, консоль — это устройство, а терминал — это коммуникационная программа внутри консоли, выполняющая следующие задачи:

1. Распознавание и прием символов, набираемых пользователем на клавиатуре.
2. Формирование из принимаемых символов одной строки с учетом управляющих кодов (например, перемещение курсора или удаление символа).
3. Обмен текстом с компьютером с помощью коммуникационного оборудования через прямое физическое соединение или по линиям связи.
4. Отображение полученного от компьютера текста на дисплее. Терминал должен распознавать и обрабатывать так называемые *управляющие последовательности ANSI* (ANSI escape codes) для задания формата, цвета и других опций вывода текста (например, перемещения курсора в произвольную позицию на экране).

Данный механизм, когда терминал посылает символы программе, запущенной на компьютере, и отображает на экране полученный от этой программы текст, остается до настоящего времени базовой моделью взаимодействия человека с компьютером через командную строку.

С середины 1980-х годов аппаратные консоли и терминалы стали вытесняться персональными компьютерами и сейчас в операционных системах терминалами и кон-

солями называют программные аналоги TTY. Это приложения, позволяющие вводить символьные команды, отправлять эти команды другому процессу и отображать на экране поступающие от этого процесса строки текста.

Сами команды, поступающие от терминала, исполняются специальной программой, которая называется *командной оболочкой* (command shell). В зависимости от полученной команды оболочка выполняет определенные действия и генерирует символьные строки, которые посылаются обратно терминалу для отображения на экране.

Для каждой операционной системы существуют разные оболочки, отличающиеся набором команд. В UNIX-подобных системах (Linux и macOS) чаще всего пользуются оболочками bash, zsh, fish, tsh. В состав Windows 10, как мы видели в *главе 1*, входят две стандартные оболочки: cmd.exe (командная строка) и Windows PowerShell.

Важно понимать, что командные оболочки не имеют собственного пользовательского интерфейса, это не терминалы. С одной и той же оболочкой можно работать с помощью разных терминалов, а из одного терминала можно запускать разные оболочки.

Терминалы в Windows

Терминал и командная оболочка — это два приложения, запущенные на одном компьютере, которые должны обмениваться текстом друг с другом. В UNIX-системах эта задача решается с помощью псевдотерминала (Pseudo TTY, PTY), предоставляющего два виртуальных устройства: подчиненное (slave) и ведущее (master). К ведущему псевдоустройству подключается терминальное приложение, а к подчиненному — командная оболочка или другое консольное приложение. Когда терминальное приложение отправляет символы на вход ведущего устройства, они перенаправляются на выход подчиненного устройства. Строки текста, которые генерирует оболочка (shell) или приложение (app), посылаются на вход подчиненного устройства и перенаправляются на выход ведущего устройства (рис. 2.3).

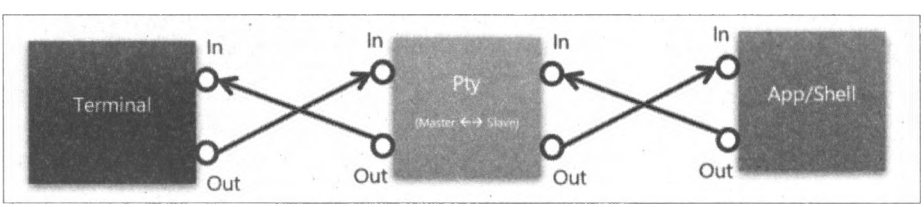


Рис. 2.3. Взаимодействие терминала и командной оболочки через псевдотерминал в UNIX-системах

При этом подчиненное устройство эмулирует поведение аппаратного терминала, выделяя из входного потока определенные управляющие комбинации символов и отсылая соответствующие этим комбинациям сигналы в подключенное приложение.

Данный механизм используется в UNIX-системах с 1980-х годов, поддерживая работу терминальных приложений (в том числе полноэкранных консольных утилит, учитывающих управляющие ANSI-последовательности).

В Windows эмуляция терминала реализована по-другому. Напомним, что все современные версии Windows — это потомки оригинальной операционной системы Windows NT, разработанной компанией Microsoft в начале 1990-х годов. Работа в режиме командной строки в Windows NT осуществлялась с помощью *терминального приложения Windows Console*, связанного с оболочкой (командным интерпретатором) cmd.exe. Эта стандартная консоль продолжает использоваться в Windows в течение почти 30 лет.

Стандартная консоль Windows

По своей функциональности терминал Windows Console (ConHost) похож на традиционный UNIX-терминал, однако спроектирован он по-другому, без использования псевдотерминала PTY. Если пользователь хочет работать с командной строкой в UNIX-системе, он сначала запускает терминал, который устанавливает связь с командной оболочкой по умолчанию. В Windows пользователь запускает не сам терминал (начиная с Windows 7, он представлен файлом conhost.exe), а исполняемый файл с оболочкой (cmd.exe или powershell.exe) или другой консольной утилитой (рис. 2.4). При этом операционная система автоматически подключает эту оболочку или утилиту к уже существующему или новому экземпляру консоли ConHost.

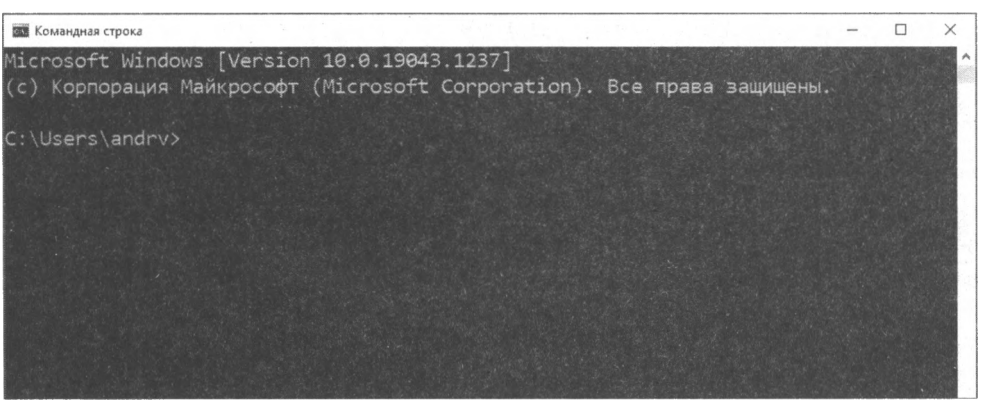


Рис. 2.4. Командная строка (cmd exe) в стандартной консоли Windows

При этом процесс ConHost взаимодействует с оболочкой с помощью системных сообщений (I/O Control messages, IOCTL messages) через специальный драйвер, а не посредством потоков текста через псевдотерминал PTY, как в UNIX-системах.

Итак, механизм поддержки консольных приложений в Windows имеет существенные отличия от подхода, принятого в UNIX-системах.

- Для любой запущенной командной оболочки или утилиты командной строки операционная система Windows всегда назначает в качестве терминала стандартную консоль ConHost (conhost.exe).

- Коммуникационные каналы для связи консоли с утилитами создает сама операционная система.
- Утилиты командной строки взаимодействуют с консолью с помощью вызовов функций из Win32 Console API, позволяющих, например, задавать цвет текста и фона, перемещать курсор в нужную позицию и т. д.

Эти архитектурные особенности консоли Windows со временем стали источником проблем.

- Затрудняется создание альтернативных программных терминалов для Windows. Разработчикам таких приложений (отметим здесь эмуляторы терминала ConEmu, Cmder, Console2, Hyper) приходилось запускать стандартную консоль Windows в окне за пределами видимости монитора, посылать в нее символы, которые вводил пользователь, затем считывать из этого окна возвращаемые оболочкой строки и отображать их в собственном окне.
- Windows-утилиты, использующие для работы с консолью функции Win32 Console API, сложно перенести на другие платформы, в которых управление консолью осуществляется посредством символьных ANSI-последовательностей. Эта несогласованность средств управления консолью также затрудняет перенос UNIX-утилит в Windows.
- В Windows 10 поддерживается подсистема WSL (Windows Subsystem for Linux), позволяющая установить внутри Windows один из дистрибутивов Linux и пользоваться оболочками и стандартными утилитами из этой операционной системы. При этом управляющие ANSI-последовательности, поступающие от этих утилит, в консоли Windows могут обрабатываться некорректно.
- Возникают трудности с реализацией подключений к командной строке Windows на сервере с удаленных терминалов на других компьютерах. Действительно, в этом случае нужно вызывать функции Win32 Console API удаленно, причем клиентская машина может работать не под Windows.

Для решения подобных проблем разработчики компании Microsoft в 2018 г. добавили в Windows инфраструктуру псевдоконсоли ConPTY, сохранив при этом обратную совместимость с традиционной Windows-консолью ConHost. Теперь ConHost в полной мере поддерживает утилиты, использующие кодировку UTF-8 и ANSI-последовательности для управления терминалом (благодаря этому, например, в консоли Windows корректно отображаются полноэкранные Linux-утилиты, запускаемые в сеансе WSL). Кроме того, теперь стало возможным создание альтернативных терминалов для Windows, работающих через ConPTY.

Windows Terminal

В 2019 г. компания Microsoft представила новый усовершенствованный терминал для Windows, который так и называется — Windows Terminal. Это программное обеспечение с открытым исходным кодом (исходники размещены на GitHub: **https://github.com/microsoft/terminal**), которое активно развивается и позиционируется Microsoft в качестве основного средства для работы с различными оболочками и утилитами командной строки в современных версиях Windows.

Перечислим основные возможности, реализованные в Windows Terminal.

- Поддержка вкладок для открытия нескольких сеансов командных оболочек в одном окне.
- Разделение окна на несколько независимых панелей, в которых можно открыть разные оболочки.
- Наличие панели для ввода или выбора команды для управления терминалом.
- Поддержка ANSI-последовательностей для управления отображением текста в терминале.
- Полноценная поддержка кодировки UTF-8.
- Использование 24-битного цвета.
- Поддержка графических тем и полупрозрачного фона в терминале.
- Поддержка разных режимов отображения окна терминала.
- Выделение гиперссылок в отображаемом в терминале тексте.
- Копирование текста в системный буфер обмена в форматах HTML и RTF.

Установка и запуск

Установить Windows Terminal проще всего из магазина приложений Microsoft Store (можно открыть с помощью ярлыка в меню **Пуск** или в браузере по ссылке **https://www.microsoft.com/ru-ru/store/apps/windows**) (рис. 2.5).

Другие варианты установки описаны в репозитории терминала на GitHub (**https://github.com/microsoft/terminal**).

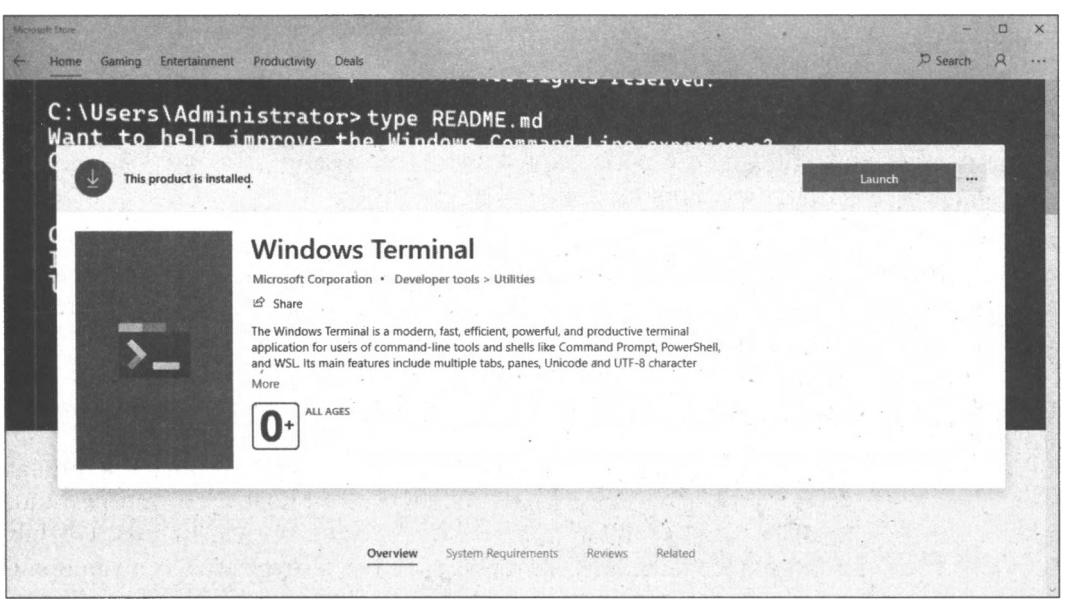


Рис. 2.5. Windows Terminal в Microsoft Store

После установки в меню **Пуск** появится ярлык **Windows Terminal**.

Для запуска терминала можно воспользоваться ярлыком **Windows Terminal** в меню **Пуск** или нажать комбинацию клавиш <Win>+<R> и в окне **Выполнить** ввести имя `wt` запускного файла терминала. В результате откроется новое окно терминала с оболочкой Windows PowerShell (рис. 2.6).

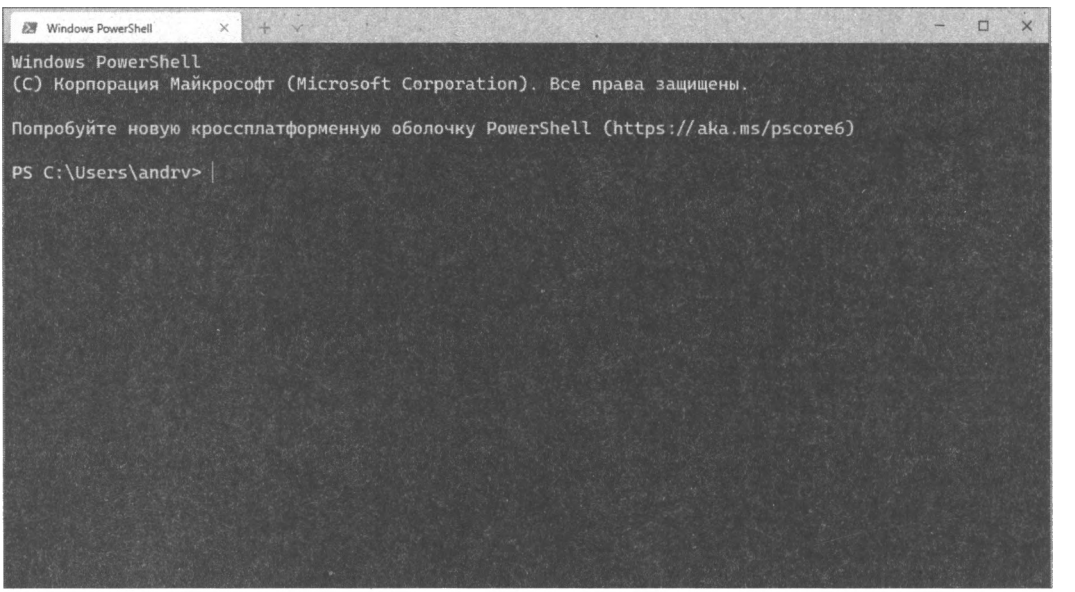


Рис. 2.6. Оболочка Windows PowerShell в Windows Terminal

Рассмотрим новые возможностями Windows Terminal, которых не было в предыдущем терминале.

Работа с вкладками

Для создания новой вкладки с оболочкой PowerShell нужно щелкнуть мышью на значке + или нажать комбинацию клавиш <Ctrl>+<Shift>+<t>.

Если щелкнуть на значке ∨, то откроется список, где можно выбрать другой профиль (командную оболочку) для новой вкладки (рис. 2.7):

- стандартная командная строка Command Prompt (интерпретатор cmd.exe);
- Windows PowerShell;
- оболочка bash операционной системы Linux (если подсистема WSL установлена и настроена).

Обратите внимание, что для каждого профиля в этом списке указана комбинация клавиш (<Ctrl>+<Shift>+<1>, <Ctrl>+<Shift>+<2> и т. д.), по которой его можно открыть в новой вкладке, не пользуясь мышью.

Переключаться между открытыми вкладками можно с помощью комбинации клавиш <Ctrl>+<Tab>.

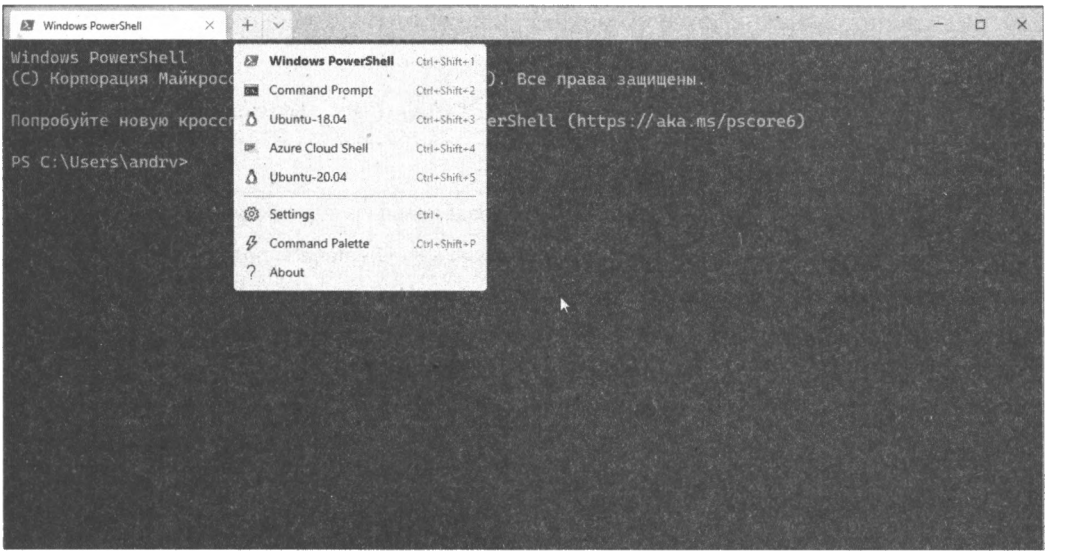


Рис. 2.7. Выбор командной оболочки для новой вкладки терминала

Разделение окна на несколько панелей

Окно в каждой вкладке можно разбить на несколько панелей как по вертикали, так и по горизонтали. Это позволяет просматривать сразу несколько сеансов работы с командной строкой, не переключаясь между вкладками (рис. 2.8).

При разделении по вертикали новая панель откроется справа от выбранной панели, а при разделении по горизонтали — под выбранной панелью.

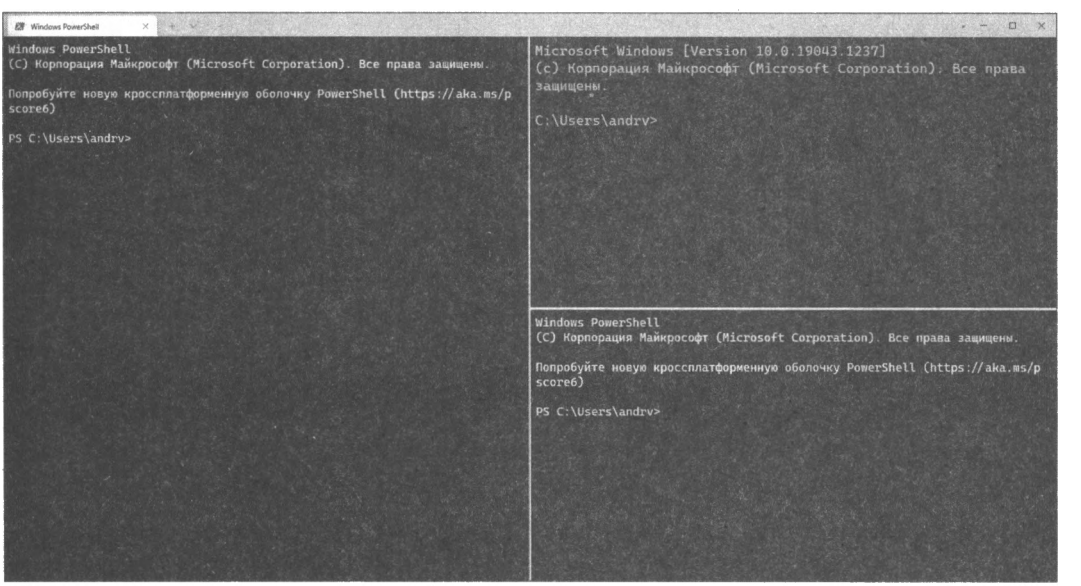


Рис. 2.8. Разделение окна терминала на независимые панели

Для разделения окна на панели можно использовать клавиатурные комбинации (табл. 2.1).

Таблица 2.1. *Разделение окна на панели*

Клавиатурная комбинация	Действие
<Alt>+<Shift>+<+>	Вертикальное разделение панели профиля по умолчанию
<Alt>+<Shift>+<->	Горизонтальное разделение панели профиля по умолчанию
<Alt> и щелчок мышью на нужном профиле	Создание новой панели для выбранной оболочки
<Alt>+<Shift>+<d>	Дублирование текущей панели
<Ctrl>+<Shift>+<w>	Закрытие текущей панели

Если во вкладке открыты несколько панелей, то переключаться между ними можно либо с помощью мыши, либо с помощью клавиш со стрелками, удерживая при этом клавишу <Alt>. Можно изменять размер панелей, удерживая одновременно клавиши <Alt>+<Shift> и используя клавиши со стрелками.

Использование палитры команд

Различные команды управления терминалом можно не только выполнять с помощью клавиатурных комбинаций, но и вводить или выбирать в палитре команд, которая вызывается нажатием клавиш <Ctrl>+<Shift>+<p> (рис. 2.9).

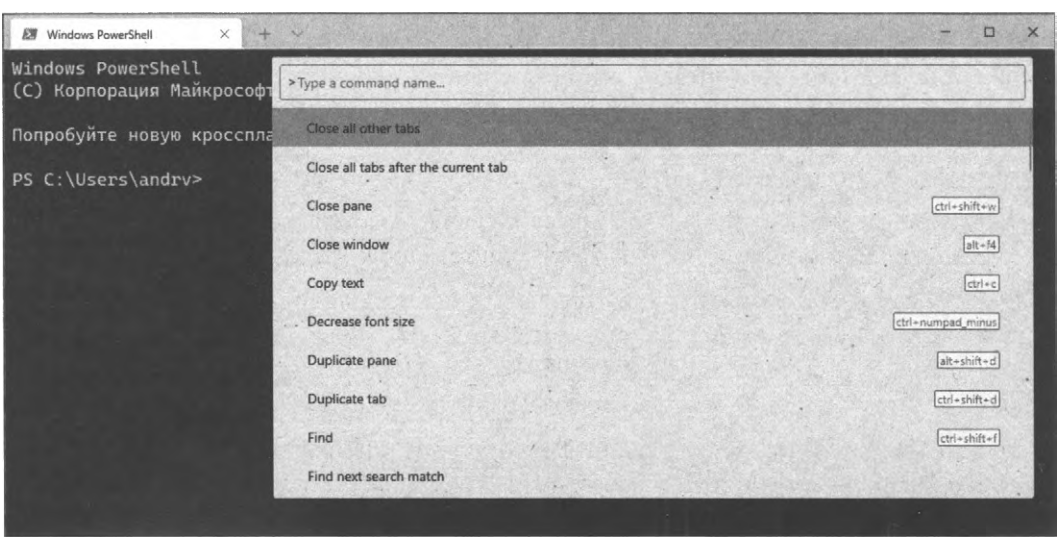


Рис. 2.9. Палитра команд Windows Terminal

Запуск терминала с аргументами командной строки

Для запуска нового экземпляра Windows Terminal из командной строки используется команда `wt`. При этом с помощью дополнительных аргументов-команд можно

задать текущий каталог, в котором будет открыт терминал, автоматически создать новые вкладки или разделить вкладку на несколько панелей. Команды для терминала разделяются между собой точкой с запятой.

Например, следующая команда:

```
wt -d C:\ ; split-pane -p "Windows PowerShell" ; split-pane -H wsl.exe
```

запустит новый терминал с тремя панелями на вкладке:

1. Сначала в корне диска C:\ открывается профиль по умолчанию, PowerShell (команда `-d C:\`).
2. Затем панель делится по вертикали и в правой половине открывается PowerShell в домашнем каталоге пользователя (команда `split-pane -p "Windows PowerShell"`).
3. Наконец, правая панель делится по горизонтали и в нижней половине открывается интерпретатор bash подсистемы WSL (команда `split-pane -H wsl.exe`).
4. Описание других аргументов, которые можно указывать при запуске терминала, можно найти в документации на сайте Microsoft (**https://docs.microsoft.com/ru-ru/windows/terminal/command-line-arguments**).

Итоги

- Аппаратные терминалы обеспечивали общение человека с компьютером через командную строку в самом начале компьютерной эпохи. До появления персональных компьютеров для работы с серверами в режиме командной строки использовались консоли — устройства со встроенной клавиатурой и монитором, в которых запускался программный аналог терминала.
- Приложение-терминал позволяет вводить символьные команды, отправляет их другому процессу и отображает поступающие от этого процесса строки текста.
- Команды, поступающие от терминала, выполняются командной оболочкой. Оболочки не имеют собственного пользовательского интерфейса.
- Для персонального компьютера терминал и командная оболочка — это два приложения, запущенные на одном компьютере, которые должны обмениваться текстом друг с другом. С одной и той же оболочкой можно работать с помощью разных терминалов, а из одного терминала можно запускать разные оболочки.
- В Windows включены две командные оболочки: стандартная командная строка cmd и Windows PowerShell.
- В качестве стандартного терминала в Windows используется консоль ConHost, к которой операционная система автоматически подключает запущенную оболочку или консольное приложение. Дополнительно в Windows можно установить усовершенствованный Windows Terminal. Это приложение с открытым исходным кодом, которое активно развивается и позиционируется Microsoft в качестве основного инструмента для работы с оболочками командной строки в современных версиях Windows.

ГЛАВА 3

Первые шаги в PowerShell. Основные понятия

Итак, приступим к работе в оболочке командной строки Windows PowerShell.

Запуск оболочки PowerShell

Для начала нового сеанса работы в PowerShell можно выбрать пункт **Windows PowerShell** в меню **Пуск**. Другой вариант запуска оболочки — пункт **Выполнить** в меню **Пуск**, ввести имя файла `powershell` и нажать кнопку **OK**.

В результате откроется новое окно стандартной консоли Windows с приглашением вводить команды (рис. 3.1).

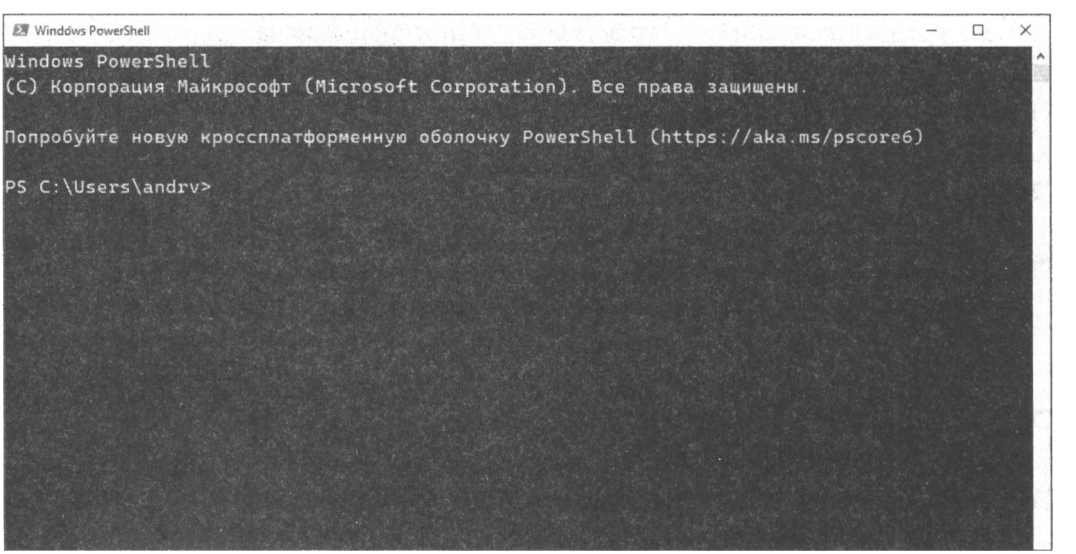


Рис. 3.1. PowerShell в стандартной консоли Windows

Если вы предпочитаете работать в Windows Terminal (напомним, что его нужно предварительно установить в систему), то сеанс PowerShell с приглашением на ввод команды откроется по умолчанию после запуска терминала (рис. 3.2).

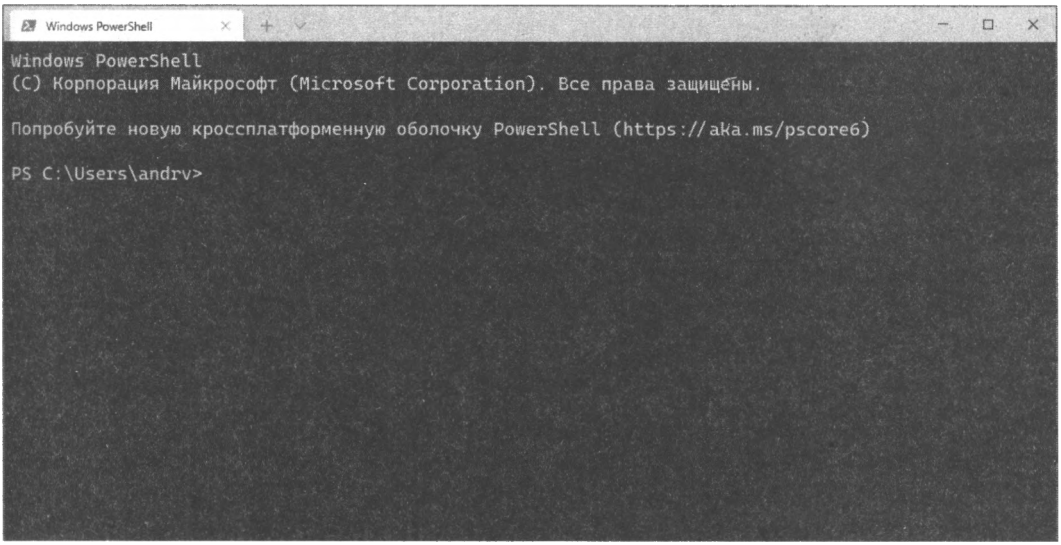


Рис. 3.2. PowerShell в Windows Terminal

Обратите внимание на предложение попробовать новую кросс-платформенную оболочку PowerShell. На момент написания книги (2021 г.) в Windows 10 по умолчанию установлена оболочка Windows PowerShell версии 5.1, базирующаяся на специфической для Windows платформе .NET Framework версии 4.*x*. Следующие версии PowerShell (шестая и седьмая) используют платформу .NET Core/.NET 5, которую можно установить в Windows, macOS и Linux. Таким образом, PowerShell 6/7 — это новая оболочка (запускать ее нужно командой `pwsh`, а не `powershell`), которая может работать в разных операционных системах, но ее нужно устанавливать отдельно.

Работают ли знакомые команды

Выполним первую команду в PowerShell — выведем содержимое текущего каталога. В оболочке cmd.exe для этого служит команда `dir`, в оболочке bash — команда `ls`. Попробуем выполнить эти же команды в PowerShell (отметим, что в PowerShell команды обрабатываются без учета регистра):

```
PS C:\Users\andrv> dir

    Каталог: C:\Users\andrv

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        15.02.2021     20:30                .config
d-----        10.10.2020     22:02                .dbus-keyrings
d-----        11.02.2021     16:12                .local
d-----        03.03.2021     20:10                .quokka
d-----        11.10.2020      7:48                .ssh
d-----        28.09.2020     23:28                .vscode
```

```
d-----        03.03.2021     20:04                .wallaby
d-r---        28.09.2020     16:18                3D Objects
d-r---        28.09.2020     16:18                Contacts
d-----        01.04.2021     19:41                Documents
d-r---        14.05.2021     20:58                Downloads
d-r---        28.09.2020     16:18                Favorites
d-r---        28.09.2020     16:18                Links
d-r---        28.09.2020     16:18                Music
dar--l        14.05.2021     20:45                OneDrive
d-r---        28.09.2020     16:18                Saved Games
d-r---        28.09.2020     16:18                Searches
d-r---        04.10.2020     23:52                Videos
-a----        18.10.2020     17:20             93 .gitconfig

PS C:\Users\andrv> ls

    Каталог: C:\Users\andrv

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        15.02.2021     20:30                .config
d-----        10.10.2020     22:02                .dbus-keyrings
d-----        11.02.2021     16:12                .local
d-----        03.03.2021     20:10                .quokka
d-----        11.10.2020      7:48                .ssh
d-----        28.09.2020     23:28                .vscode
d-----        03.03.2021     20:04                .wallaby
d-r---        28.09.2020     16:18                3D Objects
d-r---        28.09.2020     16:18                Contacts
d-----        01.04.2021     19:41                Documents
d-r---        14.05.2021     20:58                Downloads
d-r---        28.09.2020     16:18                Favorites
d-r---        28.09.2020     16:18                Links
d-r---        28.09.2020     16:18                Music
dar--l        14.05.2021     20:45                OneDrive
d-r---        28.09.2020     16:18                Saved Games
d-r---        28.09.2020     16:18                Searches
d-r---        04.10.2020     23:52                Videos
-a----        18.10.2020     17:20             93 .gitconfig
```

Как видим, обе команды срабатывают одинаково, выводя на экран список файлов и подкаталогов в текущем (домашнем) каталоге, а также дополнительную информация о них.

Оболочки cmd.exe и bash поддерживают перенаправление вывода — с помощью символа > (знак "больше") результат выполнения команд можно выводить не на экран, а в текстовый файл. Попробуем сделать то же самое в PowerShell:

```
PS C:\Users\andrv> dir > dir.txt
```

Данная команда выполнилась без каких-либо сообщений на экране. Выведем теперь на экран содержимое файла dir.txt, воспользовавшись для этого командой type:

```
PS C:\Users\andrv> type c:\dir.txt

    Каталог: C:\Users\andrv

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        15.02.2021     20:30                .config
d-----        10.10.2020     22:02                .dbus-keyrings
d-----        11.02.2021     16:12                .local
d-----        03.03.2021     20:10                .quokka
d-----        11.10.2020      7:48                .ssh
d-----        28.09.2020     23:28                .vscode
d-----        03.03.2021     20:04                .wallaby
d-r---        28.09.2020     16:18                3D Objects
d-r---        28.09.2020     16:18                Contacts
d-----        01.04.2021     19:41                Documents
d-r---        14.05.2021     20:58                Downloads
d-r---        28.09.2020     16:18                Favorites
d-r---        28.09.2020     16:18                Links
d-r---        28.09.2020     16:18                Music
dar--l        14.05.2021     20:45                OneDrive
d-r---        28.09.2020     16:18                Saved Games
d-r---        28.09.2020     16:18                Searches
d-r---        04.10.2020     23:52                Videos
-a----        18.10.2020     17:20             93 .gitconfig
```

Замечание

Если файл, в который с помощью символа > перенаправляется вывод, уже существует, то его содержимое будет перезаписано. Для перенаправления вывода в режиме добавления информации в конец существующего файла нужно использовать символы >>. Перенаправление ввода с помощью символа < в PowerShell не поддерживается.

Итак, начало положено: мы выполнили в новой оболочке несколько знакомых команд и убедились, что PowerShell поддерживает перенаправление выходной информации в текстовый файл.

Вычисление выражений

Кроме выполнения команд в PowerShell можно вычислять выражения, например пользоваться оболочкой как калькулятором. Например:

```
PS C:\Users\andrv> 2+3
5
```

```
PS C:\Users\andrv> 10-2*3
4
PS C:\Users\andrv> (10-2)*3
24
```

Как видим, результат вычислений сразу же выводится на экран, нет необходимости использовать для этого какую-либо специальную команду (эту особенность PowerShell следует запомнить на будущее).

Предыдущие примеры показывают, что PowerShell справляется с вычислением целочисленных выражений, в том числе со скобками. Проверим выражение, результатом которого является число с плавающей точкой:

```
PS C:\Users\andrv> 10/3
3.33333333333333
```

Здесь тоже все в порядке, результат вычислен верно. На самом деле в PowerShell можно выполнять и более сложные вычисления с использованием различных математических функций. Для этого используются методы .NET-класса `System.Math`. Например, следующая команда вычисляет и выводит на экран корень квадратный из числа 169:

```
PS C:\Users\andrv> [System.Math]::Sqrt(169)
13
```

В PowerShell можно результат вычисления выражения сохранять в переменной и пользоваться этой переменной в других выражениях. Например:

```
PS C:\Users\andrv> $a = 10/2
PS C:\Users\andrv> $a
5
PS C:\Users\andrv> $a * 3
15
```

Отметим здесь, что в PowerShell имена переменных должны начинаться со знака доллара (`$`); более подробно вопросы, связанные с переменными, будут обсуждаться в *главе 7*.

Типы команд PowerShell

Обычно в оболочках все команды разделялись на *внутренние*, которые распознаются и выполняются непосредственно соответствующим интерпретатором, и *внешние*, которые представляют собой отдельные исполняемые модули. В cmd.exe, например, к внутренним относятся команды `dir` и `copy`, а к внешним — `xcopy` и `more`. Кроме этого, оболочки поддерживают сценарии на каком-либо языке, позволяющие выполнять команды в пакетном режиме.

В оболочке PowerShell поддерживаются команды четырех типов: командлеты, функции, сценарии и внешние исполняемые файлы.

Командлеты

Первый тип команд PowerShell — так называемые *командлеты* (cmdlet), представляющие собой внутренние команды PowerShell. Почему для названия внутренних команд используется такое странное слово? Дело в том, что командлеты не являются полностью встроенными в PowerShell, как внутренние команды cmd.exe, но они также и не могут быть запущены вне оболочки, как обычные исполняемые утилиты. Поэтому для их наименования был предложен новый термин.

Командлет представляет собой класс .NET, порожденный от базового класса `Cmdlet`. Единый базовый класс `Cmdlet` гарантирует совместимый синтаксис всех командлетов, а также автоматизирует анализ параметров командной строки и описание синтаксиса командлетов для встроенной справки.

Данный тип команд компилируется в динамическую библиотеку (DLL) и подгружается к процессу PowerShell во время запуска оболочки (т. е. сами по себе командлеты не могут быть запущены как приложения, но в них содержатся исполняемые объекты). Так как компилированный код подгружается к процессу оболочки при запуске PowerShell, данный тип команд выполняется наиболее эффективно.

Командлеты можно считать в определенном смысле аналогом внутренних команд традиционных оболочек, хотя, в отличие от внутренних команд, новые командлеты могут быть добавлены в систему в любое время. Это делает PowerShell расширяемой оболочкой и позволяет Microsoft и другим производителям программного обеспечения добавлять в PowerShell командлеты для управления своими приложениями и службами (например, Microsoft SQL Server или Microsoft Exchange).

> ***Замечание***
>
> Подробное рассмотрение процесса создания командлетов выходит за рамки данной книги; мы в дальнейшем будем пользоваться только командлетами, входящими в стандартную поставку PowerShell.

Командлеты могут быть очень простыми или очень сложными, но каждый из них разрабатывается для решения одной узкой задачи. Работа с командлетами становится по-настоящему эффективной при использовании их композиции, когда объекты передаются от одного командлета другому по конвейеру (подробнее процедура конвейеризации объектов обсуждается в *главе 5*).

Имена и структура командлетов

Имена командлетов всегда соответствуют шаблону "глагол-существительное", где глагол задает определенное действие, а существительное в единственном числе определяет объект, над которым это действие будет совершено ("действие — объект"). Это соглашение значительно упрощает запоминание и использование командлетов. Например, для получения информации о процессе служит командлет `Get-Process`, для остановки запущенной службы — командлет `Stop-Service`, для очистки экрана консоли — командлет `Clear-Host` и т. д.

Дефис является частью имени командлета, т. е., например, `Get-Process` не является комбинацией команды `Get` и команды `Process` — это имя целиком состоит из слова `Get-Process`.

Имена командлетов нечувствительны к регистру символов. Принято писать оба слова в имени с заглавных букв для улучшения читаемости кода, но это необязательно.

При вводе имен командлетов в командной строке PowerShell можно пользоваться автодополнением. Если набрать несколько первых символов и нажимать клавишу <Tab>, то система будет по очереди подставлять полные имена всех подходящих командлетов.

Командлеты могут иметь *параметры* — элементы, предоставляющие дополнительную информацию. Параметры либо определяют элементы, с которыми должна работать команда, либо определяют, каким образом будет работать командлет. Обратиться к параметрам можно по имени, перед которым ставится дефис (-), или по позиции (в последнем случае интерпретация параметра будет выполняться в зависимости от его местоположения в командной строке).

> ***Замечание***
>
> В командах оболочки cmd exe для выделения имен параметров часто применяются символы / или \ (например, `dir /s /b`), в оболочке PowerShell для параметров командлетов данные символы не используются

Поддерживаются три типа параметров командлетов, в общем случае синтаксис имеет следующую структуру:

имя_командлета -параметр1 -параметр2 аргумент1 аргумент2

Здесь *-параметр1* — параметр, не имеющий значения (подобные параметры часто называют *переключателями*); *-параметр2* — имя параметра, имеющего значение *аргумент1*; *аргумент2* — параметр, не имеющий имени (или аргумент).

Имена параметров, как и имена командлетов, нечувствительны к регистру символов и для них доступно автодополнение при вводе по нажатии клавиши <Tab>.

В качестве примера переключателя рассмотрим параметр `-Recurse` командлета `Get-ChildItem`.

> ***Замечание***
>
> Для краткости команд вместо командлета `Get-ChildItem` мы будем применять его псевдоним `dir` (более подробно вопросы, связанные с псевдонимы командлетов, обсуждаются в следующих разделах).

Переключатель `-Recurse`, если он указан, распространяет действие команды не только на определенный каталог, но и на все его подкаталоги. Например, следующий командлет выведет информацию обо всех файлах, которые находятся в каталоге C:\Program Files\Internet Explorer или в одном из его подкаталогов и имеют имя, удовлетворяющее маске ie* (как видите, никакого аргумента после параметра `-Recurse` не указано):

```
PS C:\Users\andrv> dir -Recurse -Filter ie* 'C:\Program Files\
Internet Explorer\'

    Directory: C:\Program Files\Internet Explorer

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----         12/7/2019  12:09 PM         515072 iediagcmd.exe
-a----         12/7/2019  12:09 PM         504832 ieinstal.exe
-a----         12/7/2019  12:09 PM         224768 ielowutil.exe
-a----         12/7/2019  12:09 PM         421888 IEShims.dll
-a----         12/7/2019  12:47 AM         819136 iexplore.exe

    Directory: C:\Program Files\Internet Explorer\en-US

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----         1/12/2021  11:02 AM           2560 ieinstal.exe.mui
-a----         1/12/2021  11:03 AM           5632 iexplore.exe.mui

    Directory: C:\Program Files\Internet Explorer\ru-RU

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----         12/7/2019   5:34 PM           2560 ieinstal.exe.mui
-a----         12/7/2019   5:34 PM           6144 iexplore.exe.mui
```

В данном примере используется еще один параметр — `-Filter` с аргументом `ie*`, задающим маску файлов для поиска.

Отметим, что имена параметров не обязательно указывать полностью, однако при этом сокращенное имя должно однозначно определять соответствующий параметр. Например, вместо последнего командлета можно выполнить следующий сокращенный вариант, результат останется тем же:

```
PS C:\Users\andrv> dir -r -fi ie* 'C:\Program Files\Internet Explorer\'
```

Однако если попытаться имя параметра `-Filter` сократить до одного символа, то возникнет ошибка:

```
PS C:\Users\andrv> dir -r -f ie* 'C:\Program Files\Internet Explorer\'
Get-ChildItem : Не удается обработать параметр, т. к. имя параметра "f"
неоднозначно. Возможные совпадения: -Filter -Force.
строка:1 знак:8
+ dir -r -f ie* 'C:\Program Files\Internet Explorer\'
+        ~~
```

В нашем примере есть еще один аргумент, задающий путь к каталогу C:\Program Files\Internet Exlplorer. Как видите, параметр для этого аргумента не указан. На самом деле приведенная выше команда эквивалентна следующей:

```
PS C:\Users\andrv> dir -r -f ie* -path 'C:\Program Files\Internet Explorer\'
```

То есть в данном случае аргументу `'C:\Program Files\Internet Explorer\'` соответствует параметр `-Path`, однако командлет `Get-ChildItem` (псевдоним `dir`) устроен таким образом, что имя параметра `-Path` можно опускать — система сможет определить, что аргумент, задающий путь к каталогу файловой системы, относится именно к этому параметру.

Общие параметры командлетов

Напомним, что все командлеты являются потомками базового класса `Cmdlet`, в котором определены несколько параметров, обеспечивающих согласованный однородный интерфейс оболочки PowerShell.

В частности, некоторые параметры, называемые *общими*, поддерживаются всеми командлетами PowerShell (при этом на определенные командлеты подобные параметры могут никак не влиять). Основные общие параметры перечислены в табл. 3.1.

Таблица 3.1. *Общие параметры командлетов PowerShell*

Параметр	Описание
`-Verbose`	Тип Boolean. Включает режим вывода подробных сведений о выполняемой операции
`-Debug`	Тип Boolean. Включает режим создания подробного отчета об операции для отладки команды
`-ErrorAction`	Тип Enum. Определяет, какой будет реакция командлета на возникновение ошибки (подробнее обработка ошибок обсуждается в *главе 10*). Возможные значения: `Continue` (по умолчанию), `Stop`, `SilentlyContinue`, `Inquire`, `Ignore`
`-ErrorVariable`	Тип String. Определяет переменную, в которой будут сохраняться ошибки команды во время выполнения. Эта переменная создается дополнительно к переменной `$error`
`-OutVariable`	Тип String. Задает переменную, в которой будут сохраняться выходные данные команды во время выполнения
`-OutBuffer`	Тип Int32. Определяет количество хранящихся в буфере объектов перед вызовом следующего командлета в конвейере
`-WhatIf`	Тип Boolean. Предоставляет сведения об изменениях, которые произойдут в результате указанных действий, не производя самих этих действий. Данный параметр поддерживается командлетами в том случае, если они изменяют состояние системы
`-Confirm`	Тип Boolean. Запрашивает разрешение у пользователя на выполнение каких-либо действий, вносящих изменения в систему. Данный параметр поддерживается командлетами в том случае, если они изменяют состояние системы

Скажем дополнительно несколько слов о параметре `-WhatIf` из табл. 3.1. Этот параметр позволяет увидеть объекты, на которые будет действовать тот или иной командлет, не выполняя при этом самих действий. Например, следующая команда

позволяет увидеть, какие файлы в каталоге C:\Temp будут удалены при выполнении команды del (сами файлы при этом не удаляются):

```
PS C:\> del -WhatIf "C:\temp\*.*"
WhatIf: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\bidiSNMP.dll".
WhatIf: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\BiDiSNMP.ini".
WhatIf: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\Setup.exe".
WhatIf: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\XBASE.DLL".
WhatIf: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\XeroxLpr.dll".
WhatIf: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\XeroxPM.hlp".
WhatIf: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\XPmPrint.ini".
WhatIf: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\xrxipdis.dll".
WhatIf: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\XSNMX.DLL".
WhatIf: Выполнение операции "Удаление файла" над целевым объектом
"C:\temp\xv2p.dll".
```

Особенно полезным параметр -WhatIf может оказаться в тех случаях, когда диапазон объектов, на которые должен действовать командлет, определяется неявным образом (например, с помощью регулярных выражений).

Поиск командлетов

Чтобы просмотреть список командлетов, доступных в ходе текущего сеанса, нужно выполнить команду Get-Command. Если запустить Get-Command без параметров, то на экран будут выведены все доступные команды, в том числе функции и псевдонимы команд. Отфильтровать этот список, оставив в нем только командлеты, можно с помощью следующего конвейера команд (подробно конвейеры рассматриваются в *главе 5*):

```
PS C:\Users\andrv> Get-Command | Where-Object CommandType -eq 'Cmdlet' |
Format-Wide -Column 2

Add-AppvClientConnectionGroup          Add-AppvClientPackage
Add-AppvPublishingServer               Add-AppxPackage
Add-AppxProvisionedPackage             Add-AppxVolume
Add-Computer                           Add-Content
Add-History                            Add-JobTrigger
Add-KdsRootKey                         Add-LocalGroupMember
Add-Member                             Add-PSSnapin
Add-SignerRule                         Add-Type
. . .
```

Команда `Get-Command` имеет параметры `-Verb` и `-Noun`, позволяющие вывести командлеты с определенным глаголом или существительным соответственно. Например, посмотреть, с помощью каких команд PowerShell можно управлять запущенными в системе процессами, можно так:

```
PS C:\Users\andrv> Get-Command -Noun Process| Format-Wide -Column 2

Debug-Process                                   Get-Process
Start-Process                                   Stop-Process
Wait-Process
```

Следующая команда покажет, какие объекты можно получить с помощью команд с глаголом `Get`:

```
PS C:\Users\andrv> Get-Command -Verb Get | Format-Wide -Column 2

Get-AdlAnalyticsAccount                 Get-AdlAnalyticsComputePolicy
Get-AdlAnalyticsDataSource              Get-AdlAnalyticsFirewallRule
Get-AdlCatalogItem                      Get-AdlJob
Get-AdlJobPipeline                      Get-AdlJobRecurrence
Get-AdlStore                            Get-AdlStoreChildItem
Get-AdlStoreFirewallRule                Get-AdlStoreItem
. . .
```

Функции

Следующий тип команд PowerShell — функции. *Функция* — это блок кода на языке PowerShell, имеющий название и находящийся в памяти до завершения текущего сеанса командной оболочки. Анализ синтаксиса функции производится только один раз при ее объявлении (при повторном запуске функции подобный анализ не проводится).

Создадим простейшую функцию:

```
PS C:\Users\andrv> function MyFunc{"Всем привет!"}
```

Эта функция имеет имя `MyFunc`, в ее теле просто выводится строка "Всем привет!". Вызовем нашу функцию из командной строки. Для этого достаточно ввести ее имя:

```
PS C:\Users\andrv> MyFunc
Всем привет!
```

Функции, как и командлеты, поддерживают работу с параметрами. Например, определим функцию `MyFunc1` с одним формальным параметром:

```
PS C:\Users\andrv> function MyFunc1($a) {"Привет, $a!"}
```

Вызовем данную функцию с аргументом:

```
PS C:\Users\andrv> MyFunc1 Андрей
Привет, Андрей!
```

Как видим, для того чтобы передать аргументы в функцию, их нужно указывать через пробел после названия функции (между собой аргументы также разделяются

пробелами). Если вы работали с функциями в обычных языках программирования, где аргументы для функции указываются через запятую внутри круглых скобок, то синтаксис передачи аргументов PowerShell может показаться странным и непривычным. Но PowerShell — это оболочка, поэтому для функций используется тот же синтаксис, что и для команд.

Более подробно вопросы, связанные с функциями PowerShell, рассмотрены в *главе 9*.

Сценарии

Третий тип команд, поддерживаемый PowerShell, — это сценарии. *Сценарий* представляет собой блок кода на языке PowerShell, хранящийся во внешнем файле с расширением ps1. Анализ синтаксиса сценария производится при каждом его запуске.

Сценарии позволяют работать с PowerShell в пакетном режиме, т. е. заранее создать файл с нужными командами, определить логику работы с помощью различных управляющих инструкций языка PowerShell и пользоваться этим файлом как исполняемым модулем.

Более подробно вопросы, связанные со сценариями PowerShell, рассмотрены в *главе 9*.

Внешние исполняемые файлы

Последний тип команд, запускаемых в PowerShell, — *внешние исполняемые файлы*, которые выполняются операционной системой обычным образом. В частности, из оболочки Windows PowerShell можно запускать любые внешние утилиты командной строки (например, `xcopy`), просто указывая их имена.

Если нужно запустить файл, в имени которого (или в пути к нему) есть пробелы, то имя нужно заключить в одинарные или двойные кавычки и поставить перед именем знак амперсанда `&` (в PowerShell этот символ означает оператор вызова). Например, следующая команда запустит браузер Internet Explorer:

```
PS C:\> &'C:\Program Files\Internet Explorer\iexplore.exe'
```

Псевдонимы команд

Как отмечалось ранее, имена всех командлетов в PowerShell соответствуют шаблону "действие — объект" и могут быть довольно длинными, что затрудняет их быстрый набор. Механизм *псевдонимов*, реализованный в PowerShell, дает возможность пользователям выполнять команды по их альтернативным именам. В PowerShell заранее определено много псевдонимов, кроме этого, можно добавлять в систему собственные псевдонимы.

Например, мы уже несколько раз пользовались командой `dir`, которая в действительности является псевдонимом командлета `Get-ChildItem`. Убедиться в этом можно с помощью командлета `Get-Command` или `Get-Alias`:

```
PS C:\Users\andrv> Get-Command dir

CommandType     Name
-----------      ----
Alias           dir -> Get-ChildItem

PS C:\Users\andrv> Get-Alias dir

CommandType     Name
-----------      ----
Alias           dir -> Get-ChildItem
```

Псевдонимы в PowerShell делятся на два типа. Первый предназначен для совместимости имен с разными интерфейсами. Подобные псевдонимы позволяют пользователям, имеющим опыт работы с другими оболочками (cmd.exe или bash), использовать знакомые им имена команд для выполнения аналогичных операций в PowerShell. Это упрощает освоение новой оболочки, позволяя не тратить усилий на запоминание новых специфических команд PowerShell.

Например, пользователь хочет очистить экран. Если у него есть опыт работы с cmd.exe, то он, естественно, попробует выполнить команду `cls`. PowerShell при этом автоматически выполнит командлет `Clear-Host`, для которого `cls` является псевдонимом и который выполняет требуемое действие — очистку экрана.

Для пользователей cmd.exe в PowerShell определены псевдонимы `cd`, `cls`, `copy`, `del`, `dir`, `echo`, `erase`, `move`, `popd`, `pushd`, `ren`, `rmdir`, `sort`, `type`; для пользователей UNIX-подобных систем — псевдонимы `cat`, `chdir`, `clear`, `diff`, `h`, `history`, `kill`, `lp`, `ls`, `mount`, `ps`, `pwd`, `r`, `rm`, `sleep`, `tee`, `write`.

Псевдонимы второго типа (стандартные псевдонимы) в PowerShell предназначены для быстрого ввода команд. Такие псевдонимы образуются из имен командлетов, которым они соответствуют. Например, глагол `Get` сокращается до `g`, глагол `Set` сокращается до `s`, существительное `Location` сокращается до `l` и т. д. Таким образом, командлету `Set-Location` соответствует псевдоним `sl`, а командлету `Get-Location` — псевдоним `gl`.

Просмотреть список всех пседонимов, объявленных в системе, можно с помощью командлета `Get-Alias` без параметров:

```
PS C:\Users\andrv> Get-Alias

CommandType     Name
------------     ---
Alias           % -> ForEach-Object
Alias           ? -> Where-Object
Alias           ac -> Add-Content
Alias           asnp -> Add-PSSnapin
Alias           cat -> Get-Content
Alias           cd -> Set-Location
Alias           chdir -> Set-Location
```

```
Alias           clc -> Clear-Content
Alias           clear -> Clear-Host
Alias           clhy -> Clear-History
Alias           cli -> Clear-Item
Alias           clp -> Clear-ItemProperty
Alias           cls -> Clear-Host
Alias           clv -> Clear-Variable
Alias           cnsn -> Connect-PSSession
Alias           compare -> Compare-Object
Alias           copy -> Copy-Item
Alias           cp -> Copy-Item
Alias           cpi -> Copy-Item
Alias           cpp -> Copy-ItemProperty
Alias           curl -> Invoke-WebRequest
Alias           cvpa -> Convert-Path
Alias           dbp -> Disable-PSBreakpoint
Alias           del -> Remove-Item
Alias           diff -> Compare-Object
. . .
```

Задать собственный псевдоним можно с помощью командлета `Set-Alias` или `New-Alias`. Например, создадим для командлета `Get-ChildItem` псевдоним `list`:

```
PS C:\Users\andrv> Set-Alias -name list -value Get-ChildItem
```

Проверим, как этот псевдоним работает:

```
PS C:\Users\andrv> list

    Каталог: C:\Users\andrv

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        15.02.2021     20:30                .config
d-----        10.10.2020     22:02                .dbus-keyrings
d-----        11.02.2021     16:12                .local
d-----        03.03.2021     20:10                .quokka
d-----        11.10.2020      7:48                .ssh
d-----        28.09.2020     23:28                .vscode
d-----        03.03.2021     20:04                .wallaby
d-r---        28.09.2020     16:18                3D Objects
d-r---        28.09.2020     16:18                Contacts
d-----        01.04.2021     19:41                Documents
d-r---        14.05.2021     20:58                Downloads
d-r---        28.09.2020     16:18                Favorites
d-r---        28.09.2020     16:18                Links
d-r---        28.09.2020     16:18                Music
dar--l        14.05.2021     20:45                OneDrive
d-r---        28.09.2020     16:18                Saved Games
```

```
d-r---        28.09.2020     16:18                Searches
d-r---        04.10.2020     23:52                Videos
-a----        18.10.2020     17:20             93 .gitconfig
```

Одной команде могут соответствовать несколько псевдонимов (например, `Get-ChildItem` соответствуют псевдонимы `dir` и `ls`), но один псевдоним не может относиться к двум командам. Созданные псевдонимы можно переназначать, например:

```
PS C:\Users\andrv> Set-Alias list Get-Location
PS C:\Users\andrv> list
Path
----
C:\Users\andrv
```

В данном примере мы назначили командлету `Get-Location` псевдоним `list`, который до этого был привязан к командлету `Get-ChildItem`. Также из последнего примера видно, что названия параметров `-Name` и `-Value` в командлете `Set-Alias` можно опустить, однако в этом случае следует соблюдать порядок указания имен псевдонима (первый параметр) и соответствующего ему командлета (второй параметр).

Псевдонимы в PowerShell можно создавать не только для командлетов, но и для функций, сценариев или исполняемых файлов. Например, команда `Set-Alias np c:\windows\notepad.exe` создаст псевдоним `np`, соответствующий исполняемому файлу notepad.exe (Блокнот Windows).

В PowerShell нельзя создать псевдоним для команды с параметрами и значениями. Например, можно создать псевдоним для командлета `Set-Location`, но для команды `Set-Location C:\Windows\System32` этого сделать нельзя. Чтобы назначить псевдоним подобной команде, можно создать функцию, включающую эту команду, и определить псевдоним для этой функции. Например:

```
PS C:\Users\andrv> function CD32 {Set-Location C:\Windows\System32}
PS C:\Users\andrv> Set-Alias go cd32
PS C:\Users\andrv> go
PS C:\windows\system32>
```

Удалить псевдоним можно с помощью командлета `Remove-Item`, например:

```
PS C:\windows.1\system32> Remove-Item alias:go
PS C:\windows.1\system32> go
Условие "go" не распознано как командлет, функция, выполняемая программа или
файл сценария. Проверьте условие и повторите попытку.
строка:1 знак:2
+ go <<<<
```

Псевдонимы можно экспортировать в текстовый файл (командлет `Export-Alias`) и импортировать их из файла (командлет `Import-Alias`).

Благодаря псевдонимам синтаксис PowerShell приобретает гибкость: одни и те же команды могут быть записаны и очень кратко, и в развернутом виде. Это очень важно для языка, который одновременно является оболочкой командной строки и языком написания сценариев.

При интерактивной работе для ускорения ввода команд удобнее применять краткие псевдонимы и не полностью указывать имена параметров. Если же вы пишете сценарии, то лучше использовать полные названия командлетов и их параметров, это значительно упростит в дальнейшем разбор программного кода.

Диски PowerShell

Все мы давно привыкли к структуре файловой системы как совокупности вложенных папок (каталогов) и файлов. В операционной системе Windows интерфейс к такой структуре предоставляют Проводник Windows, оболочка cmd.exe, а также различные файловые менеджеры сторонних разработчиков. В UNIX-подобных системах понятия файлов и каталогов трактуются более широко, в качестве файлов здесь могут выступать различные компоненты системы (например, аппаратные устройства или сетевые подключения). Такой подход упрощает поиск нужных элементов в операционной системе. Оболочки командной строки или другие утилиты, обращающиеся к файлам в UNIX-системах, могут работать также и с этими компонентами.

Оболочка PowerShell в этом аспекте похожа на UNIX-системы, т. к. она позволяет просматривать различные хранилища данных и перемещаться по ним с использованием тех же привычных процедур, которые применяются для перемещения по файловой системе.

Помимо обычных локальных или сетевых дисков файловой системы (C:, D: и т. д.) оболочка поддерживает специальные (виртуальные) *диски PowerShell*, связанные с хранилищами данных разных типов. Например, корневому разделу реестра HKEY_LOCACL_MACHINE соответствует диск `HKLM:`, псевдонимам, доступным в текущем сеансе работы, соответствует диск `Alias:`, а хранилищу сертификатов цифровых подписей — диск `Cert:`.

> ***Замечание***
>
> В отличие от обычных локальных или сетевых дисков файловой системы, диски PowerShell доступны только из оболочки PowerShell, обратиться к ним из Проводника Windows нельзя. Кроме того, имена дисков PowerShell могут содержать более одного символа.

Для получения списка дисков PowerShell, доступных в текущем сеансе работы, нужно воспользоваться командлетом `Get-PSDrive`:

```
PS C:\Users\andrv> Get-PSDrive
Name        Used (GB)     Free (GB) Provider      Root
----        ---------     --------- --------      ----
Alias                               Alias
C             246.59        121.33 FileSystem    C:\
Cert                                Certificate   \
E                                   FileSystem    E:\
Env                                 Environment
Function                            Function
```

```
HKCU                    Registry    HKEY_CURRENT_USER
HKLM                    Registry    HKEY_LOCAL_MACHINE
Variable                Variable
WSMan                   WSMan
```

Как видим, командлет `Get-PSDrive` для каждого диска сообщает имя провайдера (колонка `Provider`), поддерживающего этот диск.

Провайдеры PowerShell

Провайдер PowerShell — это адаптер, предоставляющий пользователям PowerShell доступ к данным из определенного специализированного хранилища в согласованном формате, напоминающем формат обычных дисков файловой системы. Тем самым провайдеры PowerShell обеспечивают доступ к данным, к которым трудно обратиться через командную строку иными способами.

В оболочку PowerShell по умолчанию включены несколько встроенных провайдеров, которые можно использовать для доступа к различным хранилищам данных (табл. 3.2).

Таблица 3.2. *Встроенные провайдеры PowerShell*

Провайдер	Хранилище данных
`Alias`	Псевдонимы PowerShell
`Certificate`	Сертификаты X509 для цифровых подписей
`Environment`	Переменные среды Windows
`FileSystem`	Диски файловой системы, каталоги и файлы
`Function`	Функции PowerShell
`Registry`	Реестр Windows
`Variable`	Переменные PowerShell

Дополнительно к встроенным провайдерам можно создавать собственные провайдеры PowerShell и устанавливать провайдеры, созданные другими разработчиками.

Просмотреть список зарегистрированных в оболочке PowerShell провайдеров можно с помощью командлета `Get-PSProvider`:

```
PS C:\> Get-PSProvider

Name                Capabilities                Drives
----                ------------                ------
Registry            ShouldProcess, Transactions {HKLM, HKCU}
Alias               ShouldProcess               {Alias}
Environment         ShouldProcess               {Env}
FileSystem          Filter, ShouldProcess, C... {C, E}
Function            ShouldProcess               {Function}
```

```
Variable          ShouldProcess          {Variable}
Certificate       ShouldProcess          {Cert}
WSMan             Credentials            {WSMan}
```

В поле `Capabilities` здесь для каждого провайдера указаны возможности, которыми он обладает.

- `ShouldProcess`. Провайдер поддерживает общие параметры `-WhatIf` и `-Confirm`, с помощью которых можно проверить определенное действие перед его непосредственным выполнением.
- `Filter`. Провайдер поддерживает общий параметр `-Filter` для командлетов, работающих с содержимым провайдера.
- `Credentials`. Провайдер позволяет при подключении к хранилищам данных использовать другие учетные записи.
- `Transactions`. Провайдер поддерживает транзакции, позволяющие выполнять несколько операций по изменению данных, и в зависимости от результата подтверждать все операции или откатывать их.

Навигация по дискам PowerShell

Основное назначение провайдеров заключается в том, что они обеспечивают доступ к разнородным данным привычным согласованным образом. Используемая при этом модель представления данных основана на дисках файловой системы.

Предлагаемые провайдером данные можно просматривать и изменять так, как если бы они хранились в виде каталогов и файлов на жестком диске. Навигация по различным дискам PowerShell и просмотр содержимого этих дисков осуществляются с помощью одних и тех же базовых командлетов.

Работая с файловой системой, мы используем понятие *текущего* или *рабочего каталога*. К файлам в рабочем каталоге можно обращаться по имени, не указывая полного пути к ним.

Напомним, что в оболочке cmd.exe для определения или смены рабочего каталога служит команда `CD` (можно пользоваться и полным именем `CHDIR`):

```
C:\Users\andrv> cd /?
Вывод имени либо смена текущего каталога.
CHDIR [/D] [диск:][путь]
CHDIR [..]
CD [/D] [диск:][путь]
CD [..]
  ..  обозначает переход в родительский каталог.

Команда CD диск: отображает имя текущего каталога указанного диска.
Команда CD без параметров отображает имена текущих диска и каталога.
. . .
```

В оболочке PowerShell понятие рабочего (текущего) каталога распространяется и на диски PowerShell. Узнать путь к текущему каталогу можно с помощью команд-

лета Get-Location (псевдоним pwd данного командлета соответствует команде оболочки bash с аналогичной функциональностью):

```
PS C:\Users\andrv> Get-Location

Path
----
C:\Users\andrv
```

Для смены текущего каталога (в том числе для перехода на другой диск PowerShell) используется командлет Set-Location (псевдонимы cd, chdir, sl). Например:

```
PS C:\Users\andrv> Set-Location c:\
PS C:\> Set-Location HKLM:\Software
PS HKLM:\Software>
```

Как видим, при вводе командлета Set-Location на экран явно не выводится отзыв о его выполнении. При необходимости можно использовать параметр PassThru, выводящий после выполнения команды Set-Location путь к текущему каталогу:

```
PS HKLM:\Software> Set-Location 'C:\Program Files' -PassThru
Path
----
C:\Program Files
```

В оболочке cmd.exe и оболочках UNIX-систем кроме абсолютного задания путей к файлам и папкам поддерживаются относительные пути (относительно рабочего каталога). При этом текущему каталогу соответствует путь . (точка), родительскому каталогу текущего каталога — путь .. (две точки), а корневому каталогу текущего диска — путь \ (обратная косая черта). В PowerShell данная нотация сохраняется. Например (вместо командлета Set-Location мы используем его псевдоним cd):

```
PS C:\Program Files> cd \ -PassThru
Path
----
C:\

PS C:\> cd HKLM:\Software -PassThru
Path
----
HKLM:\Software

PS HKLM:\Software> cd .. -PassThru
Path
----
HKLM:\
```

Просмотр содержимого дисков и каталогов

Для просмотра элементов и контейнеров, находящихся на определенном диске PowerShell, можно воспользоваться командлетом Get-ChildItem (псевдонимы dir

и `ls`). Естественно, выводимая информация при этом будет зависеть от типа диска PowerShell. Для примера выполним командлет `Get-ChildItem` на диске, соответствующем корневому разделу реестра HKEY_CURRENT_USER, и на обычном диске файловой системы:

```
PS C:\Users\andrv> cd hkcu:
PS HKCU:\> dir

   Hive: HKEY_CURRENT_USER

Name                           Property
----                           --------
AppEvents
Console                        CtrlKeyShortcutsDisabled : 0
                               CursorColor              : 4294967295
                               CursorSize               : 25
                               DefaultBackground        : 4294967295
                               DefaultForeground        : 4294967295
                               EnableColorSelection     : 0
                               ExtendedEditKey          : 1
                               ExtendedEditKeyCustom    : 0
                               FilterOnPaste            : 1
                               ForceV2                  : 1
                               FullScreen               : 0
                               HistoryBufferSize        : 50
                               HistoryNoDup             : 0
                               InsertMode               : 1
                               LineSelection            : 1
                               LineWrap                 : 1
                               LoadConIme               : 1
                               NumberOfHistoryBuffers   : 4
                               PopupColors              : 245
                               QuickEdit                : 1
                               ScreenBufferSize         : 589889656
. . .

PS HKCU:\> cd 'C:\Program Files'
PS C:\Program Files> ls

    Directory: C:\Program Files

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         1/12/2021  11:45 AM                AMD
d-----         9/10/2019   9:10 AM                Android
d-----         1/12/2021  10:59 AM                ATI Technologies
d-----         8/15/2017   2:14 PM                Autodesk
d-----         2/25/2021   3:40 PM                CherryTree
```

```
d-----        11/20/2018   7:30 AM                Code Industry
d-----         1/12/2021  12:01 PM                Common Files
d-----         9/9/2019   3:16 PM                dotnet
da----         7/7/2017  10:36 AM                Far Manager
d-----        12/29/2017  10:33 AM                Git

. . .
```

Командлет `Get-ChildItem` имеет несколько параметров, позволяющих, в частности, просматривать содержимое вложенных каталогов, отображать скрытые элементы, применять фильтры для отображения файлов по маске и т. д. Более подробную информацию о возможностях и параметрах командлета `Get-ChildItem` можно найти в справочной системе PowerShell *(см. гл. 4)*.

Создание дисков

Помимо использования стандартных дисков PowerShell с помощью командлета `New-PSDrive` можно создавать собственные пользовательские диски. Для этого нужно указать три параметра: `-Name` (имя создаваемого диска PowerShell), `-PSProvider` (название провайдера, например `FileSystem` для диска файловой системы или `Registry` для диска, соответствующего разделу реестра) и путь к корневому каталогу нового диска.

Например, можно создать диск для определенной папки на жестком диске для того, чтобы обращаться к ней не по "настоящему" длинному пути, а просто по имени диска. Создадим диск PowerShell с именем `docs`, который будет соответствовать папке с документами определенного пользователя:

```
PS C:\Users\andrv> New-PSDrive -Name docs -PSProvider FileSystem -Root
C:\Users\andrv\Documents

Name   Used (GB)  Free (GB)  Provider       Root
----   ---------  ---------  --------      ---------
docs    0.00      121.32     FileSystem    C:\Users\andrv\Documents
```

Теперь обращаться к новому диску можно точно так же, как и к другим дискам PowerShell:

```
PS C:\Users\andrv> cd docs: -PassThru
Path
----
docs:\

PS docs:\> dir
    Directory: C:\Users\andrv\Documents

Mode            LastWriteTime         Length Name
----            -------------         ------ ----
d-----      7/7/2017  10:32 AM                FeedbackHub
d-----      9/9/2019   3:35 PM                IISExpress
```

```
d-----      9/9/2019   3:35 PM                My Web Sites
d-----     9/10/2019   9:32 AM                Visual Studio 2019
d-----     2/25/2021   4:01 PM                WindowsPowerShell
d-----      7/7/2017   8:57 AM                Записные книжки OneNote
. . .
```

В качестве еще одного примера создадим пользовательский диск `CurrVer` для ветви реестра HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion, где хранятся различные важные параметры операционной системы:

```
PS Docs:\> New-PSDrive -Name CurrVer -PSProvider Registry -Root
HKLM\Software\Microsoft\Windows\CurrentVersion

Name      Provider   Root                          CurrentLocation
----      --------   ----                          ---------------
CurrVer   Registry   HKLM\Software\Microsoft\...
```

Теперь содержимое ветви реестра можно просматривать, не вводя длинного пути, трудного для запоминания:

```
PS docs:\> dir CurrVer:\

    Hive: HKLM\Software\Microsoft\Windows\CurrentVersion

Name                          Property
----                          --------
AccountPicture
ActionCenter
AdvertisingInfo               Enabled : 0
App Management
App Paths
AppHost                       EnableWebContentEvaluation : 1
. . .
```

Итоги

- Стандартные операции, например манипуляции с файловой системой, выполняются в PowerShell аналогично другим оболочкам командной строки.
- В PowerShell можно вычислять выражения непосредственно в командной строке.
- PowerShell поддерживает команды четырех типов: командлеты, функции, сценарии и внешние исполняемые файлы.
- Командлеты — аналог внутренних команд в других оболочках. Компилированный код командлетов подгружается к процессу PowerShell во время запуска оболочки, поэтому они выполняются максимально эффективно. Синтаксис всех командлетов унифицирован, имена всегда соответствуют шаблону "действие — объект".

- Функция в PowerShell — это блок кода на языке PowerShell, имеющий название и находящийся в памяти до завершения текущего сеанса командной оболочки. При вызове функции в нее можно передавать аргументы, указывая их через пробел после названия.
- Сценарий — это блок кода на языке PowerShell, хранящийся во внешнем файле с расширением ps1.
- Внешние исполняемые файлы вызываются из оболочки по имени (если в нем нет пробелов) либо с помощью оператора вызова &.
- Для команд в PowerShell поддерживаются псевдонимы двух типов: стандартные сокращения для быстрого ввода команд и привычные названия команд из оболочек cmd и bash.
- Провайдеры PowerShell обеспечивают доступ к виртуальным дискам, связанным с хранилищами разных типов (системный реестр, хранилище цифровых сертификатов и т. д.). Обращаться к этим виртуальным дискам и работать с ними можно так же, как с дисками файловой системы.

ГЛАВА 4

Работа в оболочке PowerShell

При работе в командной строке приходится вручную вводить команды, которые могут быть довольно длинными и иметь много разных трудно запоминаемых параметров. Поэтому важно научиться грамотно пользоваться справочной системой, имеющейся в системе, а также использовать возможности оболочки для быстрого набора команд или их повторного вызова.

Редактирование в командной строке PowerShell

В командной строке PowerShell при вводе текста доступны некоторые возможности редактирования (табл. 4.1).

Таблица 4.1. *Возможности редактирования в командной строке PowerShell*

Клавиатурная комбинация	Действие
<←>	Перемещение курсора на один символ влево
<→>	Перемещение курсора на один символ вправо
<Ctrl>+<←>	Перемещение курсора влево на одно слово
<Ctrl>+<→>	Перемещение курсора вправо на одно слово
<Home>	Перемещение курсора в начало текущей строки
<End>	Перемещение курсора в конец текущей строки
<↑>/<↓>	Просмотр истории команд
<Delete>	Удаление символа, находящегося над курсором
<Backspace>	Удаление символа, находящегося перед курсором
<Ctrl>+<Home>	Удаление символов от текущей позиции курсора до начала строки
<Ctrl>+<End>	Удаление символов от текущей позиции курсора до конца строки

Таблица 4.1 *(окончание)*

Клавиатурная комбинация	Действие
<Esc>	Очистка строки (удаление всех введенных символов)
символы, затем <F8>	Перебор команд из истории, которые начинаются с введенных символов
<Tab>	Автоматическое завершение команды *(см разд "Автоматическое завершение команд" данной главы и приложение 3)*

Замечание

В *приложении 3* описаны дополнительные возможности редактирования в командной строке, предоставляемые модулем PSReadLine.

Если вы работаете в Windows Terminal, то можете с помощью специальных клавиатурных комбинаций управлять самим терминалом (например, изменять размер шрифта и открытых на экране панелей). Эти комбинации можно посмотреть на вкладке **Actions** в настройках терминала (рис. 4.1).

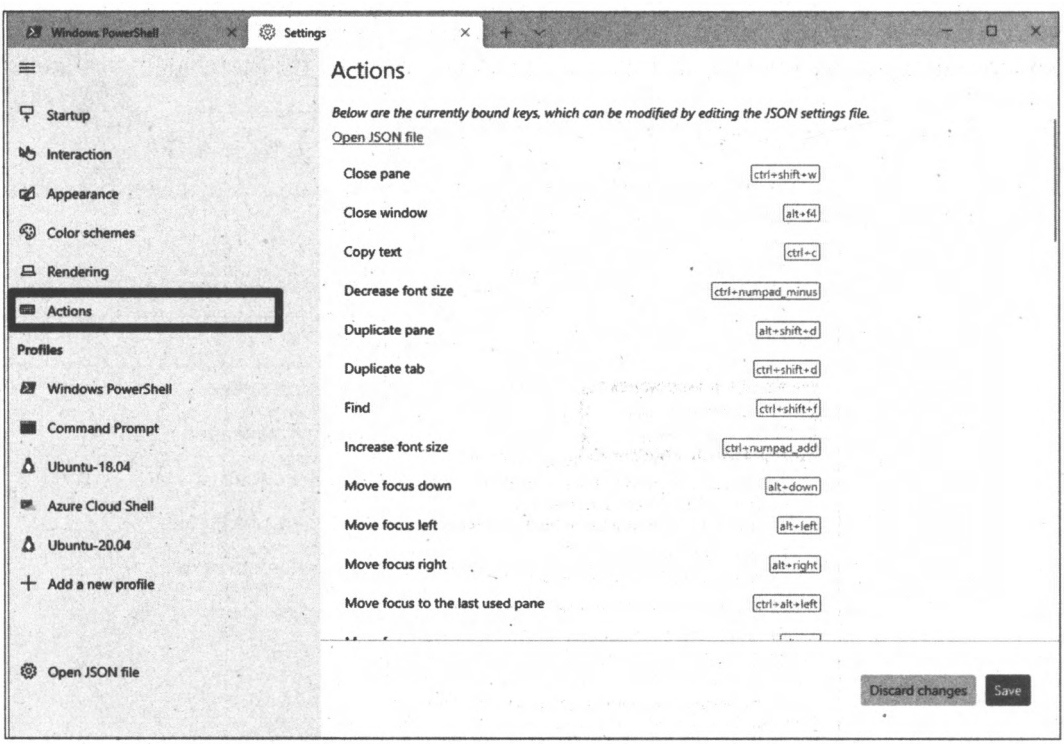


Рис. 4.1. Клавиатурные комбинации для управления Windows Terminal

При необходимости комбинации можно изменить, для этого необходимо отредактировать конфигурационный файл settings.json, ссылка для его открытия также находится на вкладке **Actions** в настройках терминала.

Работая с оболочкой PowerShell, можно пользоваться буфером Windows. Если у вас запущена стандартная консоль Windows, то выделить и скопировать текст в буфер из окна PowerShell можно следующим образом: щелкнуть правой кнопкой мыши на заголовке окна PowerShell (или любой кнопкой мыши на пиктограмме окна), последовательно выбрать пункты **Изменить** и **Пометить** в появившемся контекстном меню, затем с помощью клавиш управления курсором, удерживая нажатой клавишу <Shift>, выделить нужный блок текста (также выделение можно произвести с помощью мыши, удерживая нажатой ее левую кнопку) и нажать клавишу <Enter>. Все содержимое окна PowerShell можно выделить, выбрав пункты **Изменить** и **Выделить все** того же контекстного меню. Вставка текста из буфера Windows в командное окно PowerShell (в текущую позицию курсора) осуществляется с помощью выбора пунктов **Изменить** и **Вставить**.

Можно упростить манипуляции с буфером Windows в стандартной консоли, включив режимы выделения мышью и быстрой вставки. Для этого нужно щелкнуть правой кнопкой мыши на заголовке окна PowerShell, выбрать в контекстном меню пункт **Свойства** и установить флажки **Выделение мышью** и **Быстрая вставка** в секции **Правка** на вкладке **Настройки** диалогового окна **Свойства: "Windows PowerShell"** (рис. 4.2).

После этого выделять текст можно мышью, удерживая нажатой левую кнопку. Для копирования выделенного фрагмента в буфер достаточно нажать клавишу <Enter>

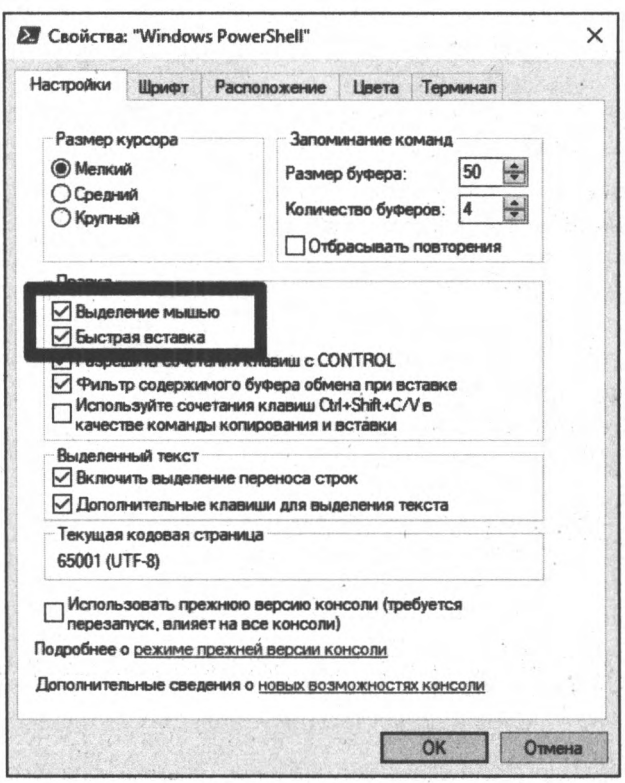


Рис. 4.2. Настройка свойств стандартной консоли Windows

или щелкнуть правой кнопкой мыши. Вставка текста из буфера в текущую позицию курсора производится также по нажатии правой кнопки мыши.

Если установлен флажок **Разрешить сочетания клавиш с CONTROL**, то копировать в буфер выделенный текст можно нажатием клавиш <Ctrl>+<c> или <Ctrl>+ +<Insert>, а вставлять текст из буфера — нажатием клавиш <Ctrl>+<v> или <Shift>+<Insert>.

Терминал Windows Terminal также поддерживает режимы выделения мышью и быстрой вставки, копировать текст в буфер и вставлять его из буфера можно с помощью тех же комбинаций <Ctrl>+<c>/<Ctrl>+<Insert> и <Ctrl>+<v>/<Shift>+<Insert>.

> ***Замечание***
> Если у вас есть сценарий PowerShell (внешний текстовый файл с командами PowerShell), содержащий несколько строк, то можно копировать в буфер Windows и вставлять в окно PowerShell сразу все команды (они будут выполняться по очереди), а не по одной.

На самом деле большинство из приведенных в табл. 4.1 клавиатурных комбинаций реализованы на уровне терминала, т. е. действуют одинаково во всех консольных приложениях Windows.

В самой оболочке PowerShell реализована еще одна важная возможность редактирования — автоматическое завершение команд и их параметров при вводе.

Автоматическое завершение команд

Находясь в оболочке PowerShell, можно ввести часть какой-либо команды, нажать клавишу <Tab>, и система попытается сама завершить эту команду.

Подобное автоматическое завершение срабатывает, во-первых, для имен файлов и путей файловой системы (подобный режим поддерживается и оболочкой cmd.exe). При нажатии клавиши <Tab> PowerShell автоматически расширит частично введенный путь файловой системы до первого найденного совпадения. При повторном нажатии клавиши <Tab> производится циклический переход по имеющимся возможностям выбора. Например, нам нужно переместиться в каталог C:\Program Files. Введем в командной строке PowerShell команду `cd` и в качестве параметра укажем начало имени нужного нам каталога:

```
PS C:\Users\andrv> cd c:\pro
```

Нажмем теперь клавишу <Tab>, и система автоматически дополнит путь к каталогу:

```
PS C:\Users\andrv> cd 'C:\Program Files'
```

Как видите, название каталога, содержащее пробелы, заключается в апострофы.

Также в PowerShell реализована возможность автоматического завершения путей файловой системы на основе шаблонных символов: ? (заменяет один произвольный символ) и * (заменяет любое количество произвольных символов). Например, если в PowerShell ввести команду `cd c:\pro*files` и нажать клавишу <Tab>, то в строке ввода вновь появится команда `cd 'C:\Program Files'`.

Во-вторых, в PowerShell реализовано автоматическое завершение имен командлетов и их параметров. Если ввести первую часть имени командлета (глагол) и дефис, нажать после этого клавишу <Tab>, то система подставит имя первого подходящего командлета (следующий вариант имени выбирается путем повторного нажатия <Tab>). Например, введем в командной строке PowerShell глагол `get-`:

```
PS C:\Users\andrv> get-
```

Нажмем клавишу <Tab>:

```
PS C:\Users\andrv> Get-Acl
```

Еще раз нажмем клавишу <Tab>:

```
PS C:\Users\andrv> Get-Alias
```

Нажимая далее клавишу <Tab>, мы можем перебрать все командлеты, начинающиеся с глагола `Get`.

Аналогичным образом автоматическое завершение срабатывает для частично введенных имен параметров командлета: нажимая клавишу <Tab>, мы будем циклически перебирать подходящие имена. Например, введем в командной строке PowerShell следующий командлет:

```
PS C:\Users\andrv> Get-Alias -
```

Нажмем клавишу <Tab>:

```
PS C:\Users\andrv> Get-Alias -Name
```

Как видите, система подставила в нашу команду параметр `-Name`. Еще раз нажмем клавишу <Tab>:

```
PS C:\Users\andrv> Get-Alias -Exclude
```

Нажимая далее клавишу <Tab>, мы можем перебрать все параметры, поддерживающиеся командлетом `Get-Alias`.

В-третьих, PowerShell позволяет автоматически завершать имена используемых переменных. Например, создадим переменную `$StringVariable`:

```
PS C:\Users\andrv> $StringVariable='asdfg'
```

Введем теперь часть имени этой переменной:

```
PS C:\Users\andrv> $str
```

Нажав клавишу <Tab>, мы получим в командной строке полное имя нашей переменной:

```
PS C:\Users\andrv> $StringVariable
```

Наконец, PowerShell поддерживает автоматическое завершение имен свойств и методов объектов. Например, введем следующие команды:

```
PS C:\Users\andrv> $s='qwerty'
PS C:\Users\andrv> $s.len
```

Нажмем клавишу <Tab>:

```
PS C:\Users\andrv> $s.Length
```

Система автоматически подставила свойство `Length`, имеющееся у символьных переменных. Если подставляется имя метода (функции), а не свойства, то после его имени автоматически ставится круглая скобка. Например, введем следующую команду:

```
PS C:\Users\andrv> $s.sub
```

Нажмем клавишу <Tab>:

```
PS C:\Users\andrv> $s.Substring(
```

Теперь можно вводить параметры метода `Substring`.

> ***Замечание***
> В *приложении 3* описаны дополнительные возможности автоматического завершения команд, предоставляемые модулем PSReadLine.

Ввод команды в несколько строках

По умолчанию нажатие клавиши <Enter> завершает ввод команды, после чего оболочка начинает разбирать и выполнять данную команду. Если же последним символом в строке указать символ обратного апострофа `, то ввод команды продолжится со следующей строки. При этом символ перевода строки рассматривается как обычный разделяющий пробел.

Например, вычислим значение выражения `10+2-3+(5*4-3)`, расположив его на трех строках:

```
PS C:\Users\andrv> 10+`
>> 2-3+(`
>> 5*4-3)
26
```

Как видим, приглашение при вводе дополнительных строк меняется на символы >> — это признак того, что продолжается ввод предыдущей команды.

> ***Замечание***
> В *приложении 3* описаны дополнительные возможности многострочного ввода команд, предоставляемые модулем PSReadLine.

Справочная система PowerShell

При работе с интерактивной командной оболочкой важно иметь под рукой подробную и удобную справочную систему с описанием возможностей команд и примерами их применения. В PowerShell такая система имеется, здесь предусмотрено несколько способов получения справочной информации внутри оболочки.

Получение справки о командлетах

Краткую справку по какому-либо одному командлету можно получить с помощью параметра -? (вопросительный знак), указанного после имени этого командлета. Например, для получения информации о командлете Get-Process следует выполнить следующую команду:

```
PS C:\Users\andrv> Get-Process -?

ИМЯ
    Get-Process

СИНТАКСИС
    Get-Process [[-Name] <string[]>] [-ComputerName <string[]>] [-Module]
                                     [-FileVersionInfo] [<CommonParameters>]

    Get-Process [[-Name] <string[]>] -IncludeUserName [<CommonParameters>]

    Get-Process -Id <int[]> -IncludeUserName [<CommonParameters>]

    Get-Process -Id <int[]> [-ComputerName <string[]>] [-Module]
                                     [-FileVersionInfo] [<CommonParameters>]

    Get-Process -InputObject <Process[]> -IncludeUserName [<CommonParameters>]

    Get-Process -InputObject <Process[]> [-ComputerName <string[]>] [-Module]
                                     [-FileVersionInfo] [<CommonParameters>]

ПСЕВДОНИМЫ
    gps
    ps

ЗАМЕЧАНИЯ
  Get-Help cannot find the Help files for this cmdlet on this computer.
  It is displaying only partial help.
  -- To download and install Help files for the module that includes this
     cmdlet, use Update-Help.
  -- To view the Help topic for this cmdlet online, type: "Get-Help Get-Process
    -Online" or go to https://go.microsoft.com/fwlink/?LinkID=113324.
```

На самом деле при вводе команды Get-Process -? срабатывает специальный командлет Get-Help, т. е. обратиться к справке для командлета Get-Process можно так:

```
PS C:\Users\andrv> Get-Help Get-Process
```

Как видим, во встроенной справке перечисляются допустимые варианты синтаксиса командлета и его псевдонимы. Необязательные параметры выводятся в квадратных скобках. Если для параметра необходимо указывать аргумент определенного типа, то после имени такого параметра в угловых скобках приводится название этого типа.

В первых версиях PowerShell справочные файлы по умолчанию находились локально, сейчас подробная справочная информация хранится в Интернете на сайтах Microsoft. Обратиться к ней из командной строки можно с помощью командлета `Get-Help` с параметром `-Online`:

```
PS C:\Users\andrv> Get-Help Get-Process -Online
```

В этом случае подробное описание командлета `Get-Process` откроется в браузере (рис. 4.3).

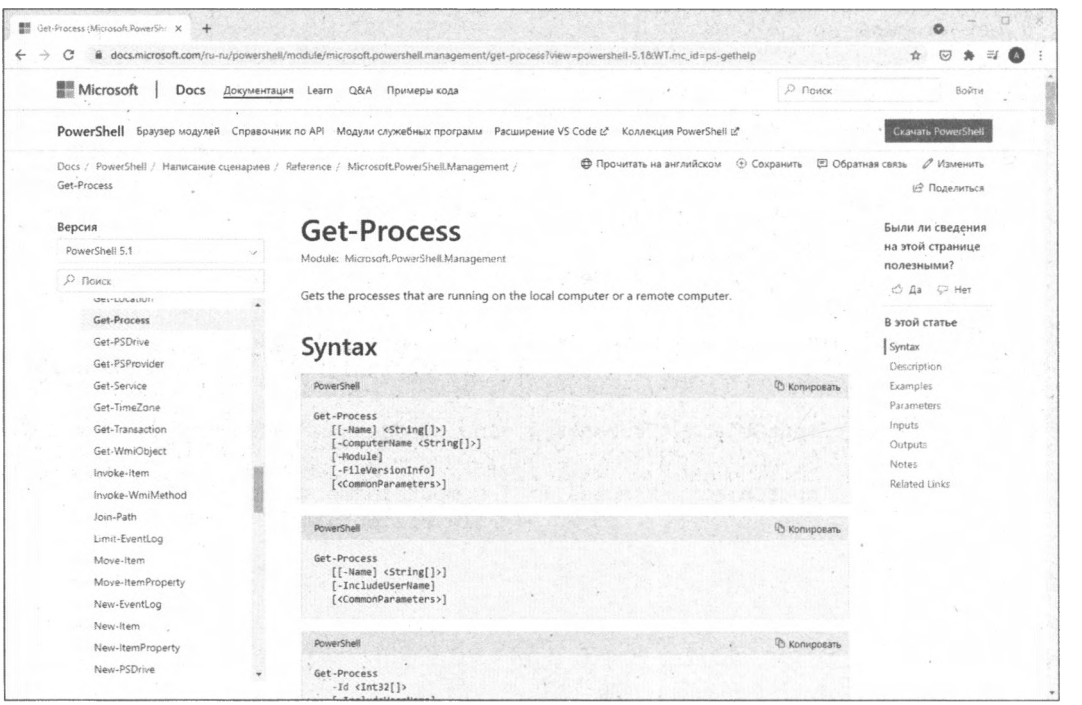


Рис. 4.3. Онлайн-информация о командлете PowerShell

Также можно открыть справочную информацию не в браузере, а в отдельном диалоговом окне. Для этого используется параметр `-ShowWindow` (рис. 4.4):

```
PS C:\Users\andrv> Get-Help Get-Process -ShowWindow
```

Командлет `Update-Help` позволяет загрузить справочные файлы на свою машину, чтобы обращение к ним происходило быстрее.

> ***Замечание***
>
> Для полной загрузки справочных файлов может понадобиться запустить оболочку PowerShell от имени учетной записи Администратора.

После локальной установки справочных файлов информация о командлетах будет выводиться в более полном формате. Например:

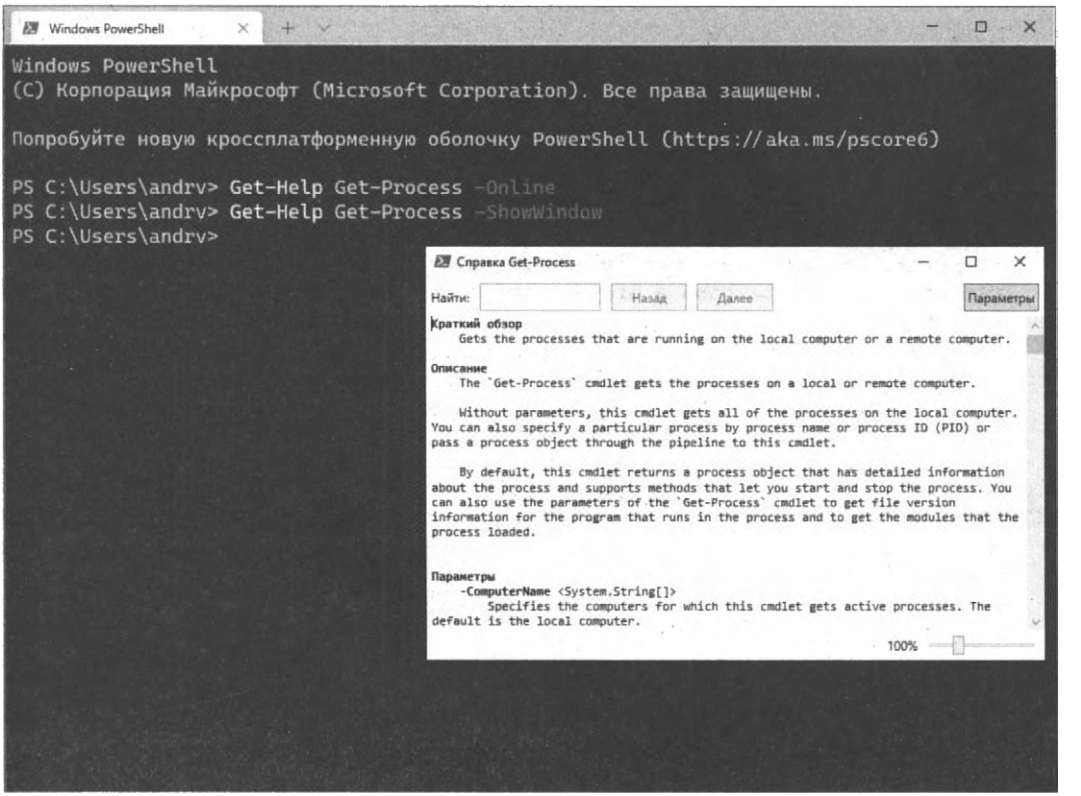


Рис. 4.4. Справочная информация о командлете в диалоговом окне

```
PS C:\Users\andrv> Get-Help Get-Process

NAME
    Get-Process

ОПИСАНИЕ
    Gets the processes that are running on the local computer or a remote
                                                                    computer.

СИНТАКСИС
    Get-Process [[-Name] <System.String[]>] [-ComputerName <System.String[]>]
                            [-FileVersionInfo] [-Module] [<CommonParameters>]

    Get-Process [-ComputerName <System.String[]>] [-FileVersionInfo]
                         -Id <System.Int32[]> [-Module] [<CommonParameters>]

    Get-Process [-ComputerName <System.String[]>] [-FileVersionInfo]
     -InputObject <System.Diagnostics.Process[]> [-Module] [<CommonParameters>]

    Get-Process -Id <System.Int32[]> -IncludeUserName [<CommonParameters>]
```

```
Get-Process [[-Name] <System.String[]>] -IncludeUserName
                                                    [<CommonParameters>]

Get-Process -IncludeUserName -InputObject <System.Diagnostics.Process[]>
                                                    [<CommonParameters>]
```

ОПИСАНИЕ

The `Get-Process` cmdlet gets the processes on a local or remote computer. Without parameters, this cmdlet gets all of the processes on the local computer. You can also specify a particular process by process name or process ID (PID) or pass a process object through the pipeline to this cmdlet.

By default, this cmdlet returns a process object that has detailed information about the process and supports methods that let you start and stop the process. You can also use the parameters of the `Get-Process` cmdlet to get file version information for the program that runs in the process and to get the modules that the process loaded.

ССЫЛКИ ПО ТЕМЕ

Online Version: https://docs.microsoft.com/powershell/module/microsoft.powershell.management/get-process?view=powershell-5.1&WT.mc_id=ps-gethelp

Debug-Process

Get-Process

Start-Process

Stop-Process

Wait-Process

ЗАМЕЧАНИЯ

To see the examples, type: "get-help Get-Process -examples".

For more information, type: "get-help Get-Process -detailed".

For technical information, type: "get-help Get-Process -full".

For online help, type: "get-help Get-Process -online"

Теперь мы дополнительно видим описание командлета `Get-Process` и ссылки на связанные с ним другие командлеты.

Для получения подробной информации командлет `Get-Help` нужно запускать с параметрами `-Detailed` или `-Full`. В этом случае будут выведены подробные описания каждого из параметров, поддерживаемых рассматриваемым командлетом, различные замечания, а также примеры запуска данного командлета с различными параметрами и аргументами (параметр `-examples` позволяет отдельно посмотреть только примеры). Например:

```
PS C:\Users\andrv> Get-Help Get-Process -Full

ИМЯ
    Get-Process

ОПИСАНИЕ
    Gets the processes that are running on the local computer or a remote
                                                                  computer.
```

```
СИНТАКСИС
    Get-Process [[-Name] <System.String[]>] [-ComputerName <System.String[]>]
                                [-FileVersionInfo] [-Module] [<CommonParameters>]

    Get-Process [-ComputerName <System.String[]>] [-FileVersionInfo]
                            -Id <System.Int32[]> [-Module] [<CommonParameters>]

    Get-Process [-ComputerName <System.String[]>] [-FileVersionInfo]
     -InputObject <System.Diagnostics.Process[]> [-Module] [<CommonParameters>]

    Get-Process -Id <System.Int32[]> -IncludeUserName [<CommonParameters>]

    Get-Process [[-Name] <System.String[]>] -IncludeUserName
                                                          [<CommonParameters>]

    Get-Process -IncludeUserName -InputObject <System.Diagnostics.Process[]>
                                                          [<CommonParameters>]

ОПИСАНИЕ
  The `Get-Process` cmdlet gets the processes on a local or remote computer.
  Without parameters, this cmdlet gets all of the processes on the local
  computer. You can also specify a particular process by process name or
  process ID (PID) or pass a process object through the pipeline to this
  cmdlet.
  By default, this cmdlet returns a process object that has detailed
  information about the process and supports methods that let you start and
  stop the process. You can also use the parameters of the `Get-Process` cmdlet
  to get file version information for the program that runs in the process and
  to get the modules that the process loaded.

ПАРАМЕТРЫ
  -ComputerName <System.String[]>
      Specifies the computers for which this cmdlet gets active processes.
      The default is the local computer.
      Type the NetBIOS name, an IP address, or a fully qualified domain name
      (FQDN) of one or more computers. To specify the local computer, type the
      computer name, a dot (.), or localhost.
    This parameter does not rely on Windows PowerShell remoting. You can use
    the ComputerName parameter of this cmdlet even if your computer is not
    configured to run remote commands.
        Требуется?                    false
        Позиция?                      named
        Значение по умолчанию         Local computer
        Принимать входные данные конвейера?  True (ByPropertyName)
        Принимать подстановочные знаки?      false

. . .
```

Как видим, в описании параметра `Name` даются сведения о пяти атрибутах. Эти атрибуты характерны для большинства параметров командлетов (табл. 4.2).

Таблица 4.2. Атрибуты параметров командлетов PowerShell

Параметр	Описание
Требуется?	Данный атрибут определяет, будет ли командлет выполняться при отсутствии этого параметра Если настройке присвоено значение `True`, то при запуске данного командлета необходимо указывать параметр Если параметр не задан, система запросит его значение
Позиция?	Данный атрибут определяет, можно ли задавать значение параметра без указания его имени Если атрибут имеет значение `0` или `named`, то при задании значения параметра необходимо указывать его имя (напомним, что параметр данного типа называется именованным) Именованные параметры могут перечисляться после имени командлета в любом порядке Если атрибут "Позиция?" имеет целое ненулевое значение, то имя параметра указывать не обязательно (позиционный параметр) Значение атрибута "Позиция?" определяет порядковый номер параметра в списке других позиционных параметров При указании имени позиционные параметры могут перечисляться после имени командлета в любом порядке
Значение по умолчанию	Данный атрибут содержит значение по умолчанию, которое используется в том случае, когда никакого иного значения не указано Обязательным параметрам, так же как и многим необязательным, никогда не присваивается значение по умолчанию Например, многие команды, чьим входным значением является параметр `-Path`, при отсутствии соответствующего значения используют текущее местоположение
Принимать входные данные конвейера?	Данный атрибут определяет, может ли параметр получать свое значение из объекта в конвейере Чтобы команду можно было включить в конвейер, соответствующая настройка ее входного параметра должна иметь значение `True`, что дает возможность принимать конвейерный ввод
Принимать подстановочные знаки?	Данный атрибут показывает, может ли значение параметра содержать подстановочные знаки, что дает возможность сопоставлять его с несколькими существующими в целевом контейнере элементами

Справочная информация, не связанная с командлетами

Все доступные разделы справочной системы PowerShell можно увидеть с помощью команды `Get-Help *`:

```
PS C:\Users\andrv> Get-Help *
Name                              Category  Module
----                              --------  ------
foreach                           Alias
%                                 Alias
where                             Alias
?                                 Alias
```

```
ac                                  Alias
clc                                 Alias
cli                                 Alias
. . .
more                                Function
cd..                                Function
cd\                                 Function
ImportSystemModules                 Function
Pause                               Function
help                                Function
. . .
Add-History                         Cmdlet     Microsoft.PowerShell....
Clear-History                       Cmdlet     Microsoft.PowerShell....
Connect-PSSession                   Cmdlet     Microsoft.PowerShell....
Debug-Job                           Cmdlet     Microsoft.PowerShell....
. . .
about_Aliases                       HelpFile
about_Alias_Provider                HelpFile
about_Arithmetic_Operators          HelpFile
about_Arrays                        HelpFile
about_Assignment_Operators          HelpFile
. . .
```

Как видим, командлет `Get-Help` позволяет просматривать справочную информацию не только о разных командлетах, но и о синтаксисе языка PowerShell, о псевдонимах, провайдерах, функциях и других аспектах работы оболочки. Список тем, обсуждение которых представлено в справочной службе PowerShell, можно увидеть с помощью следующей команды:

```
PS C:\Users\andrv> Get-Help about_*
Name                                Category  Module
----                                --------  ------
about_Aliases                       HelpFile
about_Alias_Provider                HelpFile
about_Arithmetic_Operators          HelpFile
about_Arrays                        HelpFile
about_Assignment_Operators          HelpFile
about_Automatic_Variables           HelpFile
about_Break                         HelpFile
about_Calculated_Properties         HelpFile
about_Certificate_Provider          HelpFile
about_Character_Encoding            HelpFile
about_CimSession                    HelpFile
about_Classes                       HelpFile
. . .
```

Таким образом, чтобы прочитать информацию, например, об использовании массивов в PowerShell, нужно выполнить следующую команду:

```
PS C:\Users\andrv> Get-Help about_Arrays
ABOUT_ARRAYS

Short Description
Describes arrays, which are data structures designed to store collections
of items.

Long Description
An array is a data structure that is designed to store a collection of items.
The items can be the same type or different types.
Beginning in Windows PowerShell 3.0, a collection of zero or one object has
some properties of arrays.

Creating and initializing an array
To create and initialize an array, assign multiple values to a variable.
. . .
```

Отметим, что командлет `Get-Help` выводит содержимое раздела справки на экран сразу целиком. Функции `man` и `help` позволяют выводить справочную информацию поэкранно (аналогично команде `more` интерпретатора `cmd.exe`), например: `man about_Arrays`.

История команд в сеансе работы

Информацию обо всех командах, которые мы выполняем в оболочке PowerShell, система записывает в специальный *журнал сеанса* или *журнал команд*, что дает возможность повторно использовать эти команды, не набирая их полностью на клавиатуре. Журнал сеанса сохраняется до выхода из оболочки PowerShell.

Для передвижения назад по журналу команд следует нажимать клавишу <↑>. При первом нажатии этой клавиши в командной строке будет отображена последняя команда, выполнявшаяся в текущем сеансе работы. При повторном нажатии клавиши <↑> появится предпоследняя выполненная команда и т. д. Получив нужную команду, можно отредактировать ее, после чего нажать <Enter> для выполнения команды. Клавиша <↓> позволяет перемещаться по журналу команд вперед.

Можно просматривать не все команды в журнале сеанса, а только команды, начинающиеся с определенных символов. Для этого нужно ввести эти начальные символы в командной строке и нажимать клавишу <F8>.

В дополнение к клавиатурным комбинациям в PowerShell имеются специальные командлеты для работы с журналом команд. Команда `Get-History` (псевдонимы `h`, `history` и `ghy`) позволяет вывести историю команд:

```
PS C:\> Get-History

  Id CommandLine
  -- -----------
   1 cd c:
   2 cd \
```

```
  3 del -Recurse -WhatIf "C:\Program Files"
  4 del -Recurse -WhatIf "C:\temp\*.*"
  5 del -WhatIf "C:\temp\*.*"
  6 Get-Alias del
```

По умолчанию отображаются последние 32 команды со своими порядковыми номерами (колонка `Id`). Число выводимых команд можно изменить с помощью параметра `-Count`. Например, следующая команда выведет три последние команды:

```
PS C:\> Get-History -Count 3

  Id CommandLine
  -- -----------
   5 del -WhatIf "C:\temp\*.*"
   6 Get-Alias del
   7 Get-History
```

Можно выделять из журнала сеанса команды, удовлетворяющие определенному критерию. Для этого используется процедура конвейеризации объектов и специальный командлет `Where-Object` (подробнее эти темы обсуждаются в *главе 5*). Например, для вывода всех команд, содержащих слово del, можно выполнить следующую команду:

```
PS C:\> Get-History | Where-Object {$_.commandLine -like "*del*"}

  Id CommandLine
  -- -----------
   3 del -Recurse -WhatIf "C:\Program Files"
   4 del -Recurse -WhatIf "C:\temp\*.*"
   5 del -WhatIf "C:\temp\*.*"
   6 Get-Alias del
```

Полученный с помощью `Get-History` список команд можно экспортировать во внешний файл в формате XML или CSV (текстовый файл с запятыми в качестве разделителя). Для этого вновь нужно применять конвейеризацию команд и специальные командлеты для экспорта данных в определенный формат. Например, следующая команда сохраняет журнал команд в CSV-файле c:\history.csv:

```
PS C:\> Get-History | Export-Csv c:\history.csv
```

С помощью командлета `Add-History` можно добавлять команды в журнал сеанса (например, для создания журнала, содержащего команды, введенные за несколько сеансов работы). Например, следующая команда добавит в журнал сеанса команды, сохраненные в файле c:\history.csv:

```
PS C:\> Import-Csv c:\history.csv | Add-History
```

Командлет `Invoke-History` (псевдонимы — `r`, сокращение от *repeat* или *rerun*, и `ihy`) позволяет повторно выполнять команды из журнала сеанса, при этом команды можно задавать по их порядковому номеру или первым символам, а также получать по конвейеру от командлета `Get-History`. Приведем несколько примеров.

Вызов последней команды:

```
PS C:\> Invoke-History
```

Вызов третьей команды по ее порядковому номеру:

```
PS C:\> Invoke-History -Id 3
```

или просто:

```
PS C:\> r 3
```

Вызов последней команды `Get-Help`:

```
PS C:\> Invoke-History Get-He
```

Вызов команд, полученных по конвейеру от командлета `Get-History`:

```
PS C:\> Get-History | Where-Object {$_.commandLine -like "*del*"} |
Invoke-History
```

> ***Замечание***
>
> В *приложении 3* описаны дополнительные возможности поиска в истории команд, предоставляемые модулем PSReadLine.

Протоколирование действий в сеансе работы

В предыдущем разделе мы научились вызывать список команд, выполненных во время сеанса работы, и сохранять этот список во внешний файл. В оболочке PowerShell также можно записывать в текстовый файл не только запускаемые команды, но и результат их выполнения, т. е. сохранять в файле весь сеанс работы или какую-либо его часть. Для этой цели служит командлет `Start-Transcript`, имеющий следующий синтаксис (указаны только основные параметры):

```
Start-Transcript [[-path] <string>] [-noClobber] [-append]
```

Параметр `Path` здесь задает путь к текстовому файлу с протоколом работы (путь должен быть указан явно, подстановочные знаки не поддерживаются). Давайте запустим протоколирование работы с сохранением протокола в файле C:\Users\andrv\transcript.txt:

```
PS C:\Users\andrv> Start-Transcript -Path C:\Users\andrv\transcript.txt
Транскрибирование запущено, выходной файл C:\Users\andrv\transcript.txt
```

Выполним теперь какую-нибудь команду и завершим протоколирование с помощью командлета `Stop-Transcript`:

```
PS C:\Users\andrv> Get-Help Get-Process
ИМЯ
    Get-Process

ОПИСАНИЕ
. . .

PS C:\Users\andrv> Stop-Transcript
Транскрибирование остановлено, выходной файл C:\Users\andrv\transcript.txt
```

Посмотрим на содержимое файла C:\Users\andrv\transcript.txt:

```
PS C:\> type C:\Users\andrv\transcript.txt
**********************
Начало записи сценария Windows PowerShell
Время начала: 20210523211401
Имя пользователя: DESKTOP-BU86I5T\andrv
Запуск от имени пользователя: DESKTOP-BU86I5T\andrv
Имя конфигурации:
Компьютер: DESKTOP-BU86I5T (Microsoft Windows NT 10.0.19042.0)
Ведущее приложение: powershell.exe
ИД процесса: 1824
PSVersion: 5.1.19041.906
PSEdition: Desktop
PSCompatibleVersions: 1.0, 2.0, 3.0, 4.0, 5.0, 5.1.19041.906
BuildVersion: 10.0.19041.906
CLRVersion: 4.0.30319.42000
WSManStackVersion: 3.0
PSRemotingProtocolVersion: 2.3
SerializationVersion: 1.1.0.1
**********************
Транскрибирование запущено, выходной файл C:\Users\andrv\transcript.txt
PS C:\Users\andrv> Get-Help Get-Process

ИМЯ
    Get-Process

ОПИСАНИЕ
. . .
PS C:\Users\andrv> Stop-Transcript
**********************
Конец записи протокола Windows PowerShell
Время окончания: 20210523211431
**********************
```

Как видим, в файле протокола дополнительно записана информация о дате и времени начала протоколирования и его завершения, а также об имени компьютера и учетной записи пользователя, запускавшего протоколирование.

Если имя для файла протокола не указано, то он будет сохраняться в файле, путь к которому задан в значении глобальной переменной `$Transcript`. Если эта переменная не определена, то командлет `Start-Transcript` сохраняет протоколы в каталоге $Home\Documents в файлах PowerShell_transcript.<метка-времени>.txt (в переменной `$Home` хранится путь к домашнему каталогу работающего пользователя).

Если файл протокола, в который должен начать сохраняться сеанс работы, уже существует, то по умолчанию он будет переписан. Параметр `-Append` командлета `Start-Transcript` включает режим добавления нового протокола к существующему

файлу. Если же указан параметр `-noClobber` и файл протокола уже существует, то командлет `Start-Transcript` не выполняет никаких действий.

Настройка оформления командной строки PowerShell

Внешний вид командной строки PowerShell (цвет текста и фона, используемый шрифт, степень прозрачности окна и т. п.) зависит от настроек терминала, в котором запущена оболочка.

Если вы работаете в стандартной консоли, то для настройки параметров окна нужно запустить оболочку PowerShell, щелкнуть правой кнопкой мыши на заголовке окна (или любой кнопкой мыши на пиктограмме окна) и выбрать пункт **Свойства** в появившемся контекстном меню. В результате будет открыто диалоговое окно для изменения настроек командного окна PowerShell (рис. 4.5).

Рис. 4.5. Диалоговое окно для настройки параметров окна PowerShell

Данное окно имеет несколько вкладок (**Настройки**, **Шрифт**, **Расположение**, **Цвета** и **Терминал**), на которых сгруппированы элементы управления, позволяющие управлять соответствующими категориями параметров. Подробно рассматривать эти параметры мы не будем, назначение их довольно очевидно.

При работе в новом терминале Windows (wt.exe) размер шрифта можно изменять непосредственно в командной строке с помощью комбинаций клавиш <Ctrl>+<+> (увеличить шрифт) и <Ctrl>+<-> (уменьшить шрифт). Внешний вид каждого доступного профиля в терминале можно изменять с помощью настроек терминала на вкладке **Appearance** (рис. 4.6). Здесь можно выбрать цветовую схему и шрифт, задать форму курсора, установить фоновое изображение и т. д.

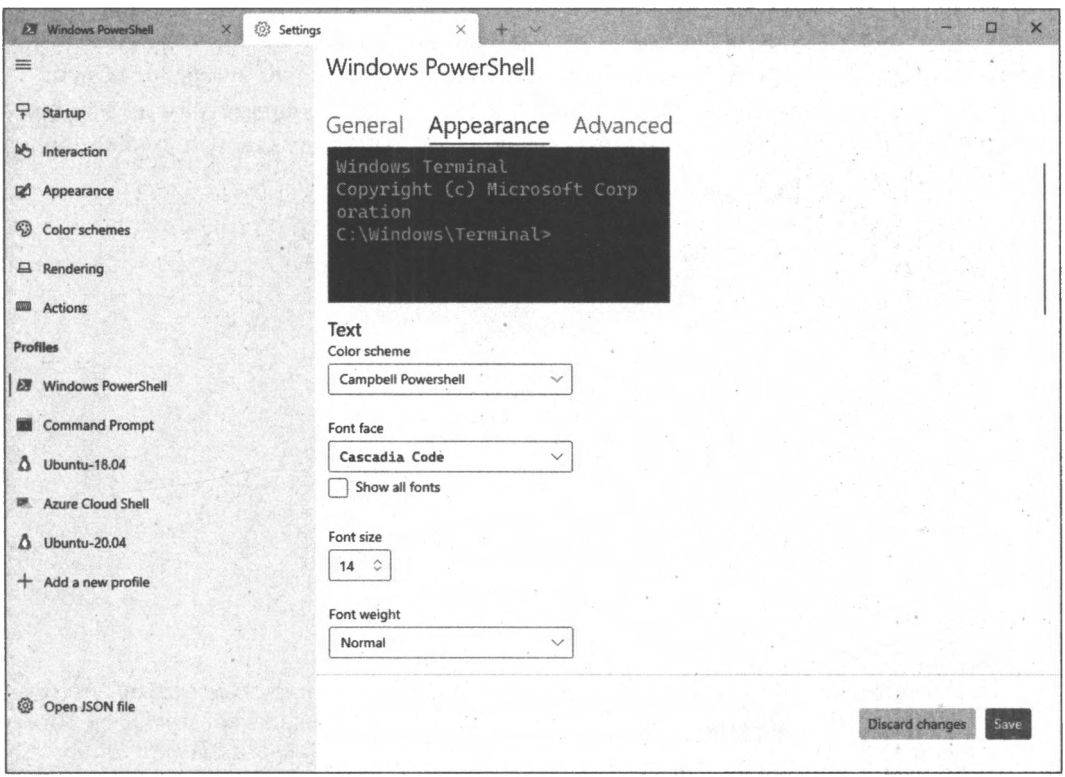


Рис. 4.6. Настройки внешнего вида профиля в терминале Windows

Заголовок командного окна

PowerShell позволяет настраивать различные параметры командного окна (размер, цвет текста и фона, заголовок окна и т. п.) непосредственно из оболочки. Для этого можно воспользоваться командлетом `Get-Host`, который по умолчанию выводит информацию о самой оболочке PowerShell (версия, региональные настройки и т. д.):

```
PS C:\Users\andrv> Get-Host

Name            : ConsoleHost
Version         : 5.1.19041.906
InstanceId      : 13dac4f0-dbb5-4382-8627-e9a1a03d2fde
```

```
UI               : System.Management.Automation.Internal.Host...
CurrentCulture   : ru-RU
CurrentUICulture : ru-RU
PrivateData      : Microsoft.PowerShell.ConsoleHost+ConsoleColor...
DebuggerEnabled  : True
IsRunspacePushed : False
Runspace         : System.Management.Automation.Runspaces.Local...
```

Нам понадобится свойство `UI` (это объект .NET-класса `System.Management.Automation.Internal.Host.InternalHostUserInterface`). В свою очередь, объект `UI` имеет свойство `RawUI`, позволяющее получить доступ к параметрам командного окна PowerShell. Для просмотра данных параметров выполним следующую команду:

```
PS C:\Users\andrv> (Get-Host).UI.RawUI

ForegroundColor       : Gray
BackgroundColor       : Black
CursorPosition        : 0,23
WindowPosition        : 0,0
CursorSize            : 25
BufferSize            : 120,39
WindowSize            : 120,39
MaxWindowSize         : 120,39
MaxPhysicalWindowSize : 1842,65
KeyAvailable          : True
WindowTitle           : Windows PowerShell
```

Замечание

В последней команде командлет `Get-Host` был заключен в круглые скобки Это означает, что система вначале запускает данный командлет и получает выходной объект в результате его работы. Затем извлекается свойство `UI` данного объекта (объект `UI`) и свойство `RawUI` объекта `UI` На экран окончательно выводятся значения свойств объекта `RawUI`

Некоторые свойства объекта `RawUI` можно изменять, настраивая тем самым соответствующие свойства командного окна. Для этого удобнее предварительно сохранить объект `RawUI` в отдельную переменную:

```
PS C:\Users\andrv> $a=(Get-Host).UI.RawUI
```

Теперь можно присвоить свойствам новые значения. По умолчанию в заголовке командного окна отображается строка "Windows PowerShell". Для изменения этого заголовка нужно записать новое значение в свойство `WindowTitle` объекта `RawUI`:

```
PS C:\Users\andrv> $a.WindowTitle="Мое командное окно"
```

Сразу после выполнения последней команды заголовком окна PowerShell станет строка "Мое командное окно" (рис. 4.7).

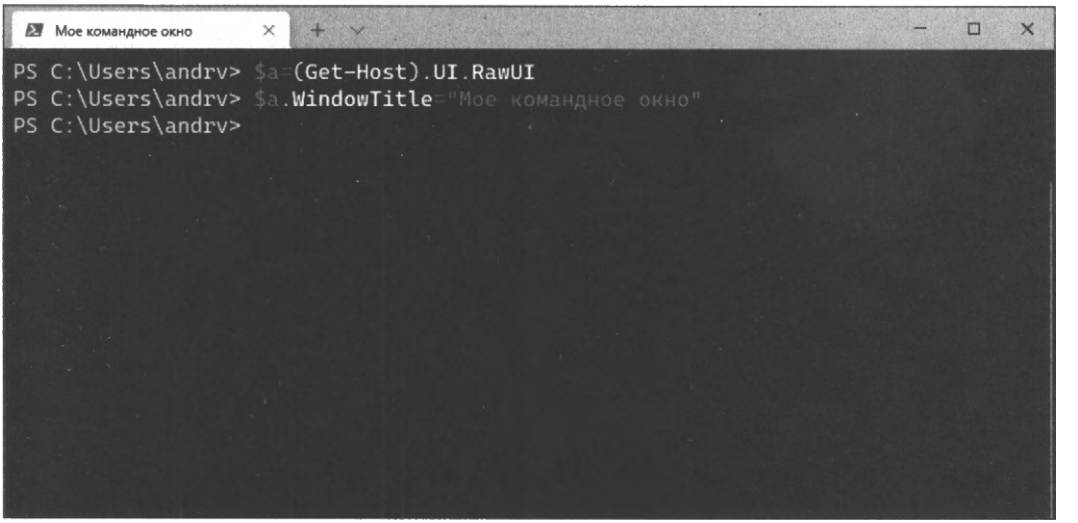


Рис. 4.7. Измененный заголовок командного окна PowerShell

Приглашение командной строки

Перейдем теперь к настройке приглашения командной строки. В оболочке PowerShell это приглашение контролируется с помощью функции `prompt`, которая должна возвращать одну строку. Посмотрим, не вдаваясь пока в подробности, какое содержание имеет данная функция по умолчанию (она отображает символы `PS`, затем путь к текущему каталогу и символ >). Для этого можно выполнить следующую команду:

```
PS C:\Users\andrv> (Get-Item function:prompt).Definition

"PS $($executionContext.SessionState.Path.CurrentLocation)$('>' *
($nestedPromptLevel + 1)) ";
# .Link
# https://go.microsoft.com/fwlink/?LinkID=225750
# .ExternalHelp System.Management.Automation.dll-help.xml
```

Для изменения приглашения нужно переопределить функцию `prompt`. Например, после выполнения следующей команды приглашение будет состоять из пути к текущему каталогу и символа >:

```
PS C:\Users\andrv> function prompt{"$(Get-Location)> "}
C:\Users\andrv>
```

Как видим, вид приглашения меняется сразу после задания нового содержимого функции `prompt`.

> ***Замечание***
>
> При определении функции `prompt` мы внутри строки, заключенной в двойные кавычки, использовали конструкцию `$(...)`, которая называется *подвыражением* (subexpression). Подвыражение — это блок кода на языке PowerShell, который в строке заменяется на значение, полученное в результате выполнения этого кода.

В функции `prompt` можно не только формировать приглашение командной строки, но и выполнять любые другие действия. Например, с помощью функции `prompt` можно решить проблему отображения длинных путей к текущему каталогу (такие пути при отображении в приглашении командной строки занимают много места, и работать становится неудобно). Если определить функцию `prompt` так, как в листинге 4.1, то путь к текущему каталогу будет отображаться не в приглашении командной строки, а в заголовке командного окна (рис. 4.8).

Листинг 4.1. Вывод на экран всех параметров сценария

```
function prompt {
  (Get-Host).UI.RawUI.WindowTitle="PS $(Get-Location)"
  "PS > "
  }
```

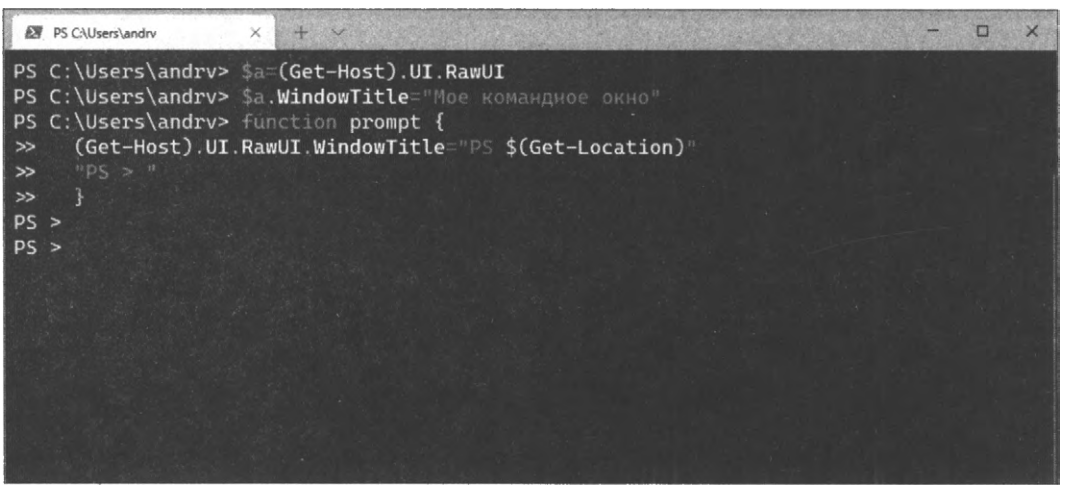


Рис. 4.8. Окно PowerShell, в качестве заголовка — путь к текущему каталогу

Дополнительные возможности по настройке приглашения командной строки, в том числе интеграция с системой управления версиями Git и применение тем оформления, предоставляемых модулем Oh My Posh, описаны в *приложении 3*.

Настройка пользовательских профилей

Ранее мы рассмотрели, каким образом можно настроить некоторые аспекты оболочки PowerShell под свои требования. Мы научились создавать собственные псевдонимы и функции для часто используемых командлетов или других команд, определять собственные переменные и диски, изменять приглашение командной строки.

Однако все эти настройки и изменения будут действовать только во время текущего сеанса работы и утратят силу после выхода из оболочки.

Для сохранения изменений необходимо создать так называемый *профиль* PowerShell и записать в него все команды, которые определяют нужные нам псевдонимы, функции, переменные и т. п. Профиль — это сценарий, который будет автоматически выполняться при каждом запуске PowerShell. Грамотно созданный профиль может упростить работу в PowerShell и администрирование операционной системы.

В PowerShell для пользователя могут быть заданы четыре разных профиля, расположенных в домашнем каталоге пользователя (путь к этому каталогу содержится в переменной `$Home`) и в установочном каталоге PowerShell (путь к нему хранится в переменной `$PSHOME`).

Во-первых, профиль может храниться в файле $PSHome\profile.ps1. Данный профиль действует на всех пользователей и на все оболочки.

Во-вторых, профиль может находиться в файле $PSHome\Microsoft.PowerShell_profile.ps1. Действие этого профиля распространяется на всех пользователей, но только на одну оболочку.

Профиль третьего типа может находиться в файле $Home\Documents\PowerShell\profile.ps1. Действие данного профиля распространяется только на текущего пользователя и на все оболочки.

Наконец, профиль может храниться в файле $Home\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1. Действие этого профиля распространяется только на текущего пользователя и только на оболочку Microsoft.PowerShell.

> ***Замечание***
>
> Если на жестком диске имеется несколько профилей, которые могут быть загружены в конкретной ситуации, то предпочтение будет отдано более узконаправленному.

Обычно при работе с оболочкой PowerShell используют профиль, специфичный для пользователя и оболочки, который называется *пользовательским профилем*. Данные о расположении этого профиля хранятся в специальной переменной `$profile`:

```
PS C:\Users\andrv> $profile
C:\Users\andrv\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
```

Находясь в оболочке PowerShell, можно с помощью командлета `Test-Path` проверить, создан ли уже пользовательский профиль:

```
PS F:\> Test-Path $profile
False
```

Если сценарий с профилем существует, эта команда возвратит `True`, в противном случае — `False`.

Для создания нового пользовательского профиля или изменения уже существующего нужно открыть в текстовом редакторе файл, путь к которому хранится в переменной `$profile`. Сделать это можно непосредственно из Проводника Windows или из оболочки PowerShell с помощью следующей команды:

```
PS C:\Users\andrv> notepad $profile
```

Если файл с профилем уже существовал на диске, то в результате выполнения последней команды будет открыт Блокнот Windows с содержимым файла Microsoft.PowerShell_profile.ps1. Если же профиль еще не был создан, то будет выдано диалоговое окно с предложением создать этот файл. Нажав кнопку **OK** в этом окне, мы попадем в пустое окно Блокнота Windows. Теперь нужно ввести команды, которые будут выполняться при загрузке PowerShell (например, функцию `prompt` из листинга 4.1), и сохранить сценарий Microsoft.PowerShell_profile.ps1 в каталоге $Home\Documents\WindowsPowerShell.

После сохранения файла с профилем можно проверить его содержимое из оболочки:

```
PS C:\Users\andrv> type $profile
function prompt {
  (Get-Host).UI.RawUI.WindowTitle="PS $(Get-Location)"
  "PS > "
  }
```

Итак, мы создали свой пользовательский профиль, который должен загружаться при каждом старте оболочки PowerShell и изменять заголовок окна и приглашение командной строки.

Завершим текущий сеанс работы и заново запустим PowerShell. Скорее всего, мы получим сообщение об ошибке наподобие следующего:

```
. : Невозможно загрузить файл
C:\Users\andrv\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1,
т. к. выполнение сценариев отключено в этой системе. Для получения
дополнительных сведений см. about_Execution_Policies по адресу
https:/go.microsoft.com/fwlink/?LinkID=135170.
строка:1 знак:3
+ . 'C:\Users\andrv\Documents\WindowsPowerShell\Microsoft.Powe ...
+   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : Ошибка безопасности: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
```

Дело в том, что для повышения уровня безопасности по умолчанию в PowerShell действует *политика выполнения*, которая запрещает загрузку профилей и вообще выполнение любых сценариев (разрешается только выполнять команды в интерактивном режиме). Поэтому нам нужно научиться настраивать политику выполнения таким образом, чтобы сценарии PowerShell начали запускаться.

Политики выполнения сценариев

Политика выполнения (execution policy) оболочки PowerShell определяет, можно ли на данном компьютере выполнять сценарии PowerShell (в том числе загружать пользовательские профили — сценарии, которые автоматически выполняются при запуске оболочки), и если да, должны ли они быть подписаны цифровой подписью.

Возможные политики выполнения PowerShell описаны в табл. 4.3 (получить аналогичную информацию в PowerShell можно с помощью команды `Get-Help about_signing`).

Таблица 4.3. *Политики выполнения PowerShell*

Название политики	Описание
Restricted	Данная политика выполнения используется по умолчанию, она запрещает выполнение сценариев и загрузку профилей (можно пользоваться только одиночными командами PowerShell в интерактивном режиме)
AllSigned	Выполнение сценариев PowerShell разрешено, однако все сценарии (как загруженные из Интернета, так и локальные) должны иметь цифровую подпись надежного издателя Перед выполнением сценариев надежных издателей запрашивается подтверждение
RemoteSigned	Выполнение сценариев PowerShell разрешено, при этом все сценарии и профили, загруженные из Интернета, должны иметь цифровую подпись надежного издателя, а локальные сценарии могут быть неподписанными При запуске сценариев надежных издателей подтверждение не запрашивается
Unrestricted	Разрешается выполнение любых сценариев PowerShell без проверки цифровой подписи При запуске сценариев и профилей, загруженных из Интернета, выдается предупреждение

Узнать, какая политика выполнения является активной, можно с помощью командлета `Get-ExecutionPolicy`:

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

По умолчанию в PowerShell действует политика Restricted, запрещающая запуск любых сценариев, включая пользовательские профили.

Командлет `Set-ExecutionPolicy` позволяет сменить политику выполнения. Параметры политики хранятся в системном реестре, поэтому для их изменения нужно запустить PowerShell от имени администратора. Например, для установки политики выполнения RemoteSigned нужно выполнить следующую команду:

```
Set-ExecutionPolicy RemoteSigned
```

> ***Замечание***
>
> Не забывайте о функции автоматического завершения команд — нет необходимости запоминать названия возможных политик, можно просто нажимать клавишу <Tab> после имени командлета, и система будет подставлять различные варианты политик

Проверим снова текущую политику:

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

Теперь завершим сеанс работы в PowerShell и вновь запустим оболочку. На этот раз никаких сообщений об ошибке выдаваться не будет, функция `prompt` из пользовательского профиля должна отработать корректно и в заголовке командного окна PowerShell отобразится путь к текущему каталогу (см. рис. 4.8).

Итоги

- Возможности по редактированию текста при работе в оболочке PowerShell зависят от используемого терминала.
- PowerShell поддерживает автоматическое завершение при вводе путей к файлам и каталогам, а также имен команд и переменных.
- Командлет `Get-Help` позволяет получить справочную информацию о командах и других аспектах PowerShell.
- Сеанс работы в командной строке PowerShell можно протоколировать — команды и результат их выполнения будут автоматически сохраняться во внешнем файле.
- Внешний вид командной строки PowerShell зависит от настроек терминала, в котором запущена оболочка.
- Дополнительные модули (PSReadLine, posh-git, Oh My Posh) расширяют возможности управления командной строкой и ее оформления.
- При запуске PowerShell автоматически выполняются сценарии-профили, в которых можно определить нужные псевдонимы, функции и переменные.
- Режимы выполнения сценариев PowerShell зависят от установленной политики. Узнать политику выполнения можно с помощью командлета `Get-ExecutionPolicy`, изменить — с помощью `Set-ExecutionPolicy`.

ГЛАВА 5

Работа с объектами

В предыдущих главах мы уже неоднократно говорили о том, что все действия в оболочке PowerShell связаны с операциями над объектами.

Нелишним будет напомнить, что объект — это совокупность данных (*поля* или *свойства* объекта) и способов работы с этими данными (*методы* объекта). Конкретная структура объекта, т. е. состав свойств и методов, задается его *типом*. Набор типов, использующихся в Windows PowerShell, базируется на типах унифицированной платформы .NET Framework, повсеместно применяемой в операционной системе Windows. Например, любому файлу на жестком диске в PowerShell соответствует .NET-объект типа `System.IO.FileInfo`.

Свойство объекта — это сведения о его атрибутах. Так, у объекта `System.IO.FileInfo` имеется свойство `Length` (длина), соответствующее размеру файла, который представлен данным объектом.

Метод объекта является действием, которое можно совершать над этим объектом. Например, у объекта `System.IO.FileInfo` имеется метод `CopyTo`, с помощью которого можно скопировать представленный данным объектом файл на уровне файловой системы.

Свойства и методы объектов используются в командлетах PowerShell для выполнения различных действий и работы с данными. Как мы уже неоднократно упоминали, в PowerShell поддерживается механизм конвейеризации (композиции) команд, упрощающий выполнение операций над объектами.

Конвейеризация объектов в PowerShell

Механизм композиции, когда выходной поток одной команды перенаправляется во входной поток другой, представляет собой, вероятно, наиболее ценную концепцию интерфейсов командной строки. Конвейеры не только снижают усилия, прилагаемые при вводе сложных команд, но и облегчают отслеживание потока работы в командах. Полезной чертой конвейеров является то, что они не зависят от числа передаваемых элементов, т. к. конвейер действует на каждый элемент отдельно. Благодаря этому, как правило, снижается потребление ресурсов для сложных команд и возникает возможность получать выводимую информацию немедленно.

В оболочке PowerShell также очень широко используется механизм конвейеризации команд, однако здесь по конвейеру передается не поток текста, как во всех других оболочках, а объекты. При этом с элементами конвейера можно производить различные манипуляции: фильтровать объекты по определенному критерию, сортировать и группировать объекты, изменять их структуру.

Конвейер в PowerShell — это последовательность команд, разделенных между собой знаком | (вертикальная черта). Каждая команда в конвейере получает объект от предыдущей команды, выполняет определенные операции над ним и передает следующей команде в конвейере. С точки зрения пользователя, объекты упаковывают связанную информацию в форму, в которой информацией проще манипулировать как единым блоком и из которой при необходимости извлекаются определенные элементы.

С данными, которые передаются между командами в виде объектов, удобнее работать, чем с текстовой информацией. Ведь команда, принимающая поток текста от другой утилиты, должна его проанализировать, разобрать и выделить нужную ей информацию, а это может быть непросто, т. к. обычно вывод команды больше ориентирован на визуальное восприятие человеком (это естественно для интерактивного режима работы), а не на удобство последующего синтаксического разбора.

При передаче по конвейеру объектов этой проблемы не возникает, здесь нужная информация извлекается из элемента конвейера простым обращением к соответствующему свойству объекта. Однако теперь возникает новый вопрос: как можно узнать, какие именно свойства есть у объектов, передаваемых по конвейеру? Ведь при выполнении того или иного командлета мы на экране видим только одну или несколько колонок отформатированного текста. Например, запустим командлет `Get-Process`, который выводит информацию об активных процессах:

```
PS C:\> Get-Process

Handles  NPM(K)   PM(K)  WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------   -----  ----- -----   ------     -- -----------
    158      11   45644  22084   126   159.69   2072 AcroRd32
     98       5    1104    284    32     0.10    256 alg
     39       1     364    364    17     0.26   1632 ati2evxx
     57       3    1028    328    30     0.38    804 atiptaxx
    434       6    2548   3680    27    21.96    800 csrss
     64       3     812    604    29     0.22   1056 ctfmon
    364      11   14120   9544    69    11.82    456 explorer
     24       2    1532   2040    29     5.34   2532 Far
. . .
```

Фактически на экране мы видим только сводную информацию (результат форматирования полученных данных), а не полное представление выходного объекта. Из этой информации непонятно, сколько точно свойств имеется у объектов, генерируемых командой `Get-Process`, и какие имена имеют эти свойства. Например, мы хотим найти все "зависшие" процессы, которые не отвечают на запросы системы.

Можно ли это сделать с помощью командлета Get-Process, какое именно свойство для этого нужно проверять у выводимых объектов?

Для ответа на подобные вопросы нужно, прежде всего, научиться исследовать структуру объектов PowerShell, узнавать, какие свойства и методы имеются у этих объектов.

Просмотр структуры объектов (командлет *Get-Member*)

Для анализа структуры объектов, возвращаемых определенной командой, проще всего направить эти объекты по конвейеру на командлет Get-Member (псевдоним gm), например:

```
PS C:\> Get-Process | Get-Member

   TypeName: System.Diagnostics.Process

Name          MemberType     Definition
----          ----------     ----------
Handles       AliasProperty  Handles = Handlecount
Name          AliasProperty  Name = ProcessName
NPM           AliasProperty  NPM = NonpagedSystemMemorySize
PM            AliasProperty  PM = PagedMemorySize
VM            AliasProperty  VM = VirtualMemorySize
WS            AliasProperty  WS = WorkingSet
. . .
Responding    Property       System.Boolean Responding {get;}
. . .
```

В результате на экране мы видим, какой .NET-тип имеют объекты, возвращаемые в ходе работы исследуемого командлета (в приведенном примере это тип System.Diagnostics.Process), а также полный список элементов объекта (в частности, интересующее нас свойство Responding, определяющее "зависшие" процессы). При этом на экран выводится очень много элементов разных типов (имена и псевдоним свойств, имена методов и т. д.), и такой длинный список становится неудобно просматривать. Командлет Get-Member имеет параметр -MemberType, позволяющий перечислить только элементы объекта определенного типа. Например, для вывода только элементов объекта, являющихся свойствами этого объекта, используется параметр -MemberType со значением Property:

```
PS C:\> Get-Process | Get-Member -MemberType Property

   TypeName: System.Diagnostics.Process

Name                MemberType Definition
----                ---------- ----------
BasePriority        Property   System.Int32 BasePriority {get;}
Container           Property   System.ComponentModel.IContainer...
```

```
EnableRaisingEvents  Property   System.Boolean EnableRaisingEven...
ExitCode             Property   System.Int32 ExitCode {get;}
ExitTime             Property   System.DateTime ExitTime {get;}
Handle               Property   System.IntPtr Handle {get;}
HandleCount          Property   System.Int32 HandleCount {get;}
HasExited            Property   System.Boolean HasExited {get;}
Id                   Property   System.Int32 Id {get;}
MachineName          Property   System.String MachineName {get;}
. . .
Responding           Property   System.Boolean Responding {get;}
. . .
```

Как видим, процессам операционной системы соответствуют объекты, имеющие очень много свойств, на экран же при работе командлета `Get-Process` выводятся лишь несколько из них. На самом деле способы отображения в оболочке PowerShell объектов различных типов задаются несколькими конфигурационными файлами с расширением ps1xml (в формате XML), находящимися в каталоге, где установлен файл powershell.exe (путь к этому каталогу хранится в переменной `$PSHome`). Список этих файлов можно получить с помощью следующей команды:

```
PS C:\> dir $pshome\*format*.ps1xml

    Directory: C:\Windows\System32\WindowsPowerShell\v1.0

Mode                LastWriteTime    Length Name
----                -------------    ------ ----
-a----       12/7/2019  12:10 PM     12825 Certificate.format.ps1xml
-a----       12/7/2019  12:10 PM      4994 Diagnostics.Format.ps1xml
-a----       12/7/2019  12:10 PM    138013 DotNetTypes.format.ps1xml
-a----       12/7/2019  12:10 PM     10112 Event.Format.ps1xml
-a----       12/7/2019  12:10 PM     25306 FileSystem.format.ps1xml
-a----       12/7/2019  12:10 PM     91655 Help.format.ps1xml
-a----       12/7/2019  12:10 PM    138625 HelpV3.format.ps1xml
-a----       12/7/2019  12:10 PM    206468 PowerShellCore.format.ps1xml
-a----       12/7/2019  12:10 PM      4097 PowerShellTrace.format.ps1xml
-a----       12/7/2019  12:10 PM      8458 Registry.format.ps1xml
-a----       12/7/2019  12:10 PM     16598 WSMan.Format.ps1xml
```

В частности, правило форматирования объекта типа `System.Diagnostics.Process` находится в файле `DotNetTypes.format.ps1xml`. Напрямую редактировать конфигурационные файлы не рекомендуется, в случае необходимости можно создать собственные файлы форматирования и с помощью командлета `Update-FormatData` включить их в состав автоматически загружаемых файлов.

Теперь, когда мы знаем, какие свойства имеют объекты, передаваемые по конвейеру, перейдем к рассмотрению возможных операций над элементами конвейера.

Фильтрация объектов (командлет *Where-Object*)

В PowerShell можно *фильтровать объекты* в конвейере, т. е. удалять из конвейера объекты, не удовлетворяющие определенному условию. Для этого используется командлет `Where-Object`, позволяющий проверить каждый объект, проходящий через конвейер, и передать его дальше по конвейеру лишь в том случае, если объект удовлетворяет условиям проверки.

Саму проверку в `Where-Object` можно организовать двумя способами: с помощью блока кода и через операторы сравнения.

Использование блока кода

Проходящие через конвейер объекты проверяются в блоке кода, который указывается после имени командлета `Where-Object`. Блок кода (script block) — это одна или несколько команд PowerShell, заключенных в фигурные скобки `{}`.

> ***Замечание***
>
> Можно считать блок кода аналогом анонимной функции в других языках программирования.

Результатом выполнения блока кода в командлете `Where-Object` должно быть значение логического типа: `$True` (истина, в этом случае объект проходит далее по конвейеру) или `$False` (ложь, в этом случае объект далее по конвейеру не передается).

Например, для вывода информации об остановленных службах в системе (объекты, возвращаемые командлетом `Get-Service`, у которых свойство `Status` равно `Stopped`) можно использовать следующий конвейер:

```
PS C:\> Get-Service | Where-Object {$_.Status -eq "Stopped"}

Status   Name              DisplayName
------   ----              -----------
Stopped  Alerter           Оповещатель
Stopped  AppMgmt           Управление приложениями
Stopped  aspnet_state      ASP.NET State Service
Stopped  cisvc             Служба индексирования
Stopped  ClipSrv           Сервер папки обмена
. . .
```

Вместо `Where-Object` можно использовать его краткие псевдонимы: `where` или просто символ `?`:

```
PS C:\> Get-Service | where {$_.Status -eq "Stopped"}
PS C:\> Get-Service | ? {$_.Status -eq "Stopped"}
```

Другой пример: оставим в конвейере только те процессы, у которых значение идентификатора (свойство `Id`) больше 1000:

```
PS C:\> Get-Process | Where-Object {$_.Id -gt 1000}

Handles  NPM(K)    PM(K)   WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----   ----- -----   ------     -- -----------
    158       9    37768   26620   125    28.49   1752 AcroRd32
     39       1      364     420    17     0.16   1632 ati2evxx
     57       3     1028     804    30     0.40   1988 atiptaxx
     24       2     1460    1052    29     0.62   2084 Far
    720      75    38516   10508   153    50.96   1756 kavsvc
     36       2      728      32    23     0.06   1792 klswd
     33       2      792    1984    25     1.15   1412 notepad
    242     158    30544    5780   180     7.96   1784 outpost
    252       5    34384   27192   137     5.15   2904 powershell
    143       5     3252    1028    42     0.31   1528 spoolsv
    194       5     2928    1340    59     0.34   1040 svchost
    301      14     1784    1316    37     0.80   1116 svchost
   1647      65    20820   11548   101    13.21   1152 svchost
     55       3     1088     724    27     0.46   1224 svchost
    170       6     1568     960    35     0.14   1604 svchost
    120       4     2356    1292    35     0.22   1876 svchost
     22       2      504     584    23     0.06   1764 winampa
    430      13     8472   11352   246    57.63   3216 WINWORD
    154       4     2112    2104    38    28.13   2032 wmiprvse
```

В блоках кода командлета `Where-Object` для обращения к текущему объекту конвейера и извлечения нужных свойств этого объекта используется специальная переменная `$_`, которая создается оболочкой PowerShell автоматически.

Замечание

Данная переменная используется и в других командлетах, производящих обработку элементов конвейера.

Как можно понять из примеров, в блоке кода используются специальные *операторы сравнения*. Основные операторы сравнения приведены в табл. 5.1.

Замечание

В PowerShell для операторов сравнения не используются обычные символы > или <, т. к. в командной строке они означают перенаправление ввода/вывода.

Таблица 5.1. *Операторы сравнения в PowerShell*

Оператор	Значение	Пример (возвращается значение True)
`-eq`	равно	`10 -eq 10`
`-ne`	не равно	`9 -ne 10`
`-lt`	меньше	`3 -lt 4`
`-le`	меньше или равно	`3 -le 4`

Таблица 5.1 (окончание)

Оператор	Значение	Пример (возвращается значение True)
-gt	больше	4 -gt 3
-ge	больше или равно	4 -ge 3
-like	сравнение на совпадение с учетом подстановочного знака в тексте	"file.doc" -like "f*.doc"
-notlike	сравнение на несовпадение с учетом подстановочного знака в тексте	"file.doc" -notlike "f*.rtf"
-contains	содержит	1,2,3 -contains 1
-notcontains	не содержит	1,2,3 -notcontains 4
-in	входит	2 -in 1..4
-notin	не входит	5 -notin 1..4

Операторы сравнения можно соединять друг с другом с помощью *логических операторов* (табл. 5.2).

Таблица 5.2. Логические операторы в PowerShell

Оператор	Значение	Пример (возвращается значение True)
-and	логическое И	(10 -eq 10) -and (1 -eq 1)
-or	логическое ИЛИ	(9 -ne 10) -or (3 -eq 4)
-not	логическое НЕ	-not (3 -gt 4)
!	логическое НЕ	!(3 -gt 4)

Замечание

Более подробно операторы сравнения и логические операторы рассматриваются в *главе 8*.

Использование оператора сравнения

Начиная с третьей версии, PowerShell поддерживает более простой и близкий к естественному языку вариант командлета `Where-Object`.

Здесь не нужно указывать блок кода в фигурных скобках и использовать переменную `$_` для доступа к объекту, поступающему по конвейеру. Достаточно просто указать свойство, по которому производится фильтрация, нужный оператор сравнения (в этом случае он будет являться параметром командлета `Where-Object`) и значение, с которым сравнивается свойство объекта. Например:

```
PS C:\Users\andrv> Get-Service | Where-Object -Property Status -eq -Value
"Stopped"
```

```
Status   Name              DisplayName
------   ----              -----------
Stopped  Alerter           Оповещатель
Stopped  AppMgmt           Управление приложениями
Stopped  aspnet_state      ASP.NET State Service
Stopped  cisvc             Служба индексирования
Stopped  ClipSrv           Сервер папки обмена
. . .
```

Названия параметров `-Property` и `-Value` можно опустить, а вместо `Where-Object` использовать псевдоним `where`. В этом случае фильтрация будет выглядеть кратко и выразительно:

```
PS C:\> Get-Service | where Status -eq "Stopped"

Status   Name              DisplayName
------   ----              -----------
Stopped  Alerter           Оповещатель
Stopped  AppMgmt           Управление приложениями
Stopped  aspnet_state      ASP.NET State Service
Stopped  cisvc             Служба индексирования
Stopped  ClipSrv           Сервер папки обмена
. . .
```

Замечание

Если условие фильтрации содержит логические операторы (является составным), то придется воспользоваться `Where-Object` с указанием блока кода, т. к в простом варианте `Where-Object` с оператором сравнения логические операторы не поддерживаются.

Сортировка объектов (командлет *Sort-Object*)

Сортировка элементов конвейера — еще одна часто применяемая операция, которую осуществляет командлет `Sort-Object` (псевдоним `sort`). Этому командлету передаются имена свойств, по которым нужно произвести сортировку объектов, проходящих по конвейеру, а он возвращает данные, упорядоченные по значениям этих свойств.

Например, для получения списка запущенных в системе процессов, упорядоченного по затраченному процессорному времени (свойство `cpu`), можно воспользоваться следующим конвейером:

```
PS C:\> Get-Process | Sort-Object -Property cpu

Handles  NPM(K)    PM(K)   WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----   ----- -----   ------     -- -----------
      0       0        0      16     0               0 Idle
     36       2      728      32    23     0.05   1792 klswd
     98       5     1104     764    32     0.09    252 alg
     21       1      164      60     4     0.09    748 smss
     39       1      364     464    17     0.12   1644 ati2evxx
```

```
   163       6     1536     1404     35      0.12   1612 svchost
    55       3     1088      852     27      0.14   1220 svchost
    22       2      504      712     23      0.14    772 winampa
   120       4     2364     1228     35      0.26   1876 svchost
   193       5     2916     1488     59      0.29   1040 svchost
    64       3      812     1080     29      0.30   1252 ctfmon
   140       5     3208     1220     41      0.32   1524 spoolsv
   281      14     1764     1688     37      0.34   1120 svchost
    57       3     1028      996     30      0.39    932 atiptaxx
   503      52     7296     3596     51      2.47    836 winlogon
   259       6     1432     1340     19      2.48    880 services
   341       8     3572     1856     40      5.36    892 lsass
   240     158    29536    10388    175      5.58   1780 outpost
   149       4     2940     1108     41      9.29   1248 kav
   398       5    36140    26408    137      9.97   1984 powershell
   375      12    15020    10456     75     14.03   1116 explorer
   376       0        0       36      2     14.97      4 System
   409       6     2500     3192     26     20.10    812 csrss
  1513      54    13528     9800     95     25.78   1156 svchost
   717      75    37432      704    145     56.97   1748 kavsvc
   152       4     2372     2716     38     58.09   2028 wmiprvse
   307      13    10952    27080    173   9128.03   1200 WINWORD
```

Параметр `-Property` в командлете `Sort-Object` используется по умолчанию, поэтому имя этого параметра можно не указывать. Для сортировки в обратном порядке используется параметр `-Descending`:

```
PS C:\> Get-Process | Sort-Object cpu -Descending

Handles  NPM(K)    PM(K)  WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----  ----- -----   ------     -- -----------
    307      13    10956  27040   173  9152.23   1200 WINWORD
    152       4     2372   2716    38    59.19   2028 wmiprvse
    717      75    37432   1220   145    57.15   1748 kavsvc
   1524      54    13528   9800    95    26.13   1156 svchost
    410       6     2508   3224    26    20.62    812 csrss
    376       0        0     36     2    15.11      4 System
    377      13    15020  10464    75    14.20   1116 explorer
    374       5    36484  26828   137    10.53   1984 powershell
    149       4     2940   1108    41     9.34   1248 kav
    240     158    29536  10388   175     5.61   1780 outpost
    344       8     3572   1856    40     5.40    892 lsass
    512      53     7324   3608    51     2.51    836 winlogon
    259       6     1432   1340    19     2.48    880 services
     57       3     1028    996    30     0.39    932 atiptaxx
    281      14     1764   1688    37     0.34   1120 svchost
    140       5     3208   1220    41     0.32   1524 spoolsv
     64       3      812   1080    29     0.30   1252 ctfmon
    193       5     2916   1488    59     0.29   1040 svchost
```

```
    120       4     2364   1228    35     0.26   1876 svchost
     22       2      504    712    23     0.15    772 winampa
     55       3     1088    852    27     0.14   1220 svchost
     39       1      364    464    17     0.13   1644 ati2evxx
    163       6     1536   1404    35     0.12   1612 svchost
     21       1      164     60     4     0.09    748 smss
     98       5     1104    764    32     0.09    252 alg
     36       2      728     32    23     0.05   1792 klswd
      0       0        0     16     0                0 Idle
```

В рассмотренных нами примерах конвейеры состояли из двух командлетов. Это не обязательное условие, конвейер может объединять и большее количество команд, например:

```
PS C:\> Get-Process | Where-Object Id -gt 1000 | Sort-Object cpu -Descending

Handles  NPM(K)    PM(K)  WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----  ----- -----   ------     -- -----------
    307      13    10956  27040   173  9152.23   1200 WINWORD
    152       4     2372   2716    38    59.19   2028 wmiprvse
    717      75    37432   1220   145    57.15   1748 kavsvc
   1524      54    13528   9800    95    26.13   1156 svchost

    377      13    15020  10464    75    14.20   1116 explorer
    374       5    36484  26828   137    10.53   1984 powershell
    149       4     2940   1108    41     9.34   1248 kav
    240     158    29536  10388   175     5.61   1780 outpost
    281      14     1764   1688    37     0.34   1120 svchost
    140       5     3208   1220    41     0.32   1524 spoolsv
     64       3      812   1080    29     0.30   1252 ctfmon
    193       5     2916   1488    59     0.29   1040 svchost
    120       4     2364   1228    35     0.26   1876 svchost
     55       3     1088    852    27     0.14   1220 svchost
     39       1      364    464    17     0.13   1644 ati2evxx
    163       6     1536   1404    35     0.12   1612 svchost
     36       2      728     32    23     0.05   1792 klswd
```

В результате выполнения последнего конвейера из трех командлетов мы получили упорядоченный по количеству затраченного процессорного времени список процессов, идентификатор которых больше 1000.

Выделение объектов и свойств (командлет *Select-Object*)

В PowerShell имеется командлет `Select-Object` (псевдоним `select`), с помощью которого можно выделять указанное количество объектов с начала или с конца конвейера, выбирать уникальные объекты из конвейера, а также выделять определенные свойства в объектах, проходящих по конвейеру.

Для выделения из конвейера нескольких первых или последних объектов следует воспользоваться соответственно параметрами `-First` или `-Last` командлета `Select-Object`. Например, следующий конвейер команд выведет на экран информацию о пяти последних процессах, использующих наибольший объем памяти:

```
PS C:\> Get-Process | Sort-Object WS | Select-Object -Last 5

Handles  NPM(K)    PM(K)  WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----  ----- -----   ------     -- -----------
    398      12    14736   8096    78    12.99    740 explorer
   1638      66    21368  12292   103    30.03   1152 svchost
    280      12    10252  14900   139   124.56   3216 WINWORD
    158       9    37776  19704   125    36.21   1752 AcroRd32
    297       6    38408  20844   137     8.53   2904 powershell
```

Разберем работу данного конвейера команд. Первый командлет в конвейере (`Get-Process`) возвращает массив объектов, соответствующих запущенным в системе процессам. Второй командлет (`Sort-Object`) упорядочивает проходящие по конвейеру объекты по значению свойства `WS` (объем памяти, занимаемой процессом). Наконец, третий командлет (`Select-Object`) выбирает из упорядоченного массива объекта последние пять элементов.

Предположим теперь, что нам нужно получить список запущенных в системе процессов, в котором были бы указаны только имена процессов и их идентификаторы. Если мы не помним названия нужных свойств, то можно с помощью командлета `Get-Member` вновь просмотреть структуру возвращаемых командой `Get-Process` объектов:

```
PS C:\> Get-Process | Get-Member -MemberType Property

   TypeName: System.Diagnostics.Process

Name              MemberType Definition
----              ---------- ----------
BasePriority      Property   System.Int32 BasePriority {get;}
. . .
Id                Property   System.Int32 Id {get;}
. . .
ProcessName       Property   System.String ProcessName {get;}
. . .
WorkingSet64      Property   System.Int64 WorkingSet64 {get;}
```

Итак, в итоговых объектах нам нужно оставить только свойства `ProcessName` и `Id`. Это можно сделать, указав имена нужных свойств в качестве параметров командлета `Select-Object`:

```
PS C:\> Get-Process | Select-Object ProcessName, Id

ProcessName                    Id
-----------                    --
AcroRd32                     1752
alg                           256
```

```
ati2evxx                    1632
atiptaxx                    1988
csrss                        804
ctfmon                       872
explorer                     740
Far                         2084
Idle                           0
kav                          884
kavsvc                      1756
klswd                       1792
lsass                        892
outpost                     1784
powershell                  2904
. . .
```

Посмотрим теперь, какой тип имеет объект, формируемый в конвейере командлетом `Select-Object`, и какие свойства имеются у этого объекта:

```
PS C:\> Get-Process | Select-Object ProcessName, Id | Get-Member

   TypeName: System.Management.Automation.PSCustomObject

Name        MemberType   Definition
----        ----------   ----------
Equals      Method       System.Boolean Equals(Object obj)
GetHashCode Method       System.Int32 GetHashCode()
GetType     Method       System.Type GetType()
ToString    Method       System.String ToString()
Id          NoteProperty System.Int32 Id=1752
ProcessName NoteProperty System.String ProcessName=AcroRd32
```

Как видим, выходной объект имеет тип `System.Management.Automation.PSCustomObject` (напомним, что командлет `Get-Process` возвращал объекты типа `System.Diagnostics.Process`) и у него имеются только два свойства — `ProcessName` и `Id`. Это связано с тем, что при использовании командлета `Select-Object` для выбора указанных свойств он копирует значения этих свойств из объектов, поступающих по конвейеру ему на вход, и создает новые объекты, которые содержат указанные свойства со скопированными значениями.

Командлет `Select-Object` может не только удалять из объектов ненужные свойства, но и добавлять новые *вычисляемые* свойства. Для этого новое свойство нужно представить в виде *хэш-таблицы*, где первый элемент (ключ `Name`) соответствует имени добавляемого свойства, а второй элемент (ключ `Expression`) — значению этого свойства для текущего элемента конвейера.

Замечание

Более подробно хэш-таблицы рассматриваются в *главе 7*.

Например, результатом выполнения следующего конвейера команд станет массив объектов, имеющих свойства ProcessName (имя запущенного процесса) и StartMin (минута, когда был запущен процесс):

```
PS C:\> Get-Process | Select-Object ProcessName, @{Name="StartMin";
Expression = {$_.StartTime.Minute}}
ProcessName           StartMin
-----------           --------
alg                         45
ati2evxx                    45
atiptaxx                    48
csrss                       45
ctfmon                      48
explorer                    48
. . .
```

Здесь свойство StartMin является вычисляемым, его значение для каждого элемента конвейера задается блоком кода {$_.StartTime.Minute}, где переменная $_ соответствует текущему объекту конвейера.

Выполнение произвольных действий над объектами в конвейере (командлет *ForEach-Object*)

Командлет ForEach-Object позволяет выполнить определенный блок кода на языке PowerShell для каждого объекта в конвейере. Другими словами, с помощью данного командлета можно производить произвольные операции над элементами конвейера.

Для примера давайте подсчитаем общий объем файлов, хранящихся в своем домашнем каталоге, путь к которому хранится в переменной окружения $HOME. Для этого сначала перейдем в данный каталог, объявим переменную $TotalLength и обнулим ее:

```
PS C:\> cd c:\
PS C:\Users\andrv> $TotalLength=0
```

Теперь выполним команду dir (напомним, что это псевдоним командлета Get-ChildItem) и результат ее работы передадим по конвейеру командлету ForEach-Object:

```
PS C:\Users\andrv> dir | ForEach-Object {$TotalLength+=$_.length}
```

В блоке кода командлета ForEach-Object к текущему значению переменной $TotalLength прибавляется значение свойства Length проходящего через конвейер объекта (размер соответствующего этому объекту файла). В результате в переменной $TotalLength будет храниться общий размер файлов в байтах:

```
PS C:\Users\andrv> $TotalLength
17809
```

Если из объектов, проходящих по конвейеру, нужно лишь извлечь определенное свойство, то можно просто указать имя этого свойства в качестве параметра ForEach-Object:

```
PS C:\Users\andrv> dir | ForEach-Object name
```

Псевдонимами для командлета ForEach-Object являются foreach и символ %:

```
PS C:\Users\andrv> dir | foreach name
PS C:\Users\andrv> dir | % {$TotalLength+=$_.length}
```

Группировка объектов (командлет *Group-Object*)

Проходящие по конвейеру объекты можно сгруппировать по значению определенных свойств с помощью командлета Group-Object. В одну группу будут попадать объекты, имеющие одинаковые значения указанных свойств (свойства могут быть вычисляемыми).

Рассмотрим пример. Командлет Get-Process генерирует объекты, имеющие свойство Company (название компании — разработчика определенного модуля, запущенного в операционной системе в качестве процесса). Выполним группировку этих объектов по значению свойства Company:

```
PS C:\> Get-Process | Group-Object Company

Count Name                      Group
----- ----                      -----
    1 Adobe Systems Incorpor... {AcroRd32}
   13 Microsoft Corporation     {alg, ctfmon, lsass, OUTLOOK...}
    7                           {csrss, Idle, kav, kavsvc...}
    5 Корпорация Майкрософт     {explorer, MAPISP32, scardsvr,...}
    1 Eugene Roshal & FAR Group {Far}
    2 Intel Corporation         {hkcmd, igfxpers}
    1 Корпорация Microsoft      {jview}
    1 Kaspersky Lab             {klnagent}
    1 Visioneer Inc             {OneTouchMon}
    1 Hewlett-Packard           {sdlaunch}
    1 Realtek Semiconductor ... {SOUNDMAN}
```

Как видите, в колонке Count отображается количество элементов в каждой из групп, а в колонке Group перечислены элементы, входящие в группы.

Если нужно просто узнать количество элементов в группах, можно запустить командлет Group-Object с параметром -NoElement:

```
PS C:\> Get-Process | Group-Object  Company -NoElement

Count Name
----- ----
    1 Adobe Systems Incorpor...
   13 Microsoft Corporation
    7
```

```
5 Корпорация Майкрософт
1 Eugene Roshal & FAR Group
2 Intel Corporation
1 Корпорация Microsoft
1 Kaspersky Lab
1
1 Visioneer Inc
1 Hewlett-Packard
1 Realtek Semiconductor ...
```

Измерение характеристик объектов (командлет *Measure-Object*)

В PowerShell имеется еще один полезный командлет — Measure-Object, предназначенный для выполнения функций агрегирования (сумма, выбор минимального, максимального или среднего значения) над свойствами элементов в конвейере объектов.

Рассмотрим пример. Ранее мы уже находили общий размер файлов в своем домашнем каталоге, применяя для этого командлет ForEach-Object:

```
PS C:\Users\andrv> $TotalLength=0
PS C:\Users\andrv> dir | ForEach-Object {$TotalLength+=$_.length}
PS C:\Users\andrv> $TotalLength
17809
```

С помощью командлета Measure-Object мы также сможем найти суммарный размер файлов. Для этого нужно указать, что Measure-Object должен для всех элементов конвейера просуммировать (параметр -Sum) значения свойства Length:

```
PS C:\Users\andrv> dir | Measure-Object -Property Length -Sum

Count    : 5
Average  :
Sum      : 17809
Maximum  :
Minimum  :
Property : Length
```

Результат будет выведен в поле Sum. Для выполнения других операций нужно указать соответствующий параметр: -Average для нахождения среднего значения, -Minimum или -Maximum для нахождения минимального или максимального значения соответственно:

```
PS C:\> dir | Measure-Object -Property Length -Minimum -Maximum -Average -Sum

Count    : 5
Average  : 6433.63636364
Sum      : 17809
```

```
Maximum  : 12458
Minimum  : 0
Property : Length
```

Также с помощью командлета `Measure-Object` можно получать статистическую информацию о текстовых файлах: количество строк, слов и символов.

Обращение к статическим методам и полям

Иногда при работе в PowerShell возникает необходимость воспользоваться методами, которые определены в классах (типах) .NET Framework, не создавая и не используя экземпляры этих классов. Такие классы называются *статическими*, т. к. они не создаются, не уничтожаются и не меняются. В частности, статическим является класс `System.Math`, методы которого часто используются для математических вычислений.

Для обращения к статическому классу его имя следует заключить в квадратные скобки, например:

```
PS C:\> [System.Math]

IsPublic IsSerial Name       BaseType
-------- -------- ----       --------
True     False    Math       System.Object
Для класса [System.Math] в PowerShell определена краткая аббревиатура [math].
PS C:\> [math]

IsPublic IsSerial Name       BaseType
-------- -------- ----       --------
True     False    Math       System.Object
```

Методы и свойства, определенные в статическом классе, также называются *статическими методами и полями*. Для их просмотра нужно передать имя нужного класса (в квадратных скобках) по конвейеру командлету `Get-Member` с параметром `-Static`:

```
PS C:\> [math] | Get-Member -Static

   TypeName: System.Math

Name            MemberType Definition
----            ---------- ----------
Abs             Method     static System.Single Abs(Single va. . .
Acos            Method     static System.Double Acos(Double d
Asin            Method     static System.Double Asin(Double d
Atan            Method     static System.Double Atan(Double d
Atan2           Method     static System.Double Atan2(Double . . .
BigMul          Method     static System.Int64 BigMul(Int32 a. . .
Ceiling         Method     static System.Double Ceiling(Doubl. . .
```

```
Cos             Method     static System.Double Cos(Double d)
Cosh            Method     static System.Double Cosh(Double v
DivRem          Method     static System.Int32 DivRem(Int32 a . .
Equals          Method     static System.Boolean Equals(Objec . .
Exp             Method     static System.Double Exp(Double d)
Floor           Method     static System.Double Floor(Double . . .
IEEERemainder   Method     static System.Double IEEERemainder. . .
Log             Method     static System.Double Log(Double d). . .
Log10           Method     static System.Double Log10(Double
Max             Method     static System.SByte Max(SByte val1. . .
Min             Method     static System.SByte Min(SByte val1. . .
Pow             Method     static System.Double Pow(Double x,. . .
ReferenceEquals Method     static System.Boolean ReferenceEqu. . .
Round           Method     static System.Double Round(Double . . .
Sign            Method     static System.Int32 Sign(SByte val. . .
Sin             Method     static System.Double Sin(Double a)
Sinh            Method     static System.Double Sinh(Double v
Sqrt            Method     static System.Double Sqrt(Double d
Tan             Method     static System.Double Tan(Double a)
Tanh            Method     static System.Double Tanh(Double v
Truncate        Method     static System.Decimal Truncate(Dec. . .
E               Property   static System.Double E {get;}
PI              Property   static System.Double PI {get;}
```

Как видим, методы и поля класса `System.Math` реализуют различные математические функции и константы, их легко распознать по названию.

Для доступа к определенному статическому методу или свойству используются два идущих подряд двоеточия (::), а не точка (.), как в обычных объектах. Например, для вычисления квадратного корня из числа (статического метода `Sqrt`) и сохранения результата в переменную используется следующая конструкция:

```
PS C:\> $a=[math]::Sqrt(25)
PS C:\> $a
5
```

Статические методы и поля есть у многих классов .NET Framework, с их помощью, например, можно:

- работать со строками (класс `[System.String]`, аббревиатура `[string]`);
- хранить показания и вычислять дату и время (класс `[System.DateTime]` с аббревиатурой `[datetime]`);
- рассчитывать промежутки времени или разницу между двумя показаниями времени (класс `[System.TimeSpan]` с аббревиатурой `[timespan]`);
- преобразовывать значения из одного числового формата в другой (класс `[System.Convert]`).

Итоги

- В PowerShell, как и во всех оболочках, используется механизм конвейеризации команд. Однако в PowerShell по конвейеру передается не поток текста, как во всех других оболочках, а объекты со свойствами и методами.
- Структуру поступающих по конвейеру объектов можно узнать с помощью командлета `Get-Member`.
- С элементами конвейера можно производить различные манипуляции: фильтровать объекты по определенному критерию, сортировать и группировать объекты, изменять их структуру.
- Использование конвейеров объектов в PowerShell — это пример декларативного подхода к программированию. Конвейеры PowerShell применяются там, где в других языках программирования прибегают к циклам и вычислениям выражений (императивный подход).
- PowerShell позволяет обращаться к свойствам и методам статических классов .NET. Таким образом, можно пользоваться множеством имеющихся в .NET математических функций или функциями для работы со строками и датами.

ГЛАВА 6

Управление выводом команд

Работая в оболочке PowerShell, мы пока не задумывались, каким образом система формирует строки текста, которые выводятся на экран в результате выполнения той или иной команды (напомним, что командлеты PowerShell возвращают .NET-объекты, которые, как правило, не знают, каким образом отображать себя на экране).

На самом деле в PowerShell имеется база данных (набор XML-файлов), содержащая *модули форматирования* по умолчанию для различных типов .NET-объектов. Эти модули определяют, какие свойства объекта отображаются при выводе и в каком формате: списка или таблицы. Располагаются модули форматирования в каталоге, где установлен PowerShell (путь к этому каталогу хранится в переменной окружения `$PSHOME`), посмотреть их список можно с помощью следующей команды:

```
PS C:\> Get-ChildItem $PSHOME/*format*

    Каталог: C:\Windows\System32\WindowsPowerShell\v1.0

Mode        LastWriteTime     Length Name
----        -------------     ------ ----
-a----   07.12.2019 12:10      12825 Certificate.format.ps1xml
-a----   07.12.2019 12:10       4994 Diagnostics.Format.ps1xml
-a----   07.12.2019 12:10     138013 DotNetTypes.format.ps1xml
-a----   07.12.2019 12:10      10112 Event.Format.ps1xml
-a----   07.12.2019 12:10      25306 FileSystem.format.ps1xml
-a----   07.12.2019 12:10      91655 Help.format.ps1xml
-a----   07.12.2019 12:10     138625 HelpV3.format.ps1xml
-a----   07.12.2019 12:10     206468 PowerShellCore.format.ps1xml
-a----   07.12.2019 12:10       4097 PowerShellTrace.format.ps1xml
-a----   07.12.2019 12:10       8458 Registry.format.ps1xml
-a----   07.12.2019 12:10      16598 WSMan.Format.ps1xml
```

Когда объект достигает конца конвейера, PowerShell определяет его тип и ищет его в списке объектов, для которых задано правило форматирования. Если данный тип в списке обнаружен, то к объекту применяется соответствующий модуль форматирования; если нет, то PowerShell просто отображает свойства этого .NET-объекта.

Также в PowerShell можно явно задавать правила форматирования данных, выводимых командлетами, и, подобно другим консольным приложениям, перенаправлять эти данные в файл, на принтер или в пустое устройство.

Форматирование выводимой информации

В традиционных оболочках команды и утилиты сами форматируют выводимые данные. Некоторые команды (например, `dir` в интерпретаторе cmd.exe) позволяют настраивать формат вывода с помощью специальных параметров-ключей.

В оболочке PowerShell вывод форматируют четыре специальных командлета `Format` (табл. 6.1). Это упрощает изучение, т. к. не нужно запоминать средства и параметры форматирования для других команд (остальные командлеты вывод не форматируют).

Таблица 6.1. *Командлеты PowerShell для форматирования вывода*

Командлет	Описание
`Format-Table`	Форматирует вывод команды в виде таблицы, столбцы которой содержат свойства объекта (также могут быть добавлены вычисляемые столбцы) Поддерживается возможность группировки выводимых данных
`Format-List`	Вывод форматируется как список свойств, в котором каждое свойство отображается на новой строке. Поддерживается возможность группировки выводимых данных
`Format-Custom`	Для форматирования вывода используется пользовательское представление (view)
`Format-Wide`	Форматирует объекты в виде широкой таблицы, в которой отображается только одно свойство каждого объекта

Как уже отмечалось, если ни один из командлетов `Format` явно не указан, то используется модуль форматирования по умолчанию, который определяется по типу отображаемых данных. Например, при выполнении командлета `Get-Service` данные по умолчанию выводятся как таблица с тремя столбцами (`Status`, `Name` и `DisplayName`):

```
PS C:\> Get-Service

Status   Name               DisplayName
------   ----               -----------
Stopped  Alerter            Оповещатель
Running  ALG                Служба шлюза уровня приложения
Stopped  AppMgmt            Управление приложениями
Stopped  aspnet_state       ASP.NET State Service
Running  Ati HotKey Poller  Ati HotKey Poller
Running  AudioSrv           Windows Audio
Running  BITS               Фоновая интеллектуальная служба пер...
```

```
Running  Browser            Обозреватель компьютеров
Stopped  cisvc              Служба индексирования
Stopped  ClipSrv            Сервер папки обмена
Stopped  clr_optimizatio... .NET Runtime Optimization Service v...
Stopped  COMSysApp          Системное приложение COM+
Running  CryptSvc           Службы криптографии
Running  DcomLaunch         Запуск серверных процессов DCOM
Running  Dhcp               DHCP-клиент
. . .
```

Для изменения формата выводимых данных нужно направить их по конвейеру соответствующему командлету `Format`. Например, следующая команда выведет список служб с помощью командлета `Format-List`:

```
PS C:\> Get-Service | Format-List

Name                  : Alerter
DisplayName           : Оповещатель
Status                : Stopped
DependentServices     : {}
ServicesDependedOn    : {LanmanWorkstation}
CanPauseAndContinue   : False
CanShutdown           : False
CanStop               : False
ServiceType           : Win32ShareProcess

Name                  : ALG
DisplayName           : Служба шлюза уровня приложения
Status                : Running
DependentServices     : {}
ServicesDependedOn    : {}
CanPauseAndContinue   : False
CanShutdown           : False
CanStop               : True
ServiceType           : Win32OwnProcess
. . .
```

Как видим, при использовании формата списка выводится больше сведений о каждой службе, чем в формате таблицы (вместо трех столбцов данных о каждой службе в формате списка выводятся девять строк данных). Однако это вовсе не означает, что командлет `Format-List` извлекает дополнительные сведения о службах. Эти данные содержатся в объектах, возвращаемых командой `Get-Service`, однако используемый по умолчанию командлет `Format-Table` отбрасывает их, потому что не может вывести на экран больше трех столбцов.

При форматировании вывода с помощью командлетов `Format-List` и `Format-Table` можно указывать имена свойства объекта, которые должны быть отображены (напомним, что просмотреть список свойств, имеющихся у объекта, позволяет командлет `Get-Member`). Например:

```
PS C:\> Get-Service | Format-List Name, Status, CanStop

Name    : Alerter
Status  : Stopped
CanStop : False

Name    : ALG
Status  : Running
CanStop : True

Name    : AppMgmt
Status  : Stopped
CanStop : False
. . .
```

Вывести все имеющиеся у объектов свойства можно с помощью параметра *, например:

```
PS C:\> Get-Service | Format-List *

Name                : Alerter
CanPauseAndContinue : False
CanShutdown         : False
CanStop             : True
DisplayName         : Оповещатель
DependentServices   : {}
MachineName         : .
ServiceName         : Alerter
ServicesDependedOn  : {LanmanWorkstation}
ServiceHandle       : SafeServiceHandle
Status              : Running
ServiceType         : Win32ShareProcess
Site                :
Container           :

Name                : ALG
CanPauseAndContinue : False
CanShutdown         : False
. . .
```

Перенаправление выводимой информации

В оболочке PowerShell имеется несколько командлетов, с помощью которых можно управлять выводом данных. Эти командлеты начинаются со слова `Out`, их список можно увидеть следующим образом:

```
PS C:\> Get-Command out-* | Format-Table Name

Name
----
Out-Default
Out-File
Out-GridView
Out-Host
Out-Null
Out-Printer
Out-String
```

По умолчанию выводимая информация передается командлету `Out-Default`, который, в свою очередь, делегирует всю работу по выводу строк на экран командлету `Out-Host`. Для понимания данного механизма нужно учитывать, что архитектура PowerShell подразумевает различие между собственно ядром оболочки (интерпретатором команд) и главным приложением-хостом (host), которое использует это ядро. В принципе, в качестве хоста может выступать любое приложение, в котором реализован ряд специальных интерфейсов, позволяющих корректно интерпретировать получаемую от PowerShell информацию. В нашем случае хостом является консольное окно (терминал), в котором мы работаем с оболочкой, и командлет `Out-Host` передает выводимую информацию в это окно.

Параметр `-Paging` командлета `Out-Host`, подобно команде `more` интерпретатора cmd.exe, позволяет организовать постраничный вывод информации, например:

```
PS C:\> Get-Help Get-Process -Full | Out-Host -Paging
ИМЯ
    Get-Process

ОПИСАНИЕ
    Отображает процессы, выполняющиеся на локальном компьютере.

СИНТАКСИС
    Get-Process [[-name] <string[]>] [<CommonParameters>]

. . .
<ПРОБЕЛ> следующая страница; <CR> следующая строка; Q выход
```

Сохранение данных в файл

Как уже упоминалось, PowerShell поддерживает перенаправление стандартного выходного потока команд в текстовые файлы с помощью стандартных операторов > и >>. Например, следующая команда выведет содержимое корневого каталога c:\ в текстовый файл d:\dir_c.txt (если данный файл существовал, то он будет перезаписан):

```
PS C:\> dir c:\ > d:\dir_c.txt
```

Если нужно перенаправить вывод команды в файл в режиме добавления (с сохранением прежнего содержимого данного файла), следует воспользоваться оператором >>:

```
PS C:\> dir c:\ >> d:\dir_c.txt
```

Кроме операторов перенаправления > и >> в PowerShell имеется командлет Out-File, также позволяющий направить выводимые данные вместо окна консоли в текстовый файл. При этом командлет Out-File имеет несколько дополнительных параметров, с помощью которых можно более гибко управлять выводом: задавать тип кодировки файла, задавать длину выводимых строк в знаках, выбирать режим перезаписи файла (табл. 6.2).

Таблица 6.2. *Некоторые параметры командлета Out-File*

Параметр	Описание
-FilePath	Указывает путь к выходному файлу
-Encoding	Определяет кодировку выходного файла Допустимые значения Unicode, UTF7, UTF8, UTF32, ASCII, BigEndianUnicode, Default и OEM По умолчанию в PowerShell используется кодировка Unicode Для сохранения текста в Windows-кодировке следует выбирать значение Default (кодировка текущей кодовой страницы ANSI), для сохранения текста в DOS-кодировке — значение OEM
-Width	Указывает число знаков в каждой выходной строке
-Append	Записывает выходные данные в конец существующего файла, а не замещает его содержимое
-noClobber	Если выходной файл уже существует, он не будет перезаписываться (по умолчанию, если файл существует по указанному пути, командлет Out-File перезаписывает его без предупреждения) Если одновременно используются параметры -Append и -NoClobber, выходные данные записываются в конец существующего файла

Например, следующая команда сохранит в файле help.txt детальный вариант встроенной справки по командлету Get-Process в Windows-кодировке:

```
PS C:\Users\andrv> Get-Help Get-Process -Detailed | Out-File -FilePath
.\help.txt -Encoding "Default"
```

Печать данных

Данные можно вывести непосредственно на принтер с помощью командлета Out-Printer. При этом печать может производиться как на принтере по умолчанию (никаких специальных параметров для этого указывать не нужно), так и на произвольном принтере (в этом случае отображаемое имя принтера должно быть указано в качестве значения параметра -Name). Например:

```
PS C:\script> Get-Process | Out-Printer -Name "Xerox Phaser 3500 PCL 6"
```

Подавление вывода

Командлет `Out-Null` служит для отбрасывания любых своих входных данных. Это может пригодиться для подавления вывода на экран ненужных сведений, полученных в качестве побочного эффекта выполнения какой-либо команды. Например, при создании каталога командой `mkdir` на экран выводится его содержимое:

```
PS C:\> mkdir klop

    Каталог: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----        26.05.2020      15:01            klop
```

Если мы не желаем видеть эту информацию, то результат выполнения команды `mkdir` нужно передать по конвейеру командлету `Out-Null`:

```
PS C:\> mkdir klop | Out-Null
PS C:\>
```

Как видим, в этом случае никаких сообщений на экран не выводится.

Табличный вывод данных в графическое окно

С помощью командлета `Out-GridView`, как и при использовании `Format-Table`, можно выводить данные в табличном виде, но не в то же самое консольное окно, а в отдельное диалоговое окно с графическим интерфейсом.

Например, выполним следующую команду:

```
PS C:\> Get-Process | Out-GridView
```

При этом информация о процессах будет доступна в отдельном окне (рис. 6.1).

В заголовке этого окна отображается выполненная команда PowerShell, при необходимости заголовок можно изменить, указав параметр `-Title`:

```
PS C:\> Get-Process | Out-GridView -Title "Процессы"
```

Выведенную таблицу можно сортировать по значениям любого столбца, щелкая мышью по его названию. Также можно отфильтровать строки по различным условиям с помощью конструктора фильтров (рис. 6.2).

Табличное представление данных, формируемое с помощью `Out-GridView`, является интерактивным, его можно изменять непосредственно в самой таблице. Более того, результирующий набор строк, полученный с помощью сортировки, фильтрации и выделения, можно по кнопке **OK** снова вернуть в PowerShell и отправить его дальше по конвейеру. Для этого нужно при вызове `Out-GridView` указать параметр `-PassThru`. Например:

```
PS C:\> Get-Process | Out-GridView -Title "Процессы" -PassThru |
Format-Table Id, ProcessName
```

Get-Process | Out-GridView

Фильтр

Добавить условие

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
1 134	85	137...	112...	12,67	1 060	8	Skype
781	30	13808	41568	1,77	1...	8	RuntimeBroker
416	20	13868	36432	1,17	1...	8	Skype
271	15	14088	35956	1,63	1 724	8	slack
211	13	14112	13360	0,31	6 284	8	igfxTray
1 631	26	14260	18532		508	0	svchost
145	8	1432	7388		1 988	0	SearchFilterHost
526	24	14556	39496	1,86	1...	8	TextInputHost
1 563	23	14868	23992		744	0	svchost
467	25	15348	39932	2,91	1...	8	Skype
133	8	1548	5148		2 912	0	svchost
224	13	15528	20360	0,83	6 052	8	chrome
121	8	1572	4888		2 640	0	svchost
130	9	1576	4092		3 836	0	svchost
161	8	1592	4672		2 772	0	svchost
443	14	15936	13176		1 520	0	svchost
123	8	1612	3944		3 508	0	svchost
137	9	1620	4468		1 424	0	svchost
237	15	16408	43616	0,16	1...	8	chrome
242	14	16412	40224	0,83	1...	8	chrome
147	10	1672	7984		6 180	0	svchost
122	8	1700	7564		1...	0	svchost
137	8	1712	7180	0,03	1...	8	svchost
163	11	1728	4824		3 500	0	svchost

Рис. 6.1. Вывод данных в окно с графическим интерфейсом

Процессы

win

Добавить условие

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
670	41	28280	73944	3,17	9 996	8	WindowsTerminal
180	11	1768	4720		836	0	wininit
278	12	2692	10256		8 504	8	winlogon
997	73	52648	1456	1,45	3 364	8	WinStore.App
644	65	29384	72616	4,97	1...	8	WINWORD

Рис. 6.2. Список процессов, содержащих в названии слово win

```
   Id ProcessName
   -- -----------
  836 wininit
12412 WINWORD
```

В этом примере в командлет `Format-Table` были переданы строки (объекты) из следующего окна (рис. 6.3).

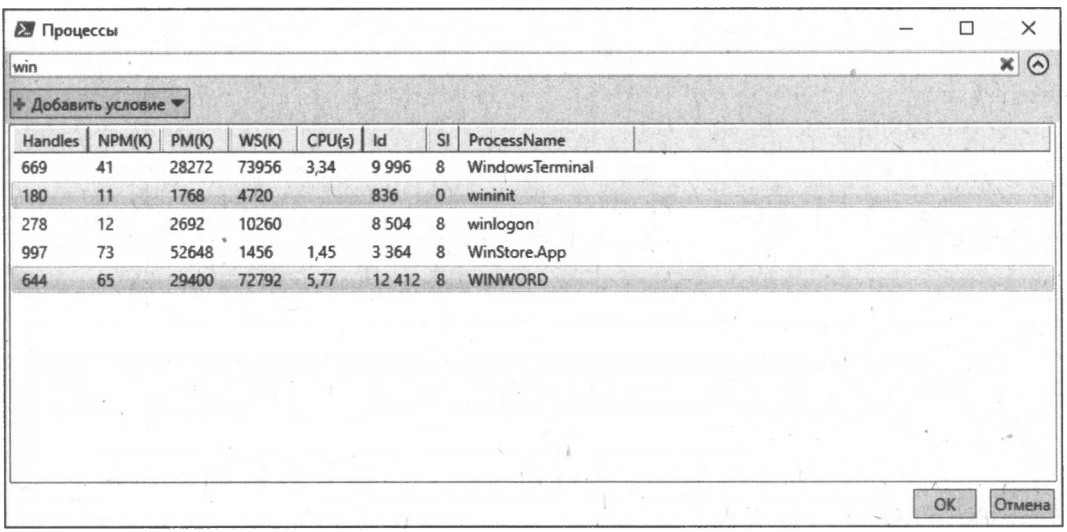


Рис. 6.3. Строки, подготовленные для возврата в конвейер PowerShell

Строки, которые нужно вернуть в конвейер PowerShell из диалогового окна, должны быть выделены в табличном представлении (по умолчанию возвращается только первая строка). Здесь поддерживаются стандартные комбинации:

- <Ctrl>+<A> для выделения всех строк;
- зажатый <Shift> и стрелки вверх или вниз для выделения непрерывного диапазона строк;
- зажатый <Ctrl> и щелчок мышью для добавления отдельных строк.

Вывод в формате HTML

В PowerShell есть стандартный командлет `ConvertTo-Html`, с помощью которого можно формировать HTML-страницы для отображения результатов выполнения команд.

Например, получим с помощью `Get-PSDrive` список дисков PowerShell и преобразуем этот список в HTML:

```
PS C:\Users\andrv> Get-PSDrive | ConvertTo-Html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<head>
<title>HTML TABLE</title>
</head><body>
<table>
<colgroup><col/><col/><col/><col/><col/><col/><col/><col/><col/><col/></colgroup>
<tr><th>Used</th><th>Free</th><th>CurrentLocation</th><th>Name</th><th>Provider
</th><th>Root</th><th>Description</th><th>MaximumSize</th><th>Credential</th>
<th>DisplayRoot</th></tr>
<tr><td></td><td></td><td></td><td></td><td>Alias</td><td>Microsoft.PowerShell.Core\
Alias</td><td></td><td>Диск, содержащий представление псевдонимов, сохраненное
в состоянии сеанса.</td><td></td><td>System.Management.Automation.
PSCredential</td><td></td></tr>
. . .
<tr><td></td><td></td><td></td><td></td><td>WSMan</td><td>Microsoft.WSMan.Management\
WSMan</td><td></td><td>Корень хранилища конфигурации WsMan.</td><td></td>
<td>System.Management.Automation.PSCredential</td><td></td></tr>
</table>
</body></html>
```

Как видим, на выходе получается корректный HTML-документ с таблицей, оформленной с помощью тега `<table>`. Этот вывод можно направить в HTML-файл и открыть этот файл в браузере (рис. 6.4) с помощью командлета `Invoke-Item`:

```
PS C:\Users\andrv> Get-PSDrive | ConvertTo-Html | Out-File psdrives.html
PS C:\Users\andrv> Invoke-Item .\psdrives.html
```

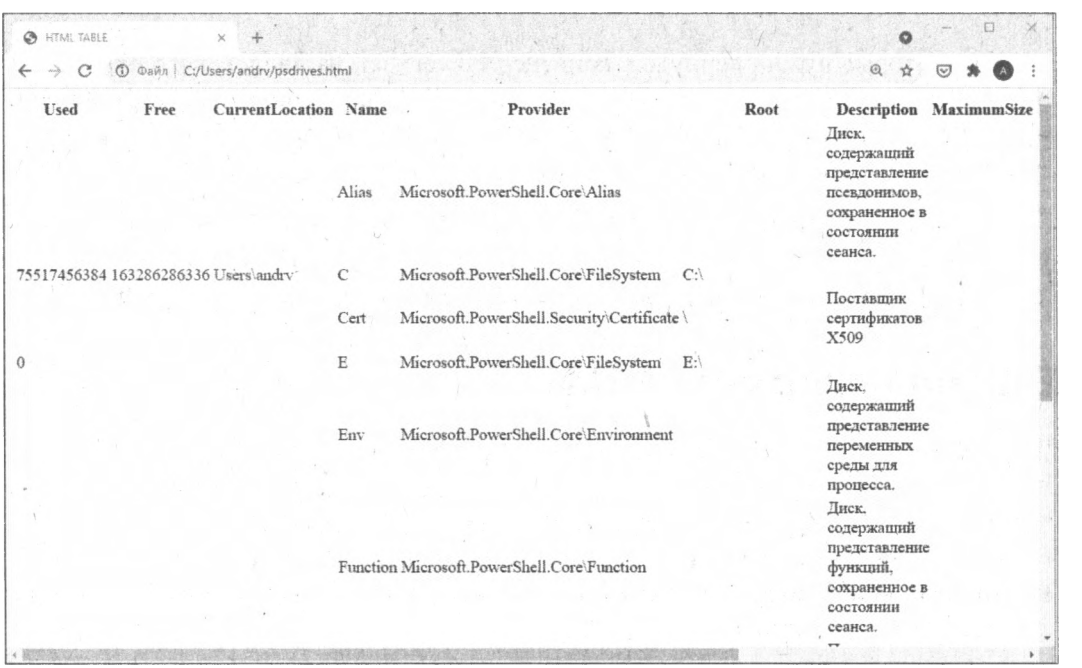


Рис. 6.4. HTML-страница с таблицей, сгенерированная с помощью командлета `ConvertTo-Html`

Параметр `-As List` позволяет изменить внешний вид таблицы — свойства каждого объекта будут расположены в виде списка (рис. 6.5), как при форматировании вывода с помощью `Format-List`:

```
PS C:\Users\andrv> Get-PSDrive | ConvertTo-Html -As List | Out-File
psdrives.html
PS C:\Users\andrv> Invoke-Item .\psdrives.html
```

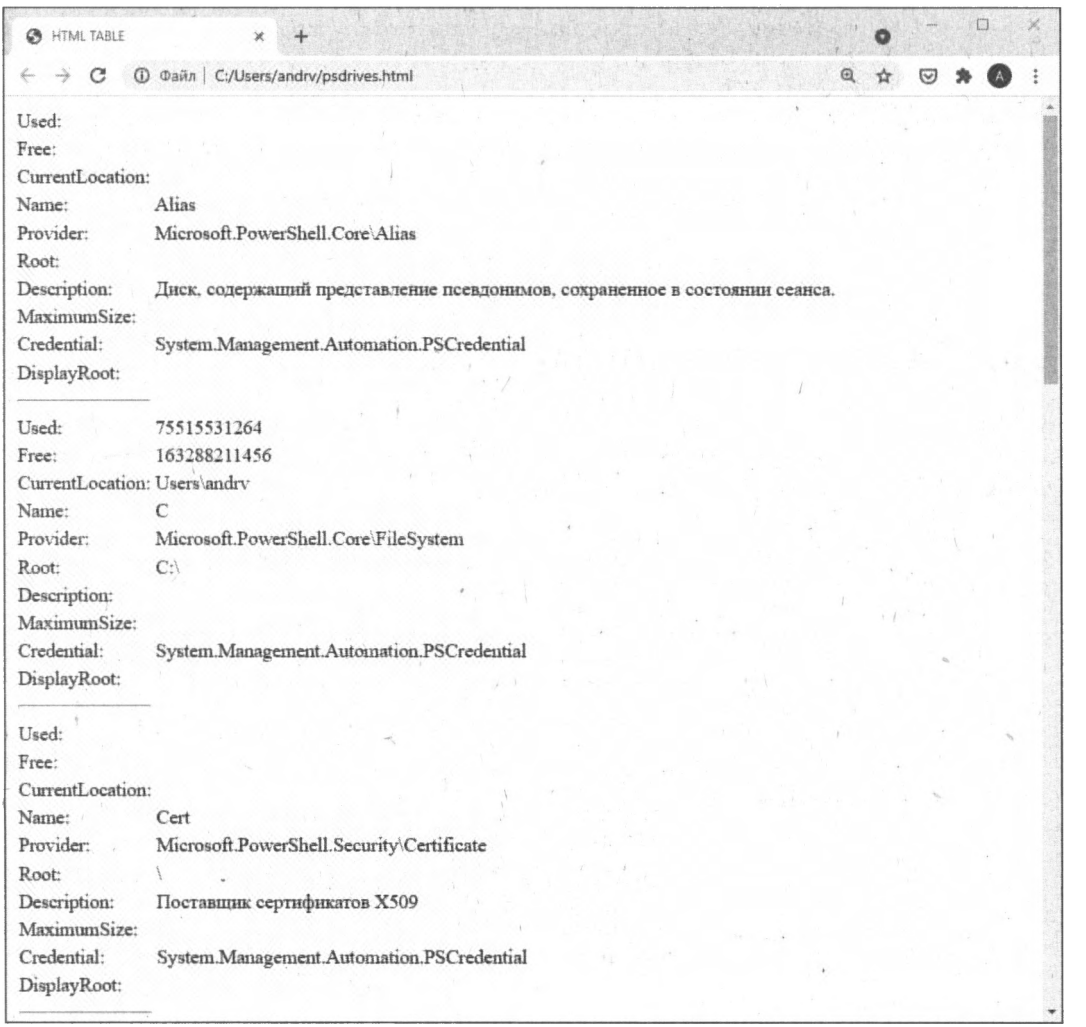


Рис. 6.5. HTML-страница с объектами в виде списка

С помощью дополнительных параметров можно задать заголовок HTML-страницы, текст до и после таблицы:

```
PS C:\Users\andrv> Get-PSDrive | ConvertTo-Html -Title "Отчет о дисках
PowerShell" -PreContent "<p>Текст до таблицы</p>" -PostContent "Текст после
таблицы" | Out-File .\psdrives.html
```

При необходимости к формируемому HTML-документу можно подключить внешний CSS-файл, путь к которому указывается в качестве значения параметра `-CssUri`.

Если в `ConvertTo-HTML` указан параметр `-Fragment`, то в HTML-разметке будет сгенерирована только таблица с данными (тег `<table>`):

```
PS C:\Users\andrv> Get-PSDrive | ConvertTo-Html -Fragment
<table>
<colgroup><col/><col/><col/><col/><col/><col/><col/><col/><col/><col/></colgroup>
<tr><th>Used</th><th>Free</th><th>CurrentLocation</th><th>Name</th><th>Provider
</th><th>Root</th><th>Description</th><th>MaximumSize</th><th>Credential</th>
<th>DisplayRoot</th></tr>
<tr><td></td><td></td><td></td><td>Alias</td><td>Microsoft.PowerShell.Core\
Alias</td><td></td><td>Диск, содержащий представление псевдонимов, сохраненное
в состоянии сеанса.</td><td></td><td>System.Management.Automation.PSCredential
</td><td></td></tr>
. . .
</table>
```

Дополнительные потоки в PowerShell

Кроме стандартного выходного потока `Output`, по которому объекты по конвейеру переходят от командлета к командлету, PowerShell поддерживает еще несколько потоков (табл. 6.3).

Таблица 6.3. *Потоки в PowerShell*

Наименование	Номер	Описание
`Output/Success`	1	Стандартный выходной поток, используется для передачи объектов по конвейеру и присвоения объектов переменным
`Error`	2	Поток для вывода объектов, соответствующих ошибкам
`Warning`	3	Поток для вывода предупреждающих сообщений
`Verbose`	4	Поток для вывода подробной информации о выполняемой операции
`Debug`	5	Поток для диагностических сообщений для отладки разрабатываемого командлета
`Information`	6	Поток для вывода информационных сообщений

Наличие нескольких непересекающихся потоков позволяет не загромождать основной поток ошибками или другими дополнительными сообщениями — командлеты получают по конвейеру только структурированные объекты, которые им нужны в качестве входных данных.

Записывать данные в различные потоки можно с помощью соответствующих командлетов с глаголом `Write`: `Write-Output`, `Write-Error`, `Write-Warning`, `Write-Verbose`, `Write-Debug` и `Write-Information`.

Например, запишем в поток ошибок новое сообщение:

```
PS C:\Users\andrv> Write-Error -Message "Произошла ошибка"
Write-Error -Message "Произошла ошибка" : Произошла ошибка
  + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
  + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException
```

При этом автоматически сформировался объект с информацией об ошибке, который был помещен в поток `Error` (более подробно обнаружение и обработку ошибок в PowerShell мы будет рассматривать в *главе 10*). Информация об ошибке из потока `Error` также выводится на экран и выделяется при этом красным цветом.

Информация из потока `Verbose` выводится в консоль и выделяется желтым цветом, только если при вызове команды указан параметр `-Verbose`:

```
PS C:\Users\andrv> Write-Verbose -Message "Сообщение из потока Verbose"
PS C:\Users\andrv> Write-Verbose -Message "Сообщение из потока Verbose"
-Verbose
ПОДРОБНО: Сообщение из потока Verbose
```

Поведение (в частности, режим вывода на экран сообщений) команд `Write-*` зависит от значений нескольких системных переменных:

```
PS C:\Users\andrv> $ErrorActionPreference
Continue
PS C:\Users\andrv> $DebugPreference
SilentlyContinue
PS C:\Users\andrv> $VerbosePreference
SilentlyContinue
PS C:\Users\andrv> $InformationPreference
SilentlyContinue
```

Если значение переменной равно `SilentlyContinue`, то информация из соответствующего потока на экран не дублируется.

Для дополнительных потоков PowerShell тоже поддерживается перенаправление в файл или в стандартный выходной поток `Output`.

Перенаправление в файл

Перенаправить данные из потока в файл можно с помощью тех же операторов `>` и `>>`, указав перед ними числовой номер этого потока. Например:

```
PS C:\Users\andrv> Write-Error -Message "Произошла ошибка" 2> error.txt
PS C:\Users\andrv> Get-Content .\error.txt
Write-Error -Message "Произошла ошибка" 2> error.txt : Произошла ошибка
  + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
  + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException

PS C:\Users\andrv> Write-Warning -Message "Сообщение из потока Warning" 3>
1.txt
PS C:\Users\andrv> Get-Content .\1.txt
Сообщение из потока Warning
```

Также с помощью оператора `*>` можно перенаправить в один файл все данные из всех потоков. Например, выполним блок кода, в котором идет запись в потоки `Output`, `Warning` и `Error`, и перенаправим данные из всех этих потоков в файл streams.txt:

```
PS C:\Users\andrv> & { Write-Output "Данные из Output"
>> Write-Warning "Данные из Warning"
>> Write-Error "Данные из Error"
>> } *> streams.txt
PS C:\Users\andrv> Get-Content .\streams.txt
Данные из Output
Данные из Warning
 Write-Output "Данные из Output"
Write-Warning "Данные из Warning"
Write-Error "Данные из Error"
 : Данные из Error
строка:1 знак:1
+ & { Write-Output "Данные из Output"
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
  + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException
```

Перенаправление в выходной поток Output

Перенаправить данные из потока с номером n в стандартный выходной поток `Output`, имеющий номер 1, можно с помощью оператора `n>&1`. Например, попробуем сохранить в переменной `$message` данные из потока `Verbose`:

```
PS C:\Users\andrv> $message = Write-Verbose -Message "Подробное сообщение"
-Verbose
ПОДРОБНО: Подробное сообщение
```

Проверим значение переменной `$message`:

```
PS C:\Users\andrv> $message
```

Как видим, в `$message` ничего не записалось, т. к. оператор присваивания получает данные из выходного потока `Output`, а не из потока `Verbose`.

Перенаправим теперь поток `Verbose` с номером 4 в выходной поток `Output` (номер 1):

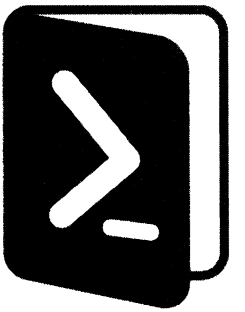
```
PS C:\Users\andrv> $message = Write-Verbose -Message "Подробное сообщение"
-Verbose 4>&1
```

После этого переменная `$message` будет иметь тип `System.Management.Automation.VerboseRecord`, в ней будет содержаться наше сообщение:

```
PS C:\Users\andrv> $message
ПОДРОБНО: Подробное сообщение
PS C:\Users\andrv> $message.getType().fullName
System.Management.Automation.VerboseRecord
```

Итоги

- В PowerShell используется внутренняя система форматирования, определяющая внешний вид и содержимое информации об объектах, которая выводится на экран. Правила форматирования можно задавать явно с помощью командлетов `Format-*`.
- По умолчанию данные выводятся на экран, с помощью командлетов `Out-*` вывод можно перенаправлять в файл, на принтер или в пустое устройство. Командлет `Out-GridView` позволяет выводить таблицу с данными в отдельное диалоговое окно с графическим интерфейсом.
- Результат выполнения команд можно оформить в виде HTML-страницы с помощью `ConvertTo-Html`.
- PowerShell поддерживает несколько независимых потоков для вывода в них дополнительной информации, которая не нужна в основном потоке, обеспечивающем прохождение объектов по конвейеру. Данные из этих дополнительных потоков могут быть перенаправлены в файл или в стандартный выходной поток `Output`.



ЧАСТЬ II

PowerShell как язык программирования

Глава 7. Переменные, массивы и хэш-таблицы

Глава 8. Операторы и управляющие инструкции

Глава 9. Функции, фильтры, сценарии и модули

Глава 10. Обработка ошибок при выполнении команд

ГЛАВА 7

Переменные, массивы и хэш-таблицы

В рассмотренных ранее примерах мы уже использовали различные числовые и символьные литералы (константы), а также переменные PowerShell, сохраняя в них результаты выполнения команд. Кроме переменных в PowerShell, как и во многих других языках программирования, поддерживаются массивы, а также более специфические структуры — ассоциативные массивы (хэш-таблицы).

Рассмотрим данные элементы языка PowerShell более подробно.

Числовые и символьные литералы

Практически в каждом языке программирования имеется возможность работы с числами и символьными строками, причем способы их задания могут быть различными (например, в одних языках строки нужно заключать в двойные кавычки, а в других — в одинарные). PowerShell также поддерживает целые и вещественные числа, а также символьные строки нескольких видов.

Числовые литералы

В PowerShell переменные являются объектами .NET Framework. Поддерживаются все основные числовые типы этой платформы: `System.Int32`, `System.Int64`, `System.Double`. При этом явно задавать тип чисел нет необходимости — система сама выбирает подходящий тип для указываемого числа. Проверим тип нескольких чисел, воспользовавшись для этого методом `GetType`:

```
PS C:\> (10).gettype().fullname
System.Int32
PS C:\> (10.23).gettype().fullname
System.Double
PS C:\> (10+10.23).gettype().fullname
System.Double
```

В PowerShell предусмотрены специальные *суффиксы-множители* для упрощения работы с величинами, часто используемыми системными администраторами: килобайтами, мегабайтами и гигабайтами (табл. 7.1).

Таблица 7.1. Суффиксы-множители в PowerShell

Суффикс-множитель	Числовой множитель	Пример	Числовое значение для примера
KB	1024	2KB	2048
kb	1024	1.1kb	1126,4
MB	1024*1024	3MB	3 145 728
mb	1024*1024	2.5mb	2 621 440
GB	1024*1024*1024	1GB	1 073 741 824
gb	1024*1024*1024	2.23gb	2 394 444 267,52

Приведем примеры:

```
PS C:\> 1mb+10kb
1058816
PS C:\> 2GB+56MB
2206203904
```

В PowerShell можно оперировать числами в шестнадцатеричном формате, используя для этого те же обозначения, что и в С-подобных языках программирования: перед числом указывается префикс `0x`, а в записи числа могут присутствовать цифры и буквы `A`, `B`, `C`, `D`, `E` и `F` (независимо от регистра). Например:

```
PS C:\> 0x10
16
PS C:\> 0xA
10
PS C:\> 0xcd
205
```

Символьные строки

Все символьные строки в PowerShell являются объектами типа `System.String` и представляют собой последовательность 32-битовых символов в кодировке Unicode. Длина строк не ограничена, содержимое строк нельзя изменять (можно только копировать).

В PowerShell поддерживается четыре вида символьных строк.

Строки в одинарных и двойных кавычках

Строки могут задаваться последовательностью символов, заключенных в одинарные или двойные кавычки:

```
PS C:\> 'Строка в одинарных кавычках'
Строка в одинарных кавычках
PS C:\> "Строка в двойных кавычках"
Строка в двойных кавычках
```

Строки могут содержать любые символы (в том числе символы разрыва строки и возврата каретки), кроме соответствующего одиночного закрывающего символа (одинарной или двойной кавычки). Строка в одинарных кавычках может содержать двойные кавычки и наоборот:

```
PS C:\> 'Строка в "одинарных" кавычках'
Строка в "одинарных" кавычках
PS C:\> "Строка в 'двойных' кавычках"
Строка в 'двойных' кавычках
```

Если внутри строки нужно поместить символ, ограничивающий данную строку (т. е. одинарную или двойную кавычку), то нужно написать этот символ два раза подряд:

```
PS C:\> 'Строка в ''одинарных кавычках'
Строка в 'одинарных кавычках
PS C:\> "Строка в ""двойных кавычках"
Строка в "двойных кавычках
```

Строки в двойных кавычках являются *расширяемыми*. Это означает, что если внутри такой строки встречается имя переменной или другое выражение, которое может быть вычислено, то в строку подставляется значение данной переменной или результат вычисления выражения. Например:

```
PS C:\> $a=123
PS C:\> "$a равно $a"
123 равен 123
```

Если имя переменной встречается внутри строки в одинарных кавычках, то никакой подстановки значения переменной не происходит:

```
PS C:\> '$a равно $a'
$a равен $a
```

При необходимости можно отключить расширение определенной переменной внутри строки в двойных кавычках. Для этого перед знаком $ этой переменной нужно указать символ обратного апострофа `, например:

```
PS C:\> "`$a равно $a"
$a равно 123
```

Символы, имеющие специальное значение, вставляются в строки в двойных кавычках с помощью escape-последовательностей, которые в PowerShell начинаются с символа обратного апострофа ` (табл. 7.2).

Таблица 7.2. *Escape-последовательности PowerShell*

Переменная	Описание
`n	Разрыв строки
`r	Возврат каретки
`t	Горизонтальная табуляция

Таблица 7.2 *(окончание)*

Переменная	Описание
`a	Звуковой сигнал
`b	Забой (backspace)
`'	Одинарная кавычка
`"	Двойная кавычка
`0	Пустой символ (null)
``	Обратный апостроф

Замечание

Традиционно во многих языках программирования для выделения специальных символов (escape-последовательностей) используется обратная косая черта (например, `\n` или `\t`) Разработчики оболочки PowerShell приняли решение ввести другой символ для escape-последовательностей, чтобы избежать проблем при использовании символа \ в качестве разделителя компонентов пути в файловой системе Windows и других пространствах имен PowerShell.

Вставим символ разрыва строки в строку в двойных кавычках:

```
PS C:\> "Строка в `nдвойных кавычках"
Строка в
двойных кавычках
```

Как видите, на экран информация выводится в двух строках. Если же вставить escape-последовательность `` `n `` в строку в одинарных кавычках, то разрыва строки не произойдет:

```
PS C:\> 'Строка в `nодинарных кавычках'
Строка в `nодинарных кавычках
```

Кроме переменных в расширяемых строках могут указываться так называемые *подвыражения* (subexpression) — ограниченные символами `$(...)` фрагменты кода на языке PowerShell, которые в строках заменяются на результаты вычисления этих фрагментов. Например:

```
PS C:\> "3+2 равно $(3+2)"
3+2 равно 5
```

Строки типа here-string

В PowerShell наряду с обычными строками в одинарных и двойных кавычках поддерживаются так называемые *строки типа here-string*. Подобные строки обычно используются для вставки в сценарий больших блоков текста или при генерации текстовой информации для других программ и имеют следующий формат:

```
@<кавычка><разрыв_строки>блок текста<разрыв_строки><кавычка>@
```

Кавычки могут быть как одинарными, так и двойными, при этом смысл их остается тем же, что и для обычных строк: переменные и подвыражения, стоящие внутри

двойных кавычек, заменяются на их значения, а стоящие внутри одинарных кавычек остаются неизменными. Например:

```
PS C:\> $a=@"
>> 1 Первая строка
>> $(1+1) Вторая строка
>> "Третья строка"
>> "@
>>
PS C:\> $a
1 Первая строка
2 Вторая строка
"Третья строка"
PS C:\> $a=@'
>> 1 Первая строка
>> $(1+1) Вторая строка
>> 'Третья строка'
>> '@
>>
PS C:\> $a
1 Первая строка
$(1+1) Вторая строка
'Третья строка'
```

Обратите внимание, что ограничитель строк типа here-string обязательно должен содержать символ разрыва строки, поэтому внутри таких строк различные специальные символы (например, одинарные или двойные кавычки) могут применяться без ограничений.

Переменные PowerShell

Как мы уже знаем, имена переменных PowerShell всегда начинаются со знака доллара ($). Переменные PowerShell не нужно предварительно объявлять или описывать, они создаются при первом присваивании переменной значения.

Если попытаться обратиться к несуществующей переменной, то система вернет значение `$null`.

> ***Замечание***
>
> `$null`, как `$true` и `$false`, является специальной переменной, определенной в системе. Изменить значения этих переменных нельзя

Проверить наличие определенной переменной можно с помощью командлета `Test-Path` с указанием виртуального диска PowerShell `Variable:`. Например, следующая команда проверяет, существует ли переменная с именем `MyVariable`:

```
PS C:\> Test-Path Variable:MyVariable
False
```

Список всех переменных, определенных в текущем сеансе работы, можно увидеть, обратившись к виртуальному диску `Variable:` с помощью команды `dir`:

```
PS C:\> dir Variable:
Name                          Value
----                          -----
$
?                             True
^
args                          {}
ConfirmPreference             High
ConsoleFileName               .
DebugPreference               SilentlyContinue
Error                         {}
ErrorActionPreference         Continue
ErrorView                     NormalView
. . .
```

Если пользователь не создавал пока своих переменных, то в системе определены только *переменные оболочки PowerShell*.

Переменные оболочки PowerShell

Переменные оболочки — это набор переменных, которые создаются, объявляются оболочкой PowerShell и присутствуют по умолчанию в каждом сеансе работы. Переменные оболочки сохраняются в течение всего сеанса и доступны всем командам, сценариям и приложениям, которые выполняются в данном сеансе.

Поддерживаются два вида переменных оболочки:

- *Автоматические переменные*. В этих переменных хранятся параметры состояния оболочки PowerShell. Автоматические переменные сохраняются и динамически изменяются самой системой. Пользователи не могут (и не должны) изменять значения этих переменных. Например, значением переменной `$PID` является идентификатор текущего процесса PowerShell.exe.
- *Переменные настроек*. В этих переменных хранятся настройки активного пользователя. Эти переменные создаются оболочкой PowerShell и заполняются значениями по умолчанию. Пользователи могут изменять значения этих переменных. Например, переменная `$MaximumHistoryCount` определяет максимальное число записей в журнале сеанса.

В табл. 7.3 приведено краткое описание некоторых переменных оболочки.

Таблица 7.3. *Переменные оболочки PowerShell*

Переменная	Описание
`$$`	Содержит последнюю лексему последней строки, полученной оболочкой
`$?`	Имеет значение True, если последняя операция завершилась успешно, иначе имеет значение False

Таблица 7.3 (окончание)

Переменная	Описание
$^	Содержит первую лексему последней строки, полученной оболочкой
$_	Содержит текущий объект конвейера, использованный в блоках сценариев, фильтрах и инструкции Where
$args	Содержит массив параметров, передаваемых в функцию
$DebugPreference	Указывает действие, которое необходимо выполнить при записи данных с помощью командлета Write-Debug
$Error	Содержит объекты, для которых возникла ошибка при обработке в командлете
$ErrorActionPreference	Указывает действие, которое необходимо выполнить при записи данных с помощью командлета Write-Error
$Home	Содержит путь к домашнему каталогу пользователя
$Input	Используется в блоках сценариев, находящихся в конвейере
$MaximumAliasCount	Содержит максимальное число псевдонимов, доступных сеансу
$MaximumDriveCount	Содержит максимальное число доступных дисков, за исключением предоставляемых операционной системой
$MaximumFunctionCount	Содержит максимальное число функций, доступных сеансу
$MaximumHistoryCount	Указывает максимальное число записей, сохраненных в истории команд
$MaximumVariableCount	Содержит максимальное число переменных, доступных сеансу
$PSHome	Каталог, в который установлен PowerShell
$Host	Содержит сведения о текущем узле
$StackTrace	Содержит подробные сведения трассировки стека последней ошибки
$VerbosePreference	Указывает действие, которое нужно выполнить, если данные записываются с помощью командлета Write-Verbose
$WarningPreference	Указывает действие, которое необходимо выполнить при записи данных с помощью командлета Write-Warning в сценарии

Переменными оболочки можно пользоваться так же, как и другими видами переменных. Например, следующая команда выведет на экран содержимое домашнего каталога PowerShell, путь к которому хранится в переменной оболочки $PSHome:

```
PS C:\Users\andrv> dir $PSHome

    Каталог: C:\Windows\System32\WindowsPowerShell\v1.0

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        28.09.2020     15:55                en-US
d-----        07.12.2019     12:14                Examples
```

```
d-----        28.09.2020     16:13                Modules
d-----        07.12.2019     17:36                ru
d-----        07.12.2019     17:34                ru-RU
d-----        07.12.2019     12:14                Schemas
d-----        07.12.2019     12:14                SessionConfig
. . .
```

Пользовательские переменные

Пользовательская переменная создается после первого присваивания ей значения. Например, создадим целочисленную переменную `$a`:

```
PS C:\> $a = 1
PS C:\> $a
1
PS C:\> Test-Path Variable:a
True
PS C:\> dir Variable:a

Name                           Value
----                           -----
a                              1
```

Типы переменных

Проверим, какой тип имеет переменная `$a`. Для этого можно воспользоваться командлетом `Get-Member` или методом `getType()`:

```
PS C:\> $a | Get-Member

   TypeName: System.Int32
. . .
PS C:\> Get-Member -InputObject $a

   TypeName: System.Int32
. . .
PS C:\> $a.getType().fullName
System.Int32
```

Итак, переменная `$a` сейчас имеет тип `System.Int32`. Присвоим этой переменной другое значение (строку) и вновь проверим тип:

```
PS C:\> $a = "aaa"
PS C:\> $a | Get-Member

   TypeName: System.String
. . .
```

Как видим, тип переменной `$a` изменился на `System.String`, т. е. тип переменной определяется типом последнего присвоенного ей значения.

Можно также явно указать тип переменной при ее определении, указав в квадратных скобках соответствующий атрибут типа. При этом выражение, стоящее в правой части после знака равенства, будет преобразовано (если это возможно) к данному типу. Например, объявим целочисленную переменную `$a` и присвоим этой переменной символьное значение, которое можно преобразовать к целому типу:

```
PS C:\> [System.Int32]$a = 10
PS C:\> $a = "123"
PS C:\> $a
123
PS C:\> $a.getType().fullName
System.Int32
```

Как видим, строка "123" была преобразована в целое число 123. Если же попытаться записать в переменную `$a` значение, которое не может быть преобразовано в целое число, то возникнет ошибка:

```
PS C:\> $a = "aaa"
Не удается преобразовать значение "aaa" в тип "System.Int32". Ошибка: "Входная
строка имела неверный формат."
строка:1 знак:1
+ $a = "aaa"
+ ~~~~~~~~~~
  + CategoryInfo          : MetadataError: (:) [],
                                    ArgumentTransformationMetadataException
  + FullyQualifiedErrorId : RuntimeException
```

Вместо явного указания .NET-типа переменной можно пользоваться более краткими *псевдонимами типов*. Например:

```
PS C:\> [int]$a = 10
PS C:\> $a.GetType().FullName
System.Int32
```

Наиболее часто используемые псевдонимы типов приведены в табл. 7.4.

Таблица 7.4. *Псевдонимы типов PowerShell*

Псевдоним типа	Соответствующий .NET-тип
`[int]`	`System.Int32`
`[int[]]`	`System.Int32[]` (массив элементов типа `System.Int32`)
`[long]`	`System.Int64`
`[long[]]`	`System.Int64[]` (массив элементов типа `System.Int64`)
`[string]`	`System.String`
`[string[]]`	`System.String[]` (массив элементов типа `System.String`)
`[char]`	`System.Char`
`[char[]]`	`System.Char[]` (массив элементов типа `System.Char`)
`[bool]`	`System.Boolean`

Таблица 7.4 *(окончание)*

Псевдоним типа	Соответствующий .NET-тип
[bool[]]	System.Boolean[] (массив элементов типа System.Boolean)
[byte]	System.Byte
[byte[]]	System.Byte[] (массив элементов типа System.Byte)
[double]	System.Double
[double[]]	System.Double[] (массив элементов типа System.Double)
[decimal]	System.Decimal
[decimal[]]	System.Decimal[] (массив элементов типа System.Decimal)
[float]	System.Float
[single]	System.Single
[regex]	System.Text.RegularExpressions.regex
[array]	System.Array
[xml]	System.Xml.XmlDocument
[scriptblock]	System.Management.Automation.ScriptBlock
[switch]	System.Management.Automation.SwitchPazameter
[hashtable]	System.Collections.Hashtable
[psobject]	System.Management.Automation.PSObject
[type]	System.Type

Приведение типов

Для явного преобразования какого-либо значения к определенному типу нужно указать этот тип (в полной форме типа платформы .NET или в виде псевдонима) перед данным значением.

Например, сохраним в переменной $a сумму двух чисел:

```
PS C:\Users\andrv> $a = 100.1 + 9.9
```

Переменная $a будет числом:

```
PS C:\Users\andrv> $a.getType().fullName
System.Double
PS C:\Users\andrv> $a
110
```

Теперь преобразуем при сложении каждое число к символьному типу.

```
PS C:\Users\andrv> $a = [string]100.1 + [string]9.9
```

В этом случае переменная $a будет строкой:

```
PS C:\Users\andrv> $a.getType().fullName
System.String
PS C:\Users\andrv> $a
100.19.9
```

Дополнительные атрибуты переменных

Для переменных в PowerShell можно указывать не только их тип, но и некоторые дополнительные атрибуты, ограничивающие набор возможных значений для этих переменных.

Например, создадим целочисленную переменную `$a` с ограниченным диапазоном значений. Для этого используется атрибут `validateRange()`:

```
PS C:\Users\andrv> [validateRange(1,5)][int]$a = 4
```

Теперь система не даст записать в эту переменную число, меньшее единицы или большее пяти:

```
PS C:\Users\andrv> $a = 6
Не удалось выполнить проверку переменной, т. к. значение 6 является
недопустимым для переменной a.
строка:1 знак:1
+ $a = 6
+ ~~~~~~
  + CategoryInfo          : MetadataError: (:) [], ValidationMetadataException
  + FullyQualifiedErrorId : ValidateSetFailure
```

С помощью атрибута `validateLength()` можно ограничить длину символьной переменной. Например:

```
PS C:\Users\andrv> [validateLength(0,4)][string]$s = 'abcd'
```

В переменную `$s` нельзя будет сохранить строку длиной более четырех символов:

```
PS C:\Users\andrv> $s = 'abcde'
Не удалось выполнить проверку переменной, т. к. значение abcde является
недопустимым для переменной s.
строка:1 знак:1
+ $s = 'abcde'
+ ~~~~~~~~~~~~
  + CategoryInfo          : MetadataError: (:) [], ValidationMetadataException
  + FullyQualifiedErrorId : ValidateSetFailure
```

Константы

В PowerShell можно создавать константы — переменные, значения которых нельзя изменять. Для этого используется командлет `New-Variable` или `Set-Variable` с параметром `-Option Constant`. Например:

```
PS C:\Users\andrv> New-Variable -Name pi -Value 3.14 -Option Constant
```

Данная команда создала переменную `$pi` (обратите внимание, что в атрибуте `-Name` знак доллара не указывается) со значением 3,14:

```
PS C:\Users\andrv> $pi
3.14
```

Попытка изменить значение такой переменной приведет к возникновению ошибки:

```
PS C:\Users\andrv> $pi = 1
Не удается перезаписать переменную pi, т. к. она является постоянной либо
доступна только для чтения.
строка:1 знак:1
+ $pi = 1
+ ~~~~~~~
  + CategoryInfo          : WriteError: (pi:String) [],
                                        SessionStateUnauthorizedAccessException
  + FullyQualifiedErrorId : VariableNotWritable
```

Переменные среды Windows

Кроме собственных переменных оболочка PowerShell позволяет работать и с *переменными среды* (или *переменными окружения*) Windows, каждая из которых хранится в оперативной памяти в течение всего сеанса работы операционной системы, имеет свое уникальное имя, а ее значением является строка. Стандартные переменные среды автоматически инициализируются в процессе загрузки операционной системы. К таким переменным относятся, например:

- `WINDIR` — путь к установочному каталогу Windows;
- `TEMP` — путь к каталогу для хранения временных файлов Windows;
- `PATH` — *системный путь* (путь поиска), т. е. список каталогов, в которых система должна искать выполняемые файлы или файлы совместного доступа (например, динамические библиотеки).

В оболочке PowerShell доступ к переменным среды можно получить через виртуальный диск `Env:`. Например, список всех переменных среды выводится командлетом `dir`:

```
PS C:\Users\andrv> dir Env:

Name                           Value
----                           -----
ALLUSERSPROFILE                C:\ProgramData
APPDATA                        C:\Users\andrv\AppData\Roaming
ChocolateyInstall              C:\ProgramData\chocolatey
ChocolateyLastPathUpdate       132578680900827918
CommonProgramFiles             C:\Program Files\Common Files
CommonProgramFiles(x86)        C:\Program Files (x86)\Common Files
CommonProgramW6432             C:\Program Files\Common Files
COMPUTERNAME                   DESKTOP-BU86I5T
. . .
```

Для получения значения определенной переменной среды нужно перед ее именем указать префикс `env:` (в оболочке cmd.exe для этой цели переменную нужно было заключать в знаки процента %). Например, следующая команда выведет на экран

путь к корневому каталогу операционной системы, который хранится в переменной среды `SystemRoot`:

```
PS C:\> $Env:SystemRoot
C:\WINDOWS.1
```

В оболочке PowerShell можно изменять значения переменных среды с помощью следующего синтаксического выражения:

```
$Env:имя_переменной = "новое_значение"
```

При этом следует иметь в виду, что изменения влияют только на текущий сеанс работы (аналогичным образом обстоит дело с командой `set` оболочки cmd.exe и с командой `setenv` в оболочках UNIX-систем). Чтобы изменения стали постоянными, необходимо их значения изменять в системном реестре.

Массивы в PowerShell

В отличие от многих языков программирования, в PowerShell не нужно с помощью каких-либо специальных символов указывать начало массива или его конец, а также предварительно объявлять массив.

Для создания и инициализации массива можно просто присвоить значения его элементам. Значения, добавляемые в массив, разделяются запятой и отделяются от имени переменной (имени массива) оператором присваивания. Например, следующая команда создаст массив `$a` из трех элементов:

```
PS C:\> $a = 1, 2, 3
PS C:\> $a
1
2
3
```

В языке PowerShell круглые скобки, окружающие какое-либо выражение, означают, что это выражение должно быть вычислено. Поэтому список значений массива можно указать в круглых скобках — это сделает массивы более заметными в сценариях:

```
PS C:\Users\andrv> $names = ('Иван', 'Сергей', 'Андрей')
PS C:\Users\andrv> $names
Иван
Сергей
Андрей
```

Дополнительно перед скобками можно указать знак `@`:

```
PS C:\Users\andrv> $numbers = @(10, 20, 45)
PS C:\Users\andrv> $numbers
10
20
45
```

Выражение `@()` создаст пустой массив, не содержащий элементов.

```
PS C:\Users\andrv> $a = @()
PS C:\Users\andrv> $a.getType().fullName
System.Object[]
PS C:\Users\andrv> $a.length
0
```

При объявлении пустого массива символ `@` нужно указывать обязательно, без него возникнет ошибка:

```
PS C:\Users\andrv> $a = ()
строка:1 знак:7
+ $a = ()
+       ~
После ''('' ожидалось выражение.
  + CategoryInfo          : ParserError: (:) [],
                                            ParentContainsErrorRecordException
  + FullyQualifiedErrorId : ExpectedExpression
```

Можно также создать и инициализировать массив, используя оператор диапазона (..). Например, следующая команда создает массив `$b`, содержащий числа от 10 до 14:

```
PS C:\> $b = 10..14
```

В результате массив `$b` будет содержать пять значений:

```
PS C:\> $b
10
11
12
13
14
```

Обращение к элементам массива

Как мы уже видели, для отображения всех элементов массива нужно просто ввести его имя. Длина массива (количество элементов) хранится в свойстве `Length`:

```
PS C:\> $a.Length
3
```

Псевдонимом свойства `Length` является свойство `Count`:

```
PS C:\> $a.Count
3
```

Для обращения к определенному элементу массива нужно указать его порядковый номер (индекс) в квадратных скобках после имени переменной. При этом следует иметь в виду, что нумерация элементов в массиве PowerShell всегда начинается с нуля, поэтому для получения значения первого элемента нужно выполнить следующую команду:

```
PS C:\> $a[0]
1
```

В качестве индекса можно указывать и отрицательные значения, при этом отсчет будет вестись с конца массива. Например, индекс –1 будет соответствовать последнему элементу массива:

```
PS C:\> $a[-1]
3
```

Язык PowerShell позволяет извлекать из массива несколько значений сразу. Для этого в качестве индекса массива можно использовать оператор диапазона или другой массив с целочисленными элементами. Например, получить элементы массива `$a` с индексами от 1 до 2 можно разными способами:

```
PS C:\> $a[1..2]
2
3
PS C:\> $a[1, 2]
2
3
PS C:\Users\andrv> $n = 1,2
PS C:\Users\andrv> $a[$n]
2
3
```

В операторе диапазона можно использовать свойство `Length`. Например, для отображения элементов от индекса 1 до конца массива (последний элемент массива имеет индекс `Length`-1) можно выполнить следующую команду:

```
PS C:\> $a[1..($a.Length-1)]
2
3
```

Для изменения элемента массива нужно присвоить новое значение элементу с соответствующим индексом:

```
PS C:\> $a[0] = 5
PS C:\> $a[1] = 3.14
PS C:\> $a[2] = "привет"
PS C:\> $a
5
3.14
привет
```

Операции с массивом

Последний пример показывает, что по умолчанию массивы PowerShell могут содержать элементы разных типов, т. е. являются *полиморфными*. Посмотрим, какой тип имеет наш массив `$a`:

```
PS C:\> $a.getType().fullName
System.Object[]
```

Итак, переменная `$a` имеет тип "массив элементов типа `System.Object[]`". Можно создать массив с жестко заданным типом, т. е. такой массив, который содержит элементы только одного типа. Для этого, как и в случае с обычными скалярными переменными, необходимо указать нужный тип в квадратных скобках перед именем переменной (см. табл. 6.5). Например, следующая команда создаст массив 32-разрядных целых чисел:

```
PS C:\> [int[]]$a = 1,2,3,4
```

Если попытаться записать в данный массив значение, которое нельзя преобразовать к целому типу, то возникнет ошибка:

```
PS C:\> $a[0] = "aaa"
Не удается преобразовать значение "aaa" в тип "System.Int32". Ошибка: "Входная
строка имела неверный формат."
строка:1 знак:1
+ $a[0]="aaa"
+ ~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [], RuntimeException
    + FullyQualifiedErrorId : InvalidCastFromStringToInteger
```

Увеличение длины массива. Объединение массивов

При попытке обратиться к элементу, выходящему за границы массива, возникнет ошибка. Например:

```
PS C:\> $a.Length
4
PS C:\> $a[4] = 5
Индекс находился вне границ массива.
строка:1 знак:1
+ $a[4]=5
+ ~~~~~~~
  + CategoryInfo          : OperationStopped: (:) [], IndexOutOfRangeException
  + FullyQualifiedErrorId : System.IndexOutOfRangeException
```

Подобные ошибки связаны с тем, что массивы PowerShell базируются на .NET-массивах, имеющих фиксированную длину. Несмотря на это, имеется способ увеличения длины массива. Для этого можно воспользоваться оператором конкатенации `+` или `+=`. Например, следующая команда добавит к массиву `$a` два новых элемента со значениями 5 и 6:

```
PS C:\> $a
1
2
3
4
```

```
PS C:\> $a += 5,6
PS C:\> $a
1
2
3
4
5
6
PS C:\>
```

При выполнении оператора += происходит следующее:

1. PowerShell создает новый массив, размер которого достаточен для помещения в него всех элементов.
2. Первоначальное содержимое массива копируется в новый массив.
3. Новые элементы копируются в конец нового массива.

Таким образом, мы на самом деле не добавляем новый элемент к массиву, а создаем новый массив большей размерности.

Можно объединить два массива в один с помощью оператора конкатенации +. Например:

```
PS C:\> $x = 1,2
PS C:\> $y = 3,4
PS C:\> $z = $x + $y
PS C:\> $z
1
2
3
4
```

Удаление элементов

Удалить элемент из массива не так просто, однако можно создать новый массив и скопировать в него все элементы кроме ненужного. Например, следующая команда создаст массив `$b`, содержащий все элементы массива `$a` кроме значения с индексом 2:

```
PS C:\> $b = $a[0,1 + 3..($a.length-1)]
PS C:\> $b
1
2
4
5
6
```

Действие оператора присваивания

Следует иметь в виду, что обычный оператор присваивания (=) действует на массивы *по ссылке*. Например, создадим массив `$a` из двух элементов и присвоим этот массив переменной `$b`:

```
PS C:\> $a = 1, 2
PS C:\> $b = $a
PS C:\> $b
1
2
```

Теперь изменим значение первого элемента массива `$a` и посмотрим еще раз на содержимое массива `$b`:

```
PS C:\> $a[0] = "Новое значение"
PS C:\> $b
Новое значение
2
PS C:\>
```

Как видим, содержимое массива `$b` также изменилось, т. к. переменная `$b` указывает на тот же объект, что и переменная `$a`.

Сохранение в массиве вывода командлетов

Если командлет генерирует поток объектов, то их можно сохранить в массиве. Например:

```
PS C:\Users\andrv> $a = Get-ChildItem
```

Теперь в переменной `$a` содержится массив объектов, соответствующих файлам и подкаталогам в каталоге C:\Users\andrv. С этим массивом можно работать обычным способом:

```
PS C:\Users\andrv> $a.Count
30
PS C:\Users\andrv> $a[0..3]

    Каталог: C:\Users\andrv

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        15.02.2021     20:30                .config
d-----        10.10.2020     22:02                .dbus-keyrings
d-----        11.02.2021     16:12                .local
d-----        03.03.2021     20:10                .quokka
```

Удаление массива

Для удаления массива можно воспользоваться командлетом `Remove-Item` (псевдоним `del`) и удалить переменную, содержащую нужный массив, с виртуального диска `Variable:`. Например:

```
PS C:\> $a
1
2
```

```
3
4
5
6
PS C:\> del Variable:a
PS C:\> $a
PS C:\>
```

Хэш-таблицы (ассоциативные массивы)

Кроме обычных массивов в PowerShell поддерживаются так называемые *ассоциативные массивы* (иногда их также называют словарями) — структуры для хранения коллекции ключей и их значений, связанных попарно. Например, можно использовать фамилию человека как ключ, а его дату рождения как значение. Ассоциативный массив обеспечивает структуру для хранения коллекции имен и дат рождения, где каждому имени сопоставлена дата рождения. Визуально массив ассоциированных значений можно представить как таблицу, состоящую из двух столбцов, где первый столбец является ключом, а второй — значением.

Ассоциативные массивы похожи на обычные массивы в PowerShell, но вместо обращения к содержимому массива по индексу, можно обратиться к элементу данных ассоциативного массива по ключу. Используя этот ключ, PowerShell возвращает соответствующее значение из ассоциативного массива.

Для хранения содержимого ассоциативного массива в PowerShell используется специальный тип данных – *хэш-таблица*, поскольку данная структура данных обеспечивает быстрый механизм поиска. Это очень важно, поскольку основным назначением ассоциативного массива является обеспечение эффективного механизма поиска.

> ***Замечание***
>
> В дальнейшем мы будем пользоваться терминами хэш-таблица и ассоциативный массив как синонимами.

В отличие от обычного массива, для объявления и инициализации хэш-таблиц используются специальные литералы:

```
$имя_массива = @{ключ1 = элемент1; <ключ2 = элемент2;...}
```

Итак, каждому значению хэш-таблицы нужно присвоить метку (ключ), перед перечислением содержимого массива следует поставить символы `@{`, а завершается перечисление элементов символом `}`. Ключи и значения разделяются знаком равенства (`=`), пары "ключ — значение" разделяются между собой точкой с запятой (`;`).

Создадим, например, хэш-таблицу, в которой будут храниться данные об одном человеке (ассоциативный массив с тремя элементами):

```
PS C:\> $user = @{Фамилия="Попов"; Имя="Андрей"; Телефон="55-55-55"}
PS C:\> $user
```

```
Name                           Value
----                           -----
Фамилия                        Попов
Имя                            Андрей
Телефон                        55-55-55
```

После создания массива нужно научиться обращаться к его элементам. В PowerShell доступ к хэш-таблицам возможен двумя способами: с использованием нотации свойств или нотации массивов. Обращение с использованием нотации свойств выглядит следующим образом:

```
PS C:\> $user.Фамилия
Попов
PS C:\> $user.Имя
Андрей
```

При данном подходе хэш-таблица рассматривается как объект: мы указываем имя нужного свойства и получаем соответствующее значение. Обращение к ассоциативному массиву с использованием нотации массивов происходит так:

```
PS C:\> $user["Фамилия"]
Попов
PS C:\> $user["Фамилия", "Имя"]
Попов
Андрей
```

Как видим, при работе с хэш-таблицей как с массивом можно получать значения сразу для нескольких ключей.

```
Базовым типом для ассоциативных массивов PowerShell является тип
System.Collections.Hashtable:
PS C:\> $user.GetType().FullName
System.Collections.Hashtable
```

В данном типе определены несколько свойств и методов, которые можно использовать (напомним, что полный список свойств и методов можно получить с помощью командлета `Get-Member`). Например, в свойствах `Keys` и `Values` хранятся все ключи и все значения соответственно:

```
PS C:\> $user.Keys
Фамилия
Имя
Телефон
PS C:\> $user.Values
Попов
Андрей
55-55-55
```

Операции с хэш-таблицей

Давайте научимся добавлять элементы в хэш-таблицу, изменять и удалять их. Добавим в хэш-таблицу `$user` данные о возрасте человека и о городе, где он проживает:

```
PS C:\> $user.Возраст = 33
PS C:\> $user

Name                           Value
----                           -----
Возраст                        33
Фамилия                        Попов
Имя                            Андрей
Телефон                        55-55-55

PS C:\> $user["Город"] = "Саранск"
PS C:\> $user

Name                           Value
----                           -----
Возраст                        33
Город                          Саранск
Фамилия                        Попов
Имя                            Андрей
Телефон                        55-55-55
```

Таким образом, добавляются элементы в ассоциативный массив с помощью простого оператора присваивания с использованием нотации свойств или массивов. Теперь изменим значение уже имеющегося в массиве ключа. Делается это также с помощью оператора присваивания:

```
PS C:\> $user.Город = "Москва"
PS C:\> $user

Name                           Value
----                           -----
Возраст                        33
Город                          Москва
Фамилия                        Попов
Имя                            Андрей
Телефон                        55-55-55
```

Для удаления элемента из ассоциативного массива используется метод `Remove()`:

```
PS C:\> $user.Remove("Возраст")
PS C:\> $user

Name              Value
----              -----
Город             Москва
Фамилия           Попов
Имя               Андрей
Телефон           55-55-55
```

Можно создать пустую хэш-таблицу, не указывая ни одной пары "ключ — значение", и затем заполнять ее последовательно по одному элементу:

```
PS C:\> $a = @{}
PS C:\> $a

PS C:\> $a.one = 1
PS C:\> $a.two = 2
PS C:\> $a

Name                           Value
----                           -----
two                            2
one                            1
```

Как и в случае с обычными массивами, оператор присваивания действует на хэш-таблицы по ссылке. Например, после выполнения следующих команд переменные `$a` и `$b` будут указывать на один и тот же объект:

```
PS C:\> $a = @{one=1;two=2}
PS C:\> $b = $a
PS C:\> $b

Name                           Value
----                           -----
two                            2
one                            1
```

Поменяв значение одного из элементов в `$a`, мы получим тот же результат в `$b`:

```
PS C:\> $a.one=3
PS C:\> $b

Name                           Value
----                           -----
two                            2
one                            3
```

Итоги

- Переменные и литералы в PowerShell являются объектами .NET.
- В PowerShell поддерживается несколько видов символьных строк. Расширение выражений и переменных выполняется для строк в двойных кавычках.
- Escape-символом в PowerShell является обратный апостроф `, а не обратная косая черта, как в большинстве других языков программирования.
- Некоторые стандартные переменные создаются и изменяются самой оболочкой PowerShell.
- Пользовательские переменные PowerShell автоматически создаются при первом присвоении им значений. Тип переменной определяется типом ее значения.

- При создании и инициализации переменных в PowerShell можно указать тип и дополнительные атрибуты, ограничивающие набор их возможных значений.
- PowerShell автоматически преобразует типы при вычислении значений выражений. Можно выполнить явное преобразование, указав нужный тип в квадратных скобках.
- PowerShell поддерживает массивы и хэш-таблицы (ассоциативные массивы). В отличие от многих языков программирования, для обозначения начала или конца массива в PowerShell никакие дополнительные символы не используются. Значения, добавляемые в массив, разделяются запятой.

ГЛАВА 8

Операторы и управляющие инструкции

Одной из особенностей операторов PowerShell является их *полиморфизм*, позволяющий применять один и тот же оператор к объектам разных типов. Например, операторы сложения (+) и умножения (*) можно применять к числам, строкам и массивам.

При этом отличие PowerShell от многих других объектно-ориентированных языков программирования состоит в том, что поведение операторов для основных типов данных (строки, числа, массивы и хэш-таблицы) реализуется непосредственно интерпретатором, а не с помощью того или иного метода объектов определенного типа.

Арифметические операторы

Основные арифметические операторы, поддерживаемые в PowerShell, приведены в табл. 8.1.

Таблица 8.1. *Основные арифметические операторы в PowerShell*

Оператор	Описание	Пример	Результат
+	Складывает два значения	2+4 "aaa"+"bbb" 1,2,3+4,5	6 "aaabbb" 1,2,3,4,5
*	Умножает два значения	2*4 "a"*3 1,2,3*2	8 "aaa" 1,2,3,1,2,3
-	Вычитает одно значение из другого	5-3	2
/	Делит одно значение на другое	6/3 7/4	2 1.75
%	Остаток от целочисленного деления одного значения на другое	7%4	3

С точки зрения полиморфного поведения наиболее интересными являются операторы сложения и умножения. Рассмотрим эти операторы более подробно.

Оператор сложения

Как упоминалось ранее, поведение операторов + и * для чисел, строк, массивов и хэш-таблиц определяется самим интерпретатором PowerShell. В результате сложения или умножения двух чисел получается число. Результатом сложения (конкатенации) двух строк является строка. При сложении двух массивов создается новый массив, являющийся объединением складываемых массивов.

А что произойдет, если попытаться сложить объекты разных типов, например число со строкой? В этом случае поведение оператора будет определяться типом операнда, стоящего слева. Следует запомнить так называемое *правило левой руки*: тип операнда, стоящего слева, задает тип результата действия оператора.

Если левый операнд является числом, то PowerShell попытается преобразовать правый операнд к числовому типу. Например:

```
PS C:\> 1+"12"
13
```

Как видим, строка "12" была преобразована к числу 12. Результат действия оператора сложения — число 13. Теперь обратный пример, когда левый операнд является строкой:

```
PS C:\> "1"+12
112
```

Здесь число 12 преобразуется к строке "12" и в результате конкатенации возвращается строка "112".

Если правый операнд нельзя преобразовать к типу левого операнда, то возникнет ошибка:

```
PS C:\> 1+"a"
Не удается преобразовать значение "a" в тип "System.Int32". Ошибка: "Входная
строка имела неверный формат."
строка:1 знак:1
+ 1+"a"
+ ~~~~~
    + CategoryInfo          : InvalidArgument: (:) [], RuntimeException
    + FullyQualifiedErrorId : InvalidCastFromStringToInteger
```

Если операнд, стоящий слева от оператора сложения, является массивом, то стоящий справа операнд добавляется к этому массиву. При этом создается новый массив типа [object[]], в который копируется содержимое операндов (это связано с тем, что размерность .NET-массивов фиксирована). В процессе создания нового массива все ограничения на типы складываемых массивов будут утрачены. Рассмотрим пример. Создадим массив целых чисел:

```
PS C:\> $a = [int[]](1,2,3,4)
PS C:\> $a.getType().fullName
System.Int32[]
```

Если попробовать изменить значение элемента этого массива на какую-либо строку, то возникнет ошибка, т. к. элементами массива $a могут быть только целые числа:

```
PS C:\> $a[0]="aaa"
Не удается преобразовать значение "aaa" в тип "System.Int32". Ошибка: "Входная
строка имела неверный формат."
строка:1 знак:1
+ $a[0]="aaa"
+ ~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [], RuntimeException
    + FullyQualifiedErrorId : InvalidCastFromStringToInteger
```

Добавим теперь к массиву $a еще один символьный элемент с помощью оператора сложения:

```
PS C:\> $a=$a+"abc"
```

Вновь попробуем изменить значение первого элемента массива $a, записав в него символьную строку:

```
PS C:\> $a[0]="aaa"
PS C:\> $a
aaa
2
3
4
abc
```

Как видим, теперь ошибка не возникает. Это связано с изменением типа массива $a:

```
PS C:\> $a.getType().fullName
System.Object[]
```

Увеличенный массив $a имеет тип [object[]], элементами его могут быть объекты любого типа.

Перейдем теперь к сложению хэш-таблиц. Как и в случае с обычными массивами, при сложении хэш-таблиц создается новый ассоциативный массив, в который копируются элементы из складываемых массивов. При этом оба операнда должны быть хэш-таблицами (никакое преобразование типов здесь не поддерживается). В новый массив сначала копируются элементы хэш-таблицы, стоящей слева от оператора сложения, а затем элементы хэш-таблицы, стоящей справа. Если ключевое значение из правого операнда уже встречалось в левом, то при сложении возникнет ошибка.

Рассмотрим пример:

```
PS C:\> $left=@{a=1;b=2;c=3}
PS C:\> $right=@{d=4;e=5}
PS C:\> $sum=$left+$right
PS C:\> $sum
```

```
Name                           Value
----                           -----
d                              4
a                              1
b                              2
e                              5
c                              3
```

Новая результирующая хэш-таблица по-прежнему имеет тип `System.Collections.Hashtable`:

```
PS C:\> $sum.getType().fullName
System.Collections.Hashtable
```

Оператор умножения

Оператор умножения может действовать на числа, строки и обычные массивы (операция умножения хэш-таблиц не определена). При этом справа от оператора умножения обязательно должно находиться число, в противном случае возникнет ошибка.

Если слева от оператора умножения расположено число, то операция умножения выполняется обычным образом:

```
PS C:\> 6*5
30
```

Если левым операндом является строка, то она повторяется указанное количество раз:

```
PS C:\> "abc"*3
abcabcabc
```

Если умножить строку на ноль, результатом будет пустая строка (тип `System.String`, длина равна 0):

```
PS C:\> "abc"*0

PS C:\> ("abc"*0).getType().fullName
System.String
PS C:\> ("abc"*0).length
0
```

Аналогичным образом оператор умножения действует на массивы:

```
PS C:\> $a=1,2,3
PS C:\> $a=$a*2
PS C:\> $a
1
2
3
1
2
3
```

Как и в случае оператора сложения, при умножении массива на число создается новый массив типа `[Object[]]` нужной размерности и в него копируются элементы первоначального массива.

Операторы вычитания, деления и остатка от деления

Операторы вычитания (`-`), деления (`/`) и остатка от деления (`%`) в PowerShell определены только для чисел. Специального оператора целочисленного деления не предусмотрено: если делятся два целых числа, то результатом будет вещественное число (тип `System.Double`). Например:

```
PS C:\> 123/4
30.75
```

Если нужно округлить результат до ближайшего целого числа, то следует просто преобразовать его к типу `[int]`. Например:

```
PS C:\> [int](123/4)
31
```

При этом следует иметь в виду, что PowerShell при преобразовании вещественного числа в целое использует так называемое округление Банкера: если число является полуцелым (0.5, 1.5, 2.5 и т. д.), то оно округляется до ближайшего четного числа. Таким образом, числа 1.5 и 2.5 округляются до 2, а 3.5 и 4.5 округляются до 4.

Если операнды можно преобразовать к числам, то PowerShell выполняет это преобразование. Например:

```
PS C:\> "123"/4
30.75
PS C:\> 123/"4"
30.75
PS C:\> "123"/"4"
30.75
```

Оператор вычитания можно применять к переменным типа `System.Datetime`, в которых хранятся дата и время. Например, пусть в переменной `$d1` хранится объект, соответствующий дате 8 марта 2021 г., а в переменной `$d2` — объект, соответствующий дате 23 февраля 2021 г.:

```
PS C:\> $d1=Get-Date -Year 2021 -Month 03 -Day 08
PS C:\> $d1.getType().fullName
System.DateTime
PS C:\> $d2=Get-Date -Year 2021 -Month 02 -Day 23
```

Теперь мы можем вычесть из одной даты другую, определив, например, количество дней между ними:

```
PS C:\> ($d1-$d2).Days
13
```

Операторы присваивания

Наряду с простым оператором присваивания (=) в PowerShell поддерживаются С-подобные составные операторы присваивания (табл. 8.2).

Таблица 8.2. *Операторы присваивания в PowerShell*

Оператор	Пример	Аналог	Описание
=	$a = 2		Присваивает переменной указанное значение
+=	$a += 3	$a = $a+3	Прибавляет указанное значение к текущему значению переменной, затем присваивает полученный результат данной переменной
-=	$a -= 3	$a = $a-3	Отнимает указанное значение от текущего значения переменной, затем присваивает полученный результат данной переменной
*=	$a *= 2	$a = $a*2	Умножает текущее значение переменной на указанное число, затем присваивает полученный результат данной переменной
/=	$a /= 2	$a = $a/2	Делит текущее значение переменной на указанное число, затем присваивает полученный результат данной переменной
%=	$a %= 2	$a = $a%2	Находит остаток от деления текущего значения переменной на указанное число, затем присваивает полученный результат данной переменной

В табл. 8.2 для каждого составного оператора присваивания приведен его аналог, составленный с помощью обычного оператора присваивания и определенного арифметического оператора. Следует иметь в виду, что для арифметических операторов здесь справедливы все правила и ограничения, описанные в предыдущем разделе. Например, оператор += можно применять не только к числам, но и к строкам:

```
PS C:\> $a = "aa"
PS C:\> $a += "bb"
PS C:\> $a
aabb
```

Важной особенностью оператора присваивания в PowerShell является то, что с его помощью можно одной командой присвоить значения сразу нескольким переменным. При этом первый элемент присваиваемого значения будет присвоен первой переменной, второй элемент будет присвоен второй переменной, третий элемент — третьей переменной и т. д. Например, в результате выполнения следующей команды переменной $a будет присвоено значение 1, а переменной $b — значение 2:

```
PS C:\> $a, $b = 1, 2
PS C:\> $a
1
PS C:\> $b
2
```

Если присваиваемое значение содержит больше элементов, чем указано переменных, то оставшиеся значения будет присвоены последней переменной. Например, после выполнения следующей команды значением переменной `$a` будет являться число 1, а значением переменной `$b` — массив из чисел 2 и 3:

```
PS C:\> $a, $b = 1, 2, 3
PS C:\> $a
1
PS C:\> $b
2
3
PS C:\> $b.getType().fullName
System.Object[]
```

Операторы сравнения

Операторы сравнения позволяют сравнивать значения параметров в командах. При каждой операции сравнения создается условие, в зависимости от выполнения или невыполнения которого оператор принимает либо значение `True` (истина), либо значение `False` (ложь).

В PowerShell поддерживается несколько операторов сравнения, причем для каждого такого оператора определены версии, учитывающие и не учитывающие регистр букв. Основные операторы сравнения приведены в табл. 8.3.

Таблица 8.3. *Операторы сравнения в PowerShell*

Оператор	Значение	Пример (возвращается значение True)
-eq -ceq -ieq	равно	10 -eq 10
-ne -cne -ine	не равно	9 -ne 10
-lt -clt -ilt	меньше	3 -lt 4
-le -cle -ile	меньше или равно	3 -le 4
-gt -cgt -igt	больше	4 -gt 3
-ge -cge -ige	больше или равно	4 -ge 3
-contains -ccontains -icintains	содержит	1,2,3 -contains 1
-notcontains -cnotcontains -inotcontains	не содержит	1,2,3 -notcontains 4

Базовый вариант операторов сравнения (`-eq`, `-ne`, `-lt` и т. д.) по умолчанию не учитывает регистр букв. Если оператор начинается с буквы `c` (`-ceq`, `-cne`, `-clt` и т. д.),

то при сравнении регистр букв будет приниматься во внимание. Если оператор начинается с буквы `i` (`-ieq`, `-ine`, `-ilt` и т. д.), то регистр букв при сравнении не учитывается.

> ***Замечание***
> В PowerShell для обозначения операторов сравнения не используются обычные символы > и <, т. к. они зарезервированы для перенаправления ввода/вывода.

Как и в случае с арифметическими операторами, для операторов сравнения справедливо "правило левой руки" (определяющим является тип левого операнда). Если число сравнивается со строкой (число стоит слева от оператора сравнения), то строка преобразовывается в число и выполняется сравнение двух чисел. Если левый операнд является строкой, то правый операнд преобразуется к символьному типу и выполняется сравнение двух строк.

Рассмотрим несколько примеров.

Сравнение числа с числом:

```
PS C:\> 01 -eq 001
True
```

Сравнение числа со строкой (строка "001" преобразуется в число 1):

```
PS C:\> 01 -eq "001"
True
```

Сравнение строки с числом (число 001 преобразуется в строку "001"):

```
PS C:\> "01" -eq 001
False
```

Сравнения с использованием массивов

В PowerShell можно сравнить массив с числом, причем результатом такого сравнения будет не логическое значение `True` или `False`, а новый массив, содержащий элементы, прошедшие это сравнение. Например:

```
PS C:\> 1,2,3,4,1,2,3 -eq 1
1
1
PS C:\> 1,2,3,4,1,2,3 -ge 3
3
4
3
```

Если тест на сравнение не пройдет ни один элемент, то результатом будет пустой массив.

Для того чтобы узнать, есть ли в массиве элементы, равные указанному значению, можно воспользоваться оператором `-contains`, в этом случае результатом будет `True` или `False`:

```
PS C:\> 1, 2, 3, 4 -contains 3
True
PS C:\> 1, 2, 3, 4 -contains 5
False
```

Операторы проверки на соответствие шаблону

Наряду с основными операторами сравнения в PowerShell имеются операторы проверки символьных строк на соответствие определенному шаблону. При этом поддерживаются два вида шаблонов: *выражения с подстановочными символами* и *регулярные выражения*.

Шаблоны с подстановочными символами

Ранее мы уже применяли подстановочные (шаблонные) символы (* и ?) при использовании, скажем, командлета dir. Например, команда dir *.doc выводит все файлы в текущем каталоге, имеющие расширение doc и любое имя (шаблонный символ * заменяет любое количество любых символов).

В PowerShell поддерживаются четыре вида подстановочных символов (табл. 8.4) и несколько операторов, проверяющих строки на соответствие шаблонам с подстановочными символами (табл. 8.5).

Таблица 8.4. *Подстановочные символы в PowerShell*

Символ	Описание	Пример	Соответствует	Не соответствует
*	Любое количество любых символов	a*	a ab abc	bc babc
?	Любой один символ	a?b	acb a1b	a ab
[<*симв1*>-<*симв2*>]	Диапазон символов от <симв1> до <симв2>	a[b-d]c	abc acc	aac ac
[<*симв1*><*симв2*>...]	Любой символ из указанного набора	a[bc]c	abc acc	a adc

Таблица 8.5. *Операторы проверки на соответствие шаблону (подстановочные символы)*

Оператор	Описание	Пример (возвращается значение True)
-like -clike -ilike	Сравнение на совпадение с учетом подстановочного символа в тексте	"file.doc" -like "f*.doc"
-notlike -cnotlike -inotlike	Сравнение на несовпадение с учетом подстановочного символа в тексте	"file.doc" -notlike "f*.rtf"

Как видим, пользоваться шаблонами с подстановочными символами очень просто, однако возможности подобных шаблонов ограничены. Если возникает необходимость проверки более сложных условий, то нужно воспользоваться шаблонами с регулярными выражениями.

Шаблоны с регулярными выражениями

Регулярные выражения обобщают и расширяют концепцию шаблонов с подстановочными символами. Для задания образца используются *литералы* и *метасимволы*. Каждый символ, который не имеет специального значения в регулярных выражениях, рассматривается как литерал и должен точно совпасть при поиске. Например, буквы и цифры являются литеральными символами. Метасимволы — это символы со специальным значением в регулярных выражениях (табл. 8.6).

Таблица 8.6. *Некоторые метасимволы для регулярных выражений в PowerShell*

Символ	Описание	Пример	Соответствует	Не соответствует
. (точка)	Символ подстановки соответствует любому символу	a..	abc a34	a ab
*	Повторитель означает ноль или более предшествующих символов или классов символов	a.*b	ab abc awerbc	a bbc
?	Повторитель. означает ноль или один предшествующий символ или класс символов	a?b	b ab cb	ac da
^	Положение в строке обозначает начало строки	^ab	abc abcd	sab acb
$	Положение в строке обозначает конец строки	ab$	sdab ab	abc abb
[<*симв1*><*симв2*>...]	Класс символов обозначает любой символ из указанного множества	a[bc]c	abc acc	a adc
[^<*симв1*><*симв2*>...]	Инвертированный класс обозначает любой символ, не входящий в указанное множество	a[^bc]c	adc a1c	abc acc
[<*симв1*>-<*симв2*>]	Диапазон обозначает любой символ из указанного промежутка	a[b-d]c	abc acc	aac ac
\<*метасимв*>	Исключение определяет использование метасимвола как литерала	\^ab	c^ab ^ab	ab ^ac

Замечание

Поддержка регулярных выражений в PowerShell реализована с помощью специальных классов платформы .NET. Язык регулярных выражений, поддерживаемый этими клас-

сами, обладает очень мощными возможностями, однако подробное рассмотрение данных вопросов выходит за рамки настоящей книги. Дополнительную информацию о некоторых аспектах регулярных выражений можно найти во встроенной справке PowerShell (команда `Get-Help about_regular_expression`).

В PowerShell с регулярными выражениями работают операторы `-match` и `-replace` (табл. 8.7).

Таблица 8.7. *Операторы, работающие с регулярными выражениями*

Оператор	Описание	Пример	Результат
-match -cmatch -imatch	Сравнение на совпадение с учетом регулярных выражений	"книга" -match "ни" "нос" -match "[к-н]ос"	True True
-notmatch -cnotmatch -inotmatch	Сравнение на несовпадение с учетом подстановочного символа в тексте	"книга" -notmatch "^кн"	False
-replace -creplace -ireplace	Замена или удаление символов в строке — левом операнде (эти операторы возвращают измененную строку) Если в качестве правого операнда указываются через запятую две подстроки, то первая из них соответствует фрагменту, который нужно изменить, а вторая — строке, которая будет вставлена в результате замены. Если в качестве правого операнда указана одна подстрока, то она соответствует фрагменту строки, который будет удален	"род" -replace "д", "т" "род" -replace "ро"	рот д

Логические операторы

Иногда внутри одной инструкции необходимо проверить сразу несколько условий. Операторы сравнения можно соединять друг с другом с помощью логических операторов, приведенных в табл. 8.8. При использовании логического оператора PowerShell проверяет каждое условие отдельно, а затем вычисляет значение инструкции целиком, связывая условия с помощью логических операторов.

Таблица 8.8. *Логические операторы в PowerShell*

Оператор	Значение	Пример (возвращается значение True)
-and	логическое И	(10 -eq 10) -and (1 -eq 1)
-or	логическое ИЛИ	(9 -ne 10) -or (3 -eq 4)
-not	логическое НЕ	-not (3 -gt 4)
!	логическое НЕ	!(3 -gt 4)

Управляющие инструкции языка PowerShell

В языке PowerShell, как и в любом другом алгоритмическом языке, имеются элементы, позволяющие выполнить логическое сравнение и предпринять различные действия в зависимости от его результата или дающие возможность повторять одну или несколько команд снова и снова.

Инструкция *If ... Elseif ... Else*

Логические сравнения лежат в основе практически всех алгоритмических языков программирования. В PowerShell с помощью инструкции `If` можно выполнять определенные блоки кода только в том случае, когда заданное условие имеет значение `True` (истина). Также можно задать одно или несколько дополнительных условий выполнения, если все предыдущие условия имели значение `False` (ложь). Наконец, можно задать дополнительный блок кода, который будет выполняться в том случае, если ни одно из условий не имеет значения `True`.

Синтаксис инструкции `If` в общем случае имеет следующий вид:

```
if (условие1)
   {блок_кода1}
[elseif (условие2)
    {блок_кода2}]
[else
   {блок_кода3}]
```

При выполнении инструкции `If` проверяется истинность условного выражения *условие1*.

> ***Замечание***
>
> Условные выражения в PowerShell формируются чаще всего с помощью операторов сравнения и логических операторов, рассмотренных ранее. Кроме того, важной особенностью языка является то, что в качестве условных выражений можно использовать конвейеры команд PowerShell.

Если *условие1* имеет значение `True`, то выполняется *блок_кода1*, после чего PowerShell завершает выполнение инструкции `If`. Если *условие1* имеет значение `False`, то PowerShell проверяет истинность условного выражения *условие2*. Если *условие2* имеет значение `True`, то выполняется *блок_кода2*, после чего PowerShell завершает выполнение инструкции `If`. Если и *условие1*, и *условие2* имеют значение `False`, то выполняется *блок_кода3*, а затем PowerShell завершает выполнение инструкции `If`.

Приведем пример использования инструкции `If` в интерактивном режиме работы. Запишем в переменную `$a` число 10:

```
PS C:\> $a=10
```

Сравним теперь значение переменной $a с числом 15:

```
PS C:\> if ($a -eq 15) {
>> 'Значение $a равно 15'
>> }
>> else {'Значение $a не равно 15'}
>>
Значение $a не равно 15
```

Из данного примера также видно, что в оболочке PowerShell в интерактивном режиме можно выполнять инструкции, состоящие из нескольких строк (это может оказаться очень кстати при отладке сценариев).

Цикл *While*

В PowerShell поддерживается несколько видов циклов. Самый простой из них — цикл While, в котором команды выполняются до тех пор, пока проверяемое условие имеет значение True.

Инструкция While имеет следующий синтаксис:

while (*условие*){*блок_команд*}

При выполнении инструкции While оболочка PowerShell вычисляет раздел *условие* инструкции, прежде чем перейти к разделу *блок_команд*. Условие в инструкции принимает значения True или False. До тех пор, пока условие имеет значение True, PowerShell повторяет выполнение раздела *блок_команд*.

> ***Замечание***
>
> Как и в инструкции If, в условном выражении цикла While может использоваться конвейер команд PowerShell.

Раздел *блок_команд* инструкции While содержит одну или несколько команд, которые выполняются каждый раз при входе и повторении цикла.

Например, следующая инструкция While отображает числа от 1 до 3, если переменная $val не была создана или была создана и инициализирована значением 0:

```
PS C:\> while($val -ne 3)
>> {
>>    $val++
>>    $val
>> }
>>
1
2
3
```

В данном примере условие (значение переменной $val не равно 3) имеет значение True, пока $val равно 0, 1 или 2. При каждом повторении цикла значение $val увеличивается на 1 с использованием унарного оператора увеличения значения ++

($val++). При последнем выполнении цикла значением $val становится 3. При этом проверяемое условие принимает значение False, и цикл завершается.

Цикл *Do ... While*

Цикл Do ... While похож на цикл While, однако условие в нем проверяется не до блока команд, а после:

do{*блок_команд*}while (*условие*)

Например:

```
PS C:\> $val=0
PS C:\> do {$val++; $val} while ($val -ne 3)
1
2
3
```

Цикл *For*

Инструкция For в PowerShell реализует еще один тип циклов — цикл со счетчиком. Обычно цикл For применяется для итерации массива значений и выполнения действий на подмножестве этих значений. В PowerShell инструкция For используется не так часто, как в других языках программирования, т. к. коллекции объектов обычно удобнее обрабатывать с помощью инструкции Foreach. Однако если необходимо знать, с каким именно элементом коллекции или массива мы работаем на данной итерации, то цикл For может помочь.

Синтаксис инструкции For:

for (*инициация; условие; повторение*){*блок_команд*}

Составные части цикла For имеют следующий смысл:

- *инициация* — это одна или несколько разделяемых запятыми команд, выполняемых перед началом цикла. Данная часть цикла обычно используется для создания и присвоения начального значения переменной, которая будет затем основой для проверяемого условия в следующей части инструкции For;
- *условие* — часть инструкции For, которая может принимать логическое значение True или False. PowerShell проводит оценку условия при каждом выполнении цикла For. Если результатом этой оценки является True, то выполняются команды в блоке *блок_команд*, после чего проводится новая оценка условия инструкции. Если результатом оценки условия вновь становится True, команды блока *блок_команд* выполняются вновь, и т. д., пока результатом проверки условия не станет False;
- *повторение* — это одна или несколько разделяемых запятыми команд, выполняемых при каждом повторении цикла. Данная часть цикла обычно используется для изменения переменной, проверяемой внутри условия;

- *блок_команд* — это набор из одной или нескольких команд, выполняющихся при вхождении в цикл или при повторении цикла. Содержимое блока *блок_команд* заключается в фигурные скобки.

Классический пример:

```
PS C:\> for ($i = 0; $i -lt 5; $i++) { $i }
0
1
2
3
4
```

Цикл *Foreach*

Инструкция Foreach позволяет последовательно перебирать элементы коллекций. Самым простым и наиболее часто используемым типом коллекции, по которой производится перемещение, является массив. Обычно в цикле Foreach одна или несколько команд выполняются на каждом элементе массива.

Особенностью цикла Foreach является то, что его синтаксис и работа зависят от того, где расположена инструкция Foreach: вне конвейера команд или внутри конвейера.

Инструкция *Foreach* вне конвейера команд

В этом случае синтаксис цикла Foreach имеет следующий вид:

```
foreach ($элемент in $коллекция){блок_команд}
```

В круглых скобках указывается коллекция, в которой производится итерация. При выполнении цикла Foreach система автоматически создает переменную $элемент. Перед каждой итерацией в цикле этой переменной присваивается значение очередного элемента в коллекции. В разделе *блок_команд* содержатся команды, выполняемые на каждом элементе коллекции.

Например, цикл Foreach в следующем примере отображает значения в массиве с именем $letterArray:

```
PS C:\> $letterArray = "a","b","c","d"
PS C:\> foreach ($letter in $letterArray){Write-Host $letter}
a
b
c
d
```

В первой команде здесь создается массив $letterArray, в который записываются четыре элемента: символы a, b, c и d. При первом выполнении инструкции Foreach переменной $letter присваивается значение, равное первому элементу в $letterArray (a), затем используется командлет Write-Host для отображения переменной $letter. При следующей итерации цикла переменной $letter присваивает-

ся значение b и т. д. После того как будут перебраны все элементы массива $letterArray, произойдет выход из цикла.

Инструкция Foreach может также использоваться совместно с командлетами, возвращающими коллекции элементов. Например:

```
PS C:\> $n = 0; foreach ($f in dir *.txt) { $n += $f.length }
PS C:\> $n
2690555
```

Здесь сначала создается и обнуляется переменная $n, затем в цикле Foreach с помощью командлета dir формируется коллекция файлов с расширением txt, находящихся в текущем каталоге. В инструкции Foreach перебираются все элементы этой коллекции, на каждом шаге к текущему элементу (соответствующему файлу) можно обратиться с помощью переменной $f. В блоке команд цикла Foreach к текущему значению переменной $n добавляется значение поля length (размер файла) переменной $f. В результате выполнения данного цикла в переменной $n будет храниться суммарный размер файлов в текущем каталоге, которые имеют расширение txt.

Инструкция *Foreach* внутри конвейера команд

Если инструкция Foreach появляется внутри конвейера команд, то PowerShell понимает под ней псевдоним Foreach, соответствующий командлету ForEach-Object. Таким образом в этом случае фактически выполняется командлет ForEach-Object и уже не нужно указывать часть инструкции ($*элемент* in $*коллекция*), т. к. элементы коллекции блоку команд предоставляет предыдущая команда в конвейере.

Синтаксис инструкции Foreach, применяемой внутри конвейера команд, в простейшем случае выглядит следующим образом:

команда | foreach {*блок_команд*}

Рассмотренный пример с подсчетом суммарного размера текстовых файлов из текущего каталога для данного варианта инструкции Foreach примет следующий вид:

```
PS C:\> $n = 0; dir *.txt | foreach { $n += $_.length }
PS C:\> $n
2690555
```

> ***Замечание***
>
> Напомним, что специальная переменная $_ используется в командлетах, производящих обработку элементов конвейера, для обращения к текущему объекту конвейера и извлечения его свойств.

В общем случае в псевдониме Foreach может указываться не один блок команд, а три: начальный, основной и конечный блоки команд. Начальный и конечный блоки команд выполняются один раз, а основной блок запускается каждый раз при очередной итерации по коллекции или массиву.

Синтаксис псевдонима Foreach, используемого в конвейере команд с начальным, основным и конечным блоками команд, выглядит следующим образом:

команда | foreach {*начальный_блок_команд*} {*основной_блок_команд*} {*конечный_блок_команд*}

Для этого варианта инструкции Foreach наш пример можно записать следующим образом:

```
PS C:\> dir *.txt | foreach {$n = 0}{ $n += $_.length }{Write-Host $n}
2690555
```

Вопросы производительности

Итак, задачу перебора элементов коллекции можно решить либо с помощью оператора цикла Foreach, либо с помощью обработки объектов в конвейере командлетом ForEach-Object.

Выбирая, каким из этих подходов воспользоваться, следует иметь в виду следующий момент. Блок кода в командлете ForEach-Object начинает выполняться, как только в конвейере появляется первый объект, а для выполнения тела цикла foreach() необходимо, чтобы все объекты были получены и коллекция была полностью сформирована. Поэтому при использовании командлета ForEach-Object требуется меньше оперативной памяти (не нужно размещать в памяти целую коллекцию объектов). Однако при массовом переборе объектов готовой коллекции с помощью foreach() срабатывают внутренние оптимизации и время обработки всех элементов будет существенно меньше, чем при запуске кода для каждого элемента в контейнере.

Поэтому выбор способа обработки коллекции объектов зависит от того, какая характеристика для нас сейчас важнее: расход оперативной памяти или скорость выполнения задачи.

Метки циклов, инструкции *Break* и *Continue*

Инструкция Break позволяет выйти из цикла любого типа, не дожидаясь окончания его итераций. Рассмотрим простой пример:

```
PS C:\> $i=0; while ($true) {if ($i++ -ge 3) { break } $i }
1
2
3
```

Условием цикла while в данном случае является логическая константа $true, поэтому этот цикл не завершился бы никогда. Инструкция Break, которая срабатывает при достижении переменной $i значения 3, позволяет выйти из данного цикла.

Инструкция Continue осуществляет переход к следующей итерации цикла любого типа. Например:

```
PS C:\> for ($i=0; $i -le 5; $i++) { if ($i -ge 4) { continue } $i }
0
1
2
3
```

В данном примере на экран выводится только число от 0 до 3, т. к. для значений переменной `$i`, больших либо равных 4, срабатывает инструкция `Continue`.

В языке PowerShell поддерживается возможность немедленного выхода или перехода к следующей итерации не только для одиночного цикла, но и для вложенных циклов. Для этого циклам присваиваются специальные метки, которые указываются в начале строки перед ключевым словом, задающим цикл того или иного типа. Данные метки затем можно использовать во вложенных циклах совместно с инструкциями `Break` или `Continue`, указывая, на какой именно цикл должны действовать данные инструкции. Рассмотрим простой пример:

```
PS C:\> :outer while ( $true ) {
>> while ( $true ) {
>>    break outer
>>    }
>> }
>>
```

Здесь инструкция `Break` осуществляет выход из внешнего цикла с меткой `outer`. Если бы метка не была указана, внешний цикл не завершился бы никогда.

Инструкция *Switch*

Инструкция `Switch`, объединяющая несколько проверок условий внутри одной конструкции, имеется во многих языках программирования. Однако в языке PowerShell данная инструкция обладает мощными дополнительными возможностями:

- может использоваться как аналог цикла, проверяя значения не единственного элемента, а целого массива;
- может проверять элементы на соответствие шаблону с подстановочными символами или регулярными выражениями;
- может обрабатывать текстовые файлы, используя в качестве проверяемых элементов строки из файлов.

Виды проверок внутри *Switch*

Рассмотрим вначале самую простую форму инструкции `Switch`, когда одно скалярное выражение по очереди сопоставляется с несколькими условиями. Например:

```
PS C:\> $a = 3
PS C:\> switch ($a) {
>>         1 {"Один"}
>>         2 {"Два"}
>>         3 {"Три"}
>>         4 {"Четыре"}
>>         }
>>
Три
```

В данном примере значение переменной $a последовательно сравнивается с числами 1, 2, 3 и 4. При совпадении выполняется соответствующий блок кода, указанный в фигурных скобках (в нашем случае просто выводится строка).

Если для проверяемого значения справедливы несколько условий из списка, то будут выполнены все действия, сопоставленные этим условиям. Например:

```
PS C:\> $a = 3
PS C:\> switch ($a) {
>>          1 {"Один"}
>>          2 {"Два"}
>>          3 {"Три"}
>>          4 {"Четыре"}
>>          3 {"Еще раз три"}
>> }
>>
Три
Еще раз три
```

Если нужно ограничиться только первым совпадением, то следует применить инструкцию Break:

```
PS C:\> $a = 3
PS C:\> switch ($a) {
>>           1 {"Один"}
>>           2 {"Два"}
>>           3 {"Три"; break}
>>           4 {"Четыре"}
>>           3 {"Еще раз три"}
>>  }
>>
Три
```

В данном случае проверка условий внутри Switch прерывается после нахождения первого соответствия. Если ни одно соответствие не найдено, то инструкция Switch не выполняет никаких действий:

```
PS C:\> switch (3) {
>> 1 {"Один"}
>> 2 {"Два"}
>> 4 {"Четыре"}
>> }
>>
PS C:\>
```

С помощью ключевого слова default можно задать действие по умолчанию, которое будет выполняться в том случае, когда не найдено ни одного соответствия. Например:

```
PS C:\> switch (3) {
>> 1 {"Один"}
>> 2 {"Два"}
```

```
>> default {"Ни один, ни два"}
>> }
>>
Ни один, ни два
```

По умолчанию в инструкции `Switch` производится прямое сравнение с объектами, указанными в условии. Сравнение строк при этом производится без учета регистров символов, например:

```
PS C:\> switch ('абв') {
>> 'абв' {"Первое совпадение"}
>> 'АБВ' {"Второе совпадение"}
>> }
>>
Первое совпадение
Второе совпадение
```

Если при сравнении следует учесть регистр символов, то нужно указать параметр `-casesensitive`:

```
PS C:\> switch -casesensitive ('абв') {
>> 'абв' {"Первое совпадение"}
>> 'АБВ' {"Второе совпадение"}
>> }
>>
Первое совпадение
```

Кроме обычного сравнения может проверять элементы на соответствие шаблону с подстановочными символами. Для этого используется переключатель `-wildcard`, например:

```
PS C:\> switch -wildcard ('абв') {
>> 'а*' {"Начинается с а"}
>> '*в' {"Оканчивается на в"}
>> }
>>
Начинается с а
Оканчивается на в
```

Проверяемый элемент (объект) доступен внутри инструкции `Switch` через специальную переменную `$_` (напомним, что в переменной с таким названием сохраняется и текущий элемент, передаваемый по конвейеру от одного командлета другому). Например:

```
PS C:\> switch -wildcard ('абв') {
>> 'а*' {"$_ начинается с а"}
>> '*в' {"$_ оканчивается на в"}
>> }
>>
абв начинается с а
абв оканчивается на в
```

Как видим, переменная $_ в строке в двойных кавычках заменяется на строку, соответствующую проверяемому значению.

Переключатель -regex позволяет проверять элементы на соответствие шаблону, содержащему регулярные выражения (вопросы, связанные с регулярными выражениями, обсуждались ранее в этой главе). Например, предыдущий пример можно записать следующим образом:

```
PS C:\> switch -regex ('абв') {
>> '^а' {"$_ начинается с а"}
>> 'в$' {"$_ оканчивается на в"}
>> }
>>
абв начинается с а
абв оканчивается на в
```

Кроме проверок на простое совпадение или соответствие шаблону инструкция Switch позволяет производить более сложные проверки, указывая для этого вместо шаблонов блоки кода на языке PowerShell. Проверяемое значение при этом вновь доступно через переменную $_. Рассмотрим пример:

```
PS C:\> switch (10) {
>> {$_ -gt 5} {"$_ больше 5"}
>> {$_ -lt 20} {"$_ меньше 20"}
>> 10 {"$_ равно 10"}
>> }
>>
10 больше 5
10 меньше 20
10 равно 10
```

В данном случае выполняются все три проверяемых условия: значения первых двух выражений "10 -gt 5" и "10 -lt 20" равны True, третье условие — обычное сравнение двух чисел на равенство (10 равно 10).

Проверка массива значений

До настоящего момента все значения, которые мы проверяли в инструкции Switch, были скалярными величинами. Язык PowerShell допускает использование в качестве проверяемого значения массивов элементов, причем массивы могут задаваться как явно, так и получаться в результате выполнения какой-либо команды или считывания строк из текстового файла. Рассмотрим пример:

```
PS C:\> switch (1,2,3,4,5,6) {
>> {$_ % 3} {"$_ не делится на три"}
>> default {"$_ делится на три"}
>> }
>>
1 не делится на три
2 не делится на три
```

```
3 делится на три
4 не делится на три
5 не делится на три
6 делится на три
```

В данном случае массив целых чисел задается явным перечислением своих элементов. Все элементы массива по очереди проверяются внутри инструкции `Switch`: если остаток от деления текущего элемента на 3 не равен нулю, то выводится сообщение "$_ не делится на три", где вместо $_ подставляется, как обычно, значение проверяемого элемента. В противном случае выводится сообщение "$_ делится на три".

Приведем еще один пример, когда массив проверяемых элементов является результатом выполнения команды PowerShell. Пусть нам нужно узнать количество файлов с расширениями txt и log, находящихся в системном каталоге Windows. Сначала обнуляем переменные-счетчики:

```
PS C:\> $txt = $log = 0
```

Выполним теперь подходящую инструкцию `Switch`. Коллекцию файлов из системного каталога Windows получим с помощью команды `dir $env:SystemRoot` (напомним, что путь к нужному каталогу хранится в переменной среды `SystemRoot`). Файлы с расширениями txt и log будем отбирать путем проверки на соответствие шаблонам с подстановочными символами (*.txt и *.log соответственно) и в соответствующих блоках кода будем увеличивать значение переменных `$txt` и `$log`:

```
PS C:\> switch -wildcard (dir $env:SystemRoot) {
>> *.txt {$txt++}
>> *.log {$log++}
>> }
>>
```

Выведем теперь значения переменных `$txt` и `$log`:

```
PS C:\> "txt-файлы: $txt    log-файлы: $log"
txt-файлы: 21    log-файлы: 156
```

Для использования в качестве входного массива строк из определенного текстового файла нужно указать в инструкции `Switch` параметр `-file` и путь к нужному файлу. Рассмотрим пример. Пусть в системном каталоге Windows имеется файл KB946627.log с протоколом установки очередного обновления операционной системы. Выведем с помощью инструкции `Switch` все строки из этого файла, содержащие подстроки "Source:" или "Destinantion:" (используются параметры `-wildcard` и `-file`, строки из файла проверяются на соответствие шаблону с подстановочными символами):

```
PS C:\> switch -widlcard -file $env:SystemRoot\KB946627.log {
>> *Source:* {$_}
>> *Destination:* {$_}
>> }
>>
```

```
10.281: Source:C:\WINDOWS\system32\SET32.tmp (5.1.2600.3173)
10.281: Destination:C:\WINDOWS\system32\rpcrt4.dll (5.1.2600.2180)
10.281: Source:C:\WINDOWS\system32\SETDB.tmp (5.1.2600.3243)
10.281: Destination:C:\WINDOWS\system32\xpsp3res.dll (5.1.2600.3132)
10.281: Source:C:\WINDOWS\system32\SET9E.tmp (6.0.2900.3231)
10.281: Destination:C:\WINDOWS\system32\wininet.dll (6.0.2900.3132)
10.281: Source:C:\WINDOWS\system32\SET9F.tmp (6.0.2900.3231)
10.281: Destination:C:\WINDOWS\system32\urlmon.dll (6.0.2900.3132)
. . .
```

Итоги

- Операторы PowerShell являются полиморфными, один и тот же оператор можно применять к объектам разных типов.
- Поведение большинства бинарных операторов определяется "правилом левой руки": тип операнда, стоящего слева, задает тип результата действия оператора.
- PowerShell поддерживает C-подобные составные операторы присваивания.
- Операторы сравнения задаются в PowerShell с помощью символьных аббревиатур, символы < и > не используются.
- Поддерживаются два варианта операторов проверки строки на соответствие шаблону: с использованием подстановочных символов и шаблонов с регулярными выражениями.
- В PowerShell поддерживаются стандартные управляющие инструкции для организации ветвлений и циклов.
- Инструкция `Switch` в PowerShell имеет больше возможностей, чем аналогичные конструкции в других языках программирования. С помощью одного `Switch`, без дополнительного цикла, можно проверять на соответствие шаблону элементы массива или строки из текстового файла.
- Задачи обработки коллекций объектов можно решать с помощью циклов или конвейеров. При использовании циклов, как правило, потребляется больше памяти, но операции выполняются быстрее, чем в случае с конвейерами.

ГЛАВА 9

Функции, фильтры, сценарии и модули

До настоящего момента мы работали преимущественно со стандартными скомпилированными командлетами, функциональность которых мы не можем изменить, т. к. их исходный код в оболочке PowerShell недоступен. Функции и сценарии — это два других типа команд PowerShell, которые можно создавать и изменять по своему усмотрению, пользуясь языком PowerShell.

Следует сразу обратить внимание, что функции в PowerShell имеют некоторые особенности по сравнению с функциями в традиционных языках программирования, т. к. PowerShell — это в первую очередь оболочка. В традиционных языках функция, как правило, является аналогом метода объекта, а в PowerShell *функция — это команда*. Отсюда различие в способах вызова функций и передачи им аргументов.

Обычно функции в традиционных языках возвращают единственное значение того или иного типа. Значением функции PowerShell может быть целый массив различных объектов, т. к. каждое вычисляемое в функции выражение помещает свой результат в выходной поток.

Кроме того, функции в PowerShell делятся на несколько типов согласно их поведению внутри конвейера команд.

Поэтому для грамотной работы в оболочке и написания корректных программ на языке PowerShell необходимо рассмотреть вопросы, связанные с реализацией функций и сценариев в этой системе.

Функции в PowerShell

Напомним, что функция в PowerShell — это блок кода, имеющий название и находящийся в памяти до завершения текущего сеанса командной оболочки. Если функция определяется без формальных параметров, то для ее задания достаточно указать ключевое слово `function`, затем имя функции и заключенный в фигурные скобки список выражений, составляющих тело функции. Например, создадим функцию `MyFunc`:

```
PS C:\> function MyFunc {"Всем привет!"}
PS C:\>
```

ЗАМЕЧАНИЕ

Здесь мы лишь для краткости используем для функции имя, которое ничего не говорит о ее назначении. Вообще, нужно стараться, чтобы имя функции описывало то, что она делает. Желательно придерживаться принятого в PowerShell соглашения об именах в формате "действие — объект", причем в качестве действия выбирать один из рекомендуемых глаголов. Список таких рекомендуемых глаголов выводит команда `Get-Verb`.

Для вызова этой функции нужно просто ввести ее имя:

```
PS C:\> MyFunc
Всем привет!
```

Отметим, что во многих языках программирования при вызове функции после ее имени нужно указывать круглые скобки. В PowerShell этого делать нельзя, возникнет ошибка:

```
PS C:\> MyFunc()
строка:1 знак:8
+ MyFunc()
+        ~
После ''('' ожидалось выражение.
  + CategoryInfo          : ParserError: (:) [],
                                          ParentContainsErrorRecordException
  + FullyQualifiedErrorId : ExpectedExpression
```

Это связано с тем, что последняя команда интерпретируется оболочкой как вызов функции `MyFunc` с аргументом `()`. Круглые скобки в PowerShell обозначают результат вычисления выражения или массив, поэтому система ожидает, что в круглых скобках будет указано выражение или перечислены элементы массива, а там ничего нет. Поэтому и возникает ошибка, ведь для корректного объявления пустого массива в PowerShell перед скобками нужно ставить знак `@`.

Функции могут работать с аргументами, которые передаются им при запуске, причем поддерживаются два варианта обработки таких аргументов: с помощью переменной `$args` и через описание формальных параметров.

Обработка аргументов с помощью переменной *$args*

Функция в PowerShell имеет доступ к аргументам, с которыми она была запущена, даже если при определении этой функции не были заданы формальные параметры. Все фактические аргументы функции автоматически сохраняются в переменной `$args`. Другими словами, в переменной `$args` содержится массив, элементами которого являются параметры функции, указанные при ее запуске. Для примера добавим переменную `$args` в нашу функцию `MyFunc`:

```
PS C:\> function MyFunc {"Привет, $args!"}
```

Так как переменная `$args` помещена в строку в двойных кавычках, то при запуске функции значение этой переменной будет вычислено (расширено) и результат будет вставлен в строку.

Вызовем функцию MyFunc с тремя параметрами:

```
PS C:\> MyFunc Андрей Сергей Иван
Привет, Андрей Сергей Иван!
```

Как видим, три указанных нами параметра (три элемента массива $args) помещены в выходную строку и разделены между собой пробелами. Можно изменить символ, разделяющий элементы массивов при их подстановке в расширяемые строки, переопределив значение специальной переменной $OFS:

```
PS C:\> function MyFunc {
>> $OFS = ","
>> "Привет, $args!"}
>>
PS C:\> MyFunc Андрей Сергей Иван
Привет, Андрей,Сергей,Иван!
```

Еще раз обратите внимание, что, в отличие от традиционных языков программирования, функции в PowerShell являются командами (это не методы объектов!), поэтому их аргументы указываются через пробел без круглых скобок, разделяющих запятых и без выделения символьных строк кавычками. Чтобы наглядно убедиться в этом, запустим функцию MyFunc обычным для других языков программирования образом:

```
PS C:\> MyFunc("Андрей", "Сергей", "Иван")
Привет, System.Object[]!
```

Напомним, что массивы в PowerShell задаются перечислением своих элементов, а круглые скобки, окружающие какое-либо выражение, означают, что это выражение должно быть вычислено. Поэтому ошибки при таком вызове функции не возникает, но в данном случае в функцию MyFunc передаются не три символьных аргумента, а один аргумент, являющийся массивом из трех элементов!

Так как переменная $args является массивом, то внутри функции можно обращаться к отдельным аргументам по их порядковому номеру (не забывайте, что нумерация элементов массивов начинается с нуля), а с помощью свойства Count или Length определять общее количество аргументов, переданных функции.

Для примера создадим функцию SumArgs, которая будет сообщать о количестве своих аргументов и вычислять их сумму:

```
PS C:\> function SumArgs{"Количество аргументов: $($args.count)"
>> $n = 0
>> for($i = 0; $i -lt $args.count; $i++) { $n += $args[$i] }
>> "Сумма аргументов: $n"}
>>
PS C:\>
```

Вызовем функцию SumArgs с тремя числовыми аргументами:

```
PS C:\> SumArgs 1 2 3
Количество аргументов: 3
Сумма аргументов: 6
```

Кроме массива `$args` в PowerShell поддерживается более привычный и удобный подход к обработке аргументов функций — с помощью задания формальных параметров.

Формальные параметры функций

В PowerShell, как и в большинстве других языков программирования, при описании функции можно задать список формальных параметров, значения которых во время выполнения функции будут заменены значениями фактически переданных аргументов.

При этом список формальных параметров можно указать в круглых скобках после имени функции (таким образом описываются параметры функций в большинстве языков программирования) или внутри самой функции с помощью специального оператора `Param`.

Позиционные и именованные параметры

Определим, например, функцию `subtract` для нахождения разности двух своих аргументов (уменьшаемому соответствует параметр `$from`, вычитаемому — параметр `$count`):

```
PS C:\> function subtract ($from, $count) {$from - $count}
```

При вызове функции `subtract` ее формальные параметры заменяются на фактические аргументы, определяемые по позиции в командной строке или по имени.

> ***Замечание***
> В очередной раз напомним, что аргументы функции указываются так же, как параметры других команд: через пробел, без круглых скобок!

Например:

```
PS C:\> subtract 10 2
8
```

В этом случае соответствие формальных параметров фактически переданным аргументам определяется по позиции: вместо первого параметра `$from` подставляется число 10, вместо второго параметра `$count` подставляется число 2.

При указании аргументов можно использовать имена формальных параметров, порядок указания аргументов при этом становится несущественным. Например:

```
PS C:\> subtract -from 10 -count 2
8
PS C:\> subtract -count 3 -from 5
2
```

Если дважды указать имя одного и того же параметра, то возникнет ошибка:

```
PS C:\> subtract -from 10 -from 2
subtract : Не удается привязать параметр, т. к. параметр "from" указан более
одного раза. Для указания множественных значений параметров, допускающих
```

```
множественные значения, используйте синтаксис массива.
Например, "-parameter value1,value2,value3".
строка:1 знак:19
+ subtract -from 10 -from 2
+                  ~~~~~
  + CategoryInfo          : InvalidArgument: (:) [subtract],
                                                    ParameterBindingException
  + FullyQualifiedErrorId : ParameterAlreadyBound,subtract
```

При вызове функции возможен и третий вариант задания аргументов, когда для некоторых задаются имена, а некоторые определяются по позиции в командной строке. При этом действует следующий алгоритм:

- все именованные аргументы сопоставляются соответствующим формальным параметрам и удаляются из списка аргументов;
- оставшиеся параметры (безымянные или имеющие имя, которому не соответствует ни один формальный параметр) сопоставляются формальным параметрам по позиции.

Например:

```
PS C:\> subtract -from 10 2
8
PS C:\> subtract -count 2 10
8
```

> ***Замечание***
>
> По возможности при вызове функций следует избегать смешивания именованных параметров и "позиционных" аргументов — это поможет избежать возможных ошибок при сопоставлении формальных и фактических параметров.

По умолчанию функции PowerShell, как и другие команды оболочки, ведут себя полиморфным образом, т. е. могут принимать аргументы разных типов. Например, определим функцию add, складывающую два своих аргумента:

```
PS C:\> function add ($x, $y) {$x + $y}
```

Выполним эту функцию с аргументами-числами и аргументами-строками:

```
PS C:\> add 2 3
5
PS C:\> add "2" "3"
23
```

В первом случае функция add возвращает число 5, а во втором — строку "23":

```
PS C:\> (add 2 3).getType().fullName
System.Int32
PS C:\> (add "2" "3").getType().fullName
System.String
```

При вызове функции может быть указано большее количество фактических аргументов, чем было задано формальных параметров. При этом "лишние" аргументы

будут помещены в массив $args. Для иллюстрации рассмотрим функцию ShowArgs с двумя формальными параметрами:

```
PS C:\> function ShowArgs ($x,$y) {
>> "Первый аргумент: $x"
>> "Второй аргумент: $y"
>> "Остальные аргументы: $args"}
>>
```

Вызовем эту функцию с одним аргументом:

```
PS C:\> ShowArgs 1
Первый аргумент: 1
Второй аргумент:
Остальные аргументы:
```

В этом случае единственный аргумент сопоставляется с параметром $x, значение параметра $y становится равным $null, а массив $args не содержит элементов ($args.length=0).

Запустим теперь функцию ShowArgs с двумя аргументами:

```
PS C:\> ShowArgs 1 2
Первый аргумент: 1
Второй аргумент: 2
Остальные аргументы:
```

Как видим, в данном случае параметры $x и $y получают значения 1 и 2, а массив $args остается пустым.

Наконец, запустим функцию ShowArgs с тремя и четырьмя аргументами:

```
PS C:\> ShowArgs 1 2 3
Первый аргумент: 1
Второй аргумент: 2
Остальные аргументы: 3
PS C:\> ShowArgs 1 2 3 4
Первый аргумент: 1
Второй аргумент: 2
Остальные аргументы: 3 4
```

Дополнительные аргументы действительно сохраняются в массиве $args.

Ограничение параметров по типу

При объявлении функции можно явно задать тип формальных параметров. Например, определим функцию int_add, которая будет складывать два своих целочисленных аргумента:

```
PS C:\> function int_add ([int]$x, [int]$y) {$x + $y}
```

Как и в предыдущем случае, вызовем данную функцию с аргументами-числами и с аргументами-строками:

```
PS C:\> int_add 2 3
5
PS C:\> int_add "2" "3"
5
```

Как видим, результат получается один и тот же, символьные аргументы автоматически преобразовались к целочисленному типу.

Если преобразование выполнить не удается, то возникнет ошибка:

```
PS C:\> int_add "2a" "3"
int_add : Не удается обработать преобразование аргументов для параметра "x".
Не удается преобразовать значение "2a" в тип "System.Int32". Ошибка: "Входная
строка имела неверный формат."
строка:1 знак:9
+ int_add "2a" "3"
+         ~~~~
  + CategoryInfo          : InvalidData: (:) [int_add],
                                ParameterBindingArgumentTransformationException
  + FullyQualifiedErrorId : ParameterArgumentTransformationError,int_add
```

Значения по умолчанию для параметров

При объявлении формальных параметров можно указать значения, которые будут принимать эти параметры по умолчанию (если явно не указан соответствующий фактический аргумент).

> ***Замечание***
>
> В качестве инициализирующих значений можно указывать как константы, так и выражения PowerShell.

Например, определим функцию `add` с двумя формальными параметрами `$x` и `$y`, которые по умолчанию инициализируются числами 2 и 3:

```
PS C:\> function add ($x = 2, $y = 3) {$x + $y}
```

Если запустить эту функцию без аргументов, то оба параметра инициализируются значениями по умолчанию и функция возвращает число 5:

```
PS C:\> add
5
```

Если запустить функцию `add` с одним аргументом, то указанное значение присвоится параметру `$x`, а параметр `$y` вновь инициализируется значением по умолчанию (`$y=3`):

```
PS C:\> add 5
8
```

При указании двух параметров функция `add` вернет их сумму:

```
PS C:\> add 6 6
12
```

Дополнительные атрибуты и валидация параметров

Перед типом параметра и его именем можно указать в квадратных скобках дополнительный набор спецификаций в формате:

```
parameter(ключ = значение [, ...])
```

Таким образом можно, например, указать, что параметр является обязательным, определить его краткий псевдоним, задать описание этого параметра. В табл. 9.1 приведены некоторые такие ключи.

Таблица 9.1. *Ключи для задания атрибутов параметров*

Ключ	Значение	Описание
`Mandatory`	`$true` или `$false`	Если значение равно `$true`, то при вызове командлета параметр нужно указывать обязательно. По умолчанию параметры командлетов являются необязательными
`HelpMessage`	*строка*	Краткое описание параметра
`ValueFromPipeline`	`$true` или `$false`	Если равно `$true`, то параметр может получать свое значение в виде объекта, поступающего по конвейеру
`ValueFromPipelineByPropertyName`	`$true` или `$false`	Если равно `$true`, то параметр может получать свое значение из одноименного свойства объекта, поступающего по конвейеру
`ValueFromRemainingArguments`	`$true` или `$false`	Если равно `$true`, то данный параметр получает все остальные значения вызывающей командной строки в виде массива

Например, создадим функцию `Print-Message`, которая будет несколько раз печатать строку, передаваемую ей в качестве аргумента. Параметр для текста выводимого сообщения назовем `-message`, количество повторов будем передавать через параметр `-repeat`.

```
function Print-Message (
    [parameter(Mandatory=$true, HelpMessage='Это очень важный параметр!')]
                                                        [string]$message,
    [int]$repeat = 1
)
{
    for ($i = 1; $i -le $repeat; $i++) {
        Write-Host($message)
    }
}
```

Здесь при определении функции мы указали, что символьный параметр `$message` является обязательным (ключ `Mandatory` со значением `$true`), и задали для него

краткое описание (ключ `HelpMessage`). Целочисленный параметр `$repeat` при вызове функции можно не указывать, по умолчанию его значение равно единице.

Если запустить `Print-Message` без параметров, то оболочка предложит указать значение обязательного параметра `-message`:

```
PS C:\Users\andrv> Print-Message

Командлет Print-Message в конвейере команд в позиции 1
Укажите значения для следующих параметров:
(Для получения справки введите "!?")
message:
```

Если вместо значения `-message` ввести символы `!?`, то будет отображено описание этого параметра:

```
PS C:\Users\andrv> Print-Message

Командлет Print-Message в конвейере команд в позиции 1
Укажите значения для следующих параметров:
(Для получения справки введите "!?")
message: !?
Это очень важный параметр!
message:
```

Также с помощью дополнительных атрибутов можно задавать правила валидации (проверки на корректность) значений параметров. Некоторые из этих атрибутов приведены в табл. 9.2.

Таблица 9.2. *Атрибуты для валидации параметров*

Атрибут	Описание
`AllowNull()`	Значением параметра может быть `Null` (имеет смысл только для обязательных параметров)
`AllowEmptyString()`	Значением параметра может быть пустая строка (имеет смысл только для обязательных параметров)
`ValidateNotNull()`	Значением необязательного параметра не может быть `Null`
`ValidateCount(min, max)`	Проверка на минимальное и максимальное количество элементов, которые могут быть переданы в параметр-массив
`ValidateLength(min, max)`	Проверка на допустимую длину строки, которая передается в качестве значения символьного параметра
`ValidateRange(min, max)`	Минимальное и максимальное значения для числового параметра
`ValidateSet(множество)`	Множество допустимых значений для параметра
`ValidatePattern(шаблон)`	Значение параметра должно соответствовать регулярному выражению *шаблон*
`ValidateScript({блок кода})`	Для валидации параметра вызывается указанный блок кода, в котором текущее значение параметра доступно через переменную `$_`. Если этот блок возвращает `True`, то значение параметра считается корректным

Атрибуты валидации очень полезны при написании функций и сценариев, они позволяют не писать шаблонный повторяющийся код для проверки входных аргументов. Например, чтобы ограничить количество выводящихся в функции `Print-Message` сообщений, можно добавить в описание параметра `-repeat` атрибут `ValidateRange()`:

```
function Print-Message (
    [parameter(Mandatory=$true, HelpMessage='Это очень важный
параметр!')][string]$message,
    [ValidateRange(1, 3)][int]$repeat = 1
)
{
    for ($i = 1; $i -le $repeat; $i++) {
        Write-Host($message)
    }
}
```

Теперь при попытке передать в параметр `-repeat` число, большее трех, возникнет ошибка:

```
PS C:\Users\andrv> Print-Message -message 'Привет всем!' -repeat 5
Print-Message : Не удается проверить аргумент для параметра "repeat".
Аргумент 5 больше максимально допустимого значения 3.
Укажите аргумент, значение которого меньше или равно 3, и повторите команду.
строка:1 знак:42
+ printmsg -message 'Привет всем!' -repeat 5
+                                          ~
  + CategoryInfo          : InvalidData: (:) [printmsg],
                                             ParameterBindingValidationException
  + FullyQualifiedErrorId : ParameterArgumentValidationError,printmsg
```

Дополнительную информацию об атрибутах параметров можно найти в разделе about_functions_advanced_parameters справочной системы PowerShell.

Параметры-переключатели

До настоящего момента мы рассматривали фактические параметры функций двух типов: именованные (указывается имя формального параметра и значение, которое должен принять данный параметр) и позиционные (указывается только значение аргумента; соответствующий формальный параметр определяется по позиции данного аргумента в командной строке).

Вспомним, что в командлетах PowerShell поддерживаются аргументы третьего типа — параметры-переключатели, которые задаются только своим именем. Таким параметром является, например, переключатель `-Recurse` в командлете `Get-ChildItem`. Если этот переключатель указан, то `Get-ChildItem` действует не только на текущий каталог, но и на все его подкаталоги.

В функциях также можно использовать подобные параметры-переключатели, которые должны иметь тип `SwitchParameter`, имеющий псевдоним `[switch]`. Значениями

переключателя могут быть только `$true` или `$false`, поэтому инициализировать подобный формальный параметр не нужно.

Для примера создадим функцию `MyFunc` с одним параметром-переключателем, определяющим поведение функции:

```
PS C:\> function MyFunc ([switch] $recurse) {
>> if ($recurse) {
>>   "Рекурсивный вариант функции"
>>   }
>> else {  "Обычный вариант функции"
>>   }
>> }
>>
```

Запустив функцию `MyFunc` без параметра, мы получим сообщение, что она работает в обычном режиме:

```
PS C:\> MyFunc
Обычный вариант функции
```

Если указан параметр `-recurse`, то функция `MyFunc` будет работать в рекурсивном режиме:

```
PS C:\> MyFunc -recurse
Рекурсивный вариант функции
```

> ***Замечание***
>
> Не рекомендуется создавать функции, сценарии или командлеты, в которых параметр-переключатель по умолчанию инициализировался бы значением `$true`, т. к. это сильно усложняет логику синтаксиса.

Описание параметров в операторе *Param()*

Все описания параметров, которые мы до этого писали в круглых скобках после имени функции, можно поместить внутрь кода самой функции. Для этого первым исполняемым оператором в блоке кода функции следует указать оператор `Param()` и перечислить через запятую внутри него все формальные параметры функции с указанием типов, дополнительных атрибутов и значений по умолчанию (формат тот же, что мы использовали ранее).

Например, рассмотренная ранее функция `Print-Message` с использованием оператора `Param()` запишется следующим образом:

```
function Print-Message ()
{
    param(
        [parameter(Mandatory=$true, HelpMessage='Это очень важный
                                                параметр!')][string]$message,
        [ValidateRange(1, 3)][int]$repeat = 1
    )
```

```
    for ($i = 1; $i -le $repeat; $i++) {
        Write-Host($message)
    }
}
```

Передача параметров с помощью сплаттинга переменных

PowerShell поддерживает особый способ передачи значений в аргументы командлета или функции с помощью предварительно подготовленной переменной. Эта техника называется *сплаттингом переменной* (variable splatting) .

Первый вариант сплаттинга позволяет с помощью переменной задать значения позиционных параметров. Для примера создадим функцию `Print-Args`, выводящую значения трех своих аргументов:

```
PS C:\Users\andrv> function Print-Args {param ($x, $y, $z) "x=$x, y=$y, z=$z" }
```

Выполним `Print-Args`, передав в качестве аргументов три числа:

```
PS C:\Users\andrv> Print-Args 1 2 3
x=1, y=2, z=3
```

Создадим теперь массив `$a` с теми же числами:

```
PS C:\Users\andrv> $a = 1, 2, 3
```

Передадим переменную `$a` в функцию `Print-Args`:

```
PS C:\Users\andrv> Print-Args $a
x=1 2 3, y=, z=
```

Как видим, внутри функции `Print-Args` все три элемента массива `$a` присвоились параметру `$x`. То есть при обычной передаче переменной в функцию ее значение рассматривается как один параметр функции, вне зависимости от внутренней структуры данной переменной.

Выполним теперь сплаттинг переменной `$a`, указав при вызове функции `Print-Agrs` перед именем переменной вместо символа `$` символ `@`:

```
PS C:\Users\andrv> Print-Args @a
x=1, y=2, z=3
```

В этом случае элементы массива `$a` соответствуют позиционным параметрам функции: `$x = $a[0]`, `$y = $a[1]`, `$z = $a[2]`.

Второй вариант сплаттинга связан с передачей значений именованным аргументам функции с помощью хэш-таблицы, ключи в которой совпадают с именами аргументов. Определим для нашего примера хэш-таблицу `$b`:

```
PS C:\Users\andrv> $b = @{x=4; y=5; z=6}
```

Вызовем функцию `Print-Args` с результатом сплаттинга переменной `$b`:

```
PS C:\Users\andrv> Print-Args @b
x=4, y=5, z=6
```

Как видим, значения их хэш-таблицы `$b` попали в соответствующие параметры функции: `$x = $b['x']`, `$y = $b['y']`, `$z = $b['z']`.

Возвращаемые значения

В традиционных языках программирования функция обычно возвращает единственное значение определенного типа. В оболочке PowerShell дело обстоит иначе — здесь результаты всех выражений или конвейеров, вычисляемые внутри функции, направляются в выходной поток. Рассмотрим, например, функцию `MyFunc`, в которой вычисляются три числовых выражения:

```
PS C:\> function MyFunc {1+2; 3*3; 12/4}
```

Как видим, в этой функции явно не возвращается ни одно значение. Запустим функцию `MyFunc`:

```
PS C:\> MyFunc
3
9
3
```

На экран выводятся три строки с результатами вычислений. Теперь запустим данную функцию и сохраним результат ее работы в переменной `$result`:

```
PS C:\> $result = MyFunc
```

Проверим тип переменной `$result`:

```
PS C:\> $result.getType().fullName
System.Object[]
```

Переменная `$result` является массивом объектов. Найдем длину этого массива и значения его элементов:

```
PS C:\> $result.Length
3
PS C:\> $result[0]
3
PS C:\> $result[1]
9
PS C:\> $result[2]
3
```

Итак, массив `$result` содержит все значения, которые попадали в стандартный выходной поток и выводились на экран во время работы функции `MyFunc`. Этот факт нужно учитывать при написании и отладке функций.

Например, определим функцию `MyFunc` следующего вида:

```
PS C:\> function MyFunc {
>> $name = Read-Host "Введите свое имя"
>> "Здравствуйте, $name!"
```

```
>> $name
>> }
>>
```

В данной функции запрашивается имя пользователя (ввод с клавиатуры осуществляется в переменную `$name` с помощью командлета `Read-Host`), выводится приветствие для этого пользователя и значение переменной `$name` выводится в выходной поток. Запустим функцию `MyFunc`:

```
PS C:\> MyFunc
Введите свое имя: Андрей
Здравствуйте, Андрей!
Андрей
```

Проверим теперь, какие данные возвращает функция:

```
PS C:\> $result = MyFunc
Введите свое имя: Андрей
PS C:\> $result.length
2
PS C:\> $result
Здравствуйте, Андрей!
Андрей
```

Итак, кроме значения переменной `$name` в выходной поток попала и строка с приветствием. Если же вместо обычной строки в двойных кавычках для вывода символьной строки воспользоваться командлетом `Write-Host`, то такая строка не будет добавляться в выходной поток:

```
PS C:\> function MyFunc {
>> $name = Read-Host "Введите свое имя"
>> Write-Host "Здравствуйте, $name!"
>> $name
>> }
>>
PS C:\> $result = MyFunc
Введите свое имя: Андрей
Здравствуйте, Андрей!
PS C:\> $result.length
6
PS C:\> $result
Андрей
```

Командлет `Write-Host` выводит строку непосредственно на экран, а не в выходной поток, поэтому в данном случае функция `MyFunc` возвращает единственную строку — значение переменной `$name`.

В языке PowerShell имеется инструкция `Return`, выполняющая немедленный выход из функции (аналог рассмотренной в *главе 8* инструкции `Break` для выхода из цикла). Например:

```
PS C:\> function MyFunc {
>> "Эта строка выводится на экран"
>> return
>> "Эта строка никогда не будет напечатана"
>> }
>>
PS C:\> MyFunc
Эта строка выводится на экран
PS C:\>
```

С помощью данной инструкции можно "возвратить" значение из функции, указав нужный объект после слова `return`. Например, `return 10*3` или `return "Возвращаемая строка"`.

> ***Замечание***
>
> Инструкция `return` включена в язык PowerShell скорее как дань традиции, т. к. аналогичные операторы есть практически во всех языках программирования. Однако всегда следует помнить, что PowerShell является оболочкой и имеет смысл говорить не о единственном значении, возвращаемом функцией, а о выходном потоке, в который функция помещает результаты вычисления выражений.

Функции внутри конвейера команд

Вся идеология PowerShell построена на применении конвейеров команд, и функции здесь не являются исключением: их тоже можно использовать внутри конвейеров. При этом для приема внутри функции входящего потока объектов, передаваемых от другой команды, служит переменная `$input`. В этой переменной будет содержаться коллекция входящих объектов.

Рассмотрим пример:

```
PS C:\> function sum {
>> $n=0
>> foreach ($i in $input) { $n+=$I }
>> $n
>> }
>>
PS C:\>
```

Здесь мы определили функцию `sum`, которая будет суммировать элементы коллекции, поступившие к ней по конвейеру. Результат суммирования помещается в выходной поток как значение переменной `$n`. Передав функции `sum` по конвейеру массив чисел, на выходе мы получим их сумму:

```
PS C:\> 1..10 | sum
55
```

Перебирать элементы входного потока можно не только с помощью цикла `Foreach`, но и путем вызова метода `MoveNext()`, при этом обратиться к текущему элементу

входного потока можно с помощью свойства `Current`. Функцию суммирования элементов входного потока можно переписать следующим образом:

```
PS C:\> function sum2 {
>> $n=0
>> while ($input.moveNext()) {
>>   $n+=$input.current
>>   }
>> $n
>> }
>>
```

Запустим функцию `sum2` и убедимся, что она выдает тот же результат, что и функция `sum`:

```
PS C:\> 1..10 | sum2
55
```

Таким образом, написать функцию для работы внутри конвейера нетрудно (достаточно знать о назначении переменной `$input`), однако при получении данных от предыдущей команды функция приостанавливает конвейер и запускается только один раз, когда сформирована вся коллекция входных элементов.

Функции в качестве командлетов. Расширенные функции

Функции, с которыми мы работали в предыдущем разделе, не поддерживали в полной мере механизм конвейеризации, в отличие от компилированных командлетов, которые обрабатывают поступающий по конвейеру элемент, не дожидаясь генерации следующих элементов. Кроме того, командлеты имеют встроенную помощь и поддерживают общие параметры.

Аналогичное поведение можно реализовать и для функций PowerShell.

Три фазы работы функции в конвейере

При разработке компилированных командлетов PowerShell можно задавать три типа действий:

1. Выполняемые до начала обработки первого входного элемента.
2. Выполняемые для каждого объекта входного потока.
3. Выполняемые после обработки последнего элемента, поступившего по конвейеру.

В PowerShell имеется возможность реализовать подобное поведение для пользовательских функций, имеющих три раздела: `BEGIN`, `PROCESS` и `END`. В этих функциях можно инициализировать некоторое состояние перед началом работы с поступающими по конвейеру элементами, обработать каждый поступающий объект (не дожидаясь при этом формирования следующих элементов) и выполнить определенные завершающие действия после обработки последнего элемента конвейера. Общий синтаксис подобных функций имеет следующий вид:

```
function имя_функции ( параметры ) {
  BEGIN {
    блок_кода_инициализация
  }
  PROCESS {
    блок_кода_обработка_элемента
  }
  END {
    блок_кода_завершение
  }
}
```

Ключевое слово BEGIN определяет секцию, команды из которой будут выполнены перед началом обработки первого объекта конвейера.

Команды и выражения из секции PROCESS будут выполняться каждый раз при получении по конвейеру нового объекта (доступ к текущему элементу в данной секции осуществляется через переменную $_).

В секции END находится код, который выполнится после обработки последнего объекта в конвейере.

Рассмотрим пример. Определим функцию MyCmdlet со всеми тремя секциями:

```
PS C:\> function MyCmdlet ($a) {
>> BEGIN {
>>   $n = 0; "Инициализация: n=$n, a=$a"}
>> PROCESS {
>>   $n++
>>   "Обработка конвейера: n=$n, a=$a, текущий объект = $_" }
>> END {"Завершение: n=$n, a=$a"}
>> }
```

В каждой из секций выводится информация о значении двух переменных: формальный параметр $a соответствует передаваемому в функцию аргументу, переменная $n определяется в секции инициализации и последовательно увеличивается в секции обработки. Кроме этого, в секции обработки выводится значение текущего объекта, поступившего по конвейеру (переменная $_).

Запустив данную функцию с аргументом (число 4), передадим ей по конвейеру числа от 5 до 8:

```
PS C:\> 5..8 | MyCmdlet 4
Инициализация: n=0, a=4
Обработка конвейера: n=1, a=4, текущий объект = 5
Обработка конвейера: n=2, a=4, текущий объект = 6
Обработка конвейера: n=3, a=4, текущий объект = 7
Обработка конвейера: n=4, a=4, текущий объект = 8
Завершение: n=4, a=4
```

Как видим, аргумент функции (число 4) и переменная $n, созданная в секции инициализации, доступны во всех трех секциях.

Запустим теперь функцию MyCmdlet без конвейера:

```
PS C:\> MyCmdlet 4
Инициализация: n=0, a=4
Обработка конвейера: n=1, a=4, текущий объект =
Завершение: n=1, a=4
```

Отсюда делаем вывод, что команды из секции PROCESS запускаются один раз даже при отсутствии конвейера.

Доступ к общим параметрам и дополнительным потокам. Расширенные функции

Напомним, что командлеты в PowerShell поддерживают ряд общих параметров (-Verbose, -Debug, -WhatIf и т. д.). Эти параметры можно обрабатывать и внутри так называемых *расширенных функций* (advanced functions), которые внутри своего тела перед оператором Param() содержат некоторые атрибуты (метаданные), влияющие на поведение функции.

Для включения поддержки в своей функции общих параметров нужно добавить в нее атрибут [CmdletBinding()], который делает поведение функции похожим на командлет.

Рассмотрим на простом примере, как влияет этот атрибут на поведение функции. Начнем с функции Sum-TwoArgs, которая суммирует значение двух своих аргументов:

```
PS C:\Users\andrv> function Sum-TwoArgs {
>>     param ($a, $b)
>>     $a + $b
>> }
```

Проверим синтаксис этой функции с помощью параметра -Syntax команды Get-Command:

```
PS C:\Users\andrv> Get-Command Sum-TwoArgs -Syntax

Sum-TwoArgs [[-a] <Object>] [[-b] <Object>]
```

Как видим, Sum-TwoArgs не поддерживает общие параметры командлетов.

Вызовем нашу функцию с тремя параметрами:

```
PS C:\Users\andrv> Sum-TwoArgs 1 2 3
3
```

В этом случае ошибки не произошло, была вычислена сумма первых двух аргументов, а третий аргумент был помещен в переменную $args.

Добавим теперь к функции Sum-TwoArgs перед оператором Param() атрибут [CmdletBinding()]:

```
PS C:\Users\andrv> function Sum-TwoArgs {
>>     [CmdletBinding()]
```

```
>>     param ($a, $b)
>>     $a + $b
>> }
```

Вновь проверим синтаксис:

```
PS C:\Users\andrv> Get-Command Sum-TwoArgs -Syntax

Sum-TwoArgs [[-a] <Object>] [[-b] <Object>] [<CommonParameters>]
```

Как видим, теперь функция `Sum-TwoArgs` поддерживает общие параметры.

Вновь вызовем функцию с тремя аргументами:

```
PS C:\Users\andrv> Sum-TwoArgs 1 2 3
Sum-TwoArgs : Не удается найти позиционный параметр, принимающий аргумент "3".
строка:1 знак:1
+ Sum-TwoArgs 1 2 3
+ ~~~~~~~~~~~~~~~~
  + CategoryInfo          : InvalidArgument: (:) [Sum-TwoArgs],
                                                     ParameterBindingException
  + FullyQualifiedErrorId : PositionalParameterNotFound,Sum-TwoArgs
```

Теперь мы получили сообщение об ошибке, т. к. при вызове командлетов количество переданных аргументов не должно превышать количество формальных параметров. Таким образом, теперь наша функция ведет себя как командлет.

В функции, для которой указан атрибут `[CmdletBinding()]`, с помощью команд `Write-*` можно записывать сообщения в дополнительные потоки, например в `Verbose` или `Debug`. Например, выведем подробное сообщение о выполняемой операции в поток `Verbose`:

```
PS C:\Users\andrv> function Sum-TwoArgs {
>>     [CmdletBinding()]
>>     param (
>>         $a, $b
>>     )
>>     Write-Verbose "Функция вычисляет сумму двух аргументов: a=$a и b=$b"
>>     $a + $b
>> }
```

При обычном запуске функции `Sum-TwoArgs` сообщения на экране мы не увидим:

```
PS C:\Users\andrv> Sum-TwoArgs 1 2
3
```

Выполним теперь функцию с общим параметром `-Verbose`. В этом случае содержимое канала `Verbose` будет отображено на экране.

```
PS C:\Users\andrv> Sum-TwoArgs 1 2 -Verbose
ПОДРОБНО: Функция вычисляет сумму двух аргументов: a=1 и b=2
3
```

Сценарии PowerShell

До настоящего момента мы работали в оболочке PowerShell интерактивно, вводя команды и получая результат их выполнения. Если завтра потребуется выполнить те же команды, что и сегодня, нам придется вводить их заново. Поэтому часто используемые последовательности команд лучше сохранять во внешних сценариях, которые выполняются в пакетном режиме, т. е. без участия человека.

Сценарии PowerShell представляют собой текстовые файлы с расширением ps1, в которых записан код (команды, операторы и другие конструкции) на языке PowerShell. Сценарии PowerShell можно писать поэтапно, непосредственно в самой оболочке, перенося затем готовый код во внешний текстовый файл. Такой подход значительно упрощает изучение языка и отладку сценариев, позволяя сразу видеть результат выполнения отдельных частей сценария.

Создание сценария

Создать файл со сценарием PowerShell можно разными способами.

- Воспользоваться внешним текстовым редактором (например, стандартным Блокнотом Windows или устанавливаемым отдельно Visual Studio Code), вручную ввести нужные команды и сохранить файл с расширением ps1.
- Выполнить нужные команды в оболочке PowerShell, затем скопировать их с консоли в буфер Windows и вставить в текстовый файл, открытый во внешнем текстовом редакторе. Полученный в результате файл сохранить с расширением ps1.
- Работая в оболочке PowerShell, включить с помощью командлета `Start-Transcript` режим протоколирования команд, описанный в *главе 4*. В результате будет создан внешний файл со всеми выполненными в сеансе работы командами. Из этого файла можно скопировать нужные команды в другой текстовый файл и сохранить его с расширением ps1.
- Находясь в оболочке PowerShell, оформить команды PowerShell в виде строк (обычных или типа here-string) и перенаправить с помощью символов > и >> эти строки во внешний файл с расширением ps1.

Воспользуемся последним из предложенных вариантов и создадим в каталоге script простейший сценарий test.ps1, который будет состоять из единственной строки:

```
"Эта строка печатается из сценария PowerShell"
```

Для начала создадим каталог script и сделаем его текущим:

```
PS C:\Users\andrv> md script

    Каталог: C:\Users\andrv

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        31.05.2021      7:04                script
```

```
PS C:\Users\andrv> cd .\script\
PS C:\Users\andrv\script>
```

Теперь запишем нужную строку (включая двойные кавычки) в файл test.ps1:

```
PS C:\Script> '"Эта строка печатается из сценария PowerShell"' > test.ps1
```

Запуск сценария из PowerShell

Попробуем теперь запустить наш сценарий. Напомним, что сценарии выполняются системой только в том случае, когда это разрешено текущей политикой выполнения (мы рассматривали данный вопрос в *главе 4*, когда обсуждали загрузку пользовательских профилей, которые также являются сценариями PowerShell). По умолчанию на клиентских машинах действует политика Restricted, полностью запрещающая выполнение сценариев PowerShell. Это сделано из соображений безопасности, т. к. в сценариях может содержаться вредоносный код, который может повредить систему или несанкционированно обратиться к пользовательским данным.

Проверим активную политику выполнения с помощью командлета `Get-ExecutionPolicy`:

```
PS C:\Users\andrv\script> Get-ExecutionPolicy
RemoteSigned
```

Если в вашей системе действует более строгая политика безопасности (Restricted или AllSigned), то установите политику RemoteSigned, позволяющую выполнять неподписанные локальные сценарии:

```
PS C:\Users\andrv\script> Set-ExecutionPolicy RemoteSigned
```

Для выполнения данной команды, разрешающей локальные сценарии на машине в постоянном режиме, нужно запустить PowerShell с повышенными привилегиями, т. е. от имени администратора. Если у вас нет прав администратора, то можно разрешить выполнение сценариев только для своего пользователя или только для текущего сеанса PowerShell, указав параметр `-Scope CurrentUser` или `-Scope Process` соответственно:

```
PS C:\Users\andrv\script> Set-ExecutionPolicy -Scope Process RemoteSigned
```

Итак, мы разрешили выполнение сценариев. Вновь попробуем запустить наш скрипт, указав его имя:

```
PS C:\Users\andrv\script> test.ps1
test.ps1 : Имя "test.ps1" не распознано как имя командлета, функции, файла
сценария или выполняемой программы. Проверьте правильность написания имени,
а также наличие и правильность пути, после чего повторите попытку.
строка:1 знак:1
+ test.ps1
+ ~~~~~~~~
  + CategoryInfo          : ObjectNotFound: (test.ps1:String) [],
                                                    CommandNotFoundException
  + FullyQualifiedErrorId : CommandNotFoundException
```

```
Suggestion [3,General]: Команда test.ps1 не найдена, однако существует
в текущем расположении. По умолчанию оболочка Windows PowerShell не загружает
команды из текущего расположения. Если вы уверены в надежности команды, введите
".\test.ps1". Для получения дополнительных сведений вызовите справку с помощью
команды "get-help about_Command_Precedence".
```

Опять ошибка... Дело в том, что при запуске сценариев PowerShell путь к файлу с кодом нужно всегда указывать явно, даже если сценарий находится в текущем каталоге, т. к. это предотвращает возможный несанкционированный запуск другой исполняемой программы с аналогичным именем, находящейся, например, в системном каталоге. Поэтому запустим сценарий, явно указав путь к нему (точка в пути к файлу соответствует текущему каталогу):

```
PS C:\Users\andrv\script> .\test.ps1
Эта строка печатается из сценария PowerShell
```

Итак, в этом случае сценарий выполняется. Можно даже не указывать расширение ps1:

```
PS C:\Script> .\test
Эта строка печатается из сценария PowerShell
```

Естественно, путь можно было задать полностью:

```
PS C:\Script> C:\Users\andrv\script\test.ps1
Эта строка печатается из сценария PowerShell
PS C:\Script> C:\Users\andrv\script\test
Эта строка печатается из сценария PowerShell
```

Если в пути к сценарию содержатся пробелы, то имя нужно заключить в кавычки, и поставить в начале знак амперсанда (&), означающий в PowerShell оператор запуска. Например:

```
PS C:\Script> &'C:\Мои скрипты\script.ps1'
```

Запуск сценария из внешней программы

Сценарии PowerShell можно запускать непосредственно из командной строки интерпретатора cmd.exe или с помощью пункта **Выполнить** меню **Пуск**. Для этого нужно указать полный путь к этому сценарию в качестве значения параметра `-File` программы powershell.exe. При этом полный путь к powershell.exe и расширение exe можно не указывать, например:

```
C:\> powershell -File C:\Users\andrv\script\test.ps1
```

Напомним, что командные файлы интерпретатора cmd.exe, равно как и сценарии WSH на языках VBScript или JScript, рассматриваются системой Windows как исполняемые, их можно запускать из Проводника, просто щелкая мышью на значках этих сценариев. Со сценариями PowerShell этот метод не работает — если дважды щелкнуть мышью на значке сценария PowerShell, то он не запустится, а откроется для редактирования в Блокноте (с точки зрения безопасности это правильный подход, позволяющий предотвратить случайный запуск сценария).

Передача аргументов в сценарии

Фактически сценарий PowerShell — это функция, которая находится не в оперативной памяти, а на внешнем носителе. Поэтому разбор и обработка аргументов, передаваемых в сценарии, производятся практически так же, как и в функциях.

Аргументы указываются после имени сценария и разделяются между собой пробелами. Переменная `$args` внутри сценария содержит массив, элементами которого являются аргументы функции, указанные при ее запуске. Для примера напишем сценарий SumArgs.ps1, который будет сообщать количество параметров, с которыми он запущен, и их сумму. Файл с исходным кодом создадим следующим образом: текст сценария оформим в виде строки типа here-string (подобные строки были описаны в *главе 7*) и перенаправим эту строку в файл SumArgs.ps1:

```
PS C:\Script> @'
>> "Количество аргументов: $($args.count)"
>> $n=0
>> for($i=0; $i -lt $args.count; $i++) { $n+=$args[$i] }
>> "Сумма аргументов: $n"
>> '@ > SumArgs.ps1
>>
```

Запустим теперь сценарий SumArgs.ps1 с несколькими числовыми параметрами:

```
PS C:\Script> .\SumArgs 1 2 3
Количество аргументов: 3
Сумма аргументов: 6
```

Как видим, массив `$args` в сценариях имеет тот же смысл и обрабатывается точно так же, как в функциях.

В сценариях можно определять формальные параметры, вместо которых во время выполнения будут подставляться фактические аргументы, переданные в сценарий. Напомним, что в функциях мы могли перечислить формальные параметры либо в круглых скобках после имени, т. е. вне тела функции, либо внутри функции с помощью оператора `Param()`.

В сценариях описать параметры можно только с помощью `Param()`, т. к. здесь все содержимое файла является телом сценария. Этот оператор должен быть первой исполняемой командой в файле, предшествовать ему могут только пустые строки, комментарии и дополнительные атрибуты в случае расширенных сценариев.

Для примера напишем сценарий add.ps1 с двумя формальными параметрами, который будет выводить сумму своих аргументов. Для создания сценария вновь воспользуемся строкой типа here-string, которую перенаправим в файл:

```
PS C:\Script> @'
>> param ($x=2, $y=3)
>> $x+$y
>> '@ > add.ps1
>>
```

Запустим полученный сценарий с аргументами и без них:

```
PS C:\Script> .\add 10 20
30
PS C:\Script> .\add
5
```

Все работает правильно: если аргументы не указаны, то внутри сценария используются значения по умолчанию.

Выход из сценариев. Код возврата

В обычном режиме выход из сценария, как и из функции, происходит после выполнения последней инструкции в нем. Напомним, что в функции можно было воспользоваться инструкцией `Return` для принудительного завершения работы. Аналог этой инструкции для сценариев — инструкция `Exit`, позволяющая завершить работу сценария и при необходимости вернуть определенный код возврата. Код возврата может анализироваться во внешних программах (например, командных файлах или сценариях WSH), запускающих сценарий PowerShell.

Рассмотрим пример. Создадим командный файл Call_PS.bat, в котором будет вызываться сценарий PowerShell, заданный строкой `"'PoSH-script is working'; exit 10"`, и выводиться значение переменной среды `ERRORLEVEL`, которое равно коду возврата последнего выполнявшегося процесса (листинг 9.1).

Листинг 9.1. Определение кода завершения сценария PowerShell (файл Call_PS.bat)

```
@echo off
echo Запускаем сценарий PowerShell...
powershell "'PoSH-script is working'; exit 10"
echo Код возврата процесса PowerShell: %ERRORLEVEL%
```

Запустив файл Call_PS.bat в оболочке cmd.exe, мы получим следующий результат:

```
C:\Script>call_ps.bat
Запускаем сценарий PowerShell...
PoSH-script is working
Код возврата процесса PowerShell: 10
```

Области видимости функций

Список всех функций, доступных из оболочки PowerShell, хранится на виртуальном диске `Function:`, посмотреть его содержимое можно с помощью команды `Get-ChildItem` (псевдоним `dir`):

```
PS C:\Users\andrv> dir function:

CommandType     Name
-----------      ----
Function        A:
Function        B:
```

```
Function        C:
Function        cd..
Function        cd\
Function        Clear-Host
. . .
```

Если мы вручную ввели функцию в окне PowerShell (или скопировали ее из буфера обмена), то она будет доступна только в текущем сеансе оболочки. В другом окне PowerShell мы этой функцией воспользоваться не сможем.

Функции, которые описаны во внешнем файле сценария, будут доступны только в этом файле. Добиться того, чтобы функции, объявленные в сценарии, остались в памяти после выполнения этого сценария, можно двумя способами.

Глобальная область видимости

Для того чтобы сделать функцию в сценарии постоянно доступной, нужно указать, что она создается в глобальной области видимости, поставив перед ее именем ключевое слово (спецификатор) `global:`.

Например, сохраним в сценарии test.ps1 функцию `MyFunc` со спецификатором `global:`.

```
PS C:\Users\andrv> "function global:MyFunc() { 'Это функция MyFunc' }"
> test.ps1
```

Выполним сценарий test.ps1:

```
PS C:\Users\andrv> .\test.ps1
```

После этого функцию `MyFunc` можно вызывать из оболочки или из другого сценария:

```
PS C:\Users\andrv> MyFunc
Это функция MyFunc
```

Таким образом, если поместить функцию со спецификатором `global:` в пользовательский профиль (настройка профиля обсуждалась в *главе 4*), то она будет загружаться при каждом запуске PowerShell и мы всегда сможем вызвать ее из командной строки.

Оператор *Dot-Source*

Если запустить сценарий PowerShell в так называемом режиме *dot-sourcing*, когда перед именем или путем к сценарию ставится точка и пробел, то его содержимое будет выполнено, как будто оно было набрано непосредственно в командной строке. Другими словами, сценарий запускается в области видимости окна командной строки, поэтому все функции и переменные, имеющиеся в этом сценарии, приобретают глобальную область видимости и будут доступны в оболочке после завершения работы сценария.

Для проверки данного режима добавим в сценарий test.ps1 новую функцию `LocalFunc` с областью видимости по умолчанию:

```
PS C:\Users\andrv> "function LocalFunc() { 'Это функция LocalFunc' }"
>> test.ps1
```

Выполним сценарий test.ps1 в режиме dot-sourcing и убедимся в доступности функции `LocalFunc` в оболочке:

```
PS C:\Users\andrv> . .\test.ps1
PS C:\Users\andrv> LocalFunc
Это функция LocalFunc
```

Подобным образом можно применить изменения в своем профиле, сделанные во время текущего сеанса работы, без перезагрузки оболочки. Путь к профилю хранится в переменной `$PROFILE`, поэтому для этого достаточно выполнить следующую команду:

```
PS C:\Users\andrv> . $PROFILE
```

В режиме dot-sourcing можно запускать не только сценарии, но и функции. Например:

```
PS C:\Users\andrv> function set-var ($x) {$x = $x}
PS C:\Users\andrv> . set-var 5
PS C:\Users\andrv> $x
5
```

Если некоторые переменные и функции из сценария должны остаться локальными даже в режиме dot-sourcing, то перед их именами нужно указать спецификаторы `local:` или `script:`.

Области видимости переменных

В PowerShell по умолчанию любая переменная является локальной относительно файла сценария, функции или блока кода, в которых эти переменные инициализируются. Также переменная будет доступна во всех программных модулях (сценариях, функциях, блоках кода), которые вызываются из данного сценария, функции или блока кода.

Например, выполним с помощью оператора вызова `&` блок кода, в котором переменной `$a` присваивается значение 1:

```
PS C:\Users\andrv> &{ $a = 1 }
```

После этого оболочка не будет знать о существовании переменной `$a`. Для проверки этого напишем функцию `Print-Var` для вывода значения переменной `$a` и выполним ее:

```
PS C:\Users\andrv> function Print-Var() { "`$a = $a" }
PS C:\Users\andrv> Print-Var
$a =
```

Как видим, в оболочке значение переменной `$a` не определено.

Теперь вызовем `Print-Var` из блока кода, в котором инициализируется переменная `$a`:

```
PS C:\Users\andrv> &{ $a = 1; Print-Var }
$a = 1
```

Как видим, переменная `$a` становится доступной внутри функции `Print-Var`, если функция вызывается на том же уровне вложенности кода.

Теперь попробуем внутри функции `Print-Var` присвоить переменной `$a` новое значение. Для этого изменим `Print-Var`:

```
PS C:\Users\andrv> function Print-Var() { "`$a = $a"; $a = 2; "`$a = $a" }
```

Выполним следующий блок кода:

```
PS C:\Users\andrv> &{ $a = 1
>> "До функции Print-Var `$a = $a"
>>  Print-Var
>> "После функции Print-Var `$a= $a" }
До функции Print-Var $a = 1
$a = 1
$a = 2
После функции Print-Var $a= 1
```

Как видим, изменения переменной `$a` внутри функции `Print-Var` не повлияли на ее значение во внешнем блоке кода. Дело в том, что PowerShell при обращении к переменной, определенной в родительском пространстве имен, по умолчанию создает новую локальную переменную с таким же именем. Таким образом, вложенный код сценария или функции работает с копией переменной из родительского пространства.

Области видимости переменных можно изменять, указывая перед их именами спецификаторы `script:` (переменная будет доступна в любом месте сценария) и `global:` (к переменной можно будет обращаться из любой функции и сценария, а также из командной строки). Например:

```
PS C:\Users\andrv> function Print-Var()
>> { "`$a = $global:a"; $global:a = 2; "`$a = $global:a" }
PS C:\Users\andrv> &{ $global:a = 1
>> "До функции Print-Var `$a = $global:a"
>>  Print-Var
>> "После функции Print-Var `$a= $global:a" }
До функции Print-Var $a = 1
$a = 1
$a = 2
После функции Print-Var $a= 1
PS C:\Users\andrv> "В командной строке `$a = $global:a"
В командной строке $a = 2
```

Модули PowerShell

Большой добродетелью у программистов и администраторов считается нежелание выполнять ненужную работу. Не нужно вновь и вновь изобретать колесо — если кто-то уже полностью или частично решил интересующую нас задачу, написал для этого код, то проще и правильнее будет воспользоваться этим решением, а не писать код заново.

Современные поисковые системы в Интернете сильно облегчают жизнь разработчикам — мы быстро можем найти подходящие примеры на разных языках программирования на специальных сайтах. Для сценариев PowerShell существует поддерживаемое компанией Microsoft централизованное хранилище (репозиторий) PowerShell Gallery (**https://www.powershellgallery.com/**), содержащее несколько тысяч пакетов на языке PowerShell, разработанных Microsoft и другими членами сообщества PowerShell. Также тысячи PowerShell-сценариев можно найти на GitHub (официальный репозиторий команды PowerShell расположен по адресу **https://github.com/powershell**).

После того как сценарий найден, его нужно применить для своих целей, и здесь возникает задача простого и удобного повторного использования кода. Для этого в PowerShell принято оформлять код в *модули*, содержащие члены модуля: командлеты, функции, переменные, псевдонимы, которые при необходимости можно загружать в оболочку и пользоваться ими из командной строки или своих сценариев.

Модули в PowerShell помогают решить две основные задачи.

- *Добавление в командную оболочку новой функциональности.* В предыдущих разделах мы рассмотрели процедуру загрузки в оболочку своих функций в режиме dot-sourcing. Модули позволяют сделать это в более удобном и надежном виде.
- *Повторное использование кода.* Модули в PowerShell, как и в других языках программирования, позволяют создавать и распространять библиотеки, функции из которых подключаются и используются в своих сценариях.

Модули-сценарии

В PowerShell используется несколько типов модулей, в том числе *двоичные*, содержащие компилированные командлеты, и *модули-сценарии*, представляющие собой сценарии на языке PowerShell в файлах с расширением psm1 (напомним, что обычные сценарии PowerShell имеют расширение ps1).

Если модуль размещен в подкаталоге одного из каталогов, указанных в переменной среды `PSModulePath`, то он будет автоматически загружаться в оболочку при первом обращении к любой команде из этого модуля:

```
PS C:\Users\andrv> $ENV:PSModulePath -split ';'
C:\Users\andrv\OneDrive\Documents\WindowsPowerShell\Modules
C:\Program Files\WindowsPowerShell\Modules
C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules
```

Таким образом, модули-сценарии позволяют нам написать собственные функции, которые будут работать аналогично встроенным командлетам, автоматически загружаясь в оболочку при каждом запуске PowerShell.

Для примера запишем две функции, `Out-Hello` и `Out-GoodBye`, в файл MyModule.psm1. Эти функции будут выводить сообщения с именем текущего пользователя:

```
PS C:\Users\andrv> 'function Out-Hello() { "Привет, $env:UserName!" }' >
MyModule.psm1
PS C:\Users\andrv> 'function Out-GoodBye() { "Пока, $env:UserName!" }' >>
MyModule.psm1
```

Проверим содержимое файла MyModule.psm1:

```
PS C:\Users\andrv> Get-Content .\MyModule.psm1
function Out-Hello() { "Привет, $env:UserName!" }
function Out-GoodBye() { "Пока, $env:UserName!" }
```

В переменную `$moduleDir` сохраним каталог для хранения пользовательских модулей (первый каталог из переменной окружения `PSModulePath`):

```
PS C:\Users\andrv> $moduleDir = ($ENV:PSModulePath -split ';')[0]
```

Создадим в этом каталоге подкаталог MyModule (имя подкаталога должно совпадать с именем файла с модулем):

```
PS C:\Users\andrv> New-Item -Path $moduleDir/MyModule -ItemType Directory

    Каталог: C:\Users\andrv\OneDrive\Documents\WindowsPowerShell\Modules

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        11.06.2021      6:06                MyModule
```

Скопируем в созданный подкаталог файл MyModule.psm1 нашего модуля:

```
PS C:\Users\andrv> Copy-Item -Path .\MyModule.psm1 -Destination
$moduleDir\MyModule
```

Теперь наш модуль стал доступным для загрузки. Проверить это можно с помощью командлета `Get-Module` с параметром `-ListAvailable`:

```
PS C:\Users\andrv> Get-Module -ListAvailable MyModule | Format-List

Name              : MyModule
Path              : C:\Users\andrv\OneDrive\Documents\WindowsPowerShell\
                    Modules\MyModule\MyModule.psm1
Description       :
ModuleType        : Script
Version           : 0.0
NestedModules     : {}
ExportedFunctions : {Out-Hello, Out-GoodBye}
ExportedCmdlets   :
```

```
ExportedVariables :
ExportedAliases   :
```

Как видим, оболочка нашла наш модуль и определила, что из него экспортируются функции `Out-Hello` и `Out-GoodBye`.

> ***Замечание***
>
> **По умолчанию из модуля экспортируются все его члены (функции, переменные, псевдонимы и т. д.). Это поведение можно изменить, явно перечислив экспортируемые члены с помощью командлета `Export-ModuleMember`.**

Загрузим наш модуль с помощью командлета `Import-Module`. Параметр `-Verbose` позволяет увидеть, какие фунции импортируются из модуля:

```
PS C:\Users\andrv> Import-Module MyModule -Verbose
ПОДРОБНО: Импорт функции "Out-GoodBye".
ПОДРОБНО: Импорт функции "Out-Hello".
```

После импорта модуля `MyModule` в оболочке станут доступны функции `Out-Hello` и `Out-GoodBye`:

```
PS C:\Users\andrv> Out-Hello
Привет, andrv!
PS C:\Users\andrv> Out-GoodBye
Пока, andrv!
```

При следующем запуске PowerShell модуль MyModule будет загружен автоматически, объявленные в нем функции останутся доступными.

Репозиторий сценариев PowerShell Gallery

По умолчанию централизованным репозиторием для модулей и сценариев PowerShell является ресурс PowerShell Gallery (**https://www.powershellgallery.com/**), которым управляет Microsoft. Информацию об этом репозитории можно получить с помощью командлета `Get-PSRepository`:

```
PS C:\Users\andrv> Get-PSRepository | Format-List

Name                      : PSGallery
SourceLocation            : https://www.powershellgallery.com/api/v2
Trusted                   : False
Registered                : True
InstallationPolicy        : Untrusted
PackageManagementProvider : NuGet
PublishLocation           : https://www.powershellgallery.com/api/v2/package/
ScriptSourceLocation      : https://www.powershellgallery.com/api/v2/items/
                            psscript
ScriptPublishLocation     : https://www.powershellgallery.com/api/v2/package/
ProviderOptions           : {}
```

Как видим, централизованный репозиторий имеет имя PSGallery.

Работать с репозиторием можно с помощью команд из модуля PowerShellGet:

```
PS C:\Users\andrv> Get-Command -Module PowerShellGet | Format-Wide -Column 2

Find-Command                    Find-DscResource
Find-Module                     Find-RoleCapability
Find-Script                     Get-InstalledModule
Get-InstalledScript             Get-PSRepository
Install-Module                  Install-Script
New-ScriptFileInfo              Publish-Module
Publish-Script                  Register-PSRepository
Save-Module                     Save-Script
Set-PSRepository                Test-ScriptFileInfo
Uninstall-Module                Uninstall-Script
Unregister-PSRepository         Update-Module
Update-ModuleManifest           Update-Script
Update-ScriptFileInfo
```

Поиск модулей выполняется с помощью командлета `Find-Module`. Если выполнить эту команду без параметров, то будет выведен полный список всех имеющихся в PowerShell Gallery модулей:

```
PS C:\Users\andrv> Find-Module -Repository PSGallery | Format-Wide -Column 2

SpeculationControl              NetworkingDsc
PSWindowsUpdate                 AzureRM.profile
PackageManagement               DellBIOSProvider
PowerShellGet                   Az.Accounts
Carbon                          ComputerManagementDsc
Azure.Storage                   Az.Storage
Az.Resources                    Az.KeyVault
Az.Automation                   Az.AnalysisServices
Az.ApiManagement                xPowerShellExecutionPolicy
Az.Compute                      PSLogging
Az.ApplicationInsights          xCertificate
Az.Aks                          Az.Network
Az.Sql                          Az.Batch
Az.DataFactory                  Az.Cdn
Az.ContainerRegistry            Az.DataLakeStore
Az.EventHub                     Az.Websites
Az.ServiceBus                   Az.HDInsight
Az.Monitor                      Az.ContainerInstance
Az.OperationalInsights          Az.DataLakeAnalytics
. . .
```

Если имя нужного модуля известно, то следует указать его в качестве значения параметра `-Name`. Например:

```
PS C:\Users\andrv> Find-Module -Repository PSGallery -Name PSLogging |
Format-List
```

```
Name                      : PSLogging
Version                   : 2.5.2
Type                      : Module
Description               : Creates and manages log files for your scripts.
Author                    : LucaSturlese
CompanyName               : 9to5IT
Copyright                 : (c) 2015 Luca Sturlese. All rights reserved.
PublishedDate             : 11/22/2015 10:26:55 AM
InstalledDate             :
UpdatedDate               :
LicenseUri                : http://9to5it.com/powershell-logging-v2-easily-
                            create-log-files
ProjectUri                : http://9to5it.com/powershell-logging-v2-easily-
create-log-files
IconUri                   :
Tags                      : {Logging, LogFiles, PSModule}
Includes                  : {Function, RoleCapability, Command, DscResource...}
PowerShellGetFormatVersion :
ReleaseNotes              : Removed HelpInfoURI from Module Manifest file as
                            was causing an issue with PowerShell 2.0
Dependencies              : {}
RepositorySourceLocation  : https://www.powershellgallery.com/api/v2
Repository                : PSGallery
PackageManagementProvider : NuGet
```

Устанавливается модуль из репозитория с помощью командлета `Install-Module`. По умолчанию модули устанавливаются в каталог C:\Program Files\WindowsPowerShell\Modules, что требует выполнения команды `Install-Module` в оболочке, запущенной от имени администратора. Для установки модуля в профиль текущего пользователя нужно указать параметр `-Scope` со значением `CurrentUser` и утвердительно ответить на вопрос об установке:

```
PS C:\Users\andrv> Install-Module -Name PSLogging -Scope CurrentUser


Untrusted repository
You are installing the modules from an untrusted repository. If you trust this
repository, change its InstallationPolicy value by running the Set-PSRepository
cmdlet. Are you sure you want to install the modules from 'PSGallery'?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default
is "N"): y
```

Для проверки загруженного модуля можно посмотреть, какие команды в нем содержатся:

```
PS C:\Users\andrv> Get-Command -Module PSLogging

CommandType     Name                                               Version
-----------     ----                                               -------
Function        Send-Log                                           2.5.2
Function        Start-Log                                          2.5.2
```

```
Function        Stop-Log                                    2.5.2
Function        Write-LogError                              2.5.2
Function        Write-LogInfo                               2.5.2
Function        Write-LogWarning                            2.5.2
```

Обновить модуль (т. е. установить его новую версию) можно с помощью командлета `Update-Module`, удалить модуль с локальной машины — с помощью `Uninstall-Module`.

Итоги

- Функции в PowerShell создаются с помощью ключевого слова `function`.
- В отличие от других языков программирования, функция в PowerShell является командой, поэтому вызывать функции нужно как команды — аргументы указываются через пробел после имени функции.
- Самый простой способ получения переданных в функцию аргументов — перебор элементов массива `$args`.
- Для функции в PowerShell можно задать формальные параметры, указав их в круглых скобках после имени функции (как это делается в других языках программирования) или внутри самой функции с помощью специального оператора `Param()`. При этом можно явно указать типы таких параметров, задать для них значения по умолчанию и другие атрибуты для валидации аргументов, передаваемых в функцию при вызове.
- При вызове функции ее формальные параметры заменяются на фактические аргументы, которые определяются по позиции в командной строке или по имени параметра.
- Тело функции может содержать разделы `BEGIN`, `PROCESS` и `END`. В этом случае функции можно корректно использовать внутри конвейеров.
- Если для функции задать атрибут `CmdletBinding()`, то она будет вести себя как командлет. Такие функции называются расширенными, в них становятся доступными общие параметры командлетов и дополнительные выходные потоки `Verbose`, `Debug`, `Error`.
- Каждое вычисляемое в функции выражение помещает свой результат в выходной поток.
- Для управления функциями, доступными в текущем сеансе оболочки PowerShell, используется виртуальный диск `Function:`.
- Сценарий PowerShell — это функция, которая находится не в оперативной памяти, а хранится в текстовом файле с расширением ps1. Определение, разбор и обработка аргументов, передаваемых в сценарии, производятся так же, как и в функциях.
- Режим запуска сценариев в PowerShell определяется политикой выполнения, которую можно изменить с помощью команды `Set-ExecutionPolicy`. Рекоменду-

ется установить политику RemoteSigned, позволяющую выполнять неподписанные локальные сценарии.

- Функции в PowerShell могут иметь разные области видимости. Для того чтобы функции, объявленные в сценарии, оставались доступными в оболочке после выполнения этого сценария, можно создать эти функции в глобальной области видимости или запустить сценарий в режиме dot-sourcing.
- По умолчанию в PowerShell переменная является локальной относительно сценария, функции или блока кода, в которых она была инициализирована. Области видимости переменных можно задавать явно, указывая спецификаторы перед их именами.
- Для упрощения повторного использования кода используются модули PowerShell, содержащие командлеты, функции, переменные и псевдонимы, которые при необходимости можно загружать в оболочку и пользоваться ими из командной строки или сценариев.
- Компания Microsoft поддерживает централизованный репозиторий модулей и сценариев на языке PowerShell — PowerShell Gallery.

ГЛАВА 10

Обработка ошибок при выполнении команд

При применении сценариев для решения реальных администраторских задач (управление учетными записями пользователей, резервное копирование информации на серверах, анализ журналов событий для выявления попыток несанкционированного доступа в систему и т. п.) актуальной становится проблема обработки возможных ошибок или исключительных ситуаций. Нужно быть уверенным, что определенная задача выполнена полностью, а в случае возникновения ошибки проанализировать и устранить ее. Язык PowerShell предлагает несколько механизмов обработки ошибок, которые мы рассмотрим подробнее в данной главе.

Возникновение ошибки при выполнении команды PowerShell приводит не просто к выводу текстовых сообщений на экран, при этом автоматически генерируется настоящий объект, в свойствах которого содержится полная информация о данной ошибке.

Кроме этого, один аспект обработки ошибок в PowerShell отличает эту систему от других языков программирования: ошибки здесь могут быть *критическими* (прерывающими выполнение команды) и *некритическими* (при их возникновении выполнение команды продолжается). Это связано с тем, что для обработки объектов в PowerShell используется модель конвейера команд. При возникновении некритической ошибки информация об ошибке записывается в соответствующий объект, а текущая команда продолжает обрабатывать поступающие к ней по конвейеру объекты. Подобная ошибка может возникнуть, например, при копировании с помощью командлета множества файлов, один из которых занят другим приложением. Если же выполняется какая-то важная операция, то может иметь смысл останавливать ее после возникновения первой ошибки (тогда подобная ошибка будет критической). При этом одна и та же ошибка в одной ситуации должна считаться критической, а в другой — некритической (в PowerShell имеется механизм, позволяющий явно задавать эти типы для ошибок).

При возникновении некритических ошибок информация о них помещается в объекты типа `ErrorRecord`, которые записываются в специальный поток ошибок. Рассмотрим, каким образом можно обращаться к данным объектам и какую информацию можно из них извлечь.

Объект *ErrorRecord* и поток ошибок

В PowerShell информация о возникающих ошибках записывается в поток ошибок, который по умолчанию отображается на экране, например:

```
PS C:\Users\andrv> dir "Несуществующий каталог"
dir : Не удается найти путь "C:\Users\andrv\Несуществующий каталог", т. к.
он не существует.
строка:1 знак:1
+ dir "Несуществующий каталог"
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  + CategoryInfo          : ObjectNotFound: (C:\Users\andrv\Несуществующий
                            каталог:String) [Get-ChildItem],
                            ItemNotFoundException
  + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.
                                                        GetChildItemCommand
```

Поток ошибок в PowerShell имеет внутренний номер 2, этот поток можно перенаправить в текстовый файл с помощью специального оператора `2>`, например:

```
PS C:\Users\andrv> dir "Несуществующий каталог" 2> err.txt
```

Проверим теперь содержимое файла err.txt:

```
PS C:\Users\andrv> Get-Content .\err.txt
dir : Не удается найти путь "C:\Users\andrv\Несуществующий каталог", т. к.
он не существует.
строка:1 знак:1
+ dir "Несуществующий каталог" 2> err.txt
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  + CategoryInfo          : ObjectNotFound: (C:\Users\andrv\Несуществующий
                            каталог:String) [Get-ChildItem],
                            ItemNotFoundException
  + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.
                            GetChildItemCommand
```

Как видим, в файл err.txt полностью записалось сообщение об ошибке.

Объект `ErrorRecord`, создающийся при возникновении ошибки, можно сохранить в переменной. Для этого используется оператор `2>&1`, который перенаправляет поток ошибок в стандартный выходной поток (имеющий внутренний номер 1). Например:

```
PS C:\Users\andrv> $err = dir "Несуществующий каталог" 2>&1
PS C:\Users\andrv> $err.getType().fullName
System.Management.Automation.ErrorRecord
```

Проверим с помощью командлета `Get-Member`, какие свойства имеет объект типа `ErrorRecord`:

```
PS C:\Users\andrv> $err | Get-Member -Type Property

   TypeName: System.Management.Automation.ErrorRecord
```

```
Name                  MemberType Definition
----                  ---------- ----------
CategoryInfo          Property   System.Management.Automation.Er...
ErrorDetails          Property   System.Management.Automation.Er...
Exception             Property   System.Exception Exception {get...
FullyQualifiedErrorId Property   System.String FullyQualifiedErr...
InvocationInfo        Property   System.Management.Automation.In...
PipelineIterationInfo Property   System.Collections.ObjectModel....
ScriptStackTrace      Property   string ScriptStackTrace {get;}
TargetObject          Property   System.Object TargetObject {get...
```

Эти свойства имеют отношение к возникающим ошибкам, описание наиболее важных из них приведено в табл. 10.1.

Таблица 10.1. *Свойства объекта ErrorRecord*

Свойство	Тип	Описание
CategoryInfo	System.Management.Automation.ErrorCategoryInfo	Категория ошибки
ErrorDetails	System.Management.Automation.ErrorDetails	Содержит дополнительное описание ошибки (свойство ErrorDetails.Message) Данное свойство может быть пустым
Exception	System.Exception	Исключение NET, соответствующее произошедшей ошибке
FullyQualifiedErrorId	System.String	Точное определение категории ошибки
InvocationInfo	System.Management.Automation.InvocationInfo	Объект, содержащий информацию о том, где именно произошла ошибка (обычно это имя сценария и номер строки в нем)
TargetObject	System.Object	Объект, над которым производились манипуляции во время возникновения ошибки. Данное свойство может быть пустым

Проверим содержимое переменной `$err`:

```
PS C:\Users\andrv> $err | Format-List * -Force

writeErrorStream      : True
PSMessageDetails      :
Exception             : System.Management.Automation.ItemNotFoundException:
                        Не удается найти путь "C:\Users\andrv\Несуществующий
                        каталог", т. к. он не существует.
  в System.Management.Automation.SessionStateInternal.GetChildItems(String
          path, Boolean recurse, UInt32 depth, CmdletProviderContext context)
  в Microsoft.PowerShell.Commands.GetChildItemCommand.ProcessRecord()
```

```
TargetObject          : C:\Users\andrv\Несуществующий каталог
CategoryInfo          : ObjectNotFound: (C:\Users\andrv\Несуществующий
                        каталог:String) [Get-ChildItem], ItemNotFoundException
FullyQualifiedErrorId : PathNotFound,
                        Microsoft.PowerShell.Commands.GetChildItemCommand
ErrorDetails          :
InvocationInfo        : System.Management.Automation.InvocationInfo
ScriptStackTrace      : в <ScriptBlock>, <Нет файла>: строка 1
PipelineIterationInfo : {0, 1}
```

Как видим, в нашем примере ошибка относится к категории `PathNotFound`, свойство `TargetObject` содержит полный путь, который командлет `dir` использовал для поиска каталога или файла. Вызвана ошибка была исключением `ItemNotFoundException`.

Выясним, что содержит свойство `InvocationInfo`:

```
PS C:\Users\andrv> $err.InvocationInfo

MyCommand             : Get-ChildItem
BoundParameters       : {}
UnboundArguments      : {}
ScriptLineNumber      : 1
OffsetInLine          : 8
HistoryId             : 1
ScriptName            :
Line                  : $err = dir "Несуществующий каталог" 2>&1
PositionMessage       : строка:1 знак:8
                        + $err = dir "Несуществующий каталог" 2>&1
                        +        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
PSScriptRoot          :
PSCommandPath         :
InvocationName        : dir
PipelineLength        : 0
PipelinePosition      : 0
ExpectingInput        : False
CommandOrigin         : Internal
DisplayScriptPosition :
```

Свойство `ScriptName` здесь не заполнено, т. к. команда была введена непосредственно в интерактивном режиме оболочки, а не из сценария.

Итак, мы изучили структуру объекта `ErrorRecord` и узнали, какую информацию можно извлечь из его свойств. Напомним, что получили мы этот объект путем перенаправления потока ошибок в стандартный выходной поток в одной конкретной команде. Понятно, что для каждой выполняемой команды повторять эти манипуляции мы не сможем (стандартный выходной поток может понадобиться для других целей) — необходим более удобный механизм, позволяющий определять, возникла ли ошибка, и обращаться к объекту `ErrorRecord`, который соответствует той или иной ошибке. Подобные механизмы реализованы в PowerShell с помощью несколь-

ких специальных переменных и параметров, к рассмотрению которых мы сейчас перейдем.

Сохранение объектов, соответствующих ошибкам

В PowerShell имеется специальная переменная `$error`, которая содержит коллекцию (массив) объектов `ErrorRecord`, соответствующих ошибкам, возникавшим в текущем сеансе работы. Максимальное количество элементов в данной коллекции задается значением переменной `$MaximumErrorCount` (по умолчанию 256):

```
PS C:\Users\andrv> $MaximumErrorCount
256
```

После заполнения массива `$error` объекты для вновь возникающих ошибок будут заменять объекты, соответствующие старым ошибкам. Возникновение каждой новой ошибки приводит к смещению элементов в массиве `$error`: объект для последней ошибки хранится в первом элементе (`$error[0]`), объект для предыдущей ошибки — во втором элементе (`$error[1]`) и т. д.

Повторим еще раз ошибку из предыдущего раздела:

```
PS C:\Users\andrv> dir "Несуществующий каталог"
dir : Не удается найти путь "C:\Users\andrv\Несуществующий каталог", т. к.
он не существует.
строка:1 знак:1
+ dir "Несуществующий каталог"
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  + CategoryInfo          : ObjectNotFound: (C:\Users\andrv\Несуществующий
                            каталог:String) [Get-Chi ldItem],
                            ItemNotFoundException
  + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.
                                                        GetChildItemCommand
```

Теперь убедимся, что элемент `$error[0]` имеет тип `ErrorRecord`, и выведем содержимое этого элемента:

```
PS C:\Users\andrv> $error[0].getType().fullName
System.Management.Automation.ErrorRecord
PS C:\Users\andrv> $error[0]
dir : Не удается найти путь "C:\Users\andrv\Несуществующий каталог", т. к.
он не существует.
строка:1 знак:1
+ dir "Несуществующий каталог"
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  + CategoryInfo          : ObjectNotFound: (C:\Users\andrv\Несуществующий
                            каталог:String) [Get-Chi ldItem],
                            ItemNotFoundException
  + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.
                                                        GetChildItemCommand
```

Естественно, мы можем обращаться ко всем свойствам объекта `ErrorRecord`:

```
PS C:\Users\andrv> $error[0].exception
Не удается найти путь "C:\Users\andrv\Несуществующий каталог", т. к.
он не существует.
PS C:\Users\andrv> $error[0].invocationInfo

MyCommand             : Get-ChildItem
BoundParameters       : {}
UnboundArguments      : {}
ScriptLineNumber      : 1
OffsetInLine          : 1
HistoryId             : 10
ScriptName            :
Line                  : dir "Несуществующий каталог"
PositionMessage       : строка:1 знак:1
                        + dir "Несуществующий каталог"
                        + ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
PSScriptRoot          :
PSCommandPath         :
InvocationName        : dir
PipelineLength        : 0
PipelinePosition      : 0
ExpectingInput        : False
CommandOrigin         : Internal
DisplayScriptPosition :

PS C:\Users\andrv> $error[0].targetobject
C:\Users\andrv\Несуществующий каталог
```

Для перехвата ошибок определенной команды без перенаправления ошибок в стандартный выходной поток можно воспользоваться стандартным параметром `-ErrorVariable`, который определен во всех командах PowerShell. Например, в следующей команде информация об ошибках будет записываться в переменную `$errs` (обратите внимание, что при задании значения параметра `-ErrorVariable` знак `$` перед именем переменной указывать не нужно):

```
PS C:\Users\andrv> dir "Каталог 1", "Каталог 2" -ErrorVariable errs
dir : Не удается найти путь "C:\Users\andrv\Каталог 1", т. к. он не существует.
строка:1 знак:1
+ dir "Каталог 1", "Каталог 2" -ErrorVariable errs
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  + CategoryInfo          : ObjectNotFound: (C:\Users\andrv\Каталог 1:String)
                            [Get-ChildItem], Item  NotFoundException
  + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.
                                                        GetChildItemCommand

dir : Не удается найти путь "C:\Users\andrv\Каталог 2", т. к. он не существует.
строка:1 знак:1
```

```
+ dir "Каталог 1", "Каталог 2" -ErrorVariable errs
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  + CategoryInfo          : ObjectNotFound: (C:\Users\andrv\Каталог 2:String)
                            [Get-ChildItem], Item NotFoundException
  + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.
                                                          GetChildItemCommand
```

При выполнении данной команды возникают две некритические ошибки, т. е. переменная `$errs` должна быть массивом, содержащим два объекта `ErrorRecord`. Проверим это:

```
C:\Users\andrv> $errs.count
2
C:\Users\andrv> $errs[0].getType().fullName
System.Management.Automation.ErrorRecord
C:\Users\andrv> $errs[1].getType().fullName
System.Management.Automation.ErrorRecord
C:\Users\andrv> $errs[0].targetObject
C:\Users\andrv\Каталог 1
C:\Users\andrv> $errs[1].targetObject
C:\Users\andrv\Каталог 2
```

Как видим, при указании параметра `ErrorVariable` сообщения о возникающих ошибках по-прежнему выводятся на экран. Для подавления этих сообщений нужно перенаправить поток ошибок на пустое устройство `$null`, например:

```
PS C:\Users\andrv> dir "Каталог 1", "Каталог 2" -ErrorVariable errs 2> $null
C:\Users\andrv>
```

На экран ничего не выводится, а в переменную `$errs` по-прежнему записываются два объекта:

```
PS C:\Users\andrv> $errs.count
2
```

При этом объекты, соответствующие ошибкам, записываются и в переменную `$error`:

```
PS C:\Users\andrv> $error[0]
dir : Не удается найти путь "C:\Users\andrv\Каталог 2", т. к. он не существует.
строка:1 знак:1
+ dir "Каталог 1", "Каталог 2" -ErrorVariable errs 2> $null
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  + CategoryInfo          : ObjectNotFound: (C:\Users\andrv\Каталог 2:String)
                            [Get-ChildItem], Item NotFoundException
  + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.
                                                          GetChildItemCommand

PS C:\Users\andrv> $error[1]
dir : Не удается найти путь "C:\Users\andrv\Каталог 1", т. к. он не существует.
строка:1 знак:1
```

```
+ dir "Каталог 1", "Каталог 2" -ErrorVariable errs 2> $null
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  + CategoryInfo          : ObjectNotFound: (C:\Users\andrv\Каталог 1:String)
                            [Get-ChildItem], Item NotFoundException
  + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.
                                                        GetChildItemCommand
```

Мониторинг возникновения ошибок

При интерактивной работе в оболочке PowerShell мы узнаем о возникновении ошибки по сообщению на экране. После этого мы можем проанализировать объект `$error[0]` и узнать всю информацию об ошибке. Но как узнать о возникновении ошибки внутри сценария?

Оказывается, в PowerShell имеется логическая переменная `$?`, которая равна `True`, если последняя выполняемая операция завершена успешно, и равна `False`, если во время выполнения последней операции возникли какие-либо ошибки.

Например, если выполнить командлет `Get-Item` для заведомо существующего каталога, то значение переменной `$?` будет равно `True`:

```
PS C:\Users\andrv> Get-Item C:\

    Каталог:

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d--hs-        01.08.2021     12:57                C:\

PS C:\Users\andrv> $?
True
```

Если же выполнить командлет `Get-Item` для несуществующего каталога, то значение переменной `$?` будет равно `False`:

```
PS C:\Users\andrv> Get-Item C:\АБВГГ
Get-Item : Не удается найти путь "C:\АБВГГ", т. к. он не существует.
строка:1 знак:1
+ Get-Item C:\АБВГГ
+ ~~~~~~~~~~~~~~~~
  + CategoryInfo          : ObjectNotFound: (C:\АБВГГ:String) [Get-Item],
                                                        ItemNotFoundException
  + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.
                                                        GetItemCommand

PS C:\Users\andrv> $?
False
```

Для внешних команд Windows и сценариев PowerShell определено понятие кода возврата (напомним, что для сценариев PowerShell этот код можно установить

с помощь инструкции `Exit`). В операционной системе код возврата последней команды доступен через переменную среды `%ERRORLEVEL%`; в оболочке PowerShell данный код возврата хранится в специальной переменной `$LASTEXITCODE`. При этом если код возврата равен нулю, то переменной `$?` присваивается значение `True`. Если же код возврата не равен нулю, то считается, что при выполнении данной команды произошла ошибка, и переменной `$?` присваивается значение `False`.

В качестве примера выполним команду интерпретатора cmd.exe, которая устанавливает нулевой код возврата. Для этого можно запустить cmd.exe с ключом `/c` (выполнить команду и завершить работу интерпретатора) и указать для исполнения команду `exit 0`:

```
PS C:\Users\andrv> cmd /c exit 0
```

Проверим значения переменных `$LASTEXITCODE` и `$?`:

```
PS C:\Users\andrv> $LASTEXITCODE
0
PS C:\Users\andrv> $?
True
```

Теперь выполним команду интерпретатора cmd.exe с ненулевым кодом возврата (пусть, например, код возврата равен 10):

```
PS C:\Users\andrv> cmd /c exit 10
```

Вновь проверим переменные `$LASTEXITCODE` и `$?`:

```
PS C:\Users\andrv> $?
False
PS C:\Users\andrv> $LASTEXITCODE
10
```

Итак, мы убедились, что переменными `$LASTEXITCODE` и `$?` можно пользоваться для обнаружения ошибок при выполнении сценариев PowerShell.

Режимы обработок ошибок

В начале данной главы мы говорили о том, что ошибки в PowerShell делятся на критические (прерывающие выполнение команды) и некритические (при их возникновении выполнение команды продолжается). При этом в зависимости от ситуации иногда может потребоваться рассматривать некритические ошибки как критические и наоборот. Подобное изменение типа ошибок реализовано в PowerShell с помощью различных режимов обработки ошибок (табл. 10.2).

Для установки нужного режима обработки ошибок следует присвоить соответствующее значение переменной `$ErrorActionPreference`, например:

```
$ErrorActionPreference = "SilentlyContinue"
```

В этом случае режим продолжения выполнения без выдачи сообщений об ошибке будет действовать на все команды.

Таблица 10.2. *Режимы обработки ошибок в PowerShell*

Режим	Символьная константа	Описание
Продолжение выполнения команды	`Continue`	Данный режим используется системой по умолчанию. Объект, соответствующий ошибке, записывается в выходной поток, добавляется в массив `$error`, переменная `$?` устанавливается в False На экран выводится сообщение об ошибке Выполнение сценария продолжается со следующей строки
Продолжение выполнения команды без выдачи сообщений об ошибке	`SilentlyContinue`	Если установлен данный режим, то при возникновении ошибки на экран не выводится никаких сообщений Объект, соответствующий ошибке, записывается в выходной поток, добавляется в массив `$error`, переменная `$?` устанавливается в False Выполнение сценария продолжается со следующей строки
Прекращение выполнения команд	`Stop`	Установка данного режима делает ошибки критическими Объект, соответствующий ошибке, не записывается в выходной поток, а генерирует исключение При этом переменные `$?` и `$error` по-прежнему обновляются Выполнение сценария прекращается
Выполнение команды с запросом	`Inquire`	Перед выполнением операции пользователю выдается запрос на подтверждение Здесь операцию можно продолжить, отменить или приостановить
Игнорирование ошибки	`Ignore`	В этом режиме возникновение ошибки игнорируется, выполнение команд продолжается Режим можно установить только в общем параметре `-ErrorActon`

Если нужно переключить режим обработки ошибок только для одного командлета, то при его вызове следует указать соответствующую символьную константу в качестве значения общего параметра `-ErrorAction` (или сокращенно — `ea`), например:

```
PS C:\Users\andrv> Get-Item "несуществующий файл" -ErrorAction SilentlyContinue
```

Обработка критических ошибок (исключений)

Напомним, что критическими ошибками в PowerShell называются ошибки, приводящие к прекращению выполнения команды. В традиционных языках программирования такие ошибки обычно называют *исключениями* (exceptions). В PowerShell перехватывать критические ошибки, т. е. выполнять определенные действия при их возникновении, можно с помощью двух инструкций: `Trap` и `Try/Catch/Finally`.

Инструкция *Trap*

Инструкция `Trap` была реализована в первой версии PowerShell и была тогда единственным инструментом для обработки исключений.

Возможны два варианта синтаксиса `Trap`:

```
trap [тип_исключения] {блок_кода}
```

или

```
trap {блок_кода}
```

В первом случае блок кода будет выполнен только при возникновении исключения определенного типа (например, для перехвата ошибок, возникающих при попытках деления на нуль, нужно в качестве типа исключения указать `DivideByZeroException`), а во втором — при возникновении любого исключения. Приведем пример.

```
PS C:\Users\andrv> trap {"Исключение перехвачено!"} $n=0; 1/$n; "n=$n"
Исключение перехвачено!
Попытка деления на нуль.
строка:1 знак:40
+ trap {"Исключение перехвачено!"} $n=0; 1/$n; "n=$n"
+                                         ~~~~
    + CategoryInfo          : NotSpecified: (:) [], RuntimeException
    + FullyQualifiedErrorId : RuntimeException

n=0
```

В данном случае при вычислении выражения `1/$n` возникает исключение (попытка деления на нуль) и управление передается в тело инструкции `Trap`, где выводится строка `Исключение перехвачено!`. Убедимся, что объект, соответствующий ошибке, записывается в массив `$error` (поэтому на экран также выводится стандартное сообщение об ошибке):

```
PS C:\Users\andrv> $error[0]
Попытка деления на нуль.
строка:1 знак:40
+ trap {"Исключение перехвачено!"} $n=0; 1/$n; "n=$n"
+                                         ~~~~
    + CategoryInfo          : NotSpecified: (:) [], RuntimeException
    + FullyQualifiedErrorId : RuntimeException
```

Обратите внимание, что после выполнения тела инструкции `Trap` управление в нашем примере передалось на команду, идущую после инструкции, вызвавшей ошибку. На самом деле так происходит не всегда. Если выход из обработчика исключения производится с помощью инструкции `Break`, то оставшиеся команды выполнены не будут:

```
PS C:\Users\andrv> trap {"Исключение перехвачено!"; break} $n=0; 1/$n; "n=$n"
Исключение перехвачено!
Попытка деления на нуль.
```

```
строка:1 знак:47
+ trap {"Исключение перехвачено!"; break} $n=0; 1/$n; "n=$n"
+                                                ~~~~
  + CategoryInfo          : NotSpecified: (:) [],
                                                ParentContainsErrorRecordException
  + FullyQualifiedErrorId : RuntimeException
```

Как видим, здесь отсутствует вывод на экран строки "n=0". Если выход из обработчика исключения производится с помощью инструкции `Continue`, то оставшиеся команды выполняются:

```
PS C:\Users\andrv> trap {"Исключение перехвачено!"; $_.Exception ; continue}
$n=0; 1/$n; "n=$n"
Исключение перехвачено!
Попытка деления на нуль.
n=0
```

Данный пример показывает также, что при обработке исключения доступ к соответствующему ему объекту осуществляется, как обычно, с помощью переменной `$_`.

Если выход из обработчика исключения выполнен обычным образом (без применения инструкций `Break` или `Continue`), то дальнейшее поведение определяется значением переменной `$ErrorActionPreference`, рассмотренной в предыдущем разделе. Так, в следующем примере выполнение команд после обработки исключения прерывается:

```
PS C:\Users\andrv> $ErrorActionPreference = "Stop"
PS C:\Users\andrv> trap {"Исключение перехвачено!"} $n=0; 1/$n; "n=$n"
Исключение перехвачено!
Попытка деления на нуль.
строка:1 знак:40
+ trap {"Исключение перехвачено!"} $n=0; 1/$n; "n=$n"
+                                           ~~~~
  + CategoryInfo          : NotSpecified: (:) [],
                                                ParentContainsErrorRecordException
  + FullyQualifiedErrorId : RuntimeException
```

Изменим теперь режим обработки ошибок и еще раз выполним те же команды после инструкции `Trap`:

```
PS C:\Users\andrv> $ErrorActionPreference = "SilentlyContinue"
PS C:\Users\andrv> trap {"Исключение перехвачено!"} $n=0; 1/$n; "n=$n"
Исключение перехвачено!
n=0
```

В данном случае стандартное сообщение об ошибке на экран не выводится и после обработки исключения выполняется следующая команда.

Инструкция *Try/Catch/Finally*

Инструкция `Try/Catch/Finally`, аналоги которой есть во многих языках программирования, поддерживается со второй версии PowerShell. Это более удобный и привычный вариант обработки исключений, чем инструкция `Trap`.

В блоке `Try` записываются команды и выражения, для которых мы хотим перехватывать возникновение критических ошибок. Этому блоку должен соответствовать хотя бы один блок `Catch` или `Finally`. Блоков `Catch` может быть несколько, в них могут проверяться типы исключений, которые нужно обработать. Если блок `Finally` указан, то код в нем будет выполнен в любом случае, независимо от возникновения исключения. Обычно в `Finally` помещают код для освобождения использованных ресурсов (закрытие подключения к базе данных, очистка сеанса удаленного доступа и т. п.).

Приведем пример обработки исключения без проверки его типа:

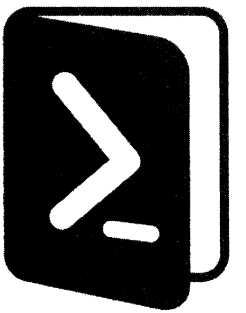
```
PS C:\Users\andrv> try {
>> 1
>> 2
>> 3/0
>> 4
>> } catch {
>> "Ошибка: $_"
>> } finally {
>> 'Завершающий блок'
>> }
1
2
Ошибка: Попытка деления на нуль.
Завершающий блок
```

В блоке `Try` здесь возникает ошибка при попытке разделить число на нуль и управление передается в блок `Catch`, где объект ошибки доступен под именем `$_`. После отработки блока `Catch` выполняются команды из блока `Finally`.

Итоги

- В PowerShell ошибки делятся на два типа: критические, прерывающие выполнение команды, и некритические, при возникновении которых выполнение команды продолжается.
- При возникновении ошибки система генерирует объект типа `ErrorRecord`, содержащий информацию об этой ошибке.
- Объекты с информацией об ошибках направляются в стандартный поток ошибок, содержимое которого по умолчанию отображается на экране.

- Объекты `ErrorRecord`, соответствующие ошибкам, возникавшим в текущем сеансе работы, автоматически сохраняются в специальный массив `$error`. Объект для последней ошибки содержится в `$error[0]`.
- Объекты, соответствующие ошибкам, можно сохранять в другие переменные, указывая их имена в стандартном параметре `-ErrorVariable`.
- В переменной `$?` система сохраняет статус завершения последней команды.
- Код возврата внешней консольной утилиты или сценария записывается в переменную `$LASTEXITCODE`.
- В PowerShell поддерживается несколько режимов обработки ошибок команд, которые можно переключать с помощью переменной `$ErrorActionPreference` и общего параметра `-ErrorAction`.
- Критические ошибки (исключения) можно обрабатывать с помощью инструкций `Trap` и `Try/Catch/Finally`.



ЧАСТЬ III

Автоматизируем рутину

Глава 11. Работа с файловой системой и оболочкой Windows

Глава 12. Обработка структурированных данных

Глава 13. Управление процессами, службами и серверами автоматизации

Глава 14. HTTP-запросы к веб-ресурсам

ГЛАВА 11

Работа с файловой системой и оболочкой Windows

Все люди, работающие с компьютером, ежедневно выполняют множество обращений к файловой системе, читая и сохраняя файлы, создавая каталоги, копируя файлы из одного места в другое и т. п. В данной главе мы научимся манипулировать объектами файловой системы с помощью PowerShell.

> ***Замечание***
> Не забывайте при работе в командной строке пользоваться автодополнением путей к файлам и каталогам. Не нужно полностью набирать длинные пути — достаточно ввести первые символы, нажать клавишу <Tab>, и оболочка сама подставит подходящий путь в файловой системе.

Рассмотрим наиболее часто выполняемые операции при работе с файлами и каталогами.

Навигация в файловой системе

В оболочках cmd.exe и bash переходить по каталогам можно с помощью команды `cd`. В PowerShell команда `cd` имеет тот же смысл, при этом она является стандартным псевдонимом командлета `Set-Location`. Например, следующая команда делает текущим каталог C:\Windows:

```
PS C:\> cd C:\WINDOWS
PS C:\WINDOWS>
```

Как и в оболочке cmd.exe, в качестве пути можно указывать символы `..` (для перехода в родительский каталог) и `\` (для перехода в корневой каталог текущего диска). Например:

```
PS C:\WINDOWS> cd \
PS C:\>
```

Получение списка файлов и каталогов

Напомним, что в оболочке cmd.exe список файлов и каталогов формируется с помощью внутренней команды `dir`, которая имеет множество ключей, позволяющих,

например, выводить только файлы с определенными атрибутами, обрабатывать вложенные подкаталоги, сортировать список по различным полям и т. д.

В PowerShell также можно использовать команду `dir`, которая является здесь псевдонимом командлета `Get-ChildItem`. Если выполнить эту команду без параметров, то будет выведено содержимое текущего каталога:

```
PS C:\> dir

    Каталог: C:\

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        03.03.2021     19:02                Far3x64
d-----        29.08.2020     22:01                Intel
d-----        13.05.2021     21:38                MRSU
d-----        07.12.2019     12:14                PerfLogs
. . .
```

В пути, который указывается для команды `dir`, можно применять подстановочные знаки. Например, следующая команда выведет все файлы с расширением log из каталога C:\Windows:

```
PS C:\> dir C:\WINDOWS\*.log

    Каталог: C:\WINDOWS

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        28.09.2020     16:16            762 comsetup.log
-a----        07.12.2019     12:17            776 DtcInstall.log
-a----        26.05.2021      3:20          63456 PFRO.log
-a----        11.05.2021     22:03            623 setupact.log
-a----        28.09.2020     16:13              0 setuperr.log
-a----        04.06.2021     16:14            276 WindowsUpdate.log
```

Параметр `-Exclude` позволяет задать маску файлов, которые не будут обрабатываться командой `dir`. Например, следующая команда выведет все файлы с расширением log из каталога C:\Windows, кроме тех, чье имя начинается на букву d:

```
PS C:\> dir C:\WINDOWS\*.log -exclude d*.log

    Каталог: C:\WINDOWS

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        28.09.2020     16:16            762 comsetup.log
-a----        26.05.2021      3:20          63456 PFRO.log
-a----        11.05.2021     22:03            623 setupact.log
-a----        28.09.2020     16:13              0 setuperr.log
-a----        04.06.2021     16:14            276 WindowsUpdate.log
```

Параметр -Name позволяет выводить на экран только имена файлов, например:

```
PS C:\> dir C:\WINDOWS\*.log -Name
comsetup.log
DtcInstall.log
PFRO.log
setupact.log
setuperr.log
WindowsUpdate.log
```

Параметр -Recurse включает режим рекурсии, при котором команда dir действует не только на содержимое указанного каталога, но и на все его подкаталоги:

```
PS C:\> dir .\Users\ -Recurse

    Каталог: C:\Users

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        31.05.2021      7:04                andrv
d-r---        28.09.2020     16:06                Public

    Каталог: C:\Users\andrv

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        15.02.2021     20:30                .config
d-----        10.10.2020     22:02                .dbus-keyrings
d-----        11.02.2021     16:12                .local
. . .
```

По умолчанию командлет dir не "видит" скрытые файлы. Если необходимо такие файлы также включать в список, то нужно указать параметр -Force:

```
PS C:\> dir -Force

    Каталог: C:\

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d--hs-        01.10.2020     14:17                $Recycle.Bin
d--h--        28.09.2020      4:27                $WinREAgent
d--hsl        29.08.2020     21:52                Documents and Settings
d-----        03.03.2021     19:02                Far3x64
d-----        29.08.2020     22:01                Intel
. . .
```

У скрытых файлов в списке атрибутов (свойство Mode) присутствует буква h (атрибут Hidden).

Иногда бывает нужно сформировать список, состоящий только из подкаталогов или только из файлов. Для этого служат ключи `-Directory` и `-File` соответственно. Например:

```
PS C:\> dir 'C:\program files\Windows Defender\' -Directory

    Каталог: C:\program files\Windows Defender

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        20.12.2020     22:19               en-US
d-----        20.12.2020     22:19               Offline
d-----        28.09.2020     16:18               platform
d-----        12.05.2021     21:23               ru-RU

PS C:\> dir 'C:\program files\Windows Defender\' -File

    Каталог: C:\program files\Windows Defender

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        07.12.2019     12:08           9398 AmMonitoringInstall.mof
-a----        23.01.2021      8:51         204624 AMMonitoringProvider.dll
-a----        07.12.2019     12:08          21004 AmStatusInstall.mof
-a----        07.12.2019     12:08           2460 ClientWMIInstall.mof
-a----        07.12.2019     12:08         318776 ConfigSecurityPolicy.exe
-a----        07.12.2019     12:08         210872 DefenderCSP.dll
-a----        07.12.2019     12:08         733200 EppManifest.dll
-a----        07.12.2019     12:08            361 FepUnregister.mof
. . .
```

Сформированный с помощью `dir` список объектов файловой системы можно сортировать по любому полю с помощью конвейера с командлетом `Sort-Object`. Например, следующая команда выведет список всех файлов и каталогов диска C:\, отсортированный в обратном порядке по времени обращения к ним:

```
PS C:\> dir | Sort-Object -descending LastWriteTime

    Каталог: C:\

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        13.05.2021     21:38               MRSU
d-----        12.05.2021     21:23               Windows
d-r---        04.04.2021     11:19               Program Files (x86)
d-----        03.04.2021     21:39               sqlite
d-----        01.04.2021     19:17               projects
d-----        03.03.2021     19:02               Far3x64
d-r---        15.02.2021     22:15               Program Files
```

```
d-----        15.02.2021     16:09                Python39
d-----        11.02.2021     16:12                vifm
d-----        11.02.2021     16:10                Soft
d-r---        01.10.2020     14:19                Users
d-----        29.08.2020     22:01                Intel
d-----        07.12.2019     12:14                PerfLogs
```

Определение размера каталогов

Любому каталогу файловой системы в PowerShell соответствует объект типа `System.IO.DirectoryInfo`, который не имеет свойства, в котором хранился бы размер этого каталога. Поэтому для определения размера каталога нужно подсчитать общий размер файлов, записанных в данном каталоге и всех его подкаталогах. Сделать это можно путем суммирования с помощью командлета `Measure-Object` значений свойства `Length` объектов-файлов, поставляемых по конвейеру командлетом `dir` (`Get-ChildItem`). Например, следующая команда показывает размер в байтах каталога C:\Program Files (свойство `Sum`) и количество файлов и подкаталогов в этом каталоге (свойство `Count`):

```
PS C:\> dir "C:\Program Files" -Recurse | Measure-Object -Property length -Sum

Count    : 21539
Average  :
Sum      : 2492086334
Maximum  :
Minimum  :
Property : length
```

Если нужно определить размер в мегабайтах или гигабайтах, то значение свойства `Sum` следует разделить на соответствующую константу-суффикс (`1Mb` или `1Gb`):

```
PS C:\> (dir "C:\Program Files" -Recurse | Measure-Object -Property length
-Sum).Sum/1Mb
2376,63873100281
```

Как видим, результат содержит 11 знаков после запятой. Можно округлить полученное число до двух знаков:

```
PS C:\> $FolderSize = (dir "C:\Program Files" -Recurse | Measure-Object
-Property length -Sum).Sum/1Mb
PS C:\> [math]::round($FolderSize,2)
2376,64
```

Создание файлов и каталогов

Создать новый файл или каталог в PowerShell позволяет командлет `New-Item` (короткий псевдоним `ni`). Путь к создаваемому элементу указывается в виде значения параметра `-Path`, а в качестве значения параметра `-ItemType` указывается `"directory"`, если нужно создать каталог, и `"file"`, если нужно создать файл.

Например, следующая команда создает на диске C:\ каталог с именем Test:

```
PS C:\> New-Item -Path C:\Test -Type "directory"

    Каталог: C:\

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        04.06.2021     20:03                Test
```

Иногда бывает нужно создавать каталоги, в имени которых присутствовала бы текущая дата в определенном формате. Требуемое текстовое представление даты может быть получено с помощью параметра `-uformat` командлета `Get-Date`. Например, следующая команда создаст на диске C:\ каталог с именем, совпадающим с текущей датой в формате ГГММДД:

```
PS C:\> New-Item -Path C:\ -Name "$(Get-Date -uformat '%y%m%d')"
-Type "directory"

    Каталог: C:\

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        04.06.2021     22:30                210604
```

Если в номере года нужно указывать четыре цифры, то следует изменить спецификатор формата на `'%Y%m%d'`:

```
PS C:\> New-Item -Path C:\ -Name "$(Get-Date -uformat '%Y%m%d')"
-Type "directory"

    Каталог: C:\

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        04.06.2021     22:31                20210604
```

При создании файла в него сразу можно записать строку, указав ее в качестве значения параметра `-Value`, например:

```
PS C:\> New-Item -Path C:\Test\test_file.txt -Type "file" -Value "Test"

    Каталог: C:\Test

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        04.06.2021     22:34              4 test_file.txt
```

Создание нескольких файлов

В качестве значения параметра `-Path` можно указать несколько файлов через запятую, они все будут созданы:

```
PS C:\Users\andrv> New-Item -Path C:\Test\test1.txt, .\test2.txt, .\test3.txt

    Каталог: C:\Test

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        12.06.2021      6:28              0 test1.txt

    Каталог: C:\Users\andrv

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        12.06.2021      6:28              0 test2.txt
-a----        12.06.2021      6:28              0 test3.txt
```

Пересоздание файла

Если попытаться создать файл с именем уже существующего файла, то возникнет ошибка:

```
PS C:\> New-Item -Path C:\Test\test_file.txt -Type "file" -Value "Test2"
new-item : Файл "C:\Test\test_file.txt" уже существует.
строка:1 знак:1
+ new-item -path C:\Test\test_file.txt -type "file" -value "Test2 ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  + CategoryInfo          : WriteError: (C:\Test\test_file.txt:String)
                            [New-Item], IOException
  + FullyQualifiedErrorId : NewItemIOError,Microsoft.PowerShell.Commands.
                                                                  NewItemCommand
```

Для пересоздания существующего файла нужно указать параметр `-Force`:

```
PS C:\> New-Item -Path C:\Test\test_file.txt -Type "file" -Value "Test2" -Force

    Каталог: C:\Test

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        04.06.2021     22:37              5 test_file.txt
```

Создание файла в несуществующем каталоге

Параметр `-Force` также помогает в том случае, когда нам нужно создать файл в каталоге, который еще не существует. По умолчанию без этого параметра в данном случае возникнет ошибка. Например, попробуем создать файл в несуществующем подкаталоге inner:

```
PS C:\Users\andrv> New-Item -Path C:\Test\inner\test_file.txt
New-Item : Не удалось найти часть пути "C:\Test\inner\test_file.txt".
```

```
строка:1 знак:1
+ New-Item -Path C:\Test\inner\test_file.txt
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  + CategoryInfo          : WriteError: (C:\Test\inner\test_file.txt:String)
                            [New-Item], DirectoryNotFoundException
  + FullyQualifiedErrorId : NewItemIOError,Microsoft.PowerShell.Commands.
                                                                  NewItemCommand
```

Если указать параметр `-Force`, то подкаталог inner будет создан автоматически:

```
PS C:\Users\andrv> New-Item -Path C:\Test\inner\test_file.txt -Force

    Каталог: C:\Test\inner

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        12.06.2021      6:34              0 test_file.txt
```

Чтение содержимого файлов

В оболочке cmd.exe имеется команда `type`, которая выводит содержимое текстового файла на экран. В PowerShell команда `type` является псевдонимом командлета `Get-Content` (другие псевдонимы — `cat` и `gc`), предназначенного для построчного считывания содержимого текстового файла с возвращением объекта для каждой строки (при этом строки отображаются на экране). Например:

```
PS C:\> Get-Content C:\WINDOWS\win.ini
; for 16-bit app support
[fonts]
[extensions]
[mci extensions]
[files]
[Mail]
MAPI=1
CMCDLLNAME32=mapi32.dll
CMC=1
MAPIX=1
MAPIXVER=1.0.0.1
OLEMessaging=1
```

Параметр `-Encoding` командлета `Get-Content` позволяет явно указывать кодировку файла для корректной обработки его содержимого. Допустимые значения данного параметра: `Unicode`, `Byte`, `BigEndianUnicode`, `UTF8`, `UTF7`, `Ascii`.

По умолчанию командлет `Get-Content` считывает все строки из файла; их количество можно ограничить с помощью параметра `-TotalCount`. Например, следующая команда считывает первые пять строк из файла C:\WINDOWS\win.ini:

```
PS C:\> Get-Content C:\WINDOWS\win.ini -TotalCount 5
; for 16-bit app support
```

```
[fonts]
[extensions]
[mci extensions]
[files]
```

Результат выполнения командлета `Get-Content` можно сохранять в переменной, обращаясь в дальнейшем к ней как к массиву строк, например:

```
PS C:\> $f = Get-Content C:\WINDOWS\win.ini -TotalCount 5
PS C:\> $f.Length
5
PS C:\> $f[0]
; for 16-bit app support
PS C:\> $f[1]
[fonts]
```

Разумеется, возвращаемый массив строк можно сразу передавать по конвейеру для обработки другими командлетами, например для какого-либо преобразования и записи в другой файл (соответствующий пример приведен в следующем разделе).

Параметр `-Raw` позволяет считать все содержимое файла в одну строку, например:

```
PS C:\Users\andrv> $s = Get-Content C:\WINDOWS\win.ini -Raw
PS C:\Users\andrv> $s
; for 16-bit app support
[fonts]
[extensions]
[mci extensions]
[files]
[Mail]
MAPI=1
CMCDLLNAME32=mapi32.dll
CMC=1
MAPIX=1
MAPIXVER=1.0.0.1
OLEMessaging=1

PS C:\Users\andrv> $s.Length
167
```

Запись файлов

В *главе 6* мы уже рассматривали варианты записи данных во внешние файлы с помощью операторов перенаправления (> и >>) и командлета `Out-File`. При этом командлет `Out-File` пытается форматировать записываемые в файл объекты. Если нужно просто записать в файл текстовую информацию (без дополнительного форматирования), то лучше воспользоваться командлетами `Set-Content` и `Add-Content`.

Замечание

Во многих языках программирования файл для чтения или записи нужно предварительно открывать с помощью специальных операторов или функций. В PowerShell об этом беспокоиться не нужно, оболочка все делает автоматически.

Данные для записи в файл могут задаваться в качестве значения параметра `-Value`. Например, следующая команда записывает в файл C:\Test\test.txt строку "Строка из PowerShell":

```
PS C:\> Set-Content C:\Test\test.txt -Value "Строка из PowerShell"
```

Проверим содержимое файла C:\Test\test.txt:

```
PS C:\> type C:\Test\test.txt
Строка из PowerShell
```

Массив записываемых строк может приниматься по конвейеру от других командлетов. В качестве примера мы считаем пять первых строк из файла C:\WINDOWS\win.ini и запишем их в файл C:\Test\num.txt с добавлением перед каждой строкой ее порядкового номера. Для этого сначала обнулим переменную `$n` (счетчик строк):

```
PS C:\> $n = 0
```

Теперь выполним следующий конвейер команд:

```
PS C:\> Get-Content C:\WINDOWS\win.ini -TotalCount 5 |
ForEach-Object {$n++;"$n $_"} | Set-Content C:\Test\num.txt
```

Командлет `Get-Content` в этом конвейере считывает строки из файла C:\WINDOWS\win.ini, которые по очереди обрабатываются командлетом `ForEach-Object`. Для каждой обрабатываемой строки значение переменной `$n` увеличивается на единицу и в выходной поток помещается значение, вычисляемое при расширении строки `"$n $_"` (значение переменной `$n` и текущая строка, поступившая по конвейеру). Затем полученная строка передается по конвейеру командлету `Set-Content`, который добавляет ее в файл C:\Test\num.txt.

Убедимся, что в файле C:\Test\num.txt записаны нужные строки:

```
PS C:\> type C:\Test\num.txt
1 ; for 16-bit app support
2 [fonts]
3 [extensions]
4 [mci extensions]
5 [files]
```

Если файл, в который производится запись, уже существует, то командлет `Set-Content` заменит его содержимое. В случае, когда требуется добавить строки в конец существующего файла, следует воспользоваться командлетом `Add-Content`.

Копирование файлов и каталогов

В PowerShell копирование файлов и каталогов осуществляется командлетом `Copy-Item`, имеющим псевдоним `copy`. Путь к копируемым файлам при этом указывается в качестве значения параметра `-Path` (данный параметр используется по умолча-

нию), а путь к целевому каталогу, в который нужно скопировать файлы, задается значением параметра `-Destination`.

Например, следующая команда копирует файл win.ini из каталога C:\Windows в каталог C:\Test:

```
PS C:\> copy C:\Windows\win.ini -Destination C:\Test
```

По умолчанию командлет `Copy-Item` не выводит информацию на экран. Для отображения результата копирования нужно указать параметр `-PassThru`:

```
PS C:\> copy C:\Windows\win.ini -Destination C:\Test -PassThru

    Каталог: C:\Test

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        21.09.2020     16:29            167 win.ini
```

Копирование каталога с файлами

Если путь к копируемым объектам указывает на каталог, то по умолчанию будет скопирован только этот каталог без своего содержимого (этим PowerShell отличается от большинства других оболочек, в том числе от cmd.exe).

Например, создадим каталог C:\Script с тремя файлами 1.ps1, 2.ps1 и 3.txt:

```
PS C:\Users\andrv> New-Item C:\Script\1.ps1, C:\Script\2.ps1, C:\Script\
3.txt -Force

    Каталог: C:\Script
Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        11.06.2021     23:23              0 1.ps1
-a----        11.06.2021     23:23              0 2.ps1
-a----        11.06.2021     23:23              0 3.txt
```

Теперь скопируем каталог C:\Script в C:\Test:

```
PS C:\Users\andrv> copy C:\Script -Destination C:\Test -PassThru

    Каталог: C:\Test

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        11.06.2021     23:25                Script
```

Как видим, скопировался только сам каталог без находящихся в нем файлов.

Копирование вложенных каталогов

Копировать содержимое вложенных каталогов позволяет параметр `-Recurse`, например:

```
PS C:\Users\andrv> copy C:\Script -Destination C:\Test -Recurse -PassThru

    Каталог: C:\Test\Script

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        11.06.2021     23:25              0 1.ps1
-a----        11.06.2021     23:25              0 2.ps1
-a----        11.06.2021     23:25              0 3.txt
```

Копирование файлов по маске

Можно копировать не все файлы из каталога, а только соответствующие определенной маске. При этом маску можно указать внутри пути для копирования или в качестве значения параметра `-Include`. Например, следующая команда копирует все файлы с расширением ps1 из каталога C:\Script в папку C:\Test:

```
PS C:\Users\andrv> copy C:\Script\*.ps1  -Destination C:\Test -PassThru

    Каталог: C:\Test

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        11.06.2021     23:25              0 1.ps1
-a----        11.06.2021     23:25              0 2.ps1
```

Однако если необходимо скопировать файлы по маске из всех вложенных подкаталогов, то одним командлетом `Copy-Item` обойтись не удастся. Предварительно требуемые файлы нужно выделить командлетом `Get-ChildItem` (`dir`), а затем передать их командлету `Copy-Item` по конвейеру.

Например, создадим два файла в подкаталогах Posh и Misc:

```
PS C:\Users\andrv> New-Item C:\Script\Posh\4.ps1, C:\Script\Misc\5.ps1 -Force

    Каталог: C:\Script\Posh

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        12.06.2021      6:39              0 4.ps1

    Каталог: C:\Script\Misc

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        12.06.2021      6:39              0 5.ps1
```

Следующая команда копирует все файлы с расширением ps1 из каталога C:\Script и всех его подкаталогов в папку C:\Test:

```
PS C:\Users\andrv> dir -Recurse -Include *.ps1  C:\script\* |
copy -Destination C:\Test -PassThru

    Каталог: C:\Test

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        12.06.2021      6:39              0 5.ps1
-a----        12.06.2021      6:39              0 4.ps1
-a----        11.06.2021     23:20              0 1.ps1
-a----        11.06.2021     23:20              0 2.ps1
```

Конкатенация файлов

Команда `copy` оболочки cmd.exe позволяла объединять несколько файлов вместе (конкатенация файлов). В PowerShell объединить файлы можно с помощью командлета `Get-Content` (псевдоним `type`) и перенаправления вывода в результирующий файл. Рассмотрим пример. Создадим файлы C:\Test\1.txt и C:\Test\2.txt:

```
PS C:\Users\andrv> New-Item -Path C:\Test\1.txt -Type "file" -Value "File 1"

    Каталог: C:\Test

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        12.06.2021      6:56              6 1.txt

PS C:\Users\andrv> New-Item -Path C:\Test\2.txt -Type "file" -Value "File 2"

    Каталог: C:\Test

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        12.06.2021      6:56              6 2.txt
```

Следующая команда объединяет файлы 1.txt и 2.txt в файл 3.txt:

```
PS C:\Users\andrv> type C:\Test\1.txt, C:\Test\2.txt > C:\Test\3.txt
```

Проверим содержимое файла 3.txt:

```
PS C:\Users\andrv> type C:\Test\3.txt
File 1
File 2
```

Как видим, конкатенация файлов 1.txt и 2.txt выполнена успешно.

Переименование и перемещение файлов и каталогов

Переименовать файл или каталог можно с помощью командлета `Rename-Item` (псевдоним `ren`). Значение параметра `-Path` этого командлета задает путь к элементам для переименования, а значение параметра `-NewName` — новое имя. Имена этих параметров можно опускать (в этом случае первым должно указываться значение параметра `-Path`). Например, создадим файл C:\1.tmp и переименуем его в файл 2.tmp:

```
PS C:\Users\andrv> New-Item -Path C:\Test\1.tmp

    Каталог: C:\Test

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        12.06.2021      7:01              0 1.tmp

PS C:\Users\andrv> ren C:\Test\1.tmp 2.tmp
```

Для того чтобы увидеть результат действия командлета `Rename-Item`, нужно указать параметр `-PassThru`:

```
PS C:\Users\andrv> ren C:\Test\2.tmp 3.tmp -PassThru

    Каталог: C:\Test

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        12.06.2021      7:01              0 3.tmp
```

Переименование группы файлов

В отличие от команды `ren` оболочки cmd.exe, командлет `Rename-Item` не поддерживает подстановочные знаки в значении параметров `-Path` или `-NewName`. Поэтому для переименования группы файлов их нужно сначала получить с помощью командлета `Get-ChildItem` (`dir`), а потом передать по конвейеру командлету `Rename-Item`.

В качестве примера мы переименуем все файлы с расширением txt в каталоге C:\Test, назначив им новое расширение — md:

```
PS C:\Users\andrv> dir C:\Test\*.txt | Rename-Item -NewName { $_.Name -replace
'.txt','.md' }
```

Перемещение файлов

Командлет `Rename-Item` позволяет переименовывать файлы или каталоги только внутри того каталога, в котором находился исходный объект. Если нужно переместить файл или каталог в другую папку, то следует воспользоваться командлетом

Move-Item (псевдоним move). Значение параметра -Path данного командлета задает путь к файлам или каталогам для перемещения (в этом пути допускается использование подстановочных знаков), а значение параметра -Destination — путь к каталогу, куда будут перемещены эти файлы или каталоги. Результат перемещения можно увидеть на экране, указав параметр -PassThru. Например, следующая команда переносит в корневой каталог диска C:\ каталог C:\Test\inner со всем его содержимым:

```
PS C:\Users\andrv> Move-Item -Path C:\Test\inner C:\ -PassThru

    Каталог: C:\

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        12.06.2021      6:34                inner
```

Вернем каталог inner на место:

```
PS C:\Users\andrv> Move-Item -Path C:\inner C:\Test -PassThru

    Каталог: C:\Test

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        12.06.2021      6:34                inner
```

Удаление файлов и каталогов

Удалять объекты файловой системы можно с помощью командлета Remove-Item (псевдоним del). Значение параметра -Path этого командлета задает путь к удаляемым файлам или каталогам (имя параметра в команде можно не указывать). В пути допускаются подстановочные символы, также командлет Remove-Item имеет параметр -Include, значение которого задает файлы, на которые будет действовать команда, и параметр и -Exclude, задающий файлы-исключения, которые удаляться не будут.

Например, следующая команда удалит все файлы с расширением ps1 в каталоге C:\Test:

```
PS C:\> del C:\Test\*.ps1
```

Если попытаться удалить все файлы в каталоге, имеющем подкаталоги, то система выдаст предупреждение:

```
PS C:\Users\andrv> del C:\Test\*

Подтверждение
Элемент в C:\Test\inner имеет дочерние объекты, и параметр Recurse не указан.
При продолжении все дочерние объекты
будут удалены вместе с элементом. Вы действительно хотите продолжить?
```

```
[Y] Да - Y  [A] Да для всех - A  [N] Нет - N  [L] Нет для всех - L  [S]
Приостановить - S  [?] Справка
(значением по умолчанию является "Y"):
```

Для удаления без предупреждения всех элементов в каталоге, включая подкаталоги, следует указать параметр `-Recurse`:

```
PS C:\> del C:\Test\* -Recurse
```

Поиск текста в файлах

В принципе, для поиска текста в файле можно считать его содержимое в массив строк с помощью командлета `Get-Content` и применить к ним один из операторов сравнения, описанных в *главе 8*. Однако в PowerShell имеется командлет `Select-String`, специально предназначенный для поиска строк текста в одном или нескольких файлах (это аналог консольной утилиты `findstr` для Windows и `grep` для UNIX-систем).

Например, следующая команда ищет слово Error во всех файлах с расширением log в системном каталоге Windows (имена параметров `-Pattern` и `-Path` можно опустить, не забывая при этом, что шаблон поиска должен стоять после имени командлета первым):

```
PS C:\Users\andrv> Select-String -Pattern Error -Path $env:windir\*.log

C:\WINDOWS\DtcInstall.log:1:12-07-2019 09:17 : DTC Install error = 0,
Enter MsDtcAdvancedInstaller::Configure, base\wcp
\plugins\msdtc\msdtcadvancedinstaller\msdtcadvancedinstaller.cpp (367)
C:\WINDOWS\DtcInstall.log:2:12-07-2019 09:17 : DTC Install error = 0,
Action: None, base\wcp\plugins\msdtc\msdtcadvance
dinstaller\msdtcadvancedinstaller.cpp (396)
. . .
```

Как видим, на экран по умолчанию выводятся: путь к проверяемом файлу, номер строки, в которой найдено совпадение, и сама эта строка. На самом деле результат поиска является объектом типа `Microsoft.PowerShell.Commands.MatchInfo`, содержащим свойства `Path` (путь к файлу), `Pattern` (шаблон, по которому производился поиск), `LineNumber` (номер найденной строки в файле), `Line` (содержимое найденной строки) и другие:

```
PS C:\Users\andrv> Select-String Error $env:windir\*.log | Format-List *

IgnoreCase : True
LineNumber : 1
Line       : 12-07-2019 09:17 : DTC Install error = 0, Enter
             MsDtcAdvancedInstaller::Configure, base\wcp\plugins\msdtc\
             msdtcadvancedinstaller\msdtcadvancedinstaller.cpp (367)
Filename   : DtcInstall.log
Path       : C:\WINDOWS\DtcInstall.log
Pattern    : Error
```

```
Context    :
Matches    : {0}

IgnoreCase : True
LineNumber : 2
Line       : 12-07-2019 09:17 : DTC Install error = 0, Action: None,
             base\wcp\plugins\msdtc\msdtcadvancedinstaller\msdt
             cadvancedinstaller.cpp (396)
Filename   : DtcInstall.log
Path       : C:\WINDOWS\DtcInstall.log
Pattern    : Error
Context    :
Matches    : {0}
. . .
```

Если при поиске нужно учитывать регистр символов, то следует указать параметр `-CaseSensitive`.

По умолчанию командлет `Select-String` находит в файлах все соответствия для строк. В случае необходимости можно ограничиться только первым найденным соответствием в каждом файле, указав параметр `-List`, например:

```
PS C:\Users\andrv> Select-String Error $env:windir\*.log -List

C:\WINDOWS\DtcInstall.log:1:12-07-2019 09:17 : DTC Install error = 0,
Enter MsDtcAdvancedInstaller::Configure, base\wcp
\plugins\msdtc\msdtcadvancedinstaller\msdtcadvancedinstaller.cpp (367)
```

Если нужно определить сам факт наличия искомой строки в файлах и неважно, в каком именно файле и на какой позиции находится эта строка, то можно использовать параметр `-Quiet`. В этом случае командлет `Select-String` возвращает `True`, если поиск завершен успешно, и `False` — в противном случае:

```
PS C:\Users\andrv> Select-String Error $env:windir\*.log -Quiet
True
```

В качестве шаблона для поиска можно использовать регулярные выражения. Например, следующая команда будет искать в файлах с расширением log в системном каталоге Windows строки, содержащие слова Error или Setup:

```
PS C:\Users\andrv> Select-String -pattern '(Error|Setup)' $env:windir\*.log

C:\WINDOWS\comsetup.log:2:COM+[4:16:25]: Setup started - [DATE:09,28,2020 TIME:
04:16 pm]
C:\WINDOWS\comsetup.log:8:COM+[4:16:28]: Setup (COMMIG) finished
- [DATE:09,28,2020 TIME: 04:16 pm]
C:\WINDOWS\DtcInstall.log:1:12-07-2019 09:17 : DTC Install error = 0,
Enter MsDtcAdvancedInstaller::Configure, base\wcp
\plugins\msdtc\msdtcadvancedinstaller\msdtcadvancedinstaller.cpp (367)
C:\WINDOWS\DtcInstall.log:2:12-07-2019 09:17 : DTC Install error = 0,
Action: None, base\wcp\plugins\msdtc\msdtcadvance
dinstaller\msdtcadvancedinstaller.cpp (396)
. . .
```

Замечание

Подробное рассмотрение языка регулярных выражений выходит за рамки данной книги. Дополнительную информацию о некоторых аспектах регулярных выражений можно найти во встроенной справке PowerShell (команда `Get-Help about_regular_expression`); более подробные справочные материалы имеются на сайте Microsoft (**https://docs.microsoft.com/ru-ru/dotnet/standard/base-types/regular-expressions**).

Файлы, в которых производится поиск, могут не только указываться в качестве значения параметра `-Path`, но и поступать командлету `Select-String` по конвейеру. Например, следующая команда ищет слово Microsoft во всех файлах с расширением txt, хранящихся в каталоге Windows\System32 и всех его подкаталогах:

```
PS C:\Users\andrv> dir $env:windir\System32\* -Include *.txt -Recurse |
Select-String -Pattern Microsoft -CaseSensitive

C:\WINDOWS\system32\catroot2\dberr.txt:1:CatalogDB: 20:43:37 08.06.2021:
DONE Adding Catalog File (78ms): Microsoft-Win
dows-Client-Desktop-Required-
Package04113~31bf3856ad364e35~amd64~~10.0.19041.1023.cat
C:\WINDOWS\system32\catroot2\dberr.txt:2:CatalogDB: 20:43:38 08.06.2021:
DONE Adding Catalog File (125ms): Microsoft-Wi
ndows-Client-Desktop-Required-WOW64-
Package~31bf3856ad364e35~amd64~~10.0.19041.1052.cat
C:\WINDOWS\system32\catroot2\dberr.txt:3:CatalogDB: 20:43:38 08.06.2021:
DONE Adding Catalog File (78ms): Microsoft-Win
dows-Client-Desktop-Required-WOW64-
Package00~31bf3856ad364e35~amd64~~10.0.19041.1052.cat
. . .
```

Также по конвейеру можно передать массив строк, в котором будет производиться поиск, например:

```
PS C:\Users\andrv> "Windows","PowerShell" | Select-String -Pattern [PR]owerShell

PowerShell
```

Замена текста в файлах

Напомним, что замена текста в строках осуществляется оператором `-replace`, поддерживающим регулярные выражения *(см. гл. 8)*. Поэтому для замены текста в файле можно считать его содержимое в массив строк с помощью командлета `Get-Content`, обработать нужные строки оператором `-replace` и сохранить результат в том же или другом файле, используя командлет `Set-Content`.

Рассмотрим простой пример. Создадим файл C:\Test\test.txt из трех строк:

```
PS C:\> "11111", "2222", "33333" > C:\Test\test.txt
PS C:\> type C:\Test\test.txt
11111
2222
33333
```

Теперь заменим в этом файле все символы "1" на "а":

```
PS C:\> (Get-Content C:\Test\test.txt) | ForEach-Object {$_ -replace "1", "a"} |
Set-Content C:\Test\test.txt
```

Первая команда в конвейере здесь взята в скобки для того, чтобы командлет `Set-Content` начал работать лишь после того, как командлет `Get-Content` полностью прочитает файл (в противном случае может возникнуть ошибка при одновременном обращении к файлу C:\Test\test.txt при чтении и записи). Убедимся, что замена прошла успешно:

```
PS C:\> type C:\Test\test.txt
aaaaa
2222
33333
```

Работа с файлами-ярлыками

Ярлыки в Windows — это особый тип файлов, которые имеют расширение lnk или url и служат указателями на другие объекты (файлы, каталоги, веб-ресурсы, приложения, команды). В PowerShell нет отдельного встроенного командлета для работы с ярлыками, но можно создать ярлык с помощью методов внешних COM-объектов, имеющихся в операционной системе.

Доступ к COM-объектам из PowerShell

Технология Component Object Model (COM), разработанная компанией Microsoft еще в 1993 г., предоставляет двоичный (не зависящий от языка программирования) стандарт для программных компонентов. Объекты, созданные по этому стандарту и зарегистрированные определенным образом в операционной системе, могут использоваться в других приложениях без применения языков, средств или систем программирования, а только с помощью двоичных компонентов (исполняемых файлов или динамических библиотек). COM-объекты до настоящего времени продолжают использоваться в Windows и в приложениях Microsoft.

Идентифицируются COM-объекты по их *программным идентификаторам* (ProgID) — символьным псевдонимам, назначаемым при регистрации объектов в системе. ProgID имеют следующий вид:

```
Библиотека_типов.Класс
```

Перед точкой в ProgID стоит имя *библиотеки типов* (type library) для объекта, которая может существовать как в виде отдельного файла с расширением tlb, так и в виде части файла с исполняемым кодом объекта (библиотека типов, содержащая сведения о COM-объекте, регистрируется в системном реестре при установке приложения, публикующего этот объект). Довольно часто имя библиотеки типов совпадает с именем приложения, являющегося сервером COM-объектов.

После точки в ProgID указывается имя класса, содержащего свойства и методы COM-объекта, доступные для использования другими приложениями. Примеры

ProgID: `InternetExplorer.Application` (приложение Internet Explorer) `Word.Application` (приложение Microsoft Word), `WScript.Shell` (класс `Shell` из объектной модели сервера сценариев Windows Script Host).

В PowerShell имеется командлет `New-Object`, позволяющий, в частности, создавать экземпляры внешних COM-объектов, указывая соответствующий ProgID в качестве значения параметра `ComObject`. Например, экземпляр COM-объекта с программным идентификатором `WScript.Shell` создается следующим образом:

```
PS C:\> $shell = New-Object -ComObject WScript.Shell
```

Выполняя командлет `New-Object`, интерпретатор PowerShell через ProgID получает из системного реестра путь к файлам нужной библиотеки типов. Затем с помощью этой библиотеки в память загружается экземпляр запрашиваемого объекта, и его интерфейсы становятся доступными для использования в PowerShell. Ссылка на созданный объект сохраняется в переменной; в дальнейшем, используя эту переменную, мы получаем доступ к свойствам и методам объекта, а также к его вложенным объектам (если они имеются).

> ***Замечание***
> "Общение" с COM-объектами в PowerShell происходит с помощью соответствующих механизмов .NET Framework (создаются экземпляры .NET-класса `System.__ComObject`), поэтому на командлет `New-Object` действуют те же ограничения, какие действуют на платформу .NET во время вызова COM-объектов.

Объект *WScript.Shell*

Посмотрим, какие свойства и методы имеются у COM-объекта `WScript.Shell`. Для этого воспользуемся, как обычно, командлетом `Get-Member`, передав ему по конвейеру переменную `$Shell`, в которой сохранена ссылка на данный COM-объект:

```
PS C:\> $shell | Get-Member

 TypeName: System.__ComObject#{41904400-be18-11d3-a28b-00104bd350...

Name                     MemberType            Definition
----                     ----------            ----------
AppActivate              Method                bool AppActivate (...
CreateShortcut           Method                IDispatch CreateSh...
Exec                     Method                IWshExec Exec (str...
ExpandEnvironmentStrings Method                string ExpandEnvir...
LogEvent                 Method                bool LogEvent (Var...
Popup                    Method                int Popup (string,...
RegDelete                Method                void RegDelete (st...
RegRead                  Method                Variant RegRead (s...
RegWrite                 Method                void RegWrite (str...
Run                      Method                int Run (string, V...
SendKeys                 Method                void SendKeys (str...
Environment              ParameterizedProperty IWshEnvironment En...
```

```
CurrentDirectory        Property        string CurrentDire...
SpecialFolders          Property        IWshCollection Spe...
```

Как видим, здесь есть методы для запуска и управления приложениями, работы с системным реестром и журналами событий. Нам понадобится метод `CreateShortcut`, позволяющий создавать ярлыки для папок и файлов, и коллекция `SpecialFolders`, содержащая пути к специальным папкам Windows для текущего пользователя.

Создание ярлыка на рабочем столе

Для примера мы создадим на рабочем столе активного пользователя ярлык PSHome.lnk для папки, в которой установлена оболочка PowerShell.

Первая задача — определить путь к рабочему столу текущего пользователя. Для этого мы воспользуемся коллекцией `SpecialFolders`, сохранив ссылку на нее в переменной `$folders`:

```
PS C:\> $folders = $shell.SpecialFolders
PS C:\> $folders
C:\Users\Public\Desktop
C:\ProgramData\Microsoft\Windows\Start Menu
C:\ProgramData\Microsoft\Windows\Start Menu\Programs
C:\ProgramData\Microsoft\Windows\Start Menu\Programs\StartUp
C:\Users\andrv\OneDrive\Рабочий стол
C:\Users\andrv\AppData\Roaming
C:\Users\andrv\AppData\Roaming\Microsoft\Windows\Printer Shortcuts
C:\Users\andrv\AppData\Roaming\Microsoft\Windows\Templates
C:\Windows\Fonts
C:\Users\andrv\AppData\Roaming\Microsoft\Windows\Network Shortcuts
C:\Users\andrv\OneDrive\Рабочий стол
C:\Users\andrv\AppData\Roaming\Microsoft\Windows\Start Menu
C:\Users\andrv\AppData\Roaming\Microsoft\Windows\SendTo
C:\Users\andrv\AppData\Roaming\Microsoft\Windows\Recent
C:\Users\andrv\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup
C:\Users\andrv\Favorites
C:\Users\andrv\OneDrive\Documents
C:\Users\andrv\AppData\Roaming\Microsoft\Windows\Start Menu\Programs
```

Путь к рабочему столу пользователя сохраним в переменной `$desktop`:

```
PS C:\> $desktop = $folders.Item('desktop')
PS C:\> $desktop
C:\Users\andrv\OneDrive\Рабочий стол
```

Для создания ярлыка на рабочем столе вызовем метод `CreateShortcut`, передав в него путь к ярлыку (напомним, что внутри строки в двойных кавычках имя переменной заменяется на ее значение):

```
PS C:\> $lnk = $shell.CreateShortcut("$desktop\PSHome.lnk")
PS C:\> $lnk = $Shell.CreateShortcut("$Home\Рабочий стол\PSHome.lnk")
```

Посмотрим, какие свойства и методы имеет объект, сохраненный в переменной $lnk:

```
PS C:\> $lnk | Get-Member

   TypeName: System.__ComObject#{f935dc23-1cf0-11d0-adb9-00c04fd58a0b}

Name             MemberType Definition
----             ---------- ----------
Load             Method     void Load (string)
Save             Method     void Save ()
Arguments        Property   string Arguments () {get} {set}
Description      Property   string Description () {get} {set}
FullName         Property   string FullName () {get}
Hotkey           Property   string Hotkey () {get} {set}
IconLocation     Property   string IconLocation () {get} {set}
RelativePath     Property   string RelativePath () {set}
TargetPath       Property   string TargetPath () {get} {set}
WindowStyle      Property   int WindowStyle () {get} {set}
WorkingDirectory Property   string WorkingDirectory () {get} {set}
```

Для создания ярлыка на папку достаточно сохранить путь к ней в свойстве TargetPath и вызвать метод Save для сохранения ярлыка. Путь к домашней папке PowerShell хранится в специальной переменной $PSHome, поэтому выполняем следующие команды:

```
PS C:\> $lnk.TargetPath = $PSHome
PS C:\> $lnk.Save()
```

Задача решена, ярлык на рабочем столе активного пользователя создан.

Удаление некорректных ярлыков

Ярлыки (файлы с расширением lnk) являются важной частью графического интерфейса Windows. Однако если объект, на который ссылается ярлык, удалить или переместить, то соответствующий ярлык станет бесполезным.

Давайте разберем команды, с помощью которых можно найти и удалить такие некорректные ярлыки на рабочем столе. Как и в предыдущем разделе, создадим экземпляр класса WScript.Shell (переменная $shell):

```
PS C:\> $shell = New-Object -ComObject WScript.Shell
```

В переменную $desktop поместим путь к рабочему столу активного пользователя:

```
PS C:\> $folders = $shell.SpecialFolders
PS C:\> $desktop = $folders.Item("desktop")
```

Теперь с помощью конвейера командлетов dir (Get-ChildItem) и ForEach-Object переберем все файлы на рабочем столе с расширением lnk. Для каждого такого файла будем создавать с помощью метода CreateShortcut соответствующий объ-

ект-ярлык и проверять с помощью командлета `Test-Path` наличие объекта, на который ссылается этот ярлык (путь к этому объекту хранится в свойстве `TargetPath` объекта-ярлыка). Если объект не существует (командлет `Test-Path` возвращает `False`), то проверяемый файл с расширением lnk удаляется:

```
PS C:\> dir "$desktop\*.lnk" | ForEach-Object{
>> $shortcut = $shell.CreateShortcut($_.FullName);
>> if (-not (Test-Path $shortcut.TargetPath)) {del $_.FullName};
>> }
>>
```

Итоги

- В PowerShell имеются командлеты для выполнения стандартных операций над файлами и каталогами.
- Для создания файлов-ярлыков и управления ими можно использовать COM-объект `WScript.Shell`. Работать с COM-объектами позволяет командлет `New-Object` с параметром `ComObject`.

ГЛАВА 12

Обработка структурированных данных

В информационных системах повсюду используются данные, имеющие определенную структуру (конфигурационные файлы, журналы операций, данные от API внешних систем и т. д.). В PowerShell имеются встроенные средства для работы с наиболее распространенными форматами таких данных.

Работа с данными в формате CSV

В самом простом случае структурированный файл содержит табличные данные и представляет собой набор строк, состоящих из отдельных полей, разделенных друг от друга определенным символом. Часто в качестве разделителя выбирается запятая, поэтому подобные файлы называют *CSV-файлами* (Comma-Separated Values — значения, разделенные запятыми).

В первой строке CSV-файла могут быть перечислены названия полей, а в остальных содержатся сами данные. Например:

```
Name,Lastname,Age
Иван,Петров,22
Сергей,Андреев,34
Ольга,Белова,25
Мария,Никитина,41
```

Рассмотрим, каким образом с помощью PowerShell можно читать и изменять данные в CSV-файлах.

Чтение из CSV-файла

Для рассмотрения примеров создадим в каталоге C:\Script файл test.csv со следующими анкетными данными (имя, фамилия и возраст):

```
Name,Lastname,Age
Иван,Петров,22
Сергей,Андреев,34
Ольга,Белова,25
Мария,Никитина,41
```

В принципе, содержимое CSV-файла, как и любого другого текстового файла, можно считать в массив строк с помощью командлета `Get-Content`.

```
PS C:\Script> $names = Get-Content .\test.csv
PS C:\Script> Get-Member -InputObject $names

   TypeName: System.Object[]
. . .
PS C:\Script> $names | Get-Member

   TypeName: System.String
. . .
PS C:\Script> $names
Name,Lastname,Age
Иван,Петров,22
Сергей,Андреев,34
Ольга,Белова,25
Мария,Никитина,41
```

Однако командлет `Get-Content` ничего не знает о внутренней структуре строк, поэтому их содержимое придется разбирать (парсить), выделяя из строк составные части.

Поэтому для парсинга структурированных данных из CSV-файла лучше воспользоваться специальным командлетом `Import-Csv`, который сразу преобразовывает данные из файла в массив объектов с нужными полями.

Считаем с помощью `Import-Csv` данные из файла test.csv в переменную `$names` и выведем ее содержимое:

```
PS C:\Script> $names = Import-Csv -Path .\test.csv
PS C:\Script> $names

Name   Lastname Age
----   -------- ---
Иван   Петров   22
Сергей Андреев  34
Ольга  Белова   25
Мария  Никитина 41
```

Как видим, командлет `Import-Csv` понял, что в первой строке файла записан заголовок, задающий структуру каждой строки. Как и в случае с `Get-Content`, переменная `$names` является массивом объектов:

```
PS C:\Script> Get-Member -InputObject $names

   TypeName: System.Object[]

Name           MemberType          Definition
----           ----------          ----------
Count          AliasProperty       Count = Length
```

```
Add             Method                int IList.Add(System...
Address         Method                System.Object&, msco...
. . .
```

Однако теперь каждый элемент массива `$names` — это не простая строка, а объект типа `PSCustomObject`, имеющий поля `Name`, `LastName` и `Age`:

```
PS C:\Script> $names | Get-Member

   TypeName: System.Management.Automation.PSCustomObject

Name        MemberType   Definition
----        ----------   ----------
Equals      Method       bool Equals(System.Object obj)
GetHashCode Method       int GetHashCode()
GetType     Method       type GetType()
ToString    Method       string ToString()
Age         NoteProperty string Age=22
Lastname    NoteProperty string Lastname=Петров
Name        NoteProperty string Name=Иван
```

Имея такой массив, мы легко можем выполнять операции поиска, сортировки или группировки данных с помощью командлетов, рассмотренных в *главе 5*. Например, отсортируем записи из нашей таблицы по возрасту:

```
PS C:\Script> $names | Sort-Object Age

Name   Lastname Age
----   -------- ---
Иван   Петров   22
Ольга  Белова   25
Сергей Андреев  34
Мария  Никитина 41
```

Названия полей при импорте можно изменить с помощью параметра `-Header`:

```
PS C:\Script> Import-Csv -Path .\test.csv -Header 'Имя', 'Фамилия', 'Возраст'

Имя    Фамилия  Возраст
---    -------  -------
Name   Lastname Age
Иван   Петров   22
Сергей Андреев  34
Ольга  Белова   25
Мария  Никитина 41
```

По умолчанию `Import-Csv` считает разделителем запятую. Если в файле в качестве разделителя используется другой символ, то его нужно указать в параметре `-Delimiter`. Например, заменим в файле test.csv запятые на символ точки с запятой:

```
PS C:\Script> (Get-Content .\test.csv -Raw).Replace(',', ';') | Set-Content
.\test.csv
```

```
PS C:\Script> Get-Content .\test.csv
Name;Lastname;Age
Иван;Петров;22
Сергей;Андреев;34
Ольга;Белова;25
Мария;Никитина;41
```

Импортировать такой файл можно следующей командой:

```
PS C:\Script> Import-Csv -Path .\test.csv -Delimiter ';'

Name   Lastname Age
----   -------- ---
Иван   Петров   22
Сергей Андреев  34
Ольга  Белова   25
Мария  Никитина 41
```

Замечание

Если значения в файле разделены символами табуляции, то в качестве значения параметра `-Delimiter` нужно указать `` `t ``.

Запись в CSV-файл

С помощью командлета `Export-Csv` можно сформировать CSV-файл из массива объектов PowerShell.

Например, получим список всех запущенных на локальном компьютере служб (статус, имя и отображаемое имя):

```
PS C:\Script> Get-Service | Where-Object Status -eq Running | Select-Object
Status, Name, DisplayName

 Status Name                        DisplayName
 ------ ----                        -----------
Running AdobeARMservice              Adobe Acrobat Update Service
Running AMD External Events Utility  AMD External Events Utility
Running AMD FUEL Service             AMD FUEL Service
Running Appinfo                      Application Information
Running AppXSvc                      AppX Deployment Service (AppXSVC)
Running AudioEndpointBuilder         Windows Audio Endpoint Builder
Running Audiosrv                     Windows Audio
Running BFE                          Base Filtering Engine
. . .
```

Сохраним этот список в переменную `$services`:

```
PS C:\Script> $services = Get-Service | Where-Object Status -eq Running |
Select-Object Status, Name, DisplayName
```

Экспортируем массив объектов `$services` в CSV-файл services.csv:

```
PS C:\Script> $services | Export-Csv -Path .\services.csv
```

Проверим содержимое файла services.csv:

```
PS C:\Script> Get-Content .\services.csv
#TYPE Selected.System.ServiceProcess.ServiceController
"Status","Name","DisplayName"
"Running","AdobeARMservice","Adobe Acrobat Update Service"
"Running","AMD External Events Utility","AMD External Events Utility"
"Running","AMD FUEL Service","AMD FUEL Service"
. . .
```

Как видим, файл создался корректно, но в самой первой его строке указан тип объектов, из которых он получен (в нашем случае это объекты типа `Selected.System.ServiceProcess.ServiceController`). Если эта информация в файле не нужна, то при вызове `Export-Csv` следует указать параметр `-NoTypeInformation`:

```
PS C:\Script> $services | Export-Csv -Path .\services.csv -NoTypeInformation
```

В этом случае в первую строку в файле будут записаны названия полей:

```
PS C:\Script> Get-Content .\services.csv
"Status","Name","DisplayName"
"Running","AdobeARMservice","Adobe Acrobat Update Service"
"Running","AMD External Events Utility","AMD External Events Utility"
"Running","AMD FUEL Service","AMD FUEL Service"
"Running","Appinfo","Application Information"
. . .
```

Обработка данных без обращения к файлу

В PowerShell есть командлеты `ConvertFrom-Csv` и `ConvertTo-Csv`, которые действуют аналогично `ImportFrom-Csv` и `ExportTo-Csv`, но при этом не обращаются к внешним файлам.

Например, повторим наш пример со списком запущенных служб:

```
PS C:\Script> $services = Get-Service | Where-Object Status -eq Running |
Select-Object Status, Name, DisplayName
```

Преобразуем массив объектов `$services` в CSV:

```
PS C:\Script> $csv = $services | ConvertTo-Csv
```

В переменной `$csv` будет содержаться строка в CSV-формате:

```
PS C:\Script> $csv | Get-Member

   TypeName: System.String
. . .

PS C:\Script> $csv
#TYPE Selected.System.ServiceProcess.ServiceController
"Status","Name","DisplayName"
"Running","AdobeARMservice","Adobe Acrobat Update Service"
```

```
"Running","AMD External Events Utility","AMD External Events Utility"
"Running","AMD FUEL Service","AMD FUEL Service"
. . .
```

Теперь конвертируем строку `$csv` в переменную `$objs`:

```
PS C:\Script> $objs = $csv | ConvertFrom-Csv
```

Массив `$objs` будет содержать объекты типа `Selected.System.ServiceProcess.ServiceController`, который был указан в комментарии `#TYPE`:

```
PS C:\Script> Get-Member -InputObject $objs

   TypeName: System.Object[]
. . .
PS C:\Script> $objs | Get-Member

   TypeName: CSV:Selected.System.ServiceProcess.ServiceController
. . .

PS C:\Script> $objs

Status  Name                          DisplayName
------  ----                          -----------
Running AdobeARMservice               Adobe Acrobat Update Service
Running AMD External Events Utility   AMD External Events Utility
Running AMD FUEL Service              AMD FUEL Service
. . .
```

Обработка данных в JSON-формате

Данные в формате JSON (JavaScript Object Notation) широко используются в веб-разработке, а также для хранения настроек приложений. В PowerShell работать с JSON можно с помощью командлетов `ConvertFrom-Json` и `ConvertTo-Json`.

> ***Замечание***
> В отличие от CSV, специальных командлетов для импорта/экспорта JSON-файлов в PowerShell нет.

Давайте преобразуем в JSON содержимое нашего CSV-файла C:\Script\test.csv. Для этого мы сначала импортируем этот файл в массив объектов PowerShell, а затем конвертируем этот массив в JSON, передав его по конвейеру командлету `ConvertTo-Json`:

```
PS C:\Script> Import-Csv -Path .\test.csv

Name   Lastname Age
----   -------- ---
Иван   Петров   22
Сергей Андреев  34
Ольга  Белова   25
Мария  Никитина 41
```

```
PS C:\Script> Import-Csv -Path .\test.csv | ConvertTo-Json
[
    {
        "Name":  "Иван",
        "Lastname":  "Петров",
        "Age":  "22"
    },
    {
        "Name":  "Сергей",
        "Lastname":  "Андреев",
        "Age":  "34"
    },
    {
        "Name":  "Ольга",
        "Lastname":  "Белова",
        "Age":  "25"
    },
    {
        "Name":  "Мария",
        "Lastname":  "Никитина",
        "Age":  "41"
    }
]
```

Убедимся, что в результате получается строка:

```
PS C:\Script> $json = Import-Csv -Path .\test.csv | ConvertTo-Json
PS C:\Script> Get-Member -InputObject $json

   TypeName: System.String
. . .
```

По умолчанию JSON-строка формируется в структурированном виде, удобном для восприятия человеком: вложенность элементов оформляется дополнительными отступами, поля разделяются переводами строк. Если JSON обрабатывается автоматически, то от лишних пробелов и переводов строк для форматирования можно отказаться с помощью параметра `-Compress`. В этом случае поля и значения записываются подряд и JSON-строка будет иметь минимальную длину:

```
PS C:\Script> Import-Csv -Path .\test.csv | ConvertTo-Json -Compress
[{"Name":"Иван","Lastname":"Петров","Age":"22"},{"Name":"Сергей","Lastname":
"Андреев","Age":"34"},{"Name":"Ольга","Lastname":"Белова","Age":"25"},{"Name":"
Мария","Lastname":"Никитина","Age":"41"}]
```

Для создания файла в формате JSON полученную в результате конвертации строку нужно сохранить с помощью командлетов `Set-Content` или `Out-File`:

```
PS C:\Script> Import-Csv -Path .\test.csv | ConvertTo-Json | Set-Content
.\test.json
```

Отметим, что, в отличие от экспорта в CSV-файл, при конвертации массива объектов в JSON информация о первоначальном типе объектов теряется. Для иллюстра-

ции этого вновь создадим массив `$services`, в котором будет храниться список запущенных на локальном компьютере служб:

```
PS C:\Script> $services = Get-Service | Where-Object Status -eq Running |
Select-Object Status, Name, DisplayName
```

Элементами этого массива будут объекты типа `Selected.System.ServiceProcess.ServiceController`:

```
PS C:\Script> $services | Get-Member

   TypeName: Selected.System.ServiceProcess.ServiceController
. . .
```

Преобразуем массив `$services` в JSON и сохраним результат в переменной `$json_services`:

```
PS C:\Script> $json_services = $services | ConvertTo-Json
```

Теперь выполним обратное преобразование — конвертируем строку `$json_services` в массив `$objs` с помощью командлета `ConvertFrom-Json`:

```
PS C:\Script> $objs = $json_services | ConvertFrom-Json
```

Проверим, какой тип имеют элементы массива `$objs`:

```
PS C:\Users\andrv> $objs | Get-Member

   TypeName: System.Management.Automation.PSCustomObject

Name        MemberType   Definition
----        ----------   ----------
Equals      Method       bool Equals(System.Object obj)
GetHashCode Method       int GetHashCode()
GetType     Method       type GetType()
ToString    Method       string ToString()
DisplayName NoteProperty string DisplayName=Adobe Acrob...
Name        NoteProperty string Name=AdobeARMservice
Status      NoteProperty int Status=4
```

Как видим, в результате преобразования из JSON-строки `$json_services` создаются объекты типа `System.Management.Automation.PSCustomObject`. Эти объекты имеют те же поля, что были у первоначальных объектов типа `Selected.System.ServiceProcess.ServiceController`, однако специфические методы для управления службами в преобразованных из JSON объектах будут недоступны.

Итоги

- Для работы с наиболее распространенными форматами структурированных данных в PowerShell используются командлеты `ConvertFrom-*`, `ConvertTo-*`.
- Читать и изменять CSV-файлы можно с помощью командлетов `Import-Csv` и `Export-Csv`.

ГЛАВА 13

Управление процессами, службами и серверами автоматизации

В данной главе мы рассмотрим вопросы, связанные с работой из оболочки PowerShell с двумя важными компонентами операционной системы Windows: запущенными процессами и службами, работающими в фоновом режиме.

Управление процессами

Основным графическим инструментом для управления процессами на локальной машине является диспетчер задач Windows (для запуска диспетчера задач можно нажать комбинацию клавиш <Ctrl>+<Shift>+<Esc>), который отображает информацию о выполняющихся процессах, позволяет останавливать их и задавать приоритет выполнения (рис. 13.1).

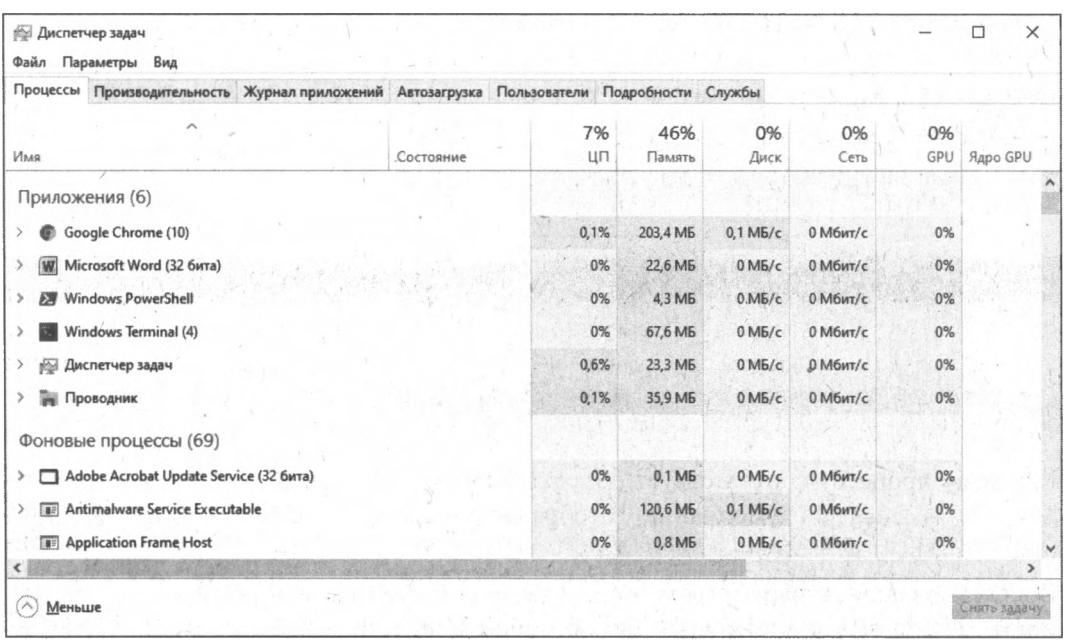


Рис. 13.1. Диспетчер задач Windows

Также в Windows имеются консольные утилиты `tasklist` (просмотр списка процессов, запущенных на локальном или удаленном компьютере) и `taskkill` (остановка процесса):

```
PS C:\Users\andrv> tasklist /?

TASKLIST [/S <система> [/U <имя пользователя> [/P [<пароль>]]]]
         [/M [<модуль>] | /SVC | /V] [/FI <фильтр>] [/FO <формат>] [/NH]

Описание:
    Отображает список процессов, которые
    сейчас выполняются на локальном или удаленном компьютере.
. . .
PS C:\Users\andrv> taskkill /?

TASKKILL [/S <система> [/U <пользователь> [/P [<пароль>]]]]
         { [/FI <фильтр>] [/PID <процесс> | /IM <образ>] } [/T] [/F]

Описание:
    Завершает процесс по его ID (PID) или имени образа.
. . .
```

Рассмотрим способы решения с помощью PowerShell некоторых типичных задач, связанных с процессами.

Просмотр списка процессов

В PowerShell получить список запущенных процессов позволяет командлет `Get-Process`. Если запустить этот командлет без параметров, то на экран будет выведена информация обо всех запущенных процессах:

```
PS C:\> Get-Process

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    109       5     1128        792    32     0.06   1360 alg
    506       6     2308       3604    29    20.34    628 csrss
     67       3      832        436    29     1.52   1736 ctfmon
    553      17    17220      11992    94    69.16   1164 explorer
     33       2     3280        400    35     2.41   3192 Far
. . .
```

Каждому процессу соответствует объект типа `System.Diagnostics.Process`; командлет `Get-Process` по умолчанию отображает несколько свойств этих объектов (табл. 13.1).

Воспользовавшись параметром `-Name` (данный параметр используется по умолчанию), можно вывести информацию об одном или нескольких процессах с определенными именами (при этом в именах можно применять подстановочные симво-

Таблица 13.1. *Свойства, отображаемые командлетом Get-Process по умолчанию*

Свойство	Описание
Handles	Счетчик дескрипторов (свойство HandleCount объекта System.Diagnostics.Process)
NPM	Объем невыгружаемого пула (свойство NonpagedSystemMemorySize объекта System.Diagnostics.Process)
PM	Объем используемой виртуальной памяти (свойство PagedSystemMemorySize объекта System.Diagnostics.Process)
WS	Объем используемой памяти (свойство WorkingSet объекта System.Diagnostics.Process)
VM	Объем выгружаемого пула (свойство VirtualMemorySize объекта System.Diagnostics.Process)
CPU	Процессорное время в секундах, затраченное процессом (свойство TotalProcessorTime.TotalSeconds объекта System.Diagnostics.Process)
ID	Идентификатор процесса (свойство Id объекта System.Diagnostics.Process)
ProcessName	Имя процесса (свойство ProcessName объекта System.Diagnostics.Process)

лы). Например, следующая команда выведет все процессы, имена которых начинаются на букву s:

```
PS C:\> Get-Process s*

Handles  NPM(K)    PM(K)  WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----  ----- -----   ------     -- -----------
    106       2      380     32    11     0.03   1880 scardsvr
    134       4     1876    200    88     0.11   3640 sdlaunch
    267       6     1504   1236    46     3.67    696 services
     19       1      164     52     4     0.06    560 smss
     71       2     1820    588    30     0.14   1352 SOUNDMAN
    297       6     6924   1768    54     5.05   1304 spoolsv
    134       3     1424    824    33     0.16    876 svchost
    314      13     1768   1280    36     0.75    944 svchost
   1185      45    12124   6532    83    14.59   1040 svchost
     59       3     1124    740    27     1.23   1076 svchost
    224       7     2356    788    38     0.27   1204 svchost
    130       4     2392   1324    35     0.81   1952 svchost
    517       0        0     52     1    26.75      4 System
```

Для сортировки или фильтрации списка процессов применяются, как обычно, командлеты Sort-Object, Where-Object и Select-Object. Например, следующий конвейер команд выводит на экран пять процессов, которые тратят наибольшее количество процессорного времени:

```
PS C:\> Get-Process | Sort-Object CPU -Descending | Select-Object -First 5

Handles  NPM(K)    PM(K)  WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----  ----- -----   ------     -- -----------
    619      19    21044  23156   110   171.19   1164 explorer
    790      50    59228  11788   191    93.17   1520 kavsvc
    476      14    10320  11120   129    88.25   2860 OUTLOOK
    448      19    61204  23412   165    83.59   3884 wjview
    411      12     9780   8920   250    61.95   3484 WINWORD
```

Для просмотра полной информации об одном или нескольких процессах можно воспользоваться командлетом `Format-List`, отобразив все свойства объекта `System.Diagnostics.Process`. Например:

```
PS C:\> Get-Process outlook | Format-List *

__NounName                  : Process
Name                        : OUTLOOK
Handles                     : 472
VM                          : 135835648
WS                          : 10010624
PM                          : 10584064
NPM                         : 14552
Path                        : C:\Program Files\Microsoft Office\Offic
                              e\OUTLOOK.EXE
Company                     : Microsoft Corporation
CPU                         : 88.875
FileVersion                 : 9.0.2416
ProductVersion              : 9.0.2416
Description                 : Microsoft Outlook
Product                     : Microsoft Outlook
Id                          : 2860
PriorityClass               : Normal
HandleCount                 : 472
WorkingSet                  : 10010624
PagedMemorySize             : 10584064
PrivateMemorySize           : 10584064
VirtualMemorySize           : 135835648
TotalProcessorTime          : 00:01:28.8750000
BasePriority                : 8
ExitCode                    :
HasExited                   : False
ExitTime                    :
Handle                      : 2276
MachineName                 : .
MainWindowHandle            : 263442
MainWindowTitle             : Входящие - Microsoft Outlook
MainModule                  : System.Diagnostics.ProcessModule (OUTLO OK.EXE)
MaxWorkingSet               : 1413120
```

```
MinWorkingSet              : 204800
Modules                    : {OUTLOOK.EXE, ntdll.dll, kernel32.dll,
                             OUTLLIB.dll...}
NonpagedSystemMemorySize   : 14552
NonpagedSystemMemorySize64 : 14552
PagedMemorySize64          : 10584064
PagedSystemMemorySize      : 109464
PagedSystemMemorySize64    : 109464
PeakPagedMemorySize        : 11055104
PeakPagedMemorySize64      : 11055104
PeakWorkingSet             : 22802432
PeakWorkingSet64           : 22802432
PeakVirtualMemorySize      : 152793088
PeakVirtualMemorySize64    : 152793088
PriorityBoostEnabled       : True
PrivateMemorySize64        : 10584064
PrivilegedProcessorTime    : 00:00:51.0156250
ProcessName                : OUTLOOK
ProcessorAffinity          : 1
Responding                 : True
SessionId                  : 0
StartInfo                  : System.Diagnostics.ProcessStartInfo
StartTime                  : 24.04.2008 8:05:59
SynchronizingObject        :
Threads                    : {1756, 316, 3108, 808...}
UserProcessorTime          : 00:00:37.8593750
VirtualMemorySize64        : 135835648
EnableRaisingEvents        : False
StandardInput              :
StandardOutput             :
StandardError              :
WorkingSet64               : 10010624
Site                       :
Container                  :
```

Для получения списка процессов на удаленных машинах нужно указать их имена или IP-адреса в качестве значения параметра `-ComputerName`.

Определение библиотек, используемых процессом

У объектов `System.Diagnostics.Process`, возвращаемых командлетом `Get-Process`, есть свойство `Modules`, содержащее список динамических библиотек, используемых соответствующими процессами. Для просмотра этого списка можно обрабатывать данные объекты командлетом `Select-Object` с параметром `-ExpandProperty`.

Например, следующий конвейер команд выводит список всех динамических библиотек, используемых оболочкой PowerShell (процесс с именем powershell):

```
PS C:\> Get-Process powershell | Select-Object -expandproperty modules |
Format-Table

Size(K) ModuleName     FileName
------- ----------     --------
    452 powershell.exe
C:\WINDOWS\System32\WindowsPowerShell\v1.0\powershell.exe
   2004 ntdll.dll      C:\WINDOWS\SYSTEM32\ntdll.dll
    756 KERNEL32.DLL   C:\WINDOWS\System32\KERNEL32.DLL
   2848 KERNELBASE.dll C:\WINDOWS\System32\KERNELBASE.dll
    632 msvcrt.dll     C:\WINDOWS\System32\msvcrt.dll
    820 OLEAUT32.dll   C:\WINDOWS\System32\OLEAUT32.dll
    628 msvcp_win.dll  C:\WINDOWS\System32\msvcp_win.dll
   1024 ucrtbase.dll   C:\WINDOWS\System32\ucrtbase.dll
    116 ATL.DLL        C:\WINDOWS\SYSTEM32\ATL.DLL
   3412 combase.dll    C:\WINDOWS\System32\combase.dll
. . .
```

Остановка процессов

Остановить процесс на локальной машине позволяет командлет `Stop-Process`, имеющий псевдоним `kill`. При этом по умолчанию используется параметр `Id`, требующий указания идентификатора останавливаемого процесса (напомним, что идентификатор процесса можно узнать с помощью командлета `Get-Process`). Например, следующая команда остановит процесс с идентификатором 764:

```
PS C:\> Stop-Process 764
```

Для остановки процесса с определенным именем нужно воспользоваться параметром `-Name` командлета `Stop-Process`. Например, следующая команда остановит все процессы с именем notepad:

```
PS C:\> Stop-Process -Name notepad
```

Параметр `-Confirm` командлета `Stop-Process` включает режим подтверждения при остановке процессов. При этом на экране отображается как имя, так и идентификатор останавливаемого процесса, например:

```
PS C:\Users\andrv> Stop-Process -Name notepad -Confirm

Подтверждение
Вы действительно хотите выполнить это действие?
Выполнение операции "Stop-Process" над целевым объектом "notepad (2812)".
[Y] Да - Y  [A] Да для всех - A  [N] Нет - N  [L] Нет для всех - L  [S]
Приостановить - S  [?] Справка
(значением по умолчанию является "Y"):Y

Подтверждение
Вы действительно хотите выполнить это действие?
Выполнение операции "Stop-Process" над целевым объектом "notepad (13984)".
[Y] Да - Y  [A] Да для всех - A  [N] Нет - N  [L] Нет для всех - L  [S]
Приостановить - S  [?] Справка
(значением по умолчанию является "Y"):Y
```

По умолчанию командлет `Stop-Process` не передает далее по конвейеру объекты, соответствующие останавливаемым процессам, и поэтому на экране ничего не отображается. Для остановки процессов с выводом информации нужно указать параметр `-PassThru`, например:

```
PS C:\> Stop-Process -Name notepad -PassThru

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     26       2      624       2172    24            2224 notepad
```

> ***Замечание***
> Для остановки процессов на удаленных компьютерах можно воспользоваться утилитой командной строки `taskkill` или методом `Terminate()` WMI-объекта `Win32_Process`.

Запуск процессов

Для запуска процесса из оболочки PowerShell можно просто указать путь к соответствующему исполняемому файлу, например:

```
PS C:\> C:\WINDOWS\system32\notepad.exe
```

В результате выполнения этой команды запустится стандартный Блокнот.

> ***Замечание***
> Напомним, что для выполнения файла, путь к которому содержит пробелы, нужно использовать оператор запуска `&`, а путь к файлу заключить в кавычки.

Также запустить процесс можно с помощью командлета `Invoke-Item`, указав в качестве аргумента путь к нужному файлу. Параметр `-Confirm` позволяет запускать процесс только после подтверждения пользователя:

```
PS C:\> Invoke-Item C:\WINDOWS\system32\notepad.exe -Confirm

Подтверждение
Вы действительно хотите выполнить это действие?
Выполнение операции "Вызов элемента" над целевым объектом "Элемент:
C:\WINDOWS\system32\notepad.exe".
[Y] Да - Y  [A] Да для всех - A  [N] Нет - N  [L] Нет для всех - L  [S]
Приостановить - S  [?] Справка
(значением по умолчанию является "Y"):Y
```

Однако командлет `Invoke-Item` не возвращает объект, соответствующий запущенному процессу, в силу чего мы не можем программно определить, скажем, идентификатор, назначенный системой данному процессу.

Поэтому процессы на локальной машине лучше запускать с помощью специального командлета `Start-Process`. Путь к исполняемому файлу или скрипту для запуска указывается в параметре `-FilePath`:

```
PS C:\Users\andrv> Start-Process -FilePath notepad.exe
```

Параметры команды Start-Process позволяют передавать аргументы запускаемому процессу, перенаправлять его входные и выходные потоки, задавать учетную запись и определять тип окна для процесса и т. д.

По умолчанию Start-Process только запускает процесс, ничего при этом не возвращая. Если нужно вернуть объект, соответствующий новому процессу, то следует использовать ключ -PassThru:

```
PS C:\Users\andrv> $process = Start-Process -FilePath notepad.exe -PassThru
PS C:\Users\andrv> $process.GetType().FullName
System.Diagnostics.Process
```

Отметим, что с помощью Start-Process можно не только запускать исполняемые файлы, но и выполнять действия, ассоциированные с файлами других типов. Например, создадим текстовый файл 1.txt и запустим его на печать, выполнив команду Start-Process с указанием действия Print:

```
PS C:\Users\andrv> "File content" > 1.txt
PS C:\Users\andrv> Start-Process -FilePath .\1.txt -Verb Print
```

Рассмотрим еще метод Create() WMI-класса Win32_Process, с помощью которого можно запускать процессы как на локальном, так и на удаленном компьютере. Данный метод является конструктором класса Win32_Process, для его вызова можно использовать спецификатор типа [wmiClass]. Например, запустим Блокнот:

```
PS C:\> $a = ([wmiClass] "win32_process").Create("notepad.exe")
```

С помощью командлета Get-Member посмотрим, какие свойства имеет объект, сохраненный в переменной $a:

```
PS C:\> $a | Get-Member

   TypeName: System.Management.ManagementBaseObject#\__PARAMETERS

Name              MemberType     Definition
----              ----------     ----------
PSComputerName    AliasProperty  PSComputerName = __SERVER
ProcessId         Property       uint32 ProcessId {get;set;}
ReturnValue       Property       uint32 ReturnValue {get;set;}
__CLASS           Property       string __CLASS {get;set;}
__DERIVATION      Property       string[] __DERIVATION {get;set;}
__DYNASTY         Property       string __DYNASTY {get;set;}
__GENUS           Property       int __GENUS {get;set;}
__NAMESPACE       Property       string __NAMESPACE {get;set;}
__PATH            Property       string __PATH {get;set;}
__PROPERTY_COUNT  Property       int __PROPERTY_COUNT {get;set;}
__RELPATH         Property       string __RELPATH {get;set;}
__SERVER          Property       string __SERVER {get;set;}
__SUPERCLASS      Property       string __SUPERCLASS {get;set;}
```

Как видим, нам доступно свойство ProcessId, в котором хранится идентификатор запущенного процесса.

Изменение приоритетов выполнения процесса

У объекта `System.Diagnostics.Process` имеется метод `set_PriorityClass`, позволяющий изменять приоритет выполнения процесса. В качестве параметра данного метода следует указать одну из следующих строк: Normal (средний), Idle (низкий), High (высокий), RealTime (реального времени), BelowNormal (ниже среднего), AboveNormal (выше среднего).

Например, запустим новый экземпляр Блокнота и с помощью метода `get_PriorityClass` проверим текущий приоритет данного процесса:

```
PS C:\Users\andrv> $b = Start-Process -FilePath notepad.exe -PassThru
PS C:\> $b.get_PriorityClass()
Normal
```

Повысим теперь приоритет до высокого:

```
PS C:\> $b.set_PriorityClass("High")
PS C:\> $b.get_PriorityClass()
High
```

Завершение неотвечающих процессов

У объектов `System.Diagnostics.Process`, соответствующих запущенным процессам, есть свойство `Responding`, которое принимает значение `False`, если процесс не реагирует на запросы. Для завершения всех неотвечающих приложений нужно выделить их из общей массы с помощью командлета `Where-Object`, после чего остановить с помощью командлета `Stop-Process`:

```
PS C:\> Get-Process | Where-Object -not Responding | Stop-Process
```

Управление службами

В операционной системе Windows *служба* (service) — это процесс, который запускается на машине в фоновом режиме для выполнения определенных действий в ответ на запросы пользователей. Например, служба веб-сервера работает в фоновом режиме, ожидая получения HTTP-запросов от клиентов. При получении такого запроса служба отвечает на него, посылая клиенту запрошенный файл или выполняя определенное действие. Состав служб зависит от версии операционной системы и установленных в ней приложений.

Основным инструментом для администрирования служб в графическом режиме на локальном компьютере является консоль **Службы**, которая находится в программной группе **Администрирование** панели управления (рис. 13.2).

В качестве утилиты, с помощью которой можно управлять службами как на локальных, так и на удаленных машинах, используется консоль **Управление компьютером**. Также в Windows входят стандартные утилиты командного интерпретатора cmd.exe для администрирования служб: `net start` и `net stop`.

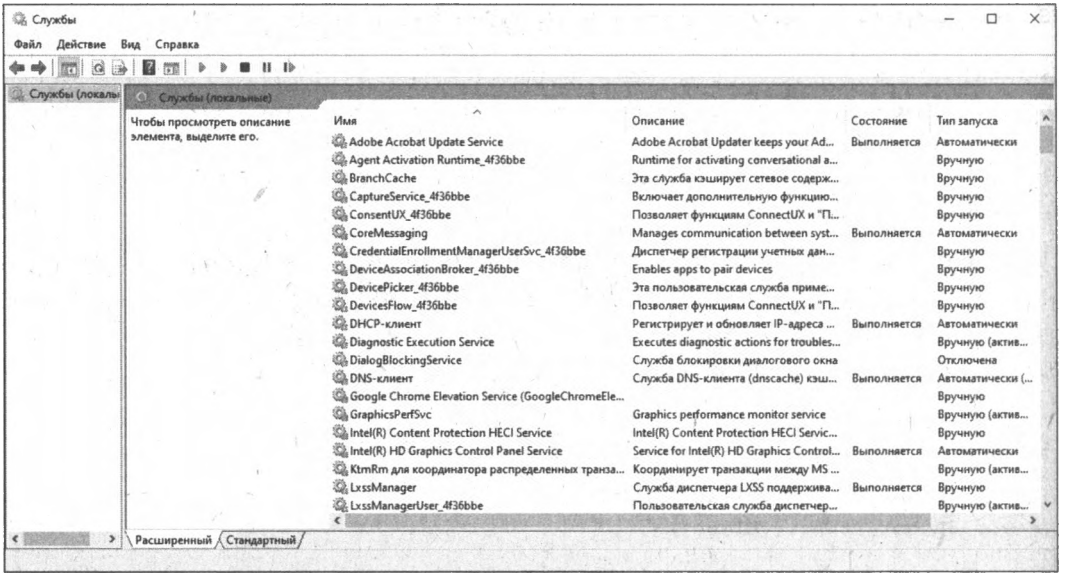


Рис. 13.2. Консоль для управления службами

Рассмотрим способы решения с помощью PowerShell некоторых типичных задач, связанных со службами.

Просмотр списка служб

Получить список служб, зарегистрированных на локальном компьютере, позволяет командлет `Get-Service`:

```
PS C:\> Get-Service

Status   Name              DisplayName
------   ----              -----------
Running  Alerter           Оповещатель
Running  ALG               Служба шлюза уровня приложения
Stopped  AppMgmt           Управление приложениями
Stopped  aspnet_state      ASP.NET State Service
Running  AudioSrv          Windows Audio
Stopped  BITS              Фоновая интеллектуальная служба пер...
Stopped  Browser           Обозреватель компьютеров
Stopped  cisvc             Служба индексирования
Stopped  ClipSrv           Сервер папки обмена
. . .
```

Как видим, по умолчанию выводится имя службы (колонка `Name`), ее отображаемое имя (колонка `DisplayName`) и состояние (колонка `Status`).

Командлет `Get-Service` имеет параметры `-Name` и `-DisplayName`, поддерживающие шаблонные символы. Например, следующая команда выводит список служб, имена которых начинаются с символов win:

```
PS C:\Users\andrv> Get-Service -Name 'win*'

Status   Name               DisplayName
------   ----               -----------
Running  WinDefend          Microsoft Defender Antivirus Service
Running  WinHttpAutoProx... WinHTTP Web Proxy Auto-Discovery Se...
Running  Winmgmt            Windows Management Instrumentation
Stopped  WinRM              Windows Remote Management (WS-Manag...
```

Если нужно вывести на экран только работающие в данный момент службы, то нужно отфильтровать объекты, у которых значением свойства `Status` является строка Running:

```
PS C:\> Get-Service | Where-Object Status -eq "Running"

Status   Name               DisplayName
------   ----               -----------
Running  Alerter            Оповещатель
Running  ALG                Служба шлюза уровня приложения
Running  AudioSrv           Windows Audio
Running  CryptSvc           Службы криптографии
. . .
```

Для просмотра служб, зарегистрированных на удаленном компьютере, нужно указать его имя в качестве значения параметра `-ComputerName`.

Остановка и приостановка служб

Локальную службу можно остановить с помощью командлета `Stop-Service` (при этом PowerShell должен быть запущен от имени администратора). Параметр `-Name` задает имя службы для остановки (здесь можно использовать подстановочные знаки), параметр `-Force` позволяет остановить указанную службу вместе со всеми службами, зависящими от нее.

Например, следующая команда остановит службу сервера `LanmanServer` (эта служба поддерживает общий доступ к файлам и принтерам по сети) со всеми зависящими от нее службами:

```
PS C:\> Stop-Service -Name LanmanServer -Force
```

По умолчанию командлет `Stop-Service` не передает далее по конвейеру объекты, соответствующие останавливаемым службам, и поэтому на экране ничего не отображается. Для остановки служб с выводом информации нужно указать параметр `-PassThru`, например:

```
PS C:\> Stop-Service -Name LanmanServer -Force -PassThru

Status   Name               DisplayName
------   ----               -----------
Stopped  LanmanServer       Сервер
```

Командлет `Suspend-Service` позволяет приостановить работу одной или нескольких служб, имена которых задаются в качестве значения параметра `-Name`. При приоста-

новке (временной остановке) службы она продолжает выполняться, однако ее действия приостанавливаются до поступления команды на возобновление работы *(см. следующий раздел)*.

Следует учитывать, что не всякую службу можно приостановить. У объектов `System.ServiceProcess.ServiceController`, соответствующих службам, имеется логическое свойство `CanPauseAndContinue`, которое равно `True`, если служба может быть приостановлена. Следующая команда приостановит все службы, для которых это возможно:

```
PS C:\> Get-Service | Where-Object {$_.CanPauseAndContinue} | Suspend-Service -
PassThru

Status   Name               DisplayName
------   ----               -----------
Paused   Irmon              Монитор инфракрасной связи
Paused   lanmanserver       Сервер
Paused   lanmanworkstation  Рабочая станция
Paused   Schedule           Планировщик заданий
Paused   seclogon           Вторичный вход в систему
Paused   ShellHWDetection   Определение оборудования оболочки
Paused   TapiSrv            Телефония
Paused   winmgmt            Инструментарий управления Windows
```

Замечание

Для остановки или приостановки служб на удаленном компьютере можно воспользоваться соответственно методами `StopService` и `PauseService` WMI-класса `Win32_Service`.

Запуск и перезапуск служб

Запустить службу на локальном компьютере можно с помощью командлета `Start-Service`. В качестве значения параметра `-Name` указывается имя запускаемой службы. Как и в предыдущих командлетах `*-Service`, после запуска службы на экран не выводится никакого сообщения; для исправления ситуации можно использовать параметр `-PassThru`. Например, следующая команда запускает службу `LanmanServer` и выводит на экран информацию об этой службе:

```
PS C:\> Start-Service -Name LanmanServer -PassThru

Status   Name               DisplayName
------   ----               -----------
Running  Spooler            Диспетчер очереди печати
```

Командлет `Restart-Service` выполняет перезапуск (т. е. остановку и запуск) служб, указанных в качестве значения параметра `-Name` или полученных по конвейеру от других командлетов. Например, следующий конвейер команд перезапускает все сетевые службы, имя которых начинается со строки net:

```
PS C:\> Get-Service -Name net* | Restart-Service
```

Изменение параметров службы

В PowerShell имеется командлет `Set-Service`, позволяющий редактировать некоторые параметры служб на локальном или удаленном компьютере:

- отображаемое имя (параметр `DisplayName`);
- описание (параметр `Description`);
- тип запуска (параметр `StartupType`), возможные значения — `Automatic` (служба запускается автоматически), `Manual` (служба запускается вручную), `Disabled` (служба отключена).

Имена изменяемых служб указываются в качестве значения параметра `-Name` (в именах могут присутствовать подстановочные знаки).

Для примера мы изменим режим запуска службы сервера `LanmanServer`. Посмотрим все свойства этой службы:

```
PS C:\> Get-Service LanmanServer | Format-List *

Name                : LanmanServer
RequiredServices    : {SamSS, Srv2}
CanPauseAndContinue : False
CanShutdown         : False
CanStop             : True
DisplayName         : Сервер
DependentServices   : {}
MachineName         : .
ServiceName         : LanmanServer
ServicesDependedOn  : {SamSS, Srv2}
ServiceHandle       : SafeServiceHandle
Status              : Running
ServiceType         : Win32OwnProcess, Win32ShareProcess
StartType           : Automatic
Site                :
Container           :
```

Как видим, сейчас эта служба запускается автоматически (значение свойства `StartType` равно `Automatic`). Изменим с помощью командлета `Set-Service` режим запуска службы `LanmanServer` с автоматического на ручной:

```
PS C:\> Set-Service -Name LanmanServer -StartupType Manual
```

Еще раз обратимся к свойствам службы `LanmanServer` для проверки сделанных изменений:

```
PS C:\Users\andrv> Get-Service LanmanServer | Format-List Name, StartType

Name      : LanmanServer
StartType : Manual
```

Как видим, значение свойства `StartType` успешно изменено.

Отметим, что описание службы не является свойством объектов `System.ServiceProcess.ServiceController`, возвращаемых командлетом `Get-Service`. Другими словами, командлет `Get-Service` не позволяет увидеть описание определенной службы.

Решить эту проблему можно с помощью WMI-класса `Win32_Service`, который имеет свойство `Description` (описание службы). Посмотрим, чему равно значение этого свойства для службы `LanmanServer`:

```
PS C:\> Get-WmiObject win32_service -Filter "name = 'LanmanServer'" |
Format-List Name, Description

Description Name : LanmanServer
Description : Поддерживает общий доступ к файлам, принтерам и именованным
каналам для данного компьютера через сетевое подключение. Если служба
остановлена, такие функции не удастся выполнить. Если данная служба отключена,
не удастся запустить любые явно зависимые службы.
```

Работа с серверами автоматизации

В *главе 11* мы рассматривали пример использования методов, предоставляемых внешними COM-объектами, для решения нужной нам задачи — создания ярлыка на рабочем столе. Этот стандарт программных компонентов Component Object Model (COM) лежал в основе технологии *связывания и внедрения объектов* (Object Linking and Embedding, OLE), которая изначально использовалась в Windows для обмена данными между приложениями.

Вначале технология OLE использовалась для создания составных документов, а затем для решения более общей задачи — предоставления приложениями друг другу собственных функций (служб) и правильного использования этих функций. Технология, позволяющая одному приложению (*клиенту автоматизации*) вызывать функции другого приложения (*сервера автоматизации*) была названа OLE Automation.

С середины 1990-х годов вместо термина OLE стал использоваться новый термин — *ActiveX*, первоначально обозначавший WWW-компоненты (объекты), созданные на базе технологии COM. До появления платформы .NET технология ActiveX была ключевой в продуктах Microsoft.

Одними из самых распространенных и часто используемых серверов автоматизации в Windows являются приложения пакета Microsoft Office. Мы рассмотрим на примерах, каким образом можно выводить из PowerShell информацию в две наиболее распространенные программы этого пакета: Microsoft Word и Microsoft Excel. Однако предварительно скажем несколько слов об объектных моделях, представляемых данными программами.

Объектные модели Microsoft Word и Excel

Для того чтобы использовать из PowerShell те возможности, которые поддерживают программы Word и Excel, необходимо знать, какие именно объекты предостав-

ляются для внешнего использования этими серверами автоматизации и как объекты соотносятся друг с другом.

Хотя объектные модели различных приложений Microsoft Office довольно обширны, они похожи друг на друга, причем для практических целей достаточно понять принцип работы с несколькими ключевыми объектами. Здесь мы не будем останавливаться на подробном рассмотрении свойств и методов объектов приложений Word и Excel, а лишь кратко упомянем, какие именно объекты будут использоваться в рассматриваемых далее примерах сценариев.

На самом верхнем уровне объектной модели Word находится объект `Application`, который представляет непосредственно само приложение Word и содержит (в качестве свойств) все остальные объекты. Таким образом, объект `Application` используется для получения доступа к любому другому объекту Word.

Семейство `Documents` является свойством объекта `Application` и содержит набор объектов `Document`, каждый из которых соответствует открытому в Word документу. Класс `Documents` понадобится нам в сценариях для создания новых документов. Объект `Document` содержит в качестве своих свойств семейства различных объектов документа: символов (`Characters`), слов (`Words`), предложений (`Sentences`), абзацев (`Paragraphs`), закладок (`Bookmarks`) и т. д.

Объект `Selection` позволяет работать с выделенным фрагментом текста (этот фрагмент может быть и пустым). Таким образом, можно сказать, что объект `Selection` открывает путь в документ, т. к. он предоставляет доступ к выделенному фрагменту документа. В частности, у объекта `Selection` имеется метод `TypeText(`*`Text`*`)`, с помощью которого можно вставлять текст в документ. Используя свойства этого объекта (которые, в свою очередь, могут являться объектами со своими свойствами), можно управлять параметрами выделенного фрагмента, например устанавливать нужный размер и гарнитуру шрифта, выравнивать абзацы по центру и т. п.

Объектная модель Excel построена по тому же принципу, что и объектная модель Word. Основным объектом, содержащим все остальные, является `Application`. Напомним, что отдельные файлы в Excel называются *рабочими книгами* (workbooks). Семейство `Workbooks` в Excel является аналогом семейства `Documents` в Word и содержит набор объектов `Workbook` (аналог объекта `Document` в Word), каждый из которых соответствует открытой в Excel рабочей книге. Новая рабочая книга создается с помощью метода `Add()` объекта `Workbooks`.

Для доступа к ячейкам активного *рабочего листа* (worksheet) в Excel используется свойство `Cells` объекта `Application`. Для получения или изменения значения отдельной ячейки применяется конструкция `Cells.Item(`*`row`*`,`*`column`*`).Value`, где *`row`* и *`column`* являются номерами строки и столбца соответственно, на пересечении которых находится данная ячейка.

В Excel, как и в Word, имеется объект `Selection`, позволяющий работать с выделенным фрагментом электронной таблицы. Самым простым способом выделить диапазон ячеек активного рабочего листа является использование метода `Select()` объекта `Range`. Например, выражение `Range("A1:C1").Select()` позволяет выделить три смежные ячейки: `A1`, `B1` и `C1`.

Для того чтобы понять, какой именно объект Word или Excel нужно использовать для решения той или иной задачи, часто проще всего бывает проделать в соответствующем приложении необходимые манипуляции вручную, включив предварительно режим записи макроса. В результате мы получим текст макроса на языке VBA (Visual Basic for Applications), из которого будет ясно, какие методы и с какими параметрами нужно вызывать и какие значения нужно присваивать свойствам объектов.

Взаимодействие с Microsoft Word

Давайте научимся управлять с помощью PowerShell программой Microsoft Word: запустим из оболочки данную программу и напечатаем в окне Word строку текста.

Сделать это несложно. Сначала создаем экземпляр главного объекта сервера автоматизации Microsoft Word, который имеет программный идентификатор `Word.Application`, и сохраняем ссылку на этот объект в переменной `$word`:

```
PS C:\> $word = New-Object -ComObject Word.Application
```

С помощью метода `Add` коллекции `Documents` создаем новый документ и сохраняем ссылку на него в переменной `$doc`:

```
PS C:\> $doc = $word.Documents.Add()
```

Делаем окно с новым документом видимым. Для этого нужно в свойство `Visible` объекта `$word` записать `$true`:

```
PS C:\> $word.Visible = $true
```

В переменную `$sel` занесем ссылку на объект `Selection`, с помощью которого можно задавать тип и размер шрифта, тип выравнивания абзацев, а также печатать в документе строки текста:

```
PS C:\> $sel = $word.Selection
```

Установим размер шрифта:

```
PS C:\> $sel.Font.Size = 14
```

Установим полужирный шрифт:

```
PS C:\> $sel.Font.Bold = $true
```

Наконец, печатаем строку текста:

```
PS C:\> $sel.TypeText("Привет из PowerShell!")
```

Взаимодействие с Microsoft Excel

Давайте создадим из оболочки PowerShell новую рабочую книгу Microsoft Excel и выведем в нее заголовок и одну запись телефонной книги.

Как и в случае с Microsoft Word, сначала мы создаем экземпляр главного объекта `Excel.Application` и сохраняем ссылку на этот экземпляр в переменной `$XL`:

```
PS C:\> $XL = New-Object -ComObject Excel.Application
```

В результате запускается программа Microsoft Excel. Для того чтобы сделать окно приложения видимым, установим значение свойства `Visible` в `$true`:

```
PS C:\> $XL.Visible = $true
```

Создать новую рабочую книгу можно с помощью метода `Add` коллекции `WorkBooks`; при выполнении данного метода на экран выводится много ненужной технической информации, которую мы перенаправим на пустое устройство `$null`:

```
PS C:\> $XL.WorkBooks.Add() > $null
```

Теперь можно устанавливать нужную ширину столбцов (свойство `ColumnWidth`) и выводить текст заголовка:

```
PS C:\> #Устанавливаем нужную ширину колонок
PS C:\> $XL.Columns.Item(1).ColumnWidth = 40
PS C:\> $XL.Columns.Item(2).ColumnWidth = 40
PS C:\> $XL.Columns.Item(3).ColumnWidth = 40
PS C:\> #Печатаем в ячейках текст
PS C:\> $XL.Cells.Item(1,1).Value="Фамилия"
PS C:\> $XL.Cells.Item(1,2).Value="Имя"
PS C:\> $XL.Cells.Item(1,3).Value="Телефон"
```

С помощью метода `Select()` выделим три ячейки с введенным заголовком и установим в них полужирный шрифт:

```
PS C:\> #Выделяем три ячейки
PS C:\> $XL.Range("A1:C1").Select() > $null
PS C:\> #Устанавливаем полужирный шрифт для выделенного диапазона
PS C:\> $XL.Selection.Font.Bold = $true
```

Во вторую строку рабочей книги выводим данные об абоненте:

```
PS C:\> $XL.Cells.Item(2,1).Value="Иванов"
PS C:\> $XL.Cells.Item(2,2).Value="Иван"
PS C:\> $XL.Cells.Item(2,3).Value="555555"
```

Итоги

- Для управления процессами на локальной машине используются командлеты `Get-Process`, `Start-Process`, `Stop-Process`. Доступ к процессам на удаленной машине можно получить с помощью WMI-класса `Win32_Process`.
- Командлеты `Get-Service`, `Start-Service`, `Stop-Service` предназначены для работы со службами на локальном компьютере.
- С помощью командлета `New-Object` с параметром `-ComObject` можно обращаться к сервисам, предоставляемым серверами автоматизации.

ГЛАВА 14

HTTP-запросы к веб-ресурсам

В Интернете имеется огромное множество полезных сервисов, с которыми можно работать, обращаясь к их ресурсам по протоколу HTTP. Веб-разработчики постоянно работают с такими HTTP-запросами для доступа к функциям внешних API или для тестирования собственных приложений. Традиционно для этого используются специальные приложения (например, Postman), консольные утилиты типа cURL и wget, а также HTTP-клиенты, интегрированные в среды разработки.

PowerShell для обращения к веб-ресурсам по протоколу HTTP предлагает два стандартных командлета: `Invoke-WebRequest` и `Invoke-RestMethod`.

Командлет *Invoke-WebRequest*

С помощью командлета `Invoke-WebRequest` можно направить веб-серверу HTTP-запрос и получить от него ответ.

Анализ HTML-страниц

Командлет `Invoke-WebRequest` хорошо подходит для анализа HTML-страниц. Также этот командлет умеет сохранять страницы на локальном диске, в этом он похож на консольную утилиту wget и даже имеет такой псевдоним:

```
PS C:\Script> Get-Alias wget

CommandType     Name
-----------     ----
Alias           wget -> Invoke-WebRequest
```

Обратимся с помощью данного командлета к какой-нибудь простой странице, например Example Domain (**https://example.com/index.html**) (рис. 14.1).

По умолчанию `Invoke-WebRequest` выполняет HTTP-запрос с методом `GET` к ресурсу на веб-сервере, адрес ресурса указывается в качестве значения параметра `-Uri`. В результате возвращается объект типа `HtmlWebResponseObject`, в котором хранится информация об ответе сервера:

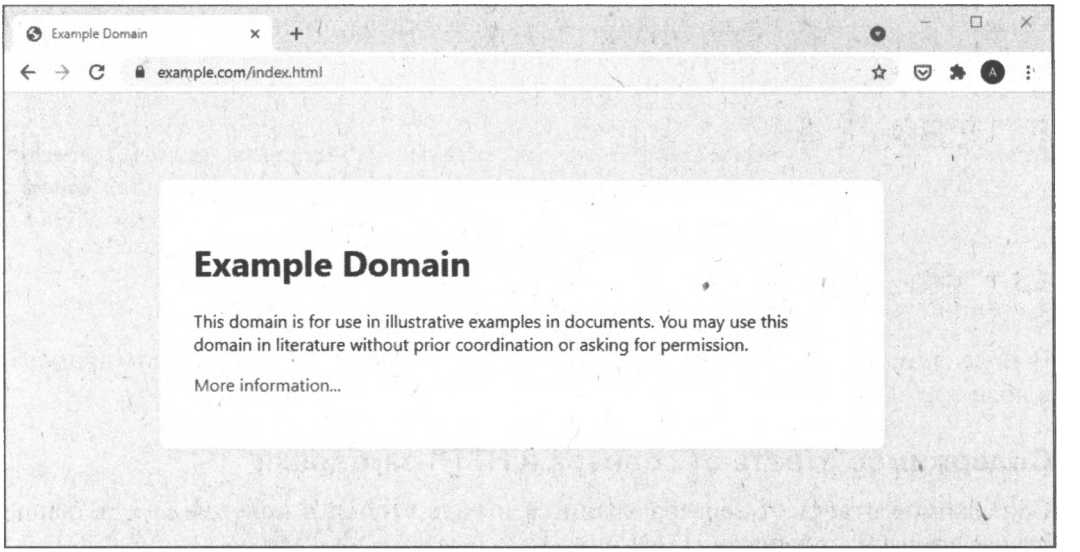


Рис. 14.1. Главная страница сайта Example Domain в браузере

```
PS C:\Script> $web = Invoke-WebRequest -Uri https://example.com/index.html
PS C:\Script> $web | Get-Member

   TypeName: Microsoft.PowerShell.Commands.HtmlWebResponseObject
. . .
PS C:\Script> $web

StatusCode        : 200
StatusDescription : OK
Content           : <!doctype html>
                    <html>
                    <head>
                      <title>Example Domain</title>
                      <meta charset="utf-8" />
                      <meta http-equiv="Content-type" content="text/html; char
                      set=utf-8" />
                      <meta name="viewport" conten...
RawContent        : HTTP/1.1 200 OK
                    Age: 497890
                    Vary: Accept-Encoding
                    X-Cache: HIT
                    Content-Length: 1256
                    Cache-Control: max-age=604800
                    Content-Type: text/html; charset=UTF-8
                    Date: Mon, 12 Jul 2021 16:05:14 GMT
                    Exp...
Forms             : {}
```

```
Headers           : {[Age, 497890], [Vary, Accept-Encoding], [X-Cache, HIT],
                    [Content-Length, 1256]...}
Images            : {}
InputFields       : {}
Links             : {@{innerHTML=More information...; innerText=More information
                    ...; outerHTML=<A href="https://www.iana.org/domains/example
                    ">More information...</A>; outerText=More information...;
                    tagName=A; href=https://www.iana.org/domains/example}}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 1256
```

В поле `StatusCode` содержится код ответа от сервера (200 для нашего примера), в поле `StatusDescription` — текстовое описание этого ответа (OK).

Содержимое ответа от сервера и HTTP-заголовки

Содержимое ответа от сервера хранится в виде строки в поле `Content`. В нашем случае здесь будет записан HTML-код:

```
PS C:\Script> $web.Content
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI",
        "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;

    }
    div {
        width: 600px;
        margin: 5em auto;
        padding: 2em;
        background-color: #fdfdff;
        border-radius: 0.5em;
        box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);
    }
    a:link, a:visited {
        color: #38488f;
        text-decoration: none;
    }
```

```
    @media (max-width: 700px) {
        div {
            margin: 0 auto;
            width: auto;
        }
    }
    </style>
</head>

<body>
<div>
    <h1>Example Domain</h1>
    <p>This domain is for use in illustrative examples in documents. You may
    use this domain in literature without prior coordination or asking
    for permission.</p>
    <p><a href="https://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
```

В поле `RawContent` записывается полный ответ от сервера с HTTP-заголовками в начале:

```
PS C:\Script> $web.RawContent
HTTP/1.1 200 OK
Age: 497890
Vary: Accept-Encoding
X-Cache: HIT
Content-Length: 1256
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Mon, 12 Jul 2021 16:05:14 GMT
Expires: Mon, 19 Jul 2021 16:05:14 GMT
ETag: "3147526947+ident"
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Server: ECS (dcb/7F83)

<!doctype html>
<html>
<head>
    <title>Example Domain</title>
. . .
```

Также заголовки ответа выделяются отдельно и хранятся в виде хэш-таблицы в свойстве `headers`:

```
PS C:\Script> $web.headers

Key             Value
---             -----
Age             497890
Vary            Accept-Encoding
```

```
X-Cache        HIT
Content-Length 1256
Cache-Control  max-age=604800
Content-Type   text/html; charset=UTF-8
Date           Mon, 12 Jul 2021 16:05:14 GMT
Expires        Mon, 19 Jul 2021 16:05:14 GMT
ETag           "3147526947+ident"
Last-Modified  Thu, 17 Oct 2019 07:18:26 GMT
Server         ECS (dcb/7F83)
```

Сохранение веб-ресурсов

Для сохранения ответа от сервера в виде локального файла нужно при вызове `Invoke-WebRequest` использовать ключ `-OutFile` и указать путь к нужному файлу. Например, сохраним страницу **https://example.com/index.html** в файле page.html в текущем каталоге:

```
PS C:\Script> Invoke-WebRequest -Uri https://example.com/index.html
-OutFile page.html
```

Проверим содержимое файла page.html и убедимся, что в нем записана HTML-разметка сохраненной страницы:

```
PS C:\Script> type .\page.html
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI",
        "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;

    }
    div {
        width: 600px;
        margin: 5em auto;
        padding: 2em;
        background-color: #fdfdff;
        border-radius: 0.5em;
        box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);
    }
```

```
    a:link, a:visited {
        color: #38488f;
        text-decoration: none;
    }
    @media (max-width: 700px) {
        div {
            margin: 0 auto;
            width: auto;
        }
    }
    </style>
</head>

<body>
<div>
    <h1>Example Domain</h1>
    <p>This domain is for use in illustrative examples in documents. You may
    use this domain in literature without prior coordination or asking
    for permission.</p>
    <p><a href="https://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
```

Подобным образом можно сохранять не только HTML-файлы, но и ресурсы других типов (текстовые, графические или мультимедийные файлы и т. д.), к которым можно обратиться на сайте.

Поиск HTML-элементов на странице

В свойствах `Forms` (формы), `Images` (изображения), `InputFields` (поля ввода) и `Links` (ссылки) объекта `HtmlWebResponseObject` сохраняются массивы объектов, которые описывают соответствующие элементы HTML-разметки, полученной от сервера. Обрабатывая эти коллекции, можно получить информацию об интересующих нас элементах на загруженной странице.

Например, сохраним в переменной `$web` страницу результатов поиска слова PowerShell в Yandex:

```
PS C:\Script> $web = Invoke-WebRequest -Uri
https://yandex.ru/search/?text=PowerShell
```

Выделим на этой странице все ссылки на сайт **habr.com**. Для этого нужно отфильтровать массив `$web.Links`, оставив в нем объекты, у которых значение свойства `href` соответствует маске `*habr.com*`:

```
PS C:\Script> $habr_links = $web.Links | Where-Object href -like '*habr.com*'
```

В `$habr_links` находятся два объекта `PSCustomObject`, содержащие различные HTML-атрибуты для ссылок:

```
PS C:\Script> $habr_links.count
2
PS C:\Script> $habr_links | Get-Member

   TypeName: System.Management.Automation.PSCustomObject

Name          MemberType   Definition
----          ----------   ----------
Equals        Method       bool Equals(System.Object obj)
GetHashCode   Method       int GetHashCode()
GetType       Method       type GetType()
ToString      Method       string ToString()
class         NoteProperty string class=Link Link_theme_normal OrganicTitl...
data-counter  NoteProperty string data-counter=["b"]
data-log-node NoteProperty string data-log-node=bf7fw01-00
href          NoteProperty string href=https://habr.com/ru/company/ruvds/b...
innerHTML     NoteProperty string innerHTML=<div class="Favicon Favicon_si...
innerText     NoteProperty string innerText=...
outerHTML     NoteProperty string outerHTML=<a tabindex="0" class="Link Li...
outerText     NoteProperty string outerText=...
tabindex      NoteProperty string tabindex=0
tagName       NoteProperty string tagName=A
target        NoteProperty string target=_blank
```

Например, выведем значение атрибутов `href` и `innerText` для этих ссылок:

```
PS C:\Script> $habr_links | ForEach-Object {$_.href + " - " + $_.innerText }
https://habr.com/ru/company/ruvds/blog/487876/ -
Что такое Windows PowerShell и с чем его едят? / Хабр
https://habr.com/ru/company/ruvds/blog/487876/ -
habr.com›ru/company/ruvds/blog/487876/
```

В свойстве `ParsedHtml` объекта `HtmlWebResponseObject` содержится объект типа `mshtml.HTMLDocumentClass`, который предоставляет доступ к DOM-дереву загруженной HTML-страницы.

```
PS C:\Users\andrv> $html = $web.ParsedHtml
PS C:\Users\andrv> Get-Member -InputObject $html

   TypeName: mshtml.HTMLDocumentClass
. . .
```

> ***Замечание***
>
> Если в HTML-коде имеются сценарии JavaScript, то по умолчанию при построении DOM-дерева они будут выполнены. При необходимости выполнение сценариев можно отключить, указав параметр `-UseBasicParsing`.

В частности, используя методы `getElementById()`, `getElementsByName()` и `getElementsByTagName()`, можно получать объекты, соответствующие HTML-элементам с заданным идентификатором, именем или тегом соответственно.

Например, посмотрим, какой текст записан в первом заголовке второго уровня (HTML-тег <h2>):

```
PS C:\Users\andrv> $html.getElementsByTagName('h2')[0].innerText

Документация по PowerShell - PowerShell | Microsoft Docs
```

Найдем элемент с идентификатором search-result:

```
PS C:\Users\andrv> $html.getElementById('search-result')

className                    : serp-list serp-list_left_yes
id                           : search-result
tagName                      : UL
parentElement                : System.__ComObject
style                        : System.__ComObject
. . .
```

Метод querySelector() позволяет найти HTML-элемент по определенному CSS-селектору. Например, обратимся к элементу с классом main__content:

```
PS C:\Users\andrv> $html.querySelector('.main__content')

className                    : main__content
id                           :
tagName                      : DIV
parentElement                : System.__ComObject
style                        : System.__ComObject
. . .
```

Подробнее о свойствах, методах и событиях объекта mshtml.HTMLDocumentClass можно прочитать в документации на сайте Microsoft (**https://docs.microsoft.com/ru-ru/dotnet/api/system.windows.forms.htmldocument?view=net-5.0**).

Выполнение POST-запросов

Командлет Invoke-WebRequest позволяет не только выполнять GET-запросы, но и вызывать другие методы, определенные в протоколе HTTP (DELETE, HEAD, MERGE, PATCH, POST, PUT, TRACE). Для этого нужный метод указывается в качестве значения параметра -Method.

Рассмотрим пример выполнения запроса с HTTP-методом POST, который часто используется для передачи данных из веб-форм или загрузки файлов на сервер. Обращаться мы будем к ресурсу **http://httpbin.org/post**, в результате сервер должен сообщить нам о полученных данных.

Передавать мы будем два параметра с именами name и lastName, которые поместим в хэш-таблицу $params:

```
PS C:\Users\andrv> $params = @{name='Andrey'; lastName='Popov'}
```

Выполним Invoke-WebRequest с методом POST, поместив передаваемые параметры в тело запроса (параметр -Body):

```
PS C:\Users\andrv> Invoke-WebRequest -Uri http://httpbin.org/post -Method POST
-Body $params

StatusCode        : 200
StatusDescription : OK
Content           : {
                      "args": {},
                      "data": "",
                      "files": {},
                      "form": {
                        "lastName": "Popov",
                        "name": "Andrey"
                      },
                      "headers": {
                        "Content-Length": "26",
                        "Content-Type": "application/x-www-form..
                    .
RawContent        : HTTP/1.1 200 OK
                    Connection: keep-alive
                    Access-Control-Allow-Origin: *
                    Access-Control-Allow-Credentials: true
                    Content-Length: 503
                    Content-Type: application/json
                    Date: Fri, 16 Jul 2021 03:03:13 GM...
Forms             : {}
Headers           : {[Connection, keep-alive], [Access-Control-Al
                    low-Origin, *], [Access-Control-Allow-Credent
                    ials, true], [Content-Length, 503]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 503
```

Отправляемые на сервер параметры командлет `Invoke-WebRequest` автоматически приводит к формату application/x-www-form-urlencoded, который используется при передаче данных из веб-форм. Ответ в поле `Content` говорит о том, что сервер принял наши параметры и определил, что они были отправлены из формы.

Иногда бывает нужно передавать на сервер данные в формате JSON, а не в application/x-www-form-urlencoded. В этом случае нужно указать параметр `-ContentType` со значением `application/json`. Например:

```
PS C:\Users\andrv> $json_params = "{ 'name':'Andrey', 'lastName':'Popov' }"
PS C:\Users\andrv> Invoke-WebRequest -Uri http://httpbin.org/post -ContentType
"application/json" -Method POST -Body $json_params

StatusCode        : 200
StatusDescription : OK
```

```
Content           : {
                      "args": {},
                      "data": "{ 'name':'Andrey', 'lastName':'Popov
                    ' }",
                      "files": {},
                      "form": {},
                      "headers": {
                        "Content-Length": "39",
                        "Content-Type": "application/json",
                        "Host": "...
RawContent        : HTTP/1.1 200 OK
                    Connection: keep-alive
                    Access-Control-Allow-Origin: *
                    Access-Control-Allow-Credentials: true
                    Content-Length: 475
                    Content-Type: application/json
                    Date: Fri, 16 Jul 2021 03:43:04 GM...
Forms             : {}
Headers           : {[Connection, keep-alive], [Access-Control-Allo
                    w-Origin, *], [Access-Control-Allow-Credentials
                    , true], [Content-Length, 475]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 475
```

Как видим, при таком способе отправки запроса серверный скрипт, обрабатывающий обращения к ресурсу **http://httpbin.org/post**, извлек полученные данные и при формировании своего JSON-ответа поместил их в поле `data`, а не в поле `form`.

Командлет *Invoke-RestMethod*

Если мы обращаемся к веб-сервису, поддерживающему REST API, то ответ сервера скорее всего будет содержать структурированные данные в формате JSON или XML. Так, в предыдущем примере мы с помощью командлета `Invoke-WebRequest` посылали POST-запрос на ресурс **http://httpbin.org/post** и получали в ответ JSON-строку:

```
PS C:\Users\andrv> $web = Invoke-WebRequest -Uri http://httpbin.org/
post -Method POST -Body @{name='Andrey'; lastName='Popov'}
PS C:\Users\andrv> $web.Content
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "lastName": "Popov",
    "name": "Andrey"
  },
```

```
  "headers": {
    "Content-Length": "26",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "Mozilla/5.0 (Windows NT; Windows NT 10.0; ru-RU)
    WindowsPowerShell/5.1.19041.1023",
    "X-Amzn-Trace-Id": "Root=1-60f2a42c-18f152db3b76b5d66d173e31"
  },
  "json": null,
  "origin": "85.95.179.209",
  "url": "http://httpbin.org/post"
}
```

Чтобы работать с полями ответа, мы должны преобразовать JSON в объект PowerShell с помощью командлета `ConvertFrom-Json` (мы рассматривали подобные преобразования в *главе 12*):

```
PS C:\Users\andrv> $response = $web.Content | ConvertFrom-Json
PS C:\Users\andrv> $response

args    :
data    :
files   :
form    : @{lastName=Popov; name=Andrey}
headers : @{Content-Length=26; Content-Type=application/x-www-form-urlencoded;
            Host=httpbin.org; User-Agent=Mozilla/5.0 (Windows NT; Windows NT 10.0;
            ru-RU) Windows PowerShell/5.1.19041.1023;
            X-Amzn-Trace-Id=Root=1-60f2a42c-18f152db3b76b5d66d173e31}
json    :
origin  : 85.95.179.209
url     : http://httpbin.org/post

PS C:\Users\andrv> $response.form.name
Andrey
```

В подобных случаях, когда от сервера мы получаем структурированные данные, удобнее пользоваться командлетом `Invoke-RestMethod`, который действует аналогично `Invoke-WebRequest` и имеет такие же параметры, но при этом автоматически преобразует ответ от сервера в объект PowerShell.

Выполним наш запрос с помощью `Invoke-RestMethod`, сохранив результат в переменной `$result`:

```
PS C:\Users\andrv> $result = Invoke-RestMethod -Uri http://httpbin.org/
post -Method POST -Body @{name='Andrey'; lastName='Popov'}
```

Проверим тип и содержимое переменной `$result`:

```
PS C:\Users\andrv> Get-Member -InputObject $result

   TypeName: System.Management.Automation.PSCustomObject
```

```
Name        MemberType   Definition
----        ----------   ----------
Equals      Method       bool Equals(System.Object obj)
GetHashCode Method       int GetHashCode()
GetType     Method       type GetType()
ToString    Method       string ToString()
args        NoteProperty System.Management.Automation.PSCusto...
data        NoteProperty string data=
files       NoteProperty System.Management.Automation.PSCusto...
form        NoteProperty System.Management.Automation.PSCusto...
headers     NoteProperty System.Management.Automation.PSCusto...
json        NoteProperty object json=null
origin      NoteProperty string origin=85.95.179.209
url         NoteProperty string url=http://httpbin.org/post
PS C:\Users\andrv> $result

args    :
data    :
files   :
form    : @{lastName=Popov; name=Andrey}
headers : @{Content-Length=26; Content-Type=application/x-www-form-urlencoded;
          Host=httpbin.org; User-Agent=Mozilla/5.0 (Windows NT; Windows NT 10.0;
          ru-RU) Windows PowerShell/5.1.19041.1023;
          X-Amzn-Trace-Id=Root=1-60f2a897-6354ce272644b6d412560a9b}
json    :
origin  : 85.95.179.209
url     : http://httpbin.org/post
```

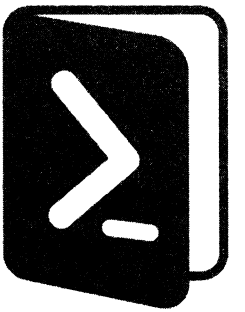
Как видим, вместо строки в формате JSON мы получаем PowerShell-объект типа `System.Management.Automation.PSCustomObject` и можем сразу обращаться к нужным свойствам:

```
PS C:\Users\andrv> $result.form.lastName
Popov
```

Итак, для выполнения HTTP-запросов к веб-ресурсам, возвращающим HTML-страницы, удобнее использовать командлет `Invoke-WebRequest`, а для работы с внешними REST API, возвращающими структурированные данные, лучше подойдет командлет `Invoke-RestMethod`.

Итоги

- Для обращения к веб-ресурсам по протоколу HTTP в PowerShell имеются два стандартных командлета: `Invoke-WebRequest` и `Invoke-RestMethod`.
- Действуют эти командлеты аналогично друг другу, за исключением того, что `Invoke-RestMethod` автоматически преобразует ответ от сервера в объект PowerShell.



ЧАСТЬ IV

Пишем сценарии

Глава 15. Разработка сценариев PowerShell

Глава 16. Отладка функций и сценариев

Глава 17. Графический интерфейс для сценариев

ГЛАВА 15

Разработка сценариев PowerShell

Удобная и однородная структура команд и конвейеров PowerShell помогает новичкам быстро освоиться в командной строке и без труда автоматизировать несложные задачи управления Windows. Можно сказать, что PowerShell идеально отвечает известному принципу KISS (*Keep it short and simple* или *Keep it simple, stupid*), утверждающему, что простые вещи должны делаться просто. Одно из главных преимуществ PowerShell состоит в том, что пользователи этой оболочки могут долго обходиться вообще без написания и применения специальных сценариев, что было бы невозможно при работе, например, с Windows Script Host.

Однако PowerShell — это не только командный интерпретатор и оболочка командной строки, но и современный скриптовый язык.

Язык PowerShell, с одной стороны, обладает мощными возможностями (он действительно оправдывает первую составляющую своего имени — Power), позволяющими создавать профессиональные сценарии, которые пользуются всей мощью платформы .NET Framework и других объектных моделей операционной системы, могут взаимодействовать с внешними приложениями и службами с помощью структурированных данных (JSON, XML) и т. д. С другой стороны, код PowerShell зачастую выглядит проще, лаконичнее и интуитивно понятнее программы на традиционных скриптовых языках.

Чтобы начать писать сценарии PowerShell для автоматизации своей работы, не нужно быть профессиональным программистом, разбирающимся в тонкостях объектно-ориентированного программирования и устройстве платформы .NET Framework. Освоив базовые принципы PowerShell, перейти от интерактивной работы к написанию сценариев будет совсем не сложно. Это еще одно подтверждение принципа KISS для PowerShell: простые сценарии пишутся просто, для этого достаточно понимать принцип конвейерной обработки объектов и знать основы синтаксиса языка, которые мы рассматривали в предыдущих главах.

Переход от команд к сценариям

В каком случае стоит создавать отдельные сценарии и каких рекомендаций следует придерживаться при их написании?

Если с помощью PowerShell нужно выполнить какие-то разовые действия (например, создать определенный каталог на жестком диске и скопировать в него, например, все загруженные файлы с расширением jpg), то проще набрать отдельные команды непосредственно в оболочке, не создавая специальные файлы со сценариями.

Если же задача выполняется (пусть даже и с модификациями) периодически, то лучше создать для нее сценарий. Написав один раз сценарий, мы сможем запускать его в любое время, не вводя вновь и вновь вручную находящиеся в нем команды.

Замечание

Количество стандартных командлетов, доступных из оболочки по умолчанию, увеличивается с каждой версией PowerShell. Поэтому прежде чем начинать писать собственный код, следует посмотреть, нет ли стандартного решения нашей задачи в самой оболочке или в общем репозитории модулей PowerShell Gallery.

Среды для разработки сценариев

В принципе, писать сценарии PowerShell можно в любом привычном для вас текстовом редакторе, например в стандартном Блокноте Windows. Однако намного удобнее это делать в специальных редакторах или средах разработки, которые поддерживают цветовое выделение синтаксиса и автоматическое дополнение вводимых команд, имеют возможности построчного запуска сценариев и просмотра значений переменных и промежуточных результатов во время выполнения.

Проще всего воспользоваться предустановленной в Windows интегрированной средой сценариев PowerShell (Integrated Scripting Environment, ISE), хотя сейчас PowerShell ISE не развивается и Microsoft рекомендует использовать свой бесплатный редактор Visual Studio Code с расширением для поддержки PowerShell. В данной главе мы кратко рассмотрим оба эти инструмента.

PowerShell ISE

Для запуска встроенной в систему интегрированной среды для написания сценариев PowerShell можно выбрать пункт **Windows PowerShell | Windows PowerShell ISE** в меню **Пуск** либо воспользоваться командной строкой или пунктом **Выполнить** в меню **Пуск** и выполнить команду `powershell-ise`.

В результате откроется окно PowerShell ISE, содержащее меню, панель инструментов, область для редактирования сценариев и консольную панель для вывода результатов, где также можно вводить команды PowerShell (рис. 15.1).

Запуск сценариев и фрагментов

Введем в области редактирования следующие строки:

```
'Привет из PowerShell!'
"Текущий каталог: $(Get-Location)"
'Пока!'
```

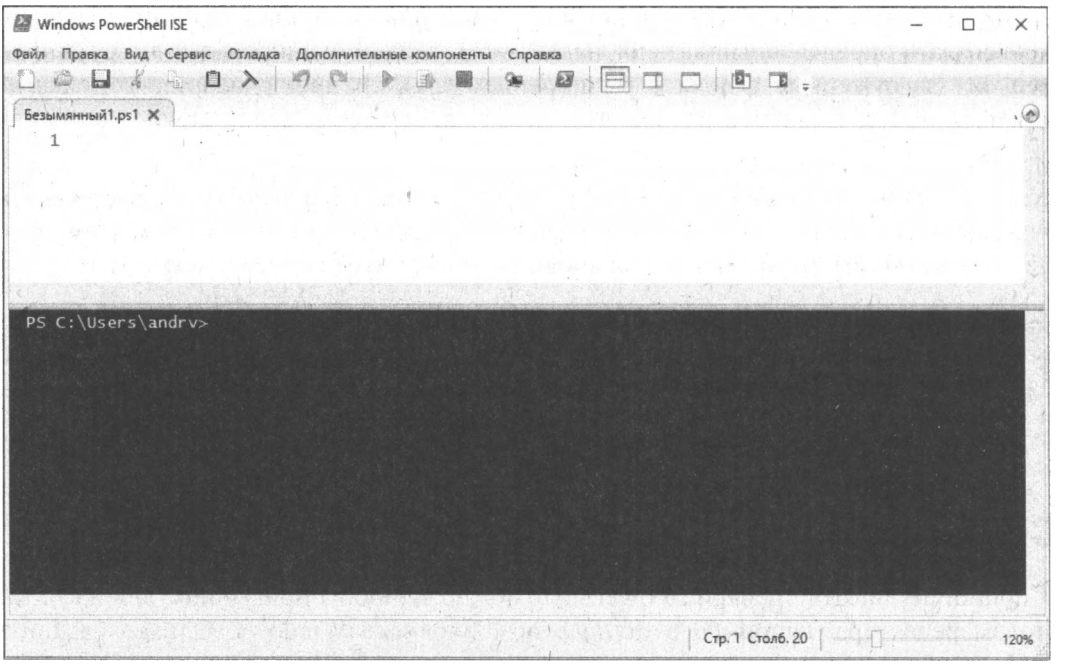


Рис. 15.1. Окно PowerShell ISE

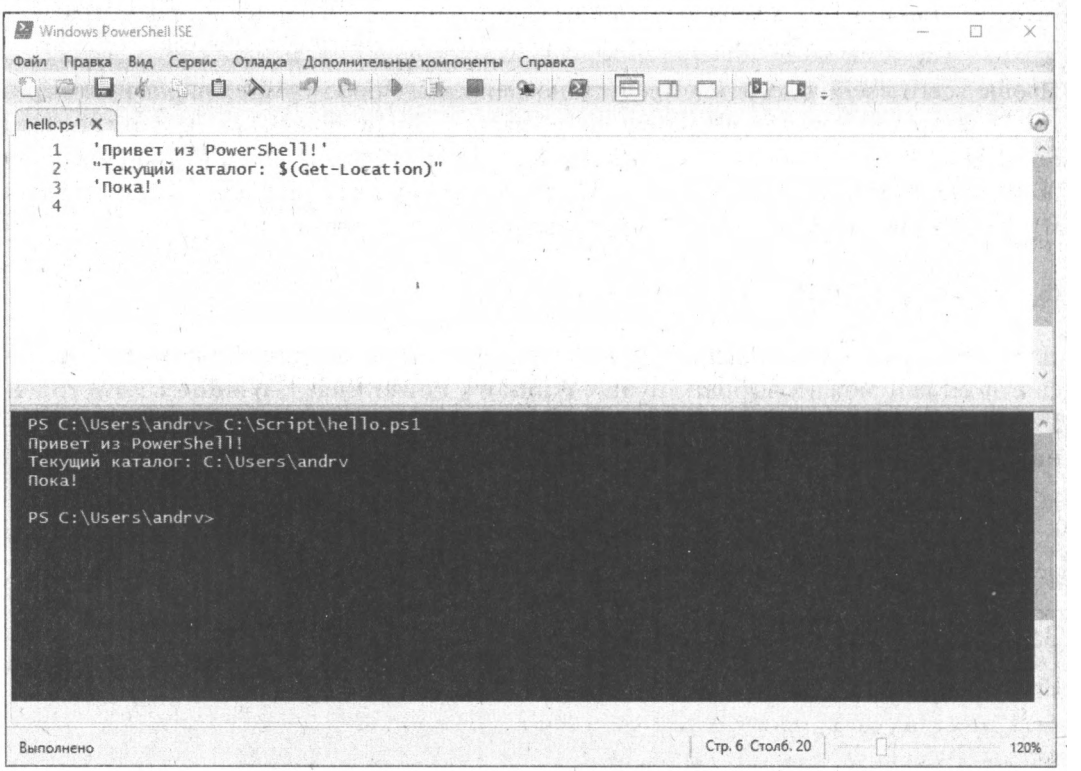


Рис. 15.2. Результат выполнения сценария hello ps1

Сохраним сценарий под именем hello.ps1 в каталоге C:\Script. Для запуска нашего сценария нужно щелкнуть на зеленом треугольнике на панели инструментов или нажать клавишу <F5>. Результат работы сценария будет выведен в консоль внизу экрана (рис. 15.2).

Можно запускать отлаживаемый скрипт по частям. Для этого следует выделить нужные строки и щелкнуть на значке с зеленым треугольником на листе бумаги или нажать клавишу <F8>. Например, выполним вторую и третью строки нашего сценария (рис. 15.3).

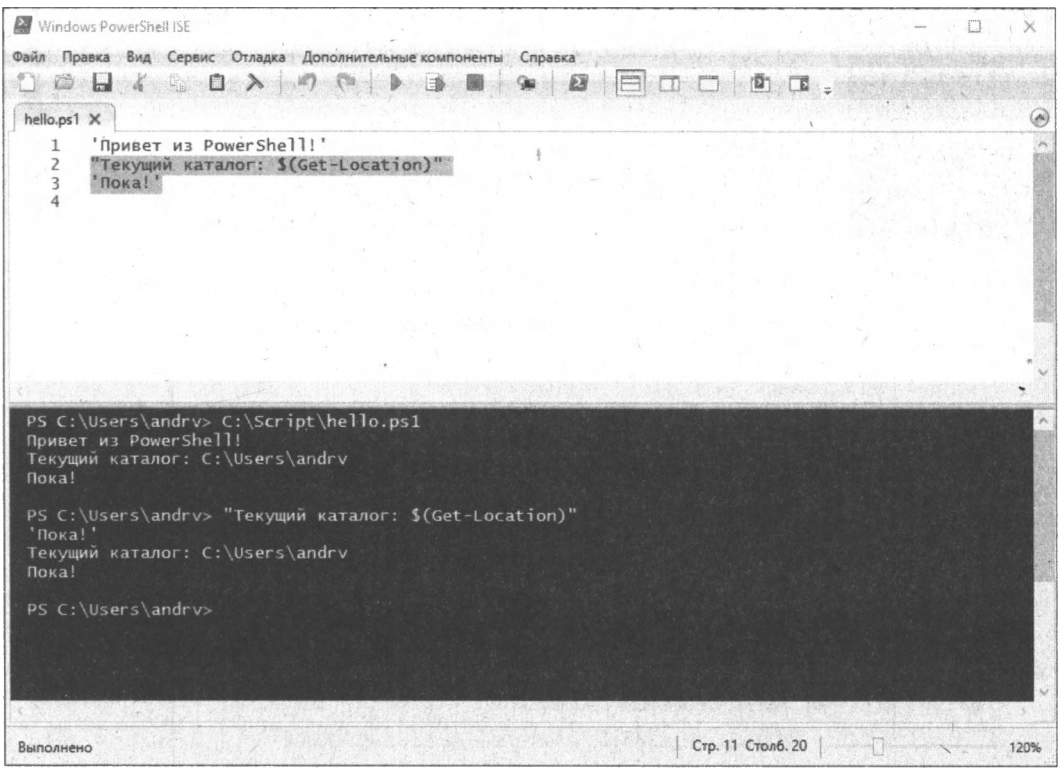


Рис. 15.3. Результат выполнения второй и третьей строк в hello ps1

Справочная система

В PowerShell ISE доступна справочная система PowerShell. Для просмотра справки по команде достаточно поместить на нее курсор и нажать клавишу <F1>. При этом по умолчанию информация берется с локальной машины и выводится в отдельном окне (рис. 15.4).

При необходимости можно переключить источник получения и при нажатии клавиши <F1> получать онлайн-справку. Для этого нужно снять флажок **Использовать локальную справку вместо справки в Интернете** на вкладке **Общие параметры** диалогового окна с настройками PowerShell ISE (пункт меню **Сервис | Параметры**) (рис. 15.5).

Рис. 15.4. Справка по командам в PowerShell ISE

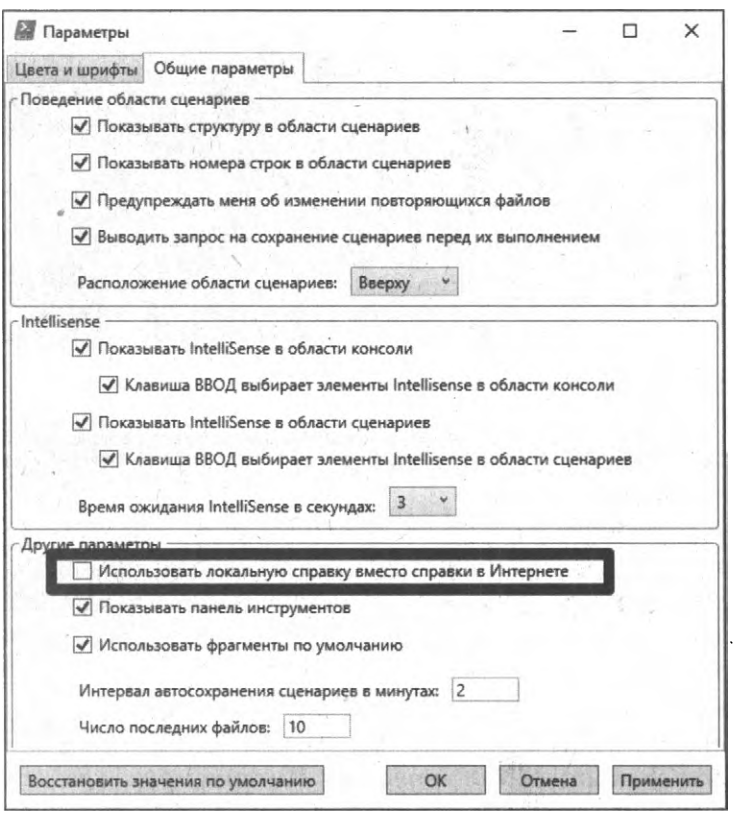


Рис. 15.5. Настройка общих параметров PowerShell ISE

Редактирование текста

Редактор кода, встроенный в PowerShell ISE, поддерживает базовые возможности для поиска и правки текста — соответствующие сочетания клавиш можно посмотреть в пункте меню **Правка** (рис. 15.6).

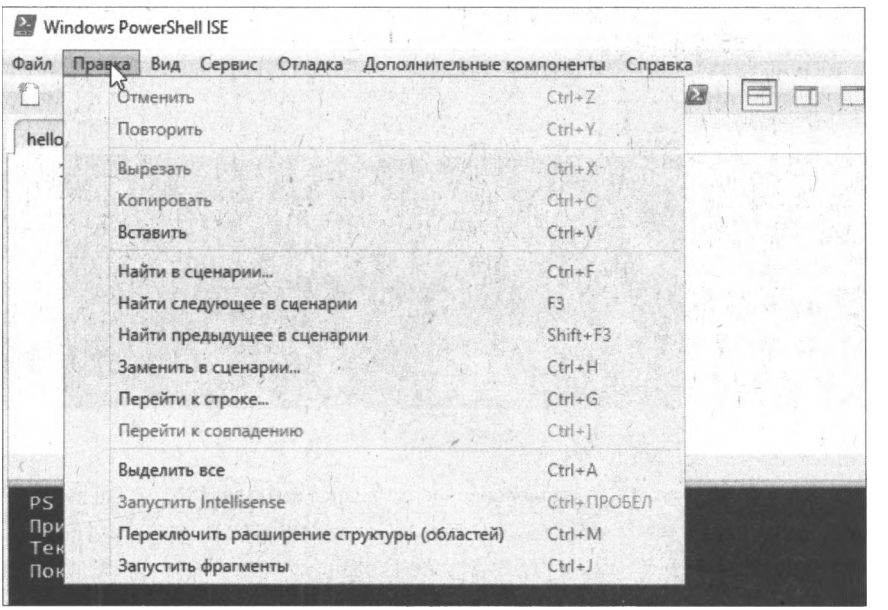


Рис. 15.6. Сочетания клавиш для поиска и редактирования текста в PowerShell ISE

В ISE имеется режим мультикурсора, когда изменения вносятся одновременно в несколько строк, в которых установлен курсор. Для установки дополнительного курсора на строку выше или ниже нужно одновременно зажать клавиши <Alt> и <Shift> и нажать клавишу <↑> или <↓> соответственно.

При вводе команд PowerShell в окне редактора срабатывает автодополнение, появляется список подходящих команд или параметров, из которого можно выбрать нужный пункт (рис. 15.7).

Также поддерживается режим автодополнения, аналогичный механизму в командной строке, когда подходящие команды и параметры подставляются по нажатии клавиши <Tab>.

Работая в PowerShell ISE, можно воспользоваться стандартными фрагментами кода (сниппетами). Для этого нужно нажать комбинацию клавиш <Ctrl>+<j> и выбрать нужный сниппет из списка (рис. 15.8).

Например, если выбрать сниппет `Cmdlet`, то в текст сценария будет вставлена расширенная функция с названием `Verb-Noun`, содержащая пустые секции `Begin`, `Process` и `End`. Для этой функции будут указаны некоторые параметры и комментарии:

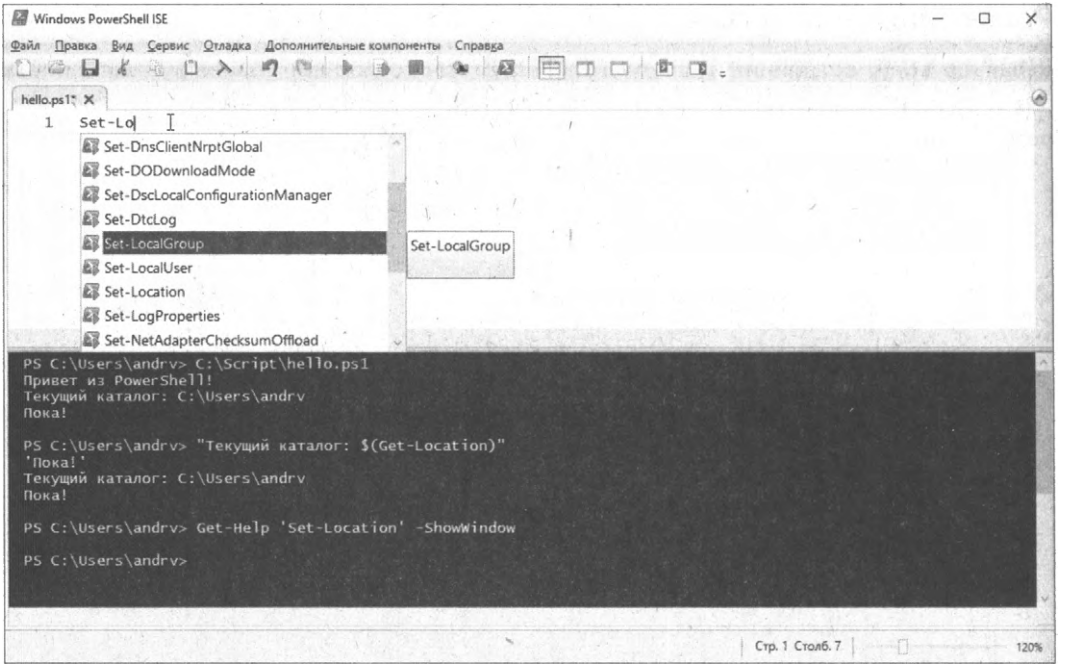


Рис. 15.7. Автодополнение команд в PowerShell ISE

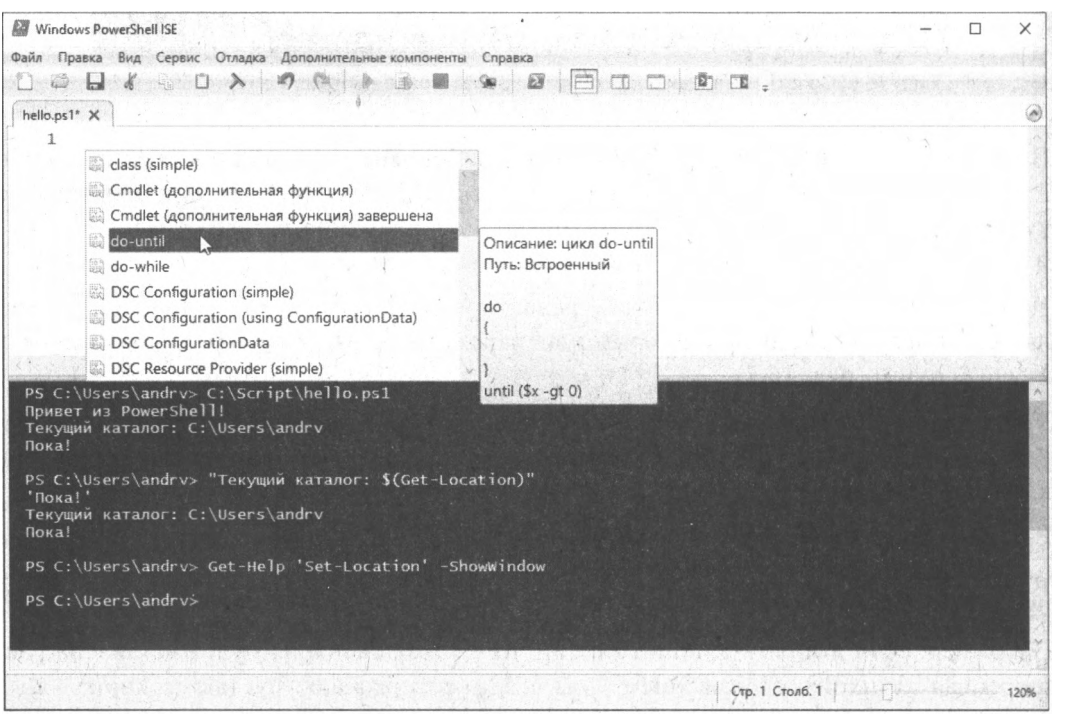


Рис. 15.8. Вставка фрагмента кода в PowerShell ISE

```
<#
.Synopsis
   Краткое описание
.DESCRIPTION
   Длинное описание
.EXAMPLE
   Пример использования этого командлета
.EXAMPLE
   Еще один пример использования этого командлета
#>
function Verb-Noun
{
    [CmdletBinding()]
    [Alias()]
    [OutputType([int])]
    Param
    (
        # Справочное описание параметра 1
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        $Param1,

        # Справочное описание параметра 2
        [int]
        $Param2
    )

    Begin
    {
    }
    Process
    {
    }
    End
    {
    }
}
```

Исправить и дополнить эту заготовку намного проще и быстрее, чем писать подобную функцию с нуля.

К сожалению, редактор ISE не поддерживает некоторые возможности редактирования программного кода, которые в других редакторах и средах разработки относятся к стандартным. Например, в ISE нет функций автоматического отступа или форматирования для придания коду структурированности, поэтому выделять пробелами вложенный код внутри условий или циклов придется вручную.

Отладка сценариев

В PowerShell ISE сценарии можно выполнять пошагово: если какой-то сценарий работает некорректно, его можно запустить в ISE и проверить, что он делает на каждом шаге, чтобы выявить ошибку.

Команды и комбинации клавиш для отладки сценариев можно посмотреть в пункте меню **Отладка** (рис. 15.9).

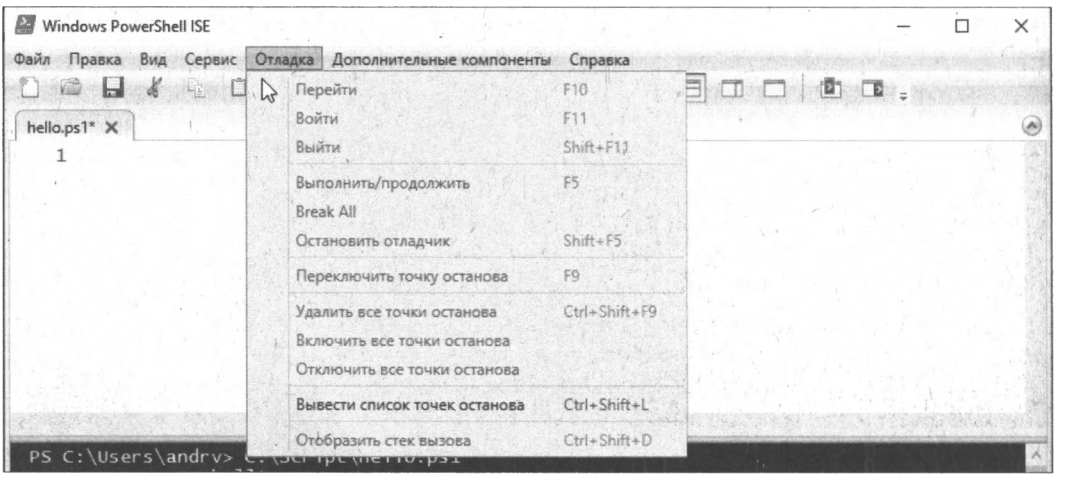


Рис. 15.9. Команды для отладки сценариев в PowerShell ISE

Для того чтобы сценарий приостановил свое выполнение перед определенной строкой, нужно поставить на ней *точку останова* (также их называют контрольными точками или точками прерывания). В ISE точки останова устанавливаются и снимаются нажатием клавиши <F9>.

Рассмотрим пример. Создадим сценарий C:\Script\sum.ps1, в котором будет вычисляться сумма чисел от 1 до 10:

```
"Начало сценария"
$sum = 0
foreach ($i in 1..10)
{
    $sum += $i
}
"Сумма равна $sum"
```

Поставим точку останова на пятой строке, где вычисляется новое значение переменной `$sum`, и запустим сценарий нажатием клавиши <F5>. В консоли мы увидим сообщение о начале сценария и новое приглашение командной строки с префиксом `[DBG]`, указывающим на то, что мы находимся в режиме отладки (рис. 15.10).

Во вложенной командной строке можно выполнять команды PowerShell, проверять или изменять значения переменных, используемых в сценарии. Посмотрим, чему равны переменные `$sum` и `$i` перед первым выполнением пятой строки:

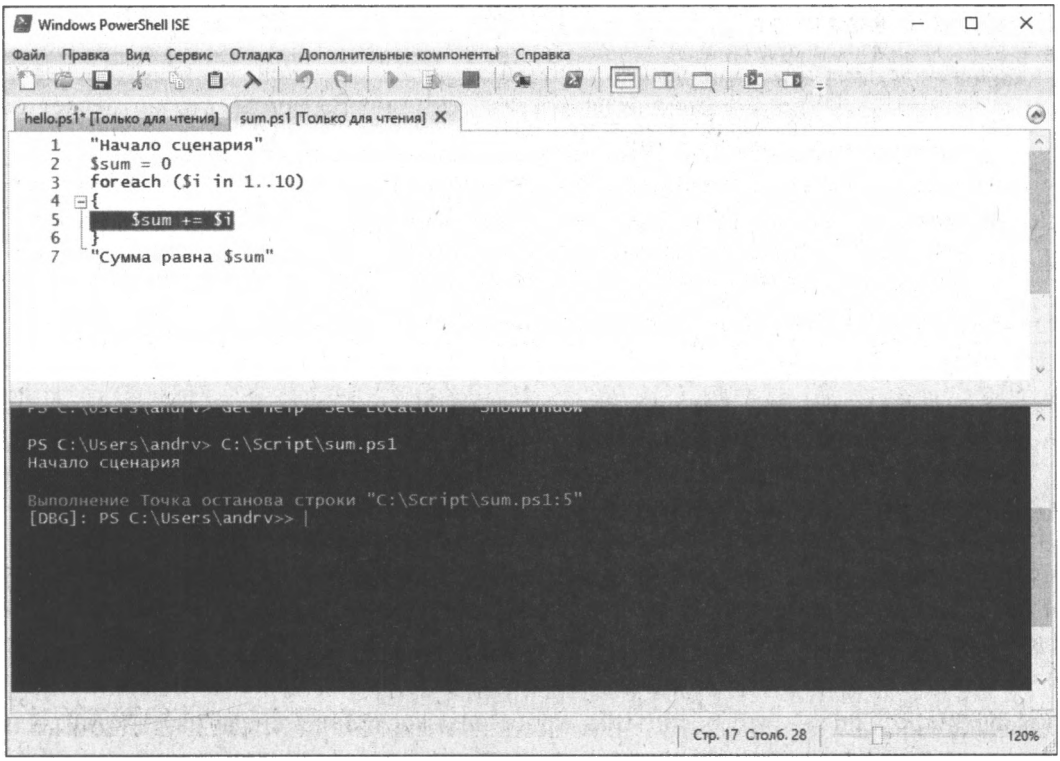


Рис. 15.10. Вложенная командная строка в режиме отладки

```
[DBG]: PS C:\Users\andrv>> $sum
0
[DBG]: PS C:\Users\andrv>> $i
1
```

Для продолжения сценария нажмем клавишу <F5>. Выполнение вновь прервется на пятой строке сценария (это будет вторая итерация цикла). Убедимся, что значения переменных `$sum` и `$i` изменились:

```
[DBG]: PS C:\Users\andrv>> $sum
1
[DBG]: PS C:\Users\andrv>> $i
2
```

После приостановки сценарий можно продолжить выполнять в разных режимах (табл. 15.1).

Таблица 15.1. *Варианты пошагового выполнения сценария*

Клавиша	Режим выполнения
<F10>	Пропуск блока Запускается только следующий оператор Если оператор является вызовом функции, определенной пользователем, то PowerShell запускает эту функцию и останавливается после ее завершения

Таблица 15.1 (окончание)

Клавиша	Режим выполнения
<F11>	Вход в блок Запускается только следующий оператор. Если оператор является вызовом функции, определенной пользователем, то PowerShell переходит на первый оператор в этой функции и там останавливается
<Shift>+<F11>	Выход из блока Если отладка производится внутри функции, то эта функция выполняется до конца, после чего управление передается в строку сценария, где данная функция была вызвана, и выполнение останавливается
<F5>	Выполнение до следующей точки останова или до конца файла

Более подробно процесс отладки сценариев из командной строки мы рассмотрим в *главе 16*.

Visual Studio Code

Сейчас компания Microsoft перестала развивать встроенную среду PowerShell ISE и официально рекомендует разрабатывать сценарии PowerShell с помощью другой своей разработки — мощного, гибкого и универсального редактора Visual Studio Code.

Visual Studio Code — это бесплатная кросс-платформенная среда разработки. Для нее существует огромное количество расширений, позволяющих быстро и удобно писать код практически на любом языке программирования в операционных системах Windows, macOS и Linux.

Устанавливается Visual Studio Code стандартный образом, скачать установочный файл для Windows можно с официального сайта **https://code.visualstudio.com/** (рис. 15.11).

Сразу после установки Visual Studio Code поддерживает подсветку синтаксиса для сценариев на PowerShell и автоматическое форматирование и структурирование кода (напомним, что в PowerShell ISE этой возможности нет). При открытии файла с расширением ps1 редактор поймет, что это сценарий на языке PowerShell, и порекомендует установить расширение для поддержки этого языка, разработанное Microsoft.

Управлять расширениями Visual Studio Code можно на специальной панели, которая открывается комбинацией клавиш <Ctrl>+<Shift>+<x>. Установим здесь нужное нам расширение (рис, 15.12).

Также это расширение можно установить непосредственно из PowerShell, выполнив следующую команду:

```
PS C:\Users\andrv> code --install-extension ms-vscode.powershell
```

Помимо подсветки синтаксиса после установки расширения PowerShell в Visual Studio Code появляются дополнительные возможности для редактирования и отладки сценариев. Перечислим основные из них.

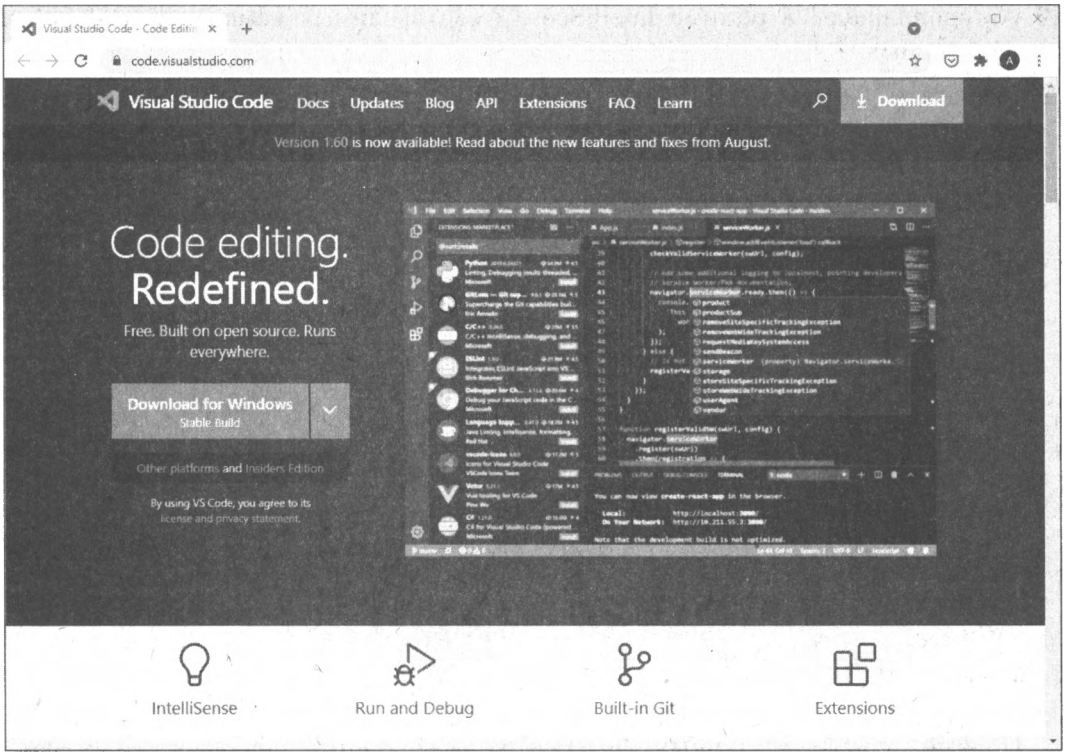


Рис. 15.11. Страница для установки Visual Studio Code

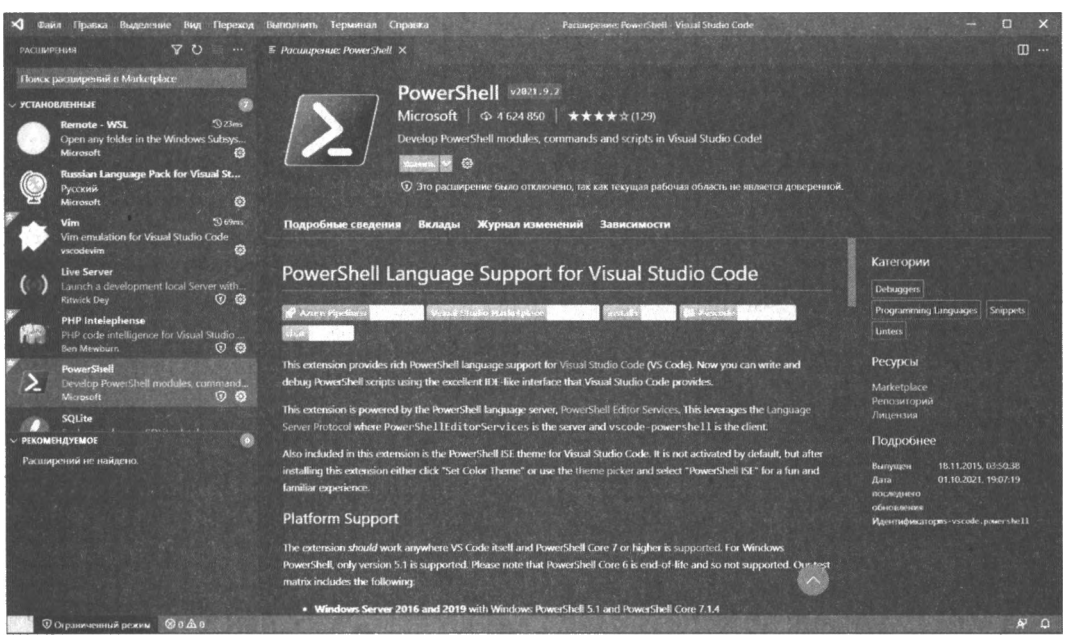


Рис. 15.12. Расширение для поддержки языка PowerShell в Visual Studio Code

- Автодополнение в режиме IntelliSense имен вводимых команд, параметров и переменных.
- Вставка в сценарий фрагментов стандартного кода (сниппетов). Нужный фрагмент можно выбрать из списка доступных вариантов, который появляется при вводе. Например, если выбрать сниппет `comment-help`, то в редактор вставится следующий код:

```
<#
.SYNOPSIS
    Short description
.DESCRIPTION
    Long description
.EXAMPLE
    PS C:\> <example usage>
    Explanation of what the example does
.INPUTS
    Inputs (if any)
.OUTPUTS
    Output (if any)
.NOTES
    General notes
#>
```

  Полный список доступных фрагментов открывается при нажатии клавиш <Ctrl>+<Alt>+<j>.
- Переход к строке в сценарии, где была инициализирована определенная переменная или описана функция. Для выполнения этого действия нужно щелкнуть правой кнопкой мыши на имени переменной или функции и в контекстном меню выбрать пункт **Перейти к определению** (Go To Definition). Также можно поместить курсор на нужное имя и нажать клавишу <F12>.
- Поиск всех строк в коде, где используется определенная переменная или функция. Для выполнения этого действия нужно щелкнуть правой кнопкой мыши на имени переменной или функции и в контекстном меню выбрать пункт **Перейти к ссылкам** (Find References). Также можно поместить курсор на нужное имя и нажать клавиши <Shift>+<F12>.
- Поиск в онлайн-справке по PowerShell для слова, в котором находится курсор, при нажатии клавиш <Ctrl>+<F1>.
- Автоматическая проверка сценария на соответствие определенным правилам написания кода с помощью *линтера* (статического анализатора) PowerShell Script Analyzer. Это позволяет писать более качественный код. Например, линтер выдаст предупреждение, если вы в тексте сценария используете псевдоним вместо полного наименования командлета или определили функцию, в имени которой используется нестандартный глагол.
- Проблемный код выделяется в редакторе волнистой линией, при наведении на него мыши можно посмотреть описание проблемы и исправить ее. Полный

список сообщений от линтера выводится на вкладке **Проблемы** (Problems) (рис. 15.13).

- Сценарии можно запускать в режиме отладки как полностью (клавиша <F5>), так и частично (выделив нужные строки и нажав клавишу <F8>). При этом отлаживаемый код будет запускаться в интегрированной консоли PowerShell (PowerShell Integrated Console), которая связана с редактируемым сценарием.

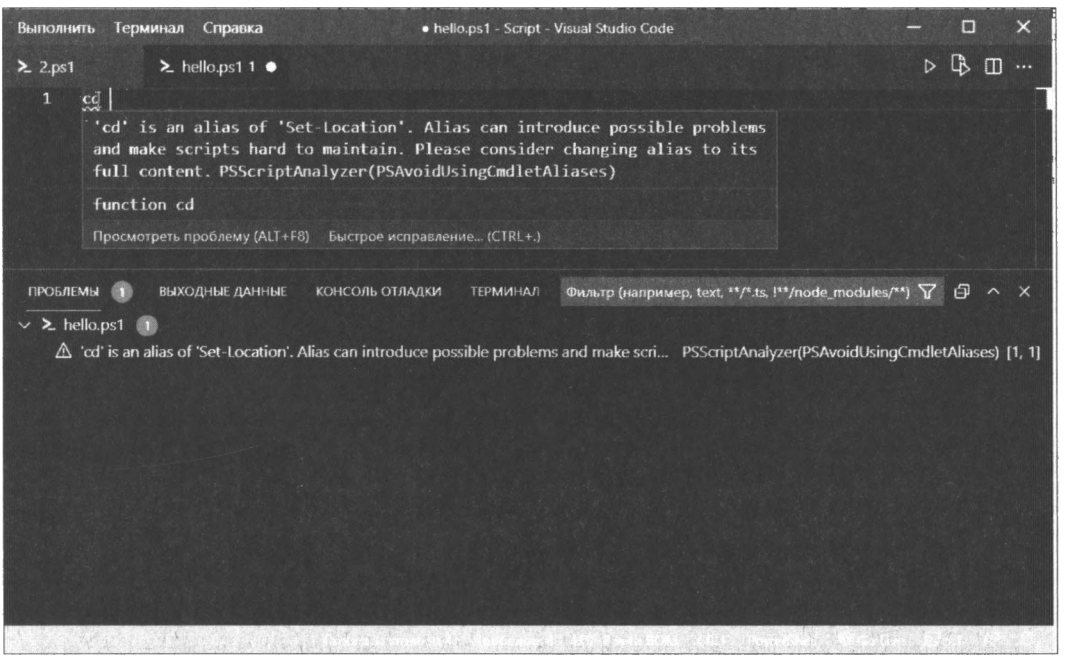


Рис. 15.13. Сообщения о проблемах в коде сценария от PowerShell Script Analyzer

Как и в PowerShell ISE, при отладке сценариев в VisualStudio Code можно работать с точками останова, которые устанавливаются на нужных строках клавишей <F9> (для снятия точки останова нужно нажать клавишу <F9> еще раз). Для продолжения приостановленного сценария используются те же горячие клавиши, что и в PowerShell ISE (см. табл. 15.1).

Все команды, используемые при отладке сценария, доступны в пункте меню **Выполнить** (Debug) (рис. 15.14).

В Visual Studio Code есть удобный инструмент — командная панель, с помощью которой можно, в частности, вызвать любую функцию редактора. Если открыть эту панель комбинацией клавиш <Ctrl>+<Shift>+<p> и ввести слово `powershell`, то будет выведен список всех команд расширения для поддержки PowerShell (рис. 15.15). При выборе команды из этого списка она будет выполнена.

Настройки самого редактора Visual Studio Code и установленных в нем расширений можно изменять в соответствии со своими предпочтениями с помощью пункта меню **Файл** (File) | **Настройки** (Preferences) | **Параметры** (Settings). В частности,

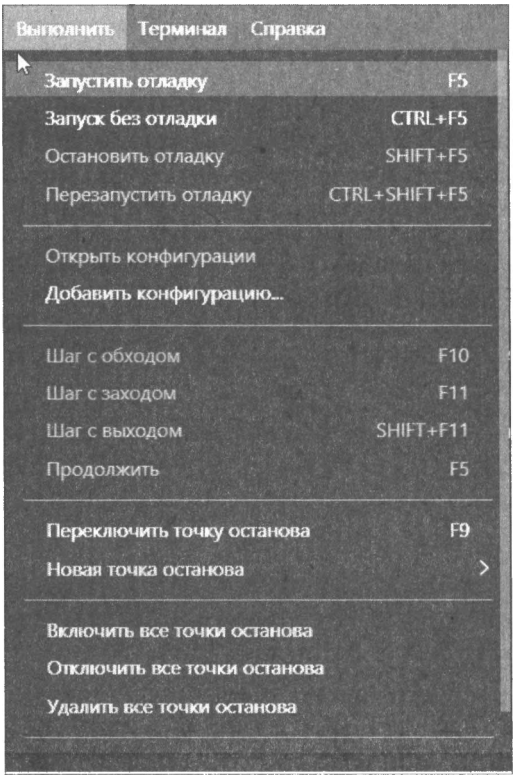


Рис. 15.14. Команды для отладки сценариев в Visual Studio Code

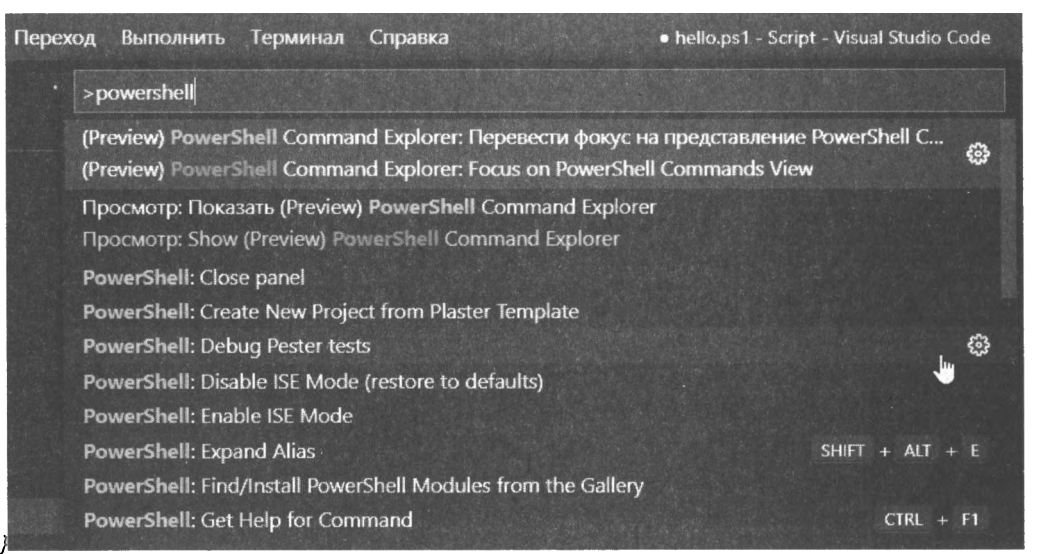


Рис. 15.15. Команды для отладки сценариев в Visual Studio Code

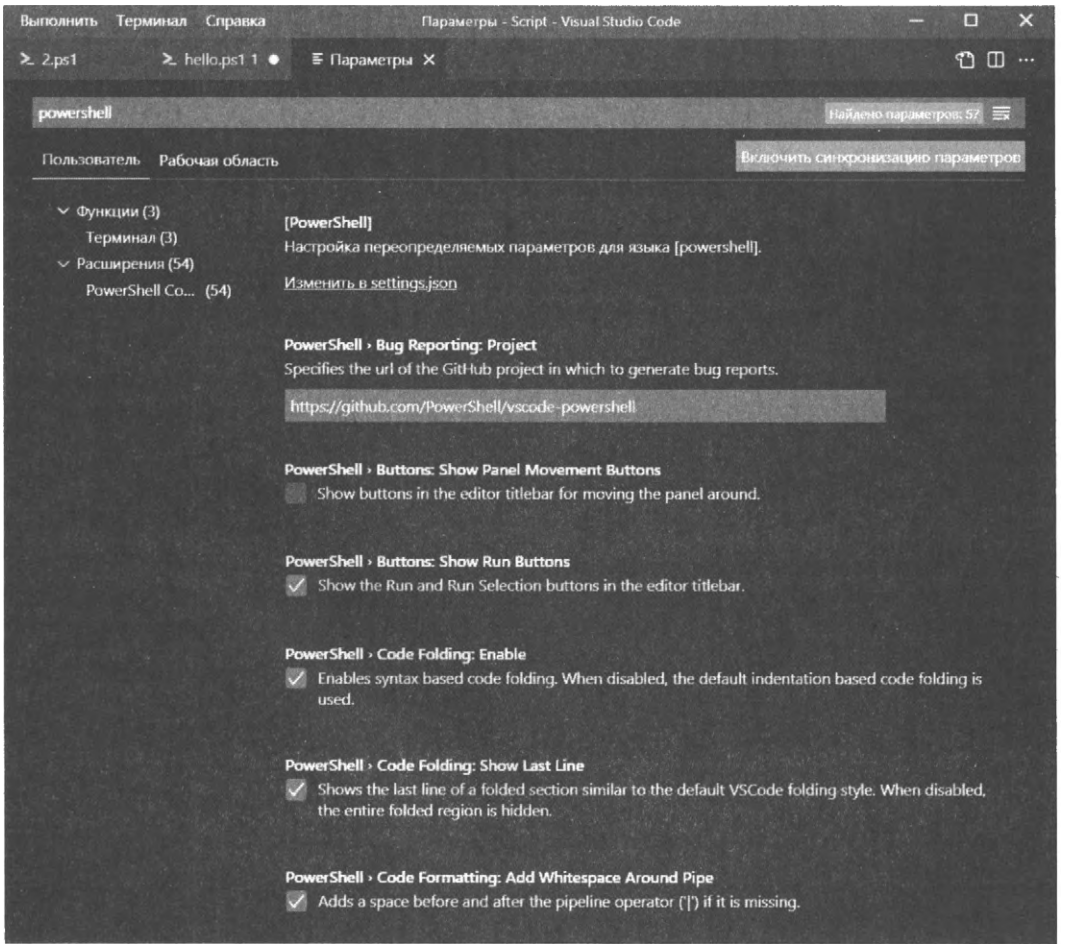


Рис. 15.16. Параметры расширения PowerShell в Visual Studio Code

здесь можно настроить различные параметры, связанные с форматированием кода и другими возможностями расширения PowerShell (рис. 15.16).

Другие редакторы и среды разработки

Несмотря на то, что PowerShell ISE и Visual Studio Code являются рекомендованными инструментами для разработки сценариев PowerShell, мы не ограничены только этими редакторами.

Существуют среды разработки (IDE), предназначенные специально для сценариев. К таким продуктам относятся, например, PowerShell Plus компании Idera (**https://www.idera.com/productssolutions/freetools/powershellplus**) или PrimalScript компании Sapien (**https://www.sapien.com/software/primalscript**).

Отдельно можно выделить очень мощную среду разработки Sapien PowerShell Studio (**https://www.sapien.com/software/powershell_studio**), позволяющую в том числе быстро создавать графический интерфейс для сценариев.

Конечно, прибегать к специальным IDE для PowerShell (как правило, это коммерческие платные продукты) имеет смысл, только если вы занимаетесь профессиональной разработкой больших и сложных сценариев. В большинстве случаев для написания и отладки скриптов будет достаточно бесплатного текстового редактора, с которым вы привыкли работать. Для большинства популярных универсальных редакторов (например, Notepad++, Sublime Text, Atom и Vim) можно найти плагин или расширение для поддержки языка PowerShell.

Рекомендации по разработке сценариев

Когда человек пишет программу, он общается не только с компьютером, который должен выполнить эту программу, но и с другими людьми, которые будут разбираться в его коде. Профессионализм программиста в большой степени связан с умением передать через исходный код свои знания другим людям. Как говорил Мартин Фаулер: "Любой дурак может написать код, понятный компьютеру. Хорошие программисты пишут код, понятный людям".

Оформляя программу или сценарий, мы объясняем и описываем, что же именно мы хотели тут сделать. Поэтому при написании сценария важно придерживаться определенных рекомендаций по оформлению и документированию кода. Это требует дополнительных усилий, однако если об этом не задумываться, то даже вам самим через месяц будет трудно разобраться в этом сценарии и вспомнить, что и как вы в нем делали.

Общая структура сценария

На простоту чтения и понимания сценариев влияет их структура и стиль, в котором написаны эти сценарии. Две программы, выполняющие одну и ту же задачу, могут восприниматься человеком по-разному. Приведем простой пример. Пусть первый сценарий hello1.ps1 состоит всего из одной строки:

```
"Привет всем!"
```

Второй сценарий hello2.ps1 напишем в следующем виде:

```
function WriteHello {
   $s = "Привет всем!"
   $s
}
WriteHello
```

Запустив эти сценарии, мы получим одинаковый результат — сценарий hello2.ps1 выводит на экран ту же строку, что и сценарий hello1.ps1. Однако второй сценарий намного больше, он содержит описание функции, поэтому для восприятия и анализа он сложнее.

Понятно, что структуры простейших сценариев, состоящих из одного или нескольких конвейеров команд, и сложных больших сценариев (с управляющими структурами, функциями, множеством операторов и т. п.) должны отличаться друг от дру-

га. Небольшие сценарии можно писать в "плоском" формате, без использования функций. Большие сценарии лучше разделять на логические части, определять функции, отдельно инициализировать все использующиеся переменные. Достоинства и недостатки возможных вариантов структур сценариев кратко описаны в табл. 15.2.

Таблица 15.2. *Сравнение сценариев, имеющих различную структуру*

Структура	Достоинства	Недостатки
В сценарии не используются функции	Самый простой вариант написания сценариев Хорошо подходит для небольших сценариев (до 100 строк), где отсутствует многократный вызов одного и того же кода	Тяжело выделить ключевые части сценария Могут возникнуть трудности с созданием библиотек сценариев, т к в сценариях нет функций
В сценарии используются функции, которые расположены в тексте неупорядоченно	Никаких достоинств	Подобный сценарий тяжело читать и изменять
Сценарий организован следующим образом 1 Раздел инициализации 2 Основное тело сценария 3 Все функции	Подобный сценарий легко читается Сценарий имеет ясную логическую структуру, поэтому нетрудно понять, как он работает	При разборе сценария постоянно приходится перемещаться по тексту от описания функций в основное тело сценария В небольших сценариях, где отсутствует многократный вызов одного и того же кода, данную структуру лучше не использовать
Сценарий организован следующим образом 1 Раздел инициализации 2 Все функции 3 Основное тело сценария	Подобная структура упрощает поиск основных операций, выполняемых в сценарии	При чтении кода приходится искать основное тело сценария, что может несколько осложнить разбор логики сценария
Сценарий организован следующим образом 1 Раздел инициализации 2 Вызов главной функции `Main`, содержащей основное тело сценария 3 Все остальные функции	Данный подход соответствует структуре приложений, принятой во многих языках программирования	При разборе сценария постоянно приходится перемещаться по тексту от описания функций в основное тело сценария

Имена и псевдонимы команд и параметров

Краткими псевдонимами удобно пользоваться в интерактивном сеансе оболочки для быстрого набора и однократного выполнения команды. С помощью псевдонимов можно быстро написать компактный и лаконичный конвейер из нескольких команд. Однако при создании сценариев не так важна скорость написания кода, как его понятность.

Поэтому нужно стараться использовать псевдонимы командлетов и сокращенные имена параметров только при интерактивной работе, а в сценариях всегда писать их полные имена. Вы потратите несколько лишних секунд при написании кода, но сэкономите намного больше времени человеку (или самому себе в будущем), который будет читать этот сценарий и разбираться в нем.

```
# Короткая команда с псевдонимами - быстрее набирать
ps | ? id -gt 1000

# Та же команда с полными названиями - проще читать и понимать
Get-Process | Where-Object id -gt 1000
```

По той же причине в сценариях рекомендуется явно указывать имена даже для тех параметров, которые используются в функциях и командлетах по умолчанию.

```
# Короткий вариант, подходит для интерактивного режима
dir C:\Windows

# Полный вариант команды для сценария
Get-ChildItem -Path C:\Windows
```

При использовании в функции позиционных параметров может возникнуть проблема: если сигнатура функции (т. е. состав и порядок перечисления ее параметров) в будущем изменится, то старый код может "сломаться". Поэтому позиционных параметров нужно избегать и стараться везде указывать именованные параметры.

При выборе наименования для расширенных функций и сценариев нужно придерживаться стандартного для PowerShell шаблона "действие — объект". При этом глаголы, обозначающие действие, лучше выбирать из списка рекомендованных глаголов, который выводится командой `Get-Verb`:

```
PS C:\Users\andrv> Get-Verb | Format-Wide -Column 4

Add         Clear       Close       Copy
Enter       Exit        Find        Format
Get         Hide        Join        Lock
Move        New         Open        Optimize
Pop         Push        Redo        Remove
Rename      Reset       Resize      Search
Select      Set         Show        Skip
Split       Step        Switch      Undo
Unlock      Watch       Backup      Checkpoint
Compare     Compress    Convert     ConvertFrom
ConvertTo   Dismount    Edit        Expand
Export      Group       Import      Initialize
Limit       Merge       Mount       Out
Publish     Restore     Save        Sync
Unpublish   Update      Approve     Assert
Complete    Confirm     Deny        Disable
Enable      Install     Invoke      Register
Request     Restart     Resume      Start
```

Stop	Submit	Suspend	Uninstall
Unregister	Wait	Debug	Measure
Ping	Repair	Resolve	Test
Trace	Connect	Disconnect	Read
Receive	Send	Write	Block
Grant	Protect	Revoke	Unblock
Unprotect	Use		

Имена параметров в собственных функциях и сценариях должны соответствовать аналогичным параметрам в стандартных командлетах. Например, если вам нужен параметр с именем компьютера, то назвать его нужно `-ComputerName`, как это сделано в командлетах `*-Computer`, `*-Process` или `*-EventLog`, а не `-Comp`, `-PC` или `-Station`.

Расширенные и базовые функции

Напомним, что PowerShell поддерживает два типа функций: базовые и расширенные *(см. гл. 9)*. Базовые функции имеет смысл применять только в простых случаях, когда функция состоит из нескольких строк. В остальных случаях используйте в своих сценариях расширенные функции и пользуйтесь их дополнительными возможностями.

Если в функцию могут поступать какие-либо параметры по конвейеру, то в ней обязательно нужно определить раздел `Process`.

Для возврата значения из функций не нужно использовать оператор `return`, достаточно просто указать в строке нужную переменную или объект. Для эффективного использования конвейеров возвращать значения в расширенных функциях следует в разделе `Process` (напомним, что он выполняется каждый раз при получении по конвейеру нового объекта), а не в разделах `Begin` или `End`.

В расширенных функциях всегда нужно указывать атрибут `[CmdletBinding()]`, в этом случае функция будет вести себя как командлет, в том числе будет поддерживать флаги `-Verbose` (вывод подробной информации о выполняемых командах) и `-Debug` (вывод отладочной информации).

Проверять на корректность (валидировать) значения параметров в расширенных функциях нужно с помощью специальных атрибутов `[Validate*]`, `[Allow*]`, а не писать для этого свой код внутри тела функции.

Вообще, всегда следует использовать встроенные возможности языка или платформы, а не изобретать велосипед. Это относится и к выбору команд PowerShell. Если задачу можно решить как с помощью "родных" команд PowerShell, так и через обращения к методам объектов COM, WMI или .NET, то предпочтение следует отдавать стандартным командлетам оболочки.

Комментарии

Комментарии помогают пояснить назначение и логику работы сценария целиком и отдельных его фрагментов. Рекомендуется комментировать следующие части сценария:

- весь сценарий в целом;
- все функции, объявленные в сценарии;
- большие управляющие структуры, в том числе не входящие в функции;
- все строки сценария, требующие дополнительного пояснения.

В языке PowerShell комментарии могут быть однострочными и блочными, состоящими из нескольких строк. Однострочные комментарии начинаются со знака #; все символы в строке, стоящие после этого знака, рассматриваются как часть комментария и не интерпретируются системой. Например:

```
PS C:\> # Вся строка - комментарий. Выражение 2+2 не вычисляется
PS C:\> 2 + 2 # Комментарий в конце строки. Выражение до комментария
вычисляется
4
```

Блочные (многострочные) комментарии располагаются между символами <# и #>:

```
<#
Пример
   комментария
из трех строк
#>
```

Комментарии нужно поддерживать в актуальном состоянии, обновлять их при внесении изменений в программный код. Если комментарий не соответствует коду, то это даже хуже полного отсутствия комментариев.

Если вы создаете сценарий не только для собственного использования, то комментарии нужно писать на английском языке. Текст комментария должен представлять собой законченное предложение. В случае короткого предложения точку в конце можно не ставить.

Справка, основанная на комментариях

При работе с PowerShell мы с помощью команды `Get-Help` постоянно обращаемся к справочной системе для просмотра синтаксиса стандартных встроенных командлетов, поддерживаемых ими параметров и примеров их использования.

Аналогичную справку можно реализовать и для своих функций и сценариев. Для этого достаточно добавить в сценарий или функцию многострочный комментарий, содержащий определенные теги (ключевые слова). После этого данная справка будет доступна в справочной системе PowerShell, ее можно будет просматривать через вызов `Get-Help`. Такой вид документирования работы сценария или функции называется *справкой, основанной на комментариях* (comment-based help).

В функциях комментарии для генерации справки можно размещать в одном из трех мест:

- В начале тела функции после открывающей фигурной скобки.
- В конце тела функции. В этом случае между последней строкой комментария и закрывающей фигурной скобкой, указывающей на завершение функции, должна быть добавлена пустая строка.

- Перед ключевым словом `function`. Между последней строкой комментария и словом `function` не должно быть более одной пустой строки.

В сценариях справочные комментарии располагаются либо в самом начале файла (в этом случае перед ними могут быть только обычные комментарии и пустые строки), либо в конце файла.

В качестве примера рассмотрим сценарий Out-Sum.ps1, который выводит на экран сумму двух своих аргументов:

```
param ( [int]$a, [int]$b )
$c = $a + $b
"$a + $b = $c"
```

Добавим к этому сценарию справочный комментарий с тегами `.SYNOPSIS` и `.DESCRIPTION`:

```
<#
.SYNOPSIS
   Сценарий Out-Sum
.DESCRIPTION
   Этот сценарий складывает два своих аргумента
и выводит сумму на экран
#>
param ( [int]$a, [int]$b )
$c = $a + $b
"$a + $b = $c"
```

Запросим у PowerShell справочную информацию о сценарии `Out-Sum`:

```
PS C:\Script> Get-Help .\Out-Sum.ps1

ИМЯ
    C:\Script\Out-Sum.ps1

ОПИСАНИЕ
    Сценарий Out-Sum

СИНТАКСИС
    C:\Script\Out-Sum.ps1 [[-a] <Int32>] [[-b] <Int32>] [<CommonParameters>]

ОПИСАНИЕ
    Этот сценарий складывает два своих аргумента
    и выводит сумму на экран

ССЫЛКИ ПО ТЕМЕ

ЗАМЕЧАНИЯ
  Для просмотра примеров введите: "get-help C:\Script\Out-Sum.ps1 -examples".
  Для получения дополнительных сведений введите: "get-help C:\Script\
                                                    Out-Sum.ps1 -detailed"
  Для получения технических сведений введите: "get-help C:\Script\Out-Sum.ps1
  -full".
```

Как видим, PowerShell показывает справку для сценария `Out-Sum.ps1`, содержащую краткое и полное описание сценария (текст берется из содержимого разделов `.SYNOPSIS` и `.DESCRIPTION` справочных комментариев)

Некоторые часто используемые в справочных комментариях теги приведены в табл. 15.3.

Таблица 15.3. *Теги в справочных комментариях*

Тег	Описание
`.SYNOPSIS`	Краткое описание функции или сценария
`.DESCRIPTION`	Детальное описание функции или сценария
`.PARAMETER`	Описание отдельного параметра
`.EXAMPLE`	Пример использования команды
`.INPUTS`	Тип объекта, который может поступать в команду по конвейеру
`.OUTPUTS`	Типы возвращаемых командой объектов
`.NOTES`	Дополнительная информация о функции или сценарии
`.LINK`	Имя связанного раздела в справочной системе или ссылка на сетевой ресурс, содержащий онлайн-версию данной справки

Во многих случаях нет необходимости в справочных комментариях использовать все имеющиеся теги. При написании сценариев и функций нужно стремиться к тому, чтобы код был самодокументируемым, понятным без дополнительных комментариев. Для этого самим сценариям, функциям и параметрам нужно давать полные, осмысленные имена.

Расположение и форматирование кода

Во многих языках программирования применяются стандарты и рекомендации по оформлению кода (например, PEP 8 для Python или PSR-1 и PSR-12 для PHP). Для чего они нужны, ведь оформление исходного кода никак не влияет на работу программы или сценария? Дело в том, что код, отформатированный привычным и стандартным для данного языка образом, легче читать и понимать.

Хотя Microsoft не предлагает официального руководства по стилю написания кода PowerShell, сообщество PowerShell поддерживает сборник лучших практик и рекомендация по оформлению кода, который доступен на GitHub: **https://github.com/PoshCode/PowerShellPracticeAndStyle**.

Регистр символов в именах

Для именования публичных идентификаторов, в том числе модулей, командлетов, функций, методов, констант и глобальных переменных, в PowerShell используется стиль PascalCase. В этом случае первая буква каждого слова, входящего в имя, пишется с заглавной буквы. Например: `$GlobalVariable`, `MyFunction`, `Sum-Arguments`.

Имена локальных переменных можно записывать в стиле camelCase, когда с заглавной буквы пишутся имена всех слов, входящих в имя, кроме первого слова. Например, `$args` или `$mySuperVariable`.

Названия ключевых слов и операторов языка PowerShell записываются строчными буквами (`if`, `for`, `switch`, `process`, `-eq`, `-match` и т. д.).

Скобки в коде

Фигурные скобки в коде PowerShell рекомендуется расставлять "единственно верным способом" (также его называют стилем Кернигана и Ритчи), когда открывающая скобка ставится в конце строки, а закрывающая — в начале. Например:

```
enum Color {
    Black,
    White
}

function Test-Code {
    [CmdletBinding()]
    param (
        [int]$ParameterOne
    )
    end {
        if (10 -gt $ParameterOne) {
            "Greater"
        } else {
            "Lesser"
        }
    }
}
```

Исключением из этого правила являются небольшие блоки кода, которые передаются в командлеты в качестве параметров. В этом случае и открывающую, и закрывающую скобку можно указать на одной строке, например:

```
Get-ChildItem | Where-Object { $_.Length -gt 10mb }
```

Отступы, пробелы и пустые строки

Вложенные структуры в коде PowerShell рекомендуется выделять четырьмя пробелами. Например:

```
function Add-Numbers {
    $sum = 0
    foreach ($number in 1..10) {
        $sum += $number
    }
    $sum
}
```

При наборе исходного кода для смещения начала строки на один уровень вправо удобнее пользоваться клавишей табуляции, поэтому в редакторе должна быть настроена автоматическая замена символа табуляции на четыре пробела.

Код будет лучше читаться, если в командах имена параметров или переменных отделять одним пробелом от операторов (в том числе от операторов присваивания и сравнения). Например:

```
# Запись без пробелов
$a=2+3
$a-eq5

# С пробелами код становится понятнее
$a = 2 + 3
$a -eq 5
```

Есть исключение из этого правила: дополнительными пробелами не следует выделять унарные операторы (т. е. операторы, имеющие один параметр). Например:

```
# Лишние пробелы в унарных операторах смотрятся плохо
$yesterdaysDate = (Get-Date).AddDays( - 1)

$i = 0
$i ++

# Без пробелов выглядит лучше
$yesterdaysDate = (Get-Date).AddDays(-1)

$i = 0
$i++
```

Одиночные пробелы нужно вставлять после запятой, точки с запятой, вокруг фигурных скобок и вокруг содержимого внутри скобок. Например:

```
# Вариант без пробелов
$Numbers = 1,2,3,10
Get-Service | Where-Object{$_.Status -eq "Stopped"}

# С дополнительными пробелами читается лучше
$Numbers = 1, 2, 3, 10
Get-Service | Where-Object { $_.Status -eq "Stopped" }
```

При описании простой функции рекомендуется разделять пробелом имя функции и список ее параметров в круглых скобках, например:

```
function MyFunction ($param1, $param2) {
    ...
}
```

Отдельные функции в сценарии нужно окружать двумя пустыми строками. Также рекомендуется ставить пустую строку в конце каждого файла-сценария.

В конце строки не должно быть "висячих" пробелов, т. е. пробельных неотображаемых символов.

Точка с запятой как разделитель строк и значений

С помощью точки с запятой можно разделять команды, операторы или выражения внутри одной строки. Например:

```
$a = 1; $b = 2; $c = $a + $b
```

В принципе, можно поставить точку с запятой и в конце строки, ошибки при этом не возникнет. Однако в PowerShell делать это не принято, специального символа для разделения строк указывать не нужно.

Разделитель можно убрать и при перечислении элементов хэш-таблицы, если они расположены на разных строках. Например:

```
$Options = @{
    Margin   = 2
    Padding  = 2
    FontSize = 24
}
```

Обратный апостроф для многострочных команд

С помощью обратного апострофа ` можно записать одну команду на нескольких строках. Это позволяет, например, более наглядно оформить вызов командлета или функции с несколькими именованными параметрами:

```
Get-WmiObject -Class Win32_LogicalDisk
              -Filter "DriveType=3"
              -ComputerName SERVER2
```

Однако рекомендуется по возможности избегать подобного использования обратного апострофа, т. к. он усложняет отладку сценариев. Этот символ трудно заметить при чтении, а если после него в строке будут стоять пробельные символы, то при выполнении сценария возникнет ошибка.

В подобных случаях лучше предварительно поместить имена и значения параметров в хэш-таблицу, присвоив ее переменной, а при вызове функции использовать сплаттинг этой переменной *(см. гл. 9)*:

```
$GetWmiObjectParams = @{
    Class = "Win32_LogicalDisk"
    Filter = "DriveType=3"
    ComputerName = "SERVER2"
}
Get-WmiObject @GetWmiObjectParams
```

Производительность сценариев и продуктивность разработчика

В качестве недостатков PowerShell иногда указывают большое потребление оперативной памяти и медленную работу сценариев по сравнению с другими оболочками и сценариями. Да, это действительно так, ведь за все в этом мире нужно платить.

Повышенное потребление ресурсов — это оборотная сторона удобства и мощных возможностей PowerShell. Команды и сценарии PowerShell не сразу выполняются интерпретатором, как это происходит в других оболочках. Они проходят через дополнительный слой .NET Framework, при этом создаются объекты, которые постоянно преобразуются к единому формату для передачи по конвейеру между командлетами.

Однако на практике вопросы производительности запускаемых сценариев редко являются основными критериями при выборе инструмента автоматизации. Сами базовые понятия языков сценариев выбираются из соображений выразительной мощности и простоты, а не в расчете на максимально эффективную проекцию на архитектуру компьютера.

Основной приоритет — повышение продуктивности труда разработчиков сценариев за счет упрощения используемого языка и применения в сценариях готовых компонентов. По сути сценарии — это склеивающие приложения, где сложность скрыта в используемых компонентах (внешних утилитах и объектах, имеющихся в операционной системе).

К сценариям для автоматизации своих действий в операционной системе Windows часто обращаются люди, не занимающиеся программированием профессионально. Для них самым главным является простота освоения языка и скорость написания сценариев, которые будут работать только на их компьютере. В этом случае оболочка PowerShell будет идеальным инструментом, ведь она позволяет практически со скоростью мысли создавать конвейеры из простых и понятных команд, которые потом можно легко оформить в виде сценария. При этом не нужно отвлекаться на сооружение дополнительных конструкций, как в сценариях WSH, или вспоминать названия утилит и их разнородных параметров и аргументов, как в традиционных оболочках командной строки.

К PowerShell в полной мере можно отнести описание языков сценариев из статьи "Природа и эволюция сценарных языков" Р. Богатырева: "Сценарный язык должен ориентироваться на скорость и простоту освоения базовых возможностей, быстро дающих видимый результат. Но из этого не следует, что язык должен быть примитивным. Путь к решению обычных типовых задач даже для начинающего должен быть коротким, простым и понятным. В этом отношении он должен напоминать родной язык: освоить несколько слов и фраз бывает достаточно, чтобы тебя понимали, однако научиться четко, грамотно и красиво выражать мысли на родном языке не всем и не всегда удается даже к глубокой старости".

Пример. Статистика по объектам файловой системы (cmd и PowerShell)

PowerShell легко и изящно справляется с автоматизацией задач вида "Получить множество каких-то объектов, выбрать из них нужные и обработать их" — человек должен только сказать, что нужно сделать, не задумываясь о деталях того, как это будет сделано. Для иллюстрации эффективности сценариев PowerShell с точки зрения времени и сложности разработки рассмотрим один практический пример.

Несколько филиалов банка ежедневно формируют операционные дневники в виде файлов od*.htm, которые записываются в папки ГГГГММДД\NNN, где *ГГГГММДД* — текущая дата, *NNN* — номер филиала. Например:

```
├──20201201
│   ├──001
│   ├──008
│   ├──011
│   ├──049
│   ├──067
│   ├──076
│   └──099
├──20201202
│   ├──001
│   ├──008
│   ├──011
│   ├──026
│   ├──067
│   ├──076
│   └──100
```

Нужно создать сценарий для сбора статистики, который для каждого филиала подсчитывал бы количество операционных дневников, созданных в течение месяца.

Сначала эта задача решалась в "полуавтоматическом" режиме с помощью командных файлов и электронных таблиц Microsoft Excel. Первый шаг — подготовка файла для анализа:

1. Формируем текстовый файл temp.txt с путями к каталогам, в которых есть дневники.
2. Сортируем строки в файле temp.txt по номеру филиала и выделяем из них этот номер.
3. Записываем результат в файл odb.txt.

Эти действия выполнялись командным файлом интерпретатора cmd.exe, в котором указывался нужный год и месяц:

```
@echo off
if exist temp.txt del temp.txt
if exist odb.txt del odb.txt

rem Формирование файла со списком каталогов, в которых есть
rem дневники за декабрь 2020 года
```

```
for /D %%f in (d:\odb\202012*) do (
   dir /s/b %%f\od*.htm  >> temp.txt)

rem Сортировка строк в файле по номеру филиала (17-й столбец),
rem выделение из путей этих номеров и запись их в файл odb.txt
for /f "delims=\ tokens=4" %%i in ('sort /+17 temp.txt') do (
  echo %%i >> odb.txt)
del temp.txt
```

Затем полученный файл odb.txt открывался Microsoft Excel, где с помощью стандартной функции формировались промежуточные итоги с подсчетом количества филиалов.

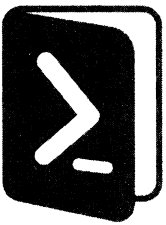
С помощью PowerShell данная задача решается в одну строку с помощью всего двух команд, объединенных в конвейер. Первая команда, `Get-ChildItem`, формирует список объектов-файлов, содержащихся в нужных каталогах, а вторая команда, `Group-Object`, группирует полученные объекты-файлы по имени и выводит на экран нужную нам информацию:

```
Get-ChildItem -Path D:\odb\202012*\*\od*.htm | Group-Object -Property
{$_.DirectoryName} -NoElement
```

Итоги

- В Windows имеется интегрированная среда сценариев PowerShell ISE, в которой можно писать и отлаживать сценарии PowerShell.
- Встроенный в ISE редактор предоставляет только базовые возможности по редактированию и форматированию кода, развивать этот продукт Microsoft перестала и рекомендует пользоваться своим бесплатным редактором Visual Studio Code с расширением для поддержки языка PowerShell.
- Существуют специальные среды разработки для сценариев PowerShell, которыми можно пользоваться при профессиональной работе с большими и сложными сценариями.
- При написании сценариев желательно придерживаться рекомендаций и лучших практик, принятых в сообществе PowerShell (**https://github.com/PoshCode/PowerShellPracticeAndStyle**).
- Для документирования сценариев в PowerShell используется механизм справки, основанной на комментариях, благодаря которому справка по конкретному сценарию включается в общую справочную систему PowerShell.

ГЛАВА 16



Отладка функций и сценариев

В PowerShell поддерживается несколько механизмов отладки сценариев, помогающих находить ошибки в программном коде, а также анализировать ход выполнения сценариев и состояние системы при этом. Мы рассмотрим лишь некоторые из них: вывод на консоль диагностических сообщений, трассировка выполняемых команд и установка точек прерывания.

Вывод диагностических сообщений

Самый простой метод отладки — это вывод на экран сообщений о ходе выполнения сценария. С помощью таких сообщений можно проверить значения переменных или убедиться, что определенная строка сценария выполнена. При этом желательно, чтобы диагностические сообщения выделялись от других цветом и не попадали в основной выходной поток `Output`, куда записываются все объекты, генерируемые командами PowerShell.

Такую функциональность предоставляет несколько командлетов с глаголом `Write-*`, которые мы рассматривали в *главе 6*, когда говорили о дополнительных потоках PowerShell.

Попробуем вывести сообщение с помощью `Write-Host`:

```
PS C:\Users\andrv> Write-Host "Тестовое сообщение"
Тестовое сообщение
```

Как видим, на экран выводится строка текста. Проверим, что при этом не генерируется никакой объект. Для этого присвоим переменной `$a` результат выполнения командлета `Write-Host` и убедимся, что переменная останется неопределенной (`$a` совпадает с `$null`):

```
PS C:\Users\andrv> $a = (Write-Host "Тестовое сообщение")
Тестовое сообщение
PS C:\Users\andrv> $a -eq $null
True
```

Командлет `Write-Host` имеет параметры `-ForegroundColor` и `-BackgroundColor`, позволяющие задать цвет текста и цвет фона соответственно. Например, следующая команда выведет желтое сообщение на черном фоне:

```
PS C:\Users\andrv> Write-Host "Тест" -ForegroundColor Yellow -BackgroundColor
Black
Тест
```

Если после вывода сообщения нужно приостановить выполнение сценария до нажатия клавиши <Enter>, то можно воспользоваться командлетом `Read-Host`:

```
PS C:\Users\andrv> Read-Host "Нажмите Enter"
Нажмите Enter

PS C:\Users\andrv>
```

Можно также приостановить выполнение сценария до нажатия произвольной клавиши, вызвав следующий метод:

```
PS C:\Users\andrv> $Host.UI.RawUI.ReadKey("NoEcho,IncludeKeyDown")

 VirtualKeyCode    Character    ControlKeyState   KeyDown
 --------------    ---------    ---------------   -------
             32                       NumLockOn      True
```

Как видим, в этом случае на экран выводится информация о нажатой на клавиатуре клавише. Для подавления этой информации можно перенаправить результат выполнения метода `ReadKey` на пустое устройство:

```
PS C:\Users\andrv> $Host.UI.RawUI.ReadKey("NoEcho,IncludeKeyDown") > $null
PS C:\Users\andrv>
```

Сообщения, которые выводятся с помощью `Write-Host`, будут отображаться на экране всегда. В PowerShell также есть командлеты `Write-Verbose` и `Write-Debug`, которые выводят сообщения в потоки `Verbose` и `Debug` соответственно. Сообщения из этих потоков отображаются на экране только при определенных режимах запуска функции или сценария с атрибутом `[CmdletBinding()]`.

В поток `Debug` записываются сообщения, используемые при отладке сценариев. Отображаются они только в том случае, если функция или сценарий запускаются с общим параметром `-Debug`. Например, введем следующую функцию `Test-Debug`:

```
PS C:\Users\andrv> function Test-Debug {
>>     [CmdletBinding()]
>>     param ()
>>     $a = 1
>>     Write-Debug "a = $a"
>>     $b = $a + 2
>>     Write-Host "b = $b"
>> }
```

Если просто вызывать эту функцию, то на экран будет выведено сообщение команды `Write-Host`, т. е. значение переменной `$b`:

```
PS C:\Users\andrv> Test-Debug
b = 3
```

Теперь запустим функцию `Test-Debug` с параметром `-Debug`:

```
PS C:\Users\andrv> Test-Debug -Debug
ОТЛАДКА: a = 1
Подтверждение
Продолжить выполнение текущей операции?
[Y] Да - Y  [A] Да для всех - A  [H] Прервать команду - H  [S]
Приостановить - S  [?] Справка
(значением по умолчанию является "Y"):
b = 3
```

Как видим, теперь на экран выводится отладочное сообщение из `Write-Debug` (значение переменной `$a`), после чего функция приостанавливается и у пользователя запрашивается режим дальнейшего выполнения.

Поток `Verbose` предназначен для вывода подробной информации о выполняемой операции. Например:

```
PS C:\Users\andrv> function Sum-TwoArgs {
>>     [CmdletBinding()]
>>     param (
>>         $a, $b
>>     )
>>     Write-Verbose "Функция вычисляет сумму двух аргументов: a=$a и b=$b"
>>     $a + $b
>> }
PS C:\Users\andrv> Sum-TwoArgs 1 2
3
PS C:\Users\andrv> Sum-TwoArgs 1 2 -Verbose
ПОДРОБНО: Функция вычисляет сумму двух аргументов: a=1 и b=2
3
```

Командлет *Set-PSDebug*

В PowerShell есть несколько командлетов для отладки сценариев. Самым первым из них появился командлет `Set-PSDebug`, параметры которого позволяют включить режимы трассировки и пошагового выполнения команд, а также режим обязательного объявления переменных (табл. 16.1).

Таблица 16.1. *Параметры командлета* `Set-PSDebug`

Параметр	Описание
`-Trace 0`	Отключение трассировки
`-Trace 1`	Включение основного режима трассировки
`-Trace 2`	Включение полного режима трассировки

Таблица 16.1 (окончание)

Параметр	Описание
-Step	Включение режима пошагового выполнения
-Strict	Включение режима обязательного объявления переменных
-Off	Отключение всех механизмов отладки

Рассмотрим режимы отладки более подробно.

Трассировка выполнения команд

Основной режим трассировки включается следующим образом:

```
PS C:\Users\andrv> Set-PSDebug -Trace 1
```

В данном режиме каждая команда, выполняемая интерпретатором, будет отображаться на экране. Например:

```
PS C:\Users\andrv> 3+2
ОТЛАДКА:    1+  >>>> 3+2
5
PS C:\Users\andrv> $s="abc"
ОТЛАДКА:    1+  >>>> $s="abc"
PS C:\Users\andrv> pwd
ОТЛАДКА:    1+  >>>> pwd

Path
----
C:\Users\andrv
```

Отладочная информация выводится на экран другим цветом и с префиксом ОТЛАДКА. Определим теперь функцию MyFunc:

```
PS C:\Users\andrv> function MyFunc {
>> $x = $args[0]
>> "x=$x"
>> }
>>
ОТЛАДКА:    1+ function MyFunc {
```

Вызовем данную функцию в цикле:

```
PS C:\Users\andrv> foreach ($i in 1..3) {MyFunc $i}
ОТЛАДКА:    1+ foreach ($i in  >>>> 1..3) {MyFunc $i}
ОТЛАДКА:    1+ foreach ( >>>> $i in 1..3) {MyFunc $i}
ОТЛАДКА:    1+ foreach ($i in 1..3) { >>>> MyFunc $i}
ОТЛАДКА:    1+ function MyFunc  >>>> {
ОТЛАДКА:    2+  >>>> $x = $args[0]
ОТЛАДКА:    3+  >>>> "x=$x"
x=1
```

```
ОТЛАДКА:    4+  >>>> }
ОТЛАДКА:    1+ foreach ( >>>> $i in 1..3) {MyFunc $i}
ОТЛАДКА:    1+ foreach ($i in 1..3) { >>>> MyFunc $i}
ОТЛАДКА:    1+ function MyFunc  >>>> {
ОТЛАДКА:    2+  >>>> $x = $args[0]
ОТЛАДКА:    3+  >>>> "x=$x"
x=2
ОТЛАДКА:    4+  >>>> }
ОТЛАДКА:    1+ foreach ( >>>> $i in 1..3) {MyFunc $i}
ОТЛАДКА:    1+ foreach ($i in 1..3) { >>>> MyFunc $i}
ОТЛАДКА:    1+ function MyFunc  >>>> {
ОТЛАДКА:    2+  >>>> $x = $args[0]
ОТЛАДКА:    3+  >>>> "x=$x"
x=3
ОТЛАДКА:    4+  >>>> }
ОТЛАДКА:    1+ foreach ( >>>> $i in 1..3) {MyFunc $i}
```

В данном случае на экран выводятся все выполняемые команды, однако вызовы функций и присваивания значений переменным явным образом не выделяются. Включим теперь полный режим трассировки и вновь вызовем функцию `MyFunc` в цикле:

```
PS C:\Users\andrv> Set-PSDebug -Trace 2
ОТЛАДКА:    1+  >>>> Set-PSDebug -Trace 2
PS C:\> foreach ($i in 1..3) {MyFunc $i}
PS C:\Users\andrv> foreach ($i in 1..3) {MyFunc $i}
ОТЛАДКА:    1+ foreach ($i in  >>>> 1..3) {MyFunc $i}
ОТЛАДКА:     ! CALL function '<ScriptBlock>'
ОТЛАДКА:     ! SET $foreach = 'IEnumerator'.
ОТЛАДКА:    1+ foreach ( >>>> $i in 1..3) {MyFunc $i}
ОТЛАДКА:     ! SET $i = '1'.
ОТЛАДКА:    1+ foreach ($i in 1..3) { >>>> MyFunc $i}
ОТЛАДКА:    1+ function MyFunc  >>>> {
ОТЛАДКА:     ! CALL function 'MyFunc'
ОТЛАДКА:    2+  >>>> $x = $args[0]
ОТЛАДКА:     ! SET $x = '1'.
ОТЛАДКА:    3+  >>>> "x=$x"
x=1
ОТЛАДКА:    4+  >>>> }
ОТЛАДКА:    1+ foreach ( >>>> $i in 1..3) {MyFunc $i}
ОТЛАДКА:     ! SET $i = '2'.
ОТЛАДКА:    1+ foreach ($i in 1..3) { >>>> MyFunc $i}
ОТЛАДКА:    1+ function MyFunc  >>>> {
ОТЛАДКА:     ! CALL function 'MyFunc'
ОТЛАДКА:    2+  >>>> $x = $args[0]
ОТЛАДКА:     ! SET $x = '2'.
ОТЛАДКА:    3+  >>>> "x=$x"
x=2
```

```
ОТЛАДКА:    4+  >>>> }
ОТЛАДКА:    1+ foreach ( >>>> $i in 1..3) {MyFunc $i}
ОТЛАДКА:     ! SET $i = '3'.
ОТЛАДКА:    1+ foreach ($i in 1..3) { >>>> MyFunc $i}
ОТЛАДКА:    1+ function MyFunc  >>>> {
ОТЛАДКА:     ! CALL function 'MyFunc'
ОТЛАДКА:    2+  >>>> $x = $args[0]
ОТЛАДКА:     ! SET $x = '3'.
ОТЛАДКА:    3+  >>>> "x=$x"
x=3
ОТЛАДКА:    4+  >>>> }
ОТЛАДКА:    1+ foreach ( >>>> $i in 1..3) {MyFunc $i}
ОТЛАДКА:     ! SET $foreach = ''.
```

Как видим, в полном режиме трассировки отдельно выделяются сообщения о вызове функций (префикс `CALL`) и о фактическом присвоении значений переменным (префикс `SET`).

Пошаговое выполнение команд

Рассмотрим еще один режим отладки, позволяющий запускать сценарии в пошаговом режиме (с подтверждением выполнения каждой строки). Включается режим пошагового выполнения команд следующим образом:

```
PS C:\Users\andrv> Set-PSDebug -Step
ОТЛАДКА:    1+  >>>> Set-PSDebug -Step
```

Вызовем в цикле функцию `MyFunc`, определенную в предыдущем разделе:

```
PS C:\Users\andrv> foreach ($i in 1..3) {MyFunc $i}

Продолжить выполнение текущей операции?
   1+ foreach ($i in  >>>> 1..3) {MyFunc $i}
[Y] Да - Y  [A] Да для всех - A  [N] Нет - N  [L] Нет для всех - L  [S]
Приостановить - S  [?] Справка
(значением по умолчанию является "Y"):Y
ОТЛАДКА:    1+ foreach ($i in  >>>> 1..3) {MyFunc $i}

Продолжить выполнение текущей операции?
   1+ foreach ( >>>> $i in 1..3) {MyFunc $i}
[Y] Да - Y  [A] Да для всех - A  [N] Нет - N  [L] Нет для всех - L  [S]
Приостановить - S  [?] Справка
(значением по умолчанию является "Y"):A
ОТЛАДКА:    1+ foreach ( >>>> $i in 1..3) {MyFunc $i}
x=1
x=2
x=3
```

В данном режиме на экран выводится текущая строка для исполнения, после чего система запрашивает действие, которое необходимо выполнить. Обратите внимание, что в пошаговом режиме, в отличие от режима трассировки, подробная отладочная информация о выполняемых командах не выводится.

Вложенная командная строка

В PowerShell реализована возможность рекурсивного вызова интерпретатора во время интерактивного сеанса работы в оболочке или при выполнении сценария. При этом запускается новый (назовем его вложенным) сеанс оболочки в том же командном окне, а текущий сеанс приостанавливается.

Другими словами, можно приостановить выполнение команд в текущем сеансе или сценарии и общаться с PowerShell, используя так называемую *вложенную командную строку*. Это позволяет выполнять команды PowerShell, анализирующие или изменяющие состояние приостановленного сеанса (не нужны отдельные специальные команды для отладки).

Создать новый вложенный сеанс можно в режиме пошагового выполнения команд, выбирая в запросе на продолжение выполнения текущей операции пункт **Приостановить**. Включим данный режим:

```
PS C:\Users\andrv> Set-PSDebug -Step
```

Запустим цикл, который будет выводить числа от 1 до 10:

```
PS C:\Users\andrv> $i=0; while ($i++ -lt 10) { $i }

Продолжить выполнение текущей операции?
   1+  >>>> $i=0; while ($i++ -lt 10) { $i }
[Y] Да - Y  [A] Да для всех - A  [N] Нет - N  [L] Нет для всех - L  [S]
Приостановить - S  [?] Справка
(значением по умолчанию является "Y"):
```

Нажимая клавишу <Enter>, продолжим выполнение шагов до тех пор, пока не будет в первый раз выведено значение переменной `$i`:

```
Продолжить выполнение текущей операции?
   1+ $i=0; while ( >>>> $i++ -lt 10) { $i }
[Y] Да - Y  [A] Да для всех - A  [N] Нет - N  [L] Нет для всех - L  [S]
Приостановить - S  [?] Справка
(значением по умолчанию является "Y"):
ОТЛАДКА:    1+ $i=0; while ( >>>> $i++ -lt 10) { $i }

Продолжить выполнение текущей операции?
   1+ $i=0; while ($i++ -lt 10) {  >>>> $i }
[Y] Да - Y  [A] Да для всех - A  [N] Нет - N  [L] Нет для всех - L  [S]
Приостановить - S  [?] Справка
(значением по умолчанию является "Y"):
ОТЛАДКА:    1+ $i=0; while ($i++ -lt 10) {  >>>> $i }
1
```

```
Продолжить выполнение текущей операции?
   1+ $i=0; while ( >>>> $i++ -lt 10) { $i }
[Y] Да - Y  [A] Да для всех - A  [N] Нет - N  [L] Нет для всех - L  [S]
Приостановить - S  [?] Справка
(значением по умолчанию является "Y"):
```

На этом шаге мы приостановим выполнение команды, нажав вместо <Enter> клавишу <S>:

```
Продолжить выполнение текущей операции?
   1+ $i=0; while ( >>>> $i++ -lt 10) { $i }
[Y] Да - Y  [A] Да для всех - A  [N] Нет - N  [L] Нет для всех - L  [S]
Приостановить - S  [?] Справка
(значением по умолчанию является "Y"):S
PS C:\Users\andrv>>
```

Итак, мы попали во вложенную командную строку (обратите внимание, что изменился вид приглашения). Другим индикатором работы во вложенном сеансе является ненулевое значение переменной `$NestedPromptLevel` (уровень вложения командной строки):

```
PS C:\Users\andrv>> $NestedPromptLevel
1
```

Во вложенном сеансе можно выполнять те же команды, что и в обычном сеансе PowerShell. Например, проверим, чему равно значение счетчика цикла `$i`:

```
PS C:\Users\andrv>> $i
1
```

На самом деле можно не только проверять значения переменных, но и изменять их. Например, запишем в переменную `$i` число 8:

```
PS C:\Users\andrv>> $i=8
PS C:\Users\andrv>>
```

Выйдем теперь из вложенной командной строки с помощью инструкции `Exit`. На экран вновь будет выдано предложение продолжить выполнение текущей операции:

```
PS C:\Users\andrv>> exit

Продолжить выполнение текущей операции?
   1+ $i=0; while ( >>>> $i++ -lt 10) { $i }
[Y] Да - Y  [A] Да для всех - A  [N] Нет - N  [L] Нет для всех - L  [S]
Приостановить - S  [?] Справка
(значением по умолчанию является "Y"):
```

Запустим все команды, нажав клавишу <A>:

```
Продолжить выполнение текущей операции?
   1+ $i=0; while ( >>>> $i++ -lt 10) { $i }
[Y] Да - Y  [A] Да для всех - A  [N] Нет - N  [L] Нет для всех - L  [S]
Приостановить - S  [?] Справка
(значением по умолчанию является "Y"):A
```

```
ОТЛАДКА:    1+ $i=0; while ( >>>> $i++ -lt 10) { $i }
9
10
PS C:\>
```

Как видим, переменная `$i` принимает значения 9 и 10, т. е. изменения, произведенные во вложенном сеансе, повлияли на приостановленный сеанс.

Итак, режим пошагового выполнения команд позволяет в любой момент вызвать вложенную командную строку для анализа или изменения состояния интерпретатора. Однако отлаживать более-менее большие сценарии с помощью данного метода часто оказывается неудобно, т. к. для запуска вложенного сеанса в определенной строке сценария придется каждый раз добираться до этой строки с самого начала сценария.

Гораздо удобнее было бы использовать *точки останова*, позволяющие запускать сценарий в автоматическом режиме и приостанавливать выполнение или запускать определенный код при достижении нужной строки.

Управление точками останова (командлеты *-PSBreakPoint)

Работать с точками останова в сценариях можно с помощью графического интерфейса в редакторах PowerShell ISE (пункт меню **Отладка** (Debug), рис. 16.1) или Visual Studio Code с установленным плагином поддержки PowerShell (пункт меню **Выполнить** (Run), рис. 16.2).

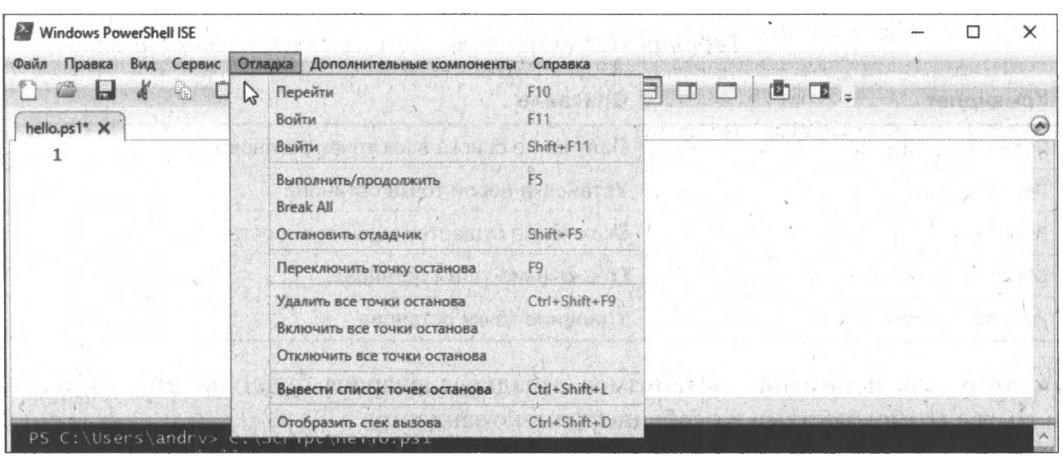


Рис. 16.1. Пункты меню для управления точками останова в PowerShell ISE

В обоих редакторах нажатием клавиши <F9> можно переключить точку останова на строке сценария, запустить выполнение сценария клавишей <F5>, пошагово выполнять команды с помощью клавиш <F10> и <F11>.

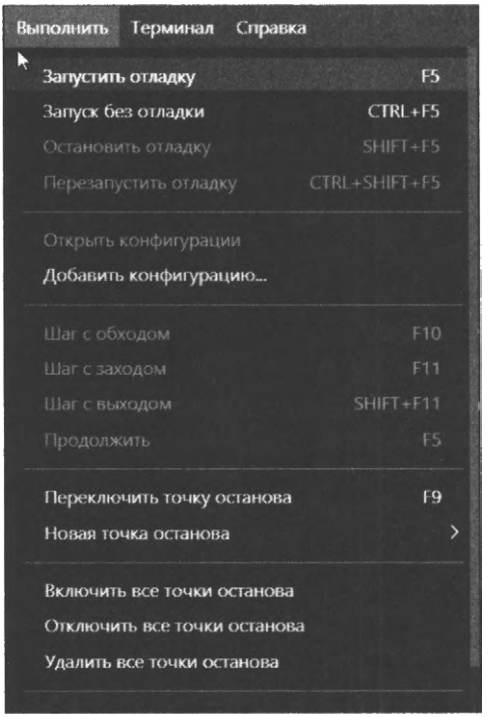


Рис. 16.2. Пункты меню для управления точками останова в Visual Studio Code

На самом деле инструменты для отладки скриптов, которые встроены в среды разработки, используют функциональность нескольких командлетов PowerShell, предназначенных для управления точками останова (табл. 16.2).

Таблица 16.2. *Командлеты для управления точками останова*

Командлет	Описание
`Get-PSBreakPoint`	Получение списка всех точек останова
`Set-PSBreakPoint`	Установка новой точки останова
`Enable-PSBreakPoint`	Включение существующей точки останова
`Disable-PSBreakPoint`	Отключение точки останова
`Remove-PSBreakPoint`	Удаление точки останова

Поэтому для понимания механизмов отладки сценариев будет полезно познакомиться с возможностями и особенностями командлетов `*-PSBreakPoint`.

Создание точки останова для сценария

Создадим в каталоге C:\Script сценарий fact.ps1 для вычисления факториала числа:

```
PS C:\Script> @'
>> $n = 8
```

```
>> "Вычисляем факториал $n"
>> $fact = 1
>> foreach ($i in 1..$n) {
>>     $fact = $fact * $i
>> }
>> "Факториал $n равен $fact"
>> '@ > fact.ps1
```

Установим с помощью команды `Set-PSBreakPoint` новую точку останова, которая сработает на пятой строке сценария fact.ps1:

```
PS C:\Script> $bp1 = Set-PSBreakpoint -Script .\fact.ps1 -Line 5 `
>> -Action {
>>     if ($i -gt 2 -and $i -lt 5)
>>     {
>>         Write-Host "*** ОТЛАДКА: i = $i, fact = $fact"
>>     }
>> }
```

Блок кода, указанный в качестве значения параметра `-Action`, будет срабатывать всякий раз перед выполнением пятой строки:

```
PS C:\Script> .\fact.ps1
Вычисляем факториал 8
*** ОТЛАДКА: i = 3, fact = 2
*** ОТЛАДКА: i = 4, fact = 6
Факториал 8 равен 40320
```

Команда `Set-PSBreakPoint` возвращает объект типа `System.Management.Automation.LineBreakpoint`, который соответствует созданной точке останова:

```
PS C:\Script> $bp1 | Get-Member

   TypeName: System.Management.Automation.LineBreakpoint

Name        MemberType Definition
----        ---------- ----------
Equals      Method     bool Equals(System.Object obj)
GetHashCode Method     int GetHashCode()
GetType     Method     type GetType()
ToString    Method     string ToString()
Action      Property   scriptblock Action {get;}
Column      Property   int Column {get;}
Enabled     Property   bool Enabled {get;}
HitCount    Property   int HitCount {get;}
Id          Property   int Id {get;}
Line        Property   int Line {get;}
Script      Property   string Script {get;}
```

Посмотрим значение свойств объекта `$bp1`:

```
PS C:\Script> $bp1 | Format-List *

Column   : 0
Line     : 5
Action   :
                if ($i -gt 2 -and $i -lt 5)
                {
                    Write-Host "*** ОТЛАДКА: i = $i, fact = $fact"
                }

Enabled  : True
HitCount : 8
Id       : 0
Script   : C:\Script\fact.ps1
```

Как видим, каждая точка останова имеет свой целочисленный идентификатор `Id` (нумерация начинается с нуля). В объекте хранятся: путь к отлаживаемому сценарию (свойство `Script`), номер строки в сценарии (свойство `Line`), статус точки останова (свойство `Enabled`). В свойстве `Action` записан блок кода, который будет выполнен при срабатывании точки останова, а в свойство `HitCount` заносится количество срабатываний данной точки останова в сценарии.

Если в блок кода `-Action` поместить команду `Break`, то выполнение сценария будет приостановлено. Создадим такую точку останова для нашего скрипта fact.ps1:

```
PS C:\Script> $bp2 = Set-PSBreakpoint -Script .\fact.ps1 -Line 5 `
>> -Action {
>>      if ($i -eq 3)
>>      {
>>          Write-Host "!!! ОТЛАДКА: i = $i, fact = $fact"
>>          break
>>      }
>> }
```

Запустим сценарий fact.ps1:

```
PS C:\Script> .\fact.ps1
Вычисляем факториал 8
*** ОТЛАДКА: i = 3, fact = 2
!!! ОТЛАДКА: i = 3, fact = 2
Выполняется запуск в режиме отладки. Введите "h" или "?" для получения справки.

Выполнение Точка останова строки "C:\Script\fact.ps1:5"

C:\Script\fact.ps1:5 знак:5
+     $fact = $fact * $i
+     ~~~~~~~~~~~~~~~~~~
[DBG]: PS C:\Script>>
```

В этом режиме мы видим строку сценария, перед которой выполнение приостановилось. Вид приглашения командной строки изменился, мы находимся в режиме отладки во вложенной командной строке. Список команд для отладки можно посмотреть, введя h или ?:

```
[DBG]: PS C:\Script>> h

 s, stepInto          Выполнение одного шага (выполнение шага со входом
                      в функции, сценарии и т. д.)
 v, stepOver          Выполнение шага с переходом к следующей инструкции (шаг
                      с обходом функций, сценариев и т. д.)
 o, stepOut           Выполнение шага с выходом из текущей функции, сценария
                      и т. д.

 c, continue          Продолжение операции
 q, quit              Прекращение операции и завершение работы отладчика
 d, detach            Продолжить операцию и отсоединить отладчик.

 k, Get-PSCallStack   Отображение стека вызова

 l, list              Вывод на экран исходного кода текущего сценария.
                      Используйте команду "list", чтобы перейти в начало текущей
                      строки, команду "list <m>", чтобы перейти в начало строки
                      <m>, или команду "list <m> <n>", чтобы вывести <n> строк,
                      начиная со строки <m>

 <ввод>               Повторение предыдущей команды, если это была команда
                      stepInto, stepOver или list

 ?, h                 Выводит эту справку.

Чтобы получить указания о настройке командной строки отладчика, введите "help
about_prompt".

[DBG]: PS C:\Script>>
```

Итак, во вложенной командной строке отладчика доступны все стандартные операции отладки. Например, чтобы продолжить выполнение сценария, нужно ввести команду c (Continue):

```
[DBG]: PS C:\Script>> c
*** ОТЛАДКА: i = 4, fact = 6
Факториал 8 равен 40320
PS C:\Script>
```

Здесь мы видим еще одно отладочное сообщение от первой точки останова и итоговое значение вычисленного факториала. Отладка и выполнение сценария завершены, командная строка вернулась в обычный режим ввода команд PowerShell.

Создание точки останова для команды

Можно установить точку останова не только на строку в файле сценария, но и на определенную команду.

Например, создадим функцию `Say-Hello` и создадим для нее точку останова (имя команды указывается в параметре `-Command`):

```
PS C:\Script> function Say-Hello {'Привет!'}
PS C:\Script> Set-PSBreakpoint -Command Say-Hello

  ID Script                  Line Command
  -- ------                  ---- -------
   2                               Say-Hello
```

Теперь при выполнении команды `Say-Hello` мы сразу войдем в режим отладки:

```
PS C:\Script> Say-Hello
Выполнение Точка останова команды на "Say-Hello"

строка:1 знак:20
+ function Say-Hello {'Привет!'}
+                    ~
[DBG]: PS C:\Script>>
```

Здесь доступны те же команды для отладки, которые мы рассматривали в предыдущем разделе. Например, выполним код нашей функции в пошаговом режиме (команда `s`):

```
[DBG]: PS C:\Script>> s
строка:1 знак:21
+ function Say-Hello {'Привет!'}
+                     ~~~~~~~~~
[DBG]: PS C:\Script>> s
Привет!
строка:1 знак:30
+ function Say-Hello {'Привет!'}
+                              ~
[DBG]: PS C:\Script>> s
PS C:\Script>
```

Создание точки останова для переменной

В PowerShell можно создать точку останова, которая будет срабатывать при чтении или записи определенной переменной.

Например, создадим подобную точку для входа в режим отладки в момент записи значения в переменную `fact` в сценарии fact.ps1:

```
PS C:\Script> $bp3 = Set-PSBreakpoint -Script .\fact.ps1 `
>> -Variable fact -Mode Write -Action {
```

```
>>     if ($fact -gt 6)
>>     {
>>         Write-Host ">>> В переменную fact записано значение $fact"
>>     }
>> }
PS C:\Script>
```

Запустим сценарий fact.ps1, на который действуют уже три точки останова (первые две мы установили ранее для пятой строки):

```
PS C:\Script> .\fact.ps1
Вычисляем факториал 8
*** ОТЛАДКА: i = 3, fact = 2
!!! ОТЛАДКА: i = 3, fact = 2
Выполнение Точка останова строки "C:\Script\fact.ps1:5"

C:\Script\fact.ps1:5 знак:5
+     $fact = $fact * $i
+     ~~~~~~~~~~~~~~~~~~
[DBG]: PS C:\Script>>
```

После приостановки сценария введем в режиме отладки команду `c`, чтобы продолжить выполнение:

```
[DBG]: PS C:\Script>> c
*** ОТЛАДКА: i = 4, fact = 6
>>> В переменную fact записано значение 24
>>> В переменную fact записано значение 120
>>> В переменную fact записано значение 720
>>> В переменную fact записано значение 5040
>>> В переменную fact записано значение 40320
Факториал 8 равен 40320
```

Как видим, блок кода, заданный для точки останова при записи нового значения в переменную `$fact`, срабатывает, сообщения на экран выводятся.

Просмотр точек останова

Команда `Get-PSBreakPoint` выводит список точек останова, созданных в текущей сессии (при следующем запуске оболочки их не будет):

```
PS C:\Script> Get-PSBreakpoint | Format-List *

Column   : 0
Line     : 5
Action   :
                if ($i -gt 2 -and $i -lt 5)
                {
                    Write-Host "*** ОТЛАДКА: i = $i, fact = $fact"
                }
```

```
Enabled  : True
HitCount : 24
Id       : 0
Script   : C:\Script\fact.ps1
Column   : 0
Line     : 5
Action   :
                if ($i -eq 3)
                {
                    Write-Host "!!! ОТЛАДКА: i = $i, fact = $fact"
                    break
                }
Enabled  : True
HitCount : 16
Id       : 1
Script   : C:\Script\fact.ps1

Command  : Say-Hello
Action   :
Enabled  : True
HitCount : 1
Id       : 2
Script   :

AccessMode : Write
Variable   : fact
Action     :
                if ($fact -gt 6)
                {
                    Write-Host ">>> В переменную fact записано значение $fact"
                }

Enabled    : True
HitCount   : 9
Id         : 3
Script     : C:\Script\fact.ps1
```

Как видим, установлены три точки останова с идентификаторами 0, 1 и 2. Параметр `-Type` командлета `Get-PSBreakPoint` позволяет получить список точек останова определенного типа. Например, следующая команда покажет точки останова для команд:

```
PS C:\Script> Get-PSBreakpoint -Type Command | Format-List *

Command  : Say-Hello
Action   :
Enabled  : True
HitCount : 1
```

```
Id        : 2
Script    :
```

Посмотреть точки останова, установленные для строк сценария fact.ps1, можно так:

```
PS C:\Script> Get-PSBreakpoint -Script fact.ps1 -Type Line | Format-List *

Column    : 0
Line      : 5
Action    :
                if ($i -gt 2 -and $i -lt 5)
                {
                    Write-Host "*** ОТЛАДКА: i = $i, fact = $fact"
                }
Enabled   : True
HitCount  : 24
Id        : 0
Script    : C:\Script\fact.ps1

Column    : 0
Line      : 5
Action    :
                 if ($i -eq 3)
                 {
                     Write-Host "!!! ОТЛАДКА: i = $i, fact = $fact"
                     break
                 }
Enabled   : True
HitCount  : 16
Id        : 1
Script    : C:\Script\fact.ps1
```

Удаление точек останова

Удалить точки останова можно командой `Remove-PSBreakPoint`.

Например, удалим точку останова с идентификатором 2:

```
PS C:\Script> Remove-PSBreakpoint -Id 2
PS C:\Script> Get-PSBreakpoint | Format-Table Id, Script, Line, Variable

Id Script              Line Variable
-- ------              ---- --------
 0 C:\Script\fact.ps1    5
 1 C:\Script\fact.ps1    5
 3 C:\Script\fact.ps1      fact
```

Для удаления нескольких (или всех) точек останова нужно сформировать их список с помощью команды `Get-PSBreakPoint` и передать его по конвейеру команде `Remove-PSBreakPoint`. Удалим все наши точки останова:

```
PS C:\Script> (Get-PSBreakpoint).count
3
PS C:\Script> Get-PSBreakpoint | Remove-PSBreakpoint
PS C:\Script> (Get-PSBreakpoint).count
0
```

Итоги

- Диагностические сообщения для отладки сценариев можно выводить с помощью командлетов `Write-Host`, `Write-Verbose`, `Write-Debug`. В основной выходной поток `Output` эти сообщения попадать не будут.
- Режимы трассировки или пошагового выполнения сценариев можно включить командой `Set-PSDebug`.
- Мощным и удобным инструментом для отладки являются точки останова, работать с которыми можно в графических средах разработки сценариев PowerShell ISE и Visual Studio Code.
- Для управления точками останова из командной строки используются командлеты `*-PSBreakPoint`.

ГЛАВА 17

Графический интерфейс для сценариев

Работая в терминале, мы управляем сценариями с помощью параметров, которые указываются в командной строке при их вызове. Если с нашим сценарием будут работать обычные пользователи, то, добавив к сценарию простой графический интерфейс (GUI), можно избавить их от необходимости запускать терминал и вводить названия параметров.

В Windows PowerShell доступны два варианта создания GUI с помощью объектов .NET Framework. Классические элементы интерфейса Windows можно добавить с помощью библиотеки .NET Windows Forms (WinForms), которая поддерживалась в .NET с самого начала. Более современный вариант дизайна реализуется с помощью технологии Windows Presentation Foundation (WPF), которая была добавлена позже в .NET Framework 3.0.

Построение GUI с помощью Windows Forms

Давайте создадим простейшую форму с надписью и кнопкой, выполняя команды непосредственно в консоли PowerShell.

Как мы знаем, платформа .NET построена таким образом, что для обращения к тем или иным объектам нужно предварительно загрузить в операционную память соответствующую *сборку* (assembly) — динамическую библиотеку определенного вида. Наиболее часто использующиеся сборки загружаются в PowerShell автоматически. Для обращения к объектам WinForms из PowerShell нужно сначала загрузить сборку, поддерживающую эти объекты:

```
PS C:\Users\andrv> Add-Type -Assemblyname System.Windows.Forms
```

Графический интерфейс в терминах WinForms — это набор *элементов управления* (кнопки, поля ввода, списки, флажки и т. д.), которые располагаются в *контейнерах* (формы, панели, вкладки и т. д.). Изменяя значения свойств объектов, соответствующих элементам управления, мы определяем внешний вид этих элементов (размер, цвет фона, используемый шрифт и т. д.).

Любое графическое приложение WinForms должно иметь главную форму — объект типа `Windows.Forms.Form`. Создадим нашу главную форму и сохраним соответствующий объект в переменной `$form`:

```
PS C:\Users\andrv> $form = New-Object System.Windows.Forms.Form
```

Наша форма пока находится только в оперативной памяти, на экране мы ее не видим.

Зададим заголовок формы (свойство `Text` объекта `$form`):

```
PS C:\Users\andrv> $form.Text = 'Первая форма'
```

Поместим на форму надпись — для этого нужно создать объект типа `System.Windows.Forms.Label` и присвоить значение свойству `Text` этого объекта:

```
PS C:\Users\andrv> $label = New-Object System.Windows.Forms.Label
PS C:\Users\andrv> $label.Text = 'Привет!'
```

Кроме надписи на форму можно помещать другие элементы управления, которым соответствуют объекты из пространства имен `System.Windows.Forms`. Некоторые из этих объектов представлены в табл. 17.1.

Таблица 17.1. *Объекты, соответствующие элементам управления*

Объект	Описание
`Button`	Кнопка на форме Надпись на кнопке задается свойством `Text`
`CheckBox`	Флажок (чекбокс) — элемент управления для выбора одного или нескольких пунктов из предложенных вариантов
`CheckedListBox`	Состоит из списка элементов (ListBox), перед каждым из которых помещен флажок (чекбокс) для возможности выбора
`ComboBox`	Состоит из поля для ввода текста (TextBox) и связанного с ним раскрывающегося списка значений (ListBox)
`DataGridView`	Основной элемент управления для представления табличных данных (поддерживает связь с базой данных)
`GroupBox`	Используется для выделения групп элементов управления (чаще всего радиокнопок), возле которых отображается рамка
`ImageList`	Контейнер для размещения коллекции изображений, используемых другими элементами управления (например, ListView и TreeView)
`ListBox`	Список элементов, которые могут быть выбраны Может содержать простой текст или объекты
`Panel`	Видимый или невидимый контейнер для группировки элементов управления
`PictureBox`	Используется для размещения графических изображений в нескольких стандартных форматах
`ProgressBar`	Индикатор хода выполнения какого-либо процесса
`RadioButton`	Переключатель (радиокнопка) для выбора одного варианта из списка предложенных

Таблица 17.1 *(окончание)*

Объект	Описание
TextBox	Элемент для ввода текста в одной или нескольких строках
TreeView	Отображение данных в виде узлов дерева

Для задания размеров и расположения элементов управления используются объекты типа System.Drawing.Point, в которые передаются координаты: первая — по горизонтали, вторая — по вертикали (начало координат размещается в верхнем левом углу, ось X направлена вправо, ось Y — вниз). Нашу надпись мы сместим от верхнего левого угла на 20 пикселов вправо и 10 пикселов вниз. Объект с нужными координатами записывается в свойство Location объекта $label:

```
PS C:\Users\andrv> $label.Location = New-Object System.Drawing.Point 20,10
```

Пока надпись представляет собой отдельный объект, который не связан с основной формой. Поместим надпись на форму, добавив объект $label в коллекцию $form.Controls с помощью метода Add:

```
PS C:\Users\andrv> $form.Controls.Add($label)
```

Теперь создадим кнопку (объект типа Windows.Forms.Button) с надписью "Нажми!":

```
PS C:\Users\andrv> $button = New-Object Windows.Forms.Button
PS C:\Users\andrv> $button.Text = "Нажми!"
```

Зададим координаты для кнопки:

```
PS C:\Users\andrv> $button.Location = New-Object System.Drawing.Point 20,40
```

Поместим кнопку на форму:

```
PS C:\Users\andrv> $form.Controls.Add($button)
```

Теперь добавим нашей форме функциональность — по нажатии кнопки $button текст в надписи $label будет меняться на "Пока!". Для этого с помощью метода Add_Click нужно определить обработчик события Click на кнопке. В этот метод передается блок кода, который будет выполнен при нажатии кнопки. В нашем случае достаточно изменить значение свойства $label.Text:

```
PS C:\Users\andrv> $button.Add_Click({$label.Text='Пока!'})
```

Если бы обработчик события был более объемным, то удобнее было бы заранее сохранить соответствующий блок кода в переменной, которую затем передать в метод Add_Click.

Теперь с помощью метода ShowDialog можно отобразить созданную форму с элементами управления на экране (рис. 17.1).

```
PS C:\Users\andrv> $form.ShowDialog()
```

По нажатии кнопки надпись на форме изменится (рис. 17.2).

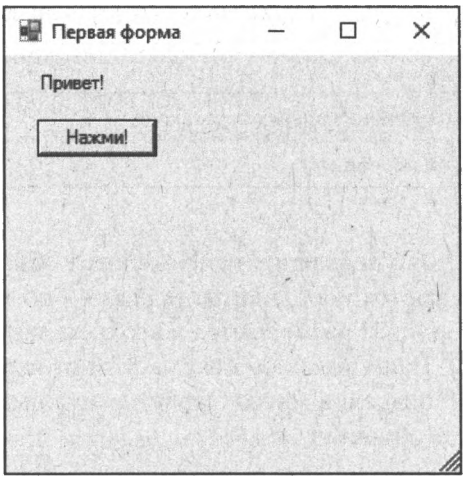


Рис. 17.1. Форма, созданная в PowerShell

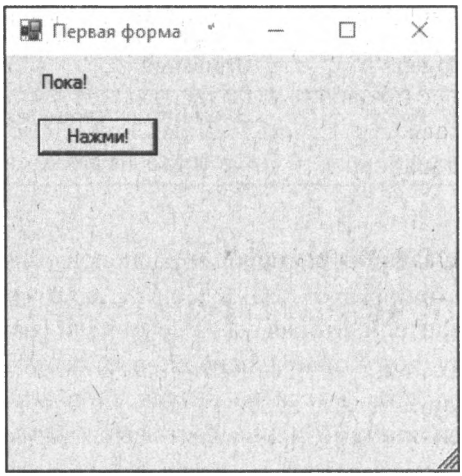


Рис. 17.2. Изменения в элементе на форме

Итак, мы рассмотрели простейший пример создания формы в PowerShell с помощью библиотеки WinForms. Более сложные интерфейсы реализуются по той же схеме:

1. Загружаем сборку `System.Windows.Forms`.
2. Создаем объекты, соответствующие главной форме, контейнерам и элементам управления.
3. Привязываем элементы управления к нужным контейнерам и форме.
4. Задаем свойства элементов управления, определяющие их внешний вид и расположение внутри контейнеров.
5. Пишем блоки кода для обработчиков событий и привязываем их к элементам управления.
6. Показываем главную форму на экране.

Как видим, при создании графического интерфейса на PowerShell с помощью WinForms значительная часть работы будет приходиться на написание шаблонного кода для создания элементов управления и задания их свойств. Конечно, намного удобнее и проще было бы строить интерфейс с помощью визуального конструктора с дальнейшей автоматической генерацией соответствующего кода на языке PowerShell. Подобные конструкторы есть в некоторых платных профессиональных средах разработки для PowerShell, например в Sapien PowerShell Studio (**https://www.sapien.com/software/powershell_studio**), или в сервисе PowerShell GUI Designer (**https://poshgui.com/**). Из бесплатных инструментов можно выделить утилиту PowerShell WinForms Creator (**https://www.pswinformscreator.com/**), которая сама написана на PowerShell.

Таким образом, при работе с WinForms код для описания интерфейса (создание объектов для элементов управления) оказывается тесно связанным с кодом, определяющим функциональность сценария (обработчики событий на элементах управ-

ления и другие функции). Вносить изменения в такие сценарии неудобно, они быстро разрастаются и становятся слишком объемными и сложными для понимания.

Более правильный подход, когда декларативное описание графического интерфейса отделяется от остального кода, обеспечивает технология Windows Presentation Foundation (WPF).

Построение GUI с помощью Windows Presentation Foundation

При использовании WPF графический интерфейс описывается с помощью базирующегося на XML языка разметки XAML (Extensible Application Markup Language). Таким образом, дизайн интерфейса отделяется от логики сценария, работающего с этим интерфейсом.

Для работы с объектами WPF в PowerShell нужно загрузить сборку PresentationFramework:

```
PS C:\Users\andrv> Add-Type -AssemblyName PresentationFramework
```

Создадим сначала простейшее диалоговое окно без элементов размером 300×200 пикселов и с заголовком "Форма WPF". На первом шаге с помощью here-string создадим строку `$xaml` с XML-разметкой, содержащей один тег `<Window/>` с атрибутами `Width`, `Height` и `Title`, и преобразуем эту строку в объект XML:

```
PS C:\Users\andrv> [xml]$xaml = @"
>> <Window
>>     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
>>     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
>>     x:Name="Window" Title="Форма WPF" Height="200" Width="300"
>> />
>> "@
```

Атрибут `xmlns` здесь определяет пространство имен, содержащее имена элементов и атрибутов, которые можно использовать в разметке Windows Presentation Foundation. Префикс `x` перед атрибутом указывает на то, что данный атрибут относится к данному пространству имен.

Создадим теперь экземпляр класса `XmlNodeReader`, передав ему объект `$xaml` в качестве аргумента. Полученный объект сохраним в переменной `$reader`:

```
PS C:\Users\andrv> $reader = (New-Object System.Xml.XmlNodeReader $xaml)
```

Объект, соответствующий диалоговому окну, создается с помощью статического метода `Load()` класса `XamlReader`. В этот метод передается объект `$reader`:

```
PS C:\Users\andrv> $window = [Windows.Markup.XamlReader]::Load($reader)
```

Теперь окно можно отобразить на экране с помощью метода `ShowDialog` (рис. 17.3):

```
PS C:\Users\andrv> $window.ShowDialog()
```

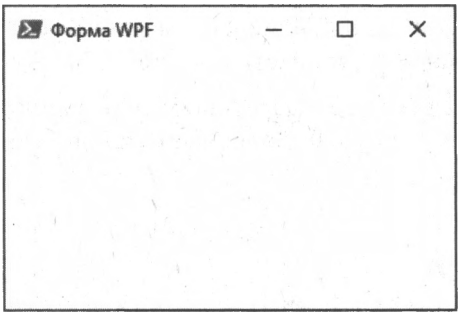


Рис. 17.3. Пустая форма WPF

При этом работа оболочки PowerShell приостановится, для ввода следующей команды нужно закрыть наше диалоговое окно.

Добавим в нашу форму элемент `<Grid>`, задающий сетку для размещения других элементов управления.

```
PS C:\Users\andrv> [xml]$xaml = @"
>> <Window
>>     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
>>     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
>>     x:Name="Window" Title="Форма WPF" Height="200" Width="300"
>> >
>>     <Grid x:Name="Grid">
>>         <Grid.RowDefinitions>
>>             <RowDefinition Height="Auto"/>
>>             <RowDefinition Height="Auto"/>
>>         </Grid.RowDefinitions>
>>         <Grid.ColumnDefinitions>
>>             <ColumnDefinition Width="Auto"/>
>>             <ColumnDefinition Width="Auto"/>
>>         </Grid.ColumnDefinitions>
>>     </Grid>
>> </Window>
>> "@
```

Сетка не отображается на экране, поэтому внешний вид главной формы (диалогового окна) не изменится, форма останется пустой.

Поместим теперь на форму надпись (тег `<Label>` с именем `Message`) и кнопку (тег `<Button>` с именем `ChangeMessage`). Надписи в этих элементах задаются с помощью атрибута `Content`, расположение элемента в ячейке сетки определяется с помощью атрибутов `Grid.Column` и `Grid.Row`.

```
PS C:\Users\andrv> [xml]$xaml = @"
>> <Window
>>     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
>>     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
>>     x:Name="Window" Title="Форма WPF" Height="200" Width="300"
>> >
>>     <Grid x:Name="Grid">
>>         <Grid.RowDefinitions>
>>             <RowDefinition Height="Auto"/>
>>             <RowDefinition Height="Auto"/>
>>         </Grid.RowDefinitions>
>>         <Grid.ColumnDefinitions>
>>             <ColumnDefinition Width="Auto"/>
>>             <ColumnDefinition Width="Auto"/>
>>         </Grid.ColumnDefinitions>
>>         <Label x:Name="Message"
>>             Content="Привет!"
>>             Grid.Column="0"
>>             Grid.Row="0"
>>         />
>>         <Button x:Name="ChangeMessage"
>>             Content="Нажми!"
>>             Grid.Column="1"
>>             Grid.Row="1"
>>         />
>>     </Grid>
>> </Window>
>> "@
```

Отобразим форму на экране (рис. 17.4):

```
C:\Users\andrv> $reader = (New-Object System.Xml.XmlNodeReader $xaml)
PS C:\Users\andrv> $window = [Windows.Markup.XamlReader]::Load($reader)
PS C:\Users\andrv> $window.ShowDialog()
```

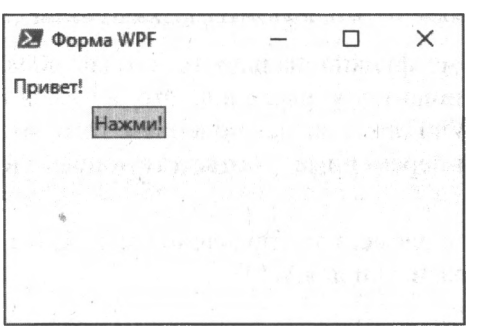


Рис. 17.4. WPF-форма с элементами управления

С помощью команды `Set-Content` поместим XAML-разметку для нашей формы в файл wpf_test.xaml:

```
PS C:\Users\andrv> @"
>> <Window
>>     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
>>     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
>>     x:Name="Window" Title="Форма WPF" Height="200" Width="300"
>> >
>>     <Grid x:Name="Grid">
>>         <Grid.RowDefinitions>
>>             <RowDefinition Height="Auto"/>
>>             <RowDefinition Height="Auto"/>
>>         </Grid.RowDefinitions>
>>         <Grid.ColumnDefinitions>
>>             <ColumnDefinition Width="Auto"/>
>>             <ColumnDefinition Width="Auto"/>
>>         </Grid.ColumnDefinitions>
>>         <Label x:Name="Message"
>>             Content="Привет!"
>>             Grid.Column="0"
>>             Grid.Row="0"
>>         />
>>         <Button x:Name="ChangeMessage"
>>             Content="Нажми!"
>>             Grid.Column="1"
>>             Grid.Row="1"
>>         />
>>     </Grid>
>> </Window>
>> "@ | Set-Content -Path ./wpf_test.xaml -Encoding UTF8
```

Теперь для отображения формы мы можем прочитать ее разметку из файла:

```
PS C:\Users\andrv> [xml]$xaml = Get-Content -Path .\wpf_test.xaml -Encoding
UTF8
PS C:\Users\andrv> $reader = (New-Object System.Xml.XmlNodeReader $xaml)
PS C:\Users\andrv> $window = [Windows.Markup.XamlReader]::Load($reader)
```

Осталось добавить форме функциональность, создав обработчики нужных событий, возникающих на элементах управления. Это делается практически так же, как и при использовании WinForms, за исключением того, что нам нужно предварительно создать объекты-переменные, соответствующие элементам управления загруженной формы.

Напомним, что для тегов элементов управления мы задавали атрибут `Name` из пространства имен XAML-разметки для WPF:

```
<Label x:Name="Message"
   . . .
/>
<Button x:Name="ChangeMessage"
   . . .
/>
```

По значению атрибута `Name` с помощью метода `$window.FindName()` получим ссылки на соответствующие элементы управления:

```
PS C:\Users\andrv> $messageLabel = $window.FindName("Message")
PS C:\Users\andrv> $changeButton = $window.FindName("ChangeMessage")
```

Пусть, как и в примере с WinForms, при нажатии кнопки текст надписи в нашей форме должен измениться на "Пока!". Для этого зададим обработчик события `Click` на кнопке с помощью метода `Add_Click`:

```
PS C:\Users\andrv> $changeButton.Add_Click({$messageLabel.Content='Пока!'})
```

Теперь форму можно отобразить на экране и проверить результат нажатия кнопки (рис. 17.5).

```
PS C:\Users\andrv> $window.ShowDialog()
```

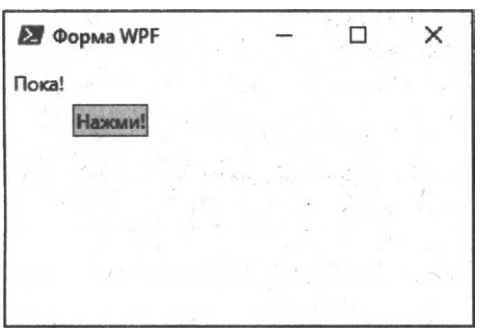


Рис. 17.5. Изменение содержимого элемента управления на WPF-форме

Итак, при работе с графическим интерфейсом с помощью WPF нужно выполнить следующие шаги:

1. Создать XAML-файл с описанием интерфейса.
2. Подключить сборку PresentationFramework.
3. Создать объект, соответствующий форме, загрузив ее описание в формате XAML.
4. Определить переменные (объекты), соответствующие элементам управления в форме.
5. Написать блоки кода для обработчиков событий и привязать их к объектам, созданным на предыдущем шаге.
6. Отобразить главную форму на экране.

Таким образом, при работе с WPF мы можем создавать и изменять дизайн интерфейса независимо от его функционала.

Кроме того, XAML-описание формы не зависит от специфики PowerShell, как это было в случае интерфейса на базе WinForms, поэтому вариантов выбора визуального конструктора для построения форм становится больше. Кроме платных специализированных сред разработки сценариев PowerShell (Sapien PowerShell Studio, PowerShell GUI Designer) можно воспользоваться бесплатной платформой Microsoft — Visual Studio Community Edition. Скачать Visual Studio можно с сайта Microsoft по адресу **https://visualstudio.microsoft.com/ru/vs/community/**.

Создание проекта

Поиск шаблонов (ALT+"B")

Последние шаблоны проектов

Здесь отобразится список недавно использованных шаблонов.

Все языки | Все платформы | Все типы проектов

Библиотека пользовательских элементов управления WPF
Проект для создания библиотеки пользовательских элементов управления для приложений WPF .NET Core.
Visual Basic Windows Рабочий стол Библиотека

Приложение WPF
Проект для создания приложения WPF .NET Core.
C# Windows Рабочий стол

Приложение WPF
Проект для создания приложения WPF .NET Core.
Visual Basic Windows Рабочий стол

Приложение WPF (.NET Framework)
Клиентское приложение Windows Presentation Foundation
C# XAML Windows Рабочий стол

Приложение WPF (.NET Framework)
Клиентское приложение Windows Presentation Foundation
Visual Basic XAML Windows Рабочий стол

Пустое приложение (универсальное приложение для Windows)

Назад Далее

Рис. 17.6. Создание WPF-приложения в Visual Studio

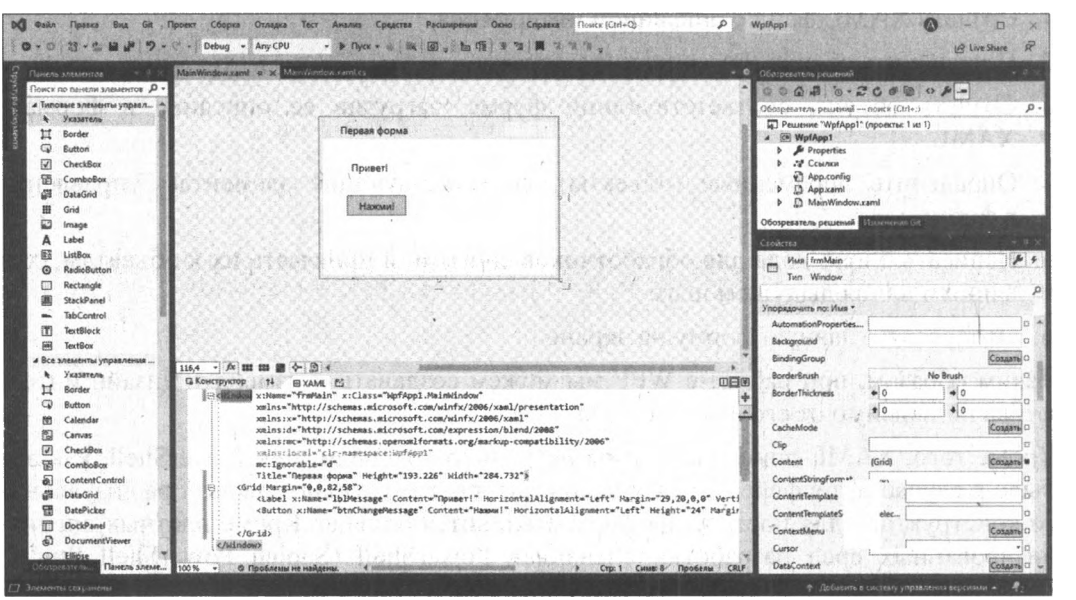


Рис. 17.7. Конструктор для WPF-форм в Visual Studio

Для построения интерфейса формы и получения ее XAML-разметки нужно при запуске Visual Studio создать новый проект "Приложение WPF (.NET Framework)" (рис. 17.6).

При создании такого приложения можно настроить дизайн формы с помощью графического конструктора (рис. 17.7).

В результате будет автоматически сформирован XAML-файл с описанием формы, с которым можно работать в PowerShell.

Итоги

- Добавить графический интерфейс к сценариям Windows PowerShell можно с помощью библиотек Windows Forms (WinForms) или Windows Presentation Foundation (WPF).
- При использовании WinForms элементы интерфейса создаются и настраиваются непосредственно в коде PowerShell. Средства для визуального редактирования подобных форм встроены в профессиональные среды разработки сценариев PowerShell.
- При работе с WPF дизайн интерфейса, описываемый с помощью разметки в формате XAML, не связан с кодом PowerShell. Создавать и редактировать WPF-формы можно с помощью визуального конструктора в бесплатной версии Microsoft Visual Studio.

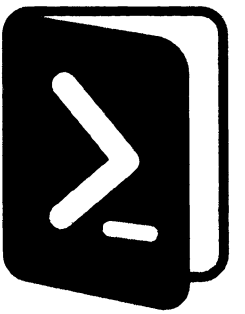
Что дальше? PowerShell для профессионалов

Мы рассмотрели лишь базовые аспекты PowerShell, которых достаточно для уверенной работы в оболочке на своем компьютере и написания несложных сценариев. Профессионалам, использующим PowerShell для решения задач системного администрирования и управления серверами, могут пригодиться следующие мощные возможности, оставшиеся за рамками данной книги.

- Подключение и работа с оболочкой PowerShell на удаленных компьютерах с помощью службы удаленного управления Windows (WinRM) или по протоколу SSH.
- Удаленное управление одним или несколькими компьютерами с помощью объектов .NET, CIM, WMI.
- Использование рабочих процессов (workflows) PowerShell, с помощью которых можно параллельно или последовательно запускать несколько заданий.
- Асинхронное выполнение команд с помощью фоновых заданий (background jobs).
- Обработка синхронных и асинхронных событий от различных источников в операционной системе.
- Настройка среды выполнения для приложений с помощью технологии Desired State Configuration (DSC), реализующей подход "инфраструктура как код".

Если вы часто пишете на PowerShell сценарии, которые становятся все сложнее и объемнее, то для их создания и поддержки лучше придерживаться испытанных подходов из традиционной разработки программного обеспечения.

- Обязательно следует применять систему контроля версий Git, с которой вы никогда не потеряете свой код и всегда сможете вернуться к любой сохраненной версии сценариев (способ интеграции Git с командной строкой PowerShell описан в *приложении 3*).
- Очень полезно будет разобраться с тем, как в PowerShell можно создавать и применять тесты, позволяющие удостовериться в том, что после внесения изменений код продолжает корректно работать. Подобно другим языкам программирования, для PowerShell существуют специальные фреймворки (frameworks), облегчающие работу с тестами, например Pester (**https://github.com/pester/Pester**).
- Связанные друг с другом функции и сценарии можно организовывать в виде модулей (напомним, что работу с готовыми модулями мы рассматривали в *главе 9*).



ПРИЛОЖЕНИЯ

Приложение 1. Что значат эти символы

Приложение 2. PowerShell в macOS и Linux

Приложение 3. Дополнительная настройка командной строки

ПРИЛОЖЕНИЕ 1

Что значат эти символы

В языке PowerShell используются различные специальные символы, которые в зависимости от контента могут иметь разный смысл и иногда применяются не так, как в других языках программирования.

- Знак ` (обратный апостроф или *гравис*) является в PowerShell экранирующим символом. Внутри строки в двойных кавычках обратный апостроф задает escape-последовательность (например, `n означает перевод строки). Использование его перед пробелом внутри параметров команды позволяет не указывать кавычки для путей к файлам или каталогам, содержащим пробелы (например, cd C:\Program` Files). Если обратный апостроф является последним символом в командной строке, то ввод команды продолжается со следующей строки (это позволяет выполнять в PowerShell многострочные команды).
- Знак ~ (тильда) используется в путях к объектам файловой системы, он соответствует пути к домашнему каталогу активного пользователя. Например, по команде cd ~ мы перейдем в этот каталог (в системе данный путь хранится в переменной среды USERPROFILE).
- Круглые скобки () в PowerShell используются двумя способами.
  - Скобки задают порядок вычисления выражений или выполнения команд. Например, в команде Get-Process -ComputerName (Get-Content .\comps.txt) сначала выполняется команда Get-Content .\comps.txt, а ее результат (содержимое файла comps.txt) передается с качестве значения параметра -ComputerName.
  - Скобки, указанные после имени метода, приводят к его выполнению. Например: " Привет ".Trim(). Скобки после имени метода нужно писать обязательно, даже если в метод не передается ни один параметр.
- Квадратные скобки [] используются двумя способами.
  - Квадратные скобки позволяют получить элемент массива по его индексу. Например, $services[3] означает четвертый элемент массива $services (первый элемент всегда имеет индекс 0).

- Внутри квадратных скобок можно указать тип, к которому требуется привести переменную или результат вычисления выражения. Например, значением выражения `[string]1.1 + [string]0.9` будет строка '1.10.9'.

- Фигурные скобки `{}` могут применяться в нескольких вариантах.
  - В фигурных скобках указывается блок исполняемого кода (аналог анонимных функций в других языках программирования). Например: `Get-Service | Where-Object {$_.Status -eq "Stopped"}`.
  - При создании хэш-таблиц внутри фигурных скобок, перед которыми стоит знак `@`, перечисляются пары "ключ=значение". Например: `$user = @{name="James"; lastName="Bond"; age=30}`.
  - С помощью фигурных скобок можно давать переменным имена, содержащие пробелы или другие недопустимые для имен символы. Например: `${My Var}`. Интересный эффект получается, если в фигурных скобках указана переменная из пространства имен файловой системы, т. е. путь к файлу с указанием имени диска. В этом случае значением переменной будет содержимое этого файла, например `$content = ${C:\Script\test.txt}`. Если подобную переменную указать в левой части оператора присваивания, то содержимое файла будет изменено на новое значение, например `${C:\Script\test.txt} = "Новое содержимое файла"`.

- Одинарными кавычками `' '` выделяют нерасширяемые строки. В таких строках не действуют escape-последовательности и не происходит подстановка значений переменных.

- Двойные кавычки `" "` используются для создания расширяемых строк. Если внутри такой строки встречается имя переменной или другое выражение, которое может быть вычислено, то в строку подставляется значение данной переменной или результат вычисления выражения. Также в расширяемых строках действуют escape-последовательности. Например, если в переменной `$name` хранится слово "Иван", то в переменной `$hello = "Привет, $name!`n"` будет записана строка 'Привет, Иван!' и символ перевода строки.

- Знак доллара `$` в PowerShell ставится перед именами переменных.

- Знак процента `%` используется в качестве псевдонима командлета `ForEach-Object`. В арифметических выражениях % означает оператор целочисленного деления.

- Вопросительный знак `?` используется в качестве псевдонима командлета `Where-Object`.

- Знак больше `>` фактически является псевдонимом командлета `Out-File`. С помощью этого символа можно перенаправить вывод какого-либо командлета во внешний файл, и выглядит это как перенаправление вывода в оболочках cmd.exe или bash.

- Знак `+` используется для обозначения операции сложения чисел и конкатенации строк.

- Дефис - ставится перед именами параметров команд (например, `-id`) и перед многими операторами (например, `-eq`). Также дефис используется для разделения глагола и существительного в именах командлетов (например, `Get-Process`).
- Для знака коммерческого at (знака "собачки") `@` возможны четыре варианта использования.
  - Ставится перед фигурными скобками при создании хэш-таблицы. Например: `$server = @{comp = "Server1"; ip = "10.170.1.1"}`.
  - Записывается перед круглыми скобками, внутри которых через запятую перечисляется список значений, формирующих массив. Например: `$numbers = @(1, 2, 45, 3)`.
  - Определяет строку типа here-string вида `@"`*блок текста*`"@` или `@'`*блок текста*`'@`. Подобные строки обычно используются для вставки в сценарий больших блоков текста или при генерации текстовой информации для других программ.
  - Обозначает splat-оператор, с помощью которого можно передавать в команду параметры, упакованные в массив, элементы которого будут соответствовать позиционным параметрам, или хэш-таблицу, ключи в которой соответствуют именам параметров. Например: `$params = @{Path = "test.txt"; Destination = "test2.txt"; WhatIf = $true}; Copy-Item @params`.
- Знак амперсанда & соответствует оператору вызова и сообщает оболочке о том, что следующая за этим знаком строка является командной, которую нужно выполнить. Например: `$command = "dir"; & $command`.
- Точка с запятой ; является разделителем, который позволяет записать и выполнить несколько команд в одной строке. Например: `dir ; Get-Process`. При такой записи команды являются независимыми, результаты выполнения одной команды следующей команде не передаются.
- Знак решетки # используется в комментариях. Оболочка игнорирует все символы, которые идут после этого знака до конца строки. Блочные комментарии, состоящие из нескольких строк, помещаются внутри символов `<# ... #>`.
- Знак равенства = обозначает оператор присваивания значения переменной: `$a = 1`. В операторе сравнения знак равенства не используется, вместо него нужно писать оператор `-eq`. Также знак равенства может применяться совместно с математическими операторами: `$a += 10` добавит число 10 к текущему значению переменной `$a`.
- Вертикальная черта | используется для организации конвейера, когда выходные данные одного командлета (т. е. объекты, которые этот командлет помещает в выходной поток) передаются на вход другому командлету.
- Символ косой черты (слэш) / в математических выражениях обозначает операцию деления. Слэш и обратный слеш \ также используются в качестве разделителя в путях к объектам файловой системы. Например: C:\Windows или C:/Windows.

- Точка . может использоваться в нескольких вариантах.
  - Точка указывается после имени объекта для получения доступа к его свойству или методу. Например, с помощью `$_.Name` мы обращаемся к свойству `Name` объекта, которому соответствует переменная `$_`.
  - Точка и пробел перед именем скрипта или перед путем к нему приводят к выполнению этого сценария в режиме dot-sourcing. Скрипт запускается в области видимости окна командной строки, все функции и переменные из него приобретают глобальную область видимости и остаются доступными в оболочке после завершения работы сценария.
  - Две точки соответствуют оператору диапазона. Например, выражение `1 .. 10` генерирует числа от 1 до 10.
  - Две точки в файловых путях соответствуют родительскому каталогу. Например, если текущим каталогом является C:\Script\PowerShell, то путь `..\..` будет соответствовать корневому каталогу на диске C:\.
- Запятая , является разделителем элементов в массиве или списке. Например: `$arr = "Иванов", "Петров", "Сидоров", 23`. Также запятая может разделять несколько значений параметров командлета или функции, например: `Get-Services -computername Server1,Server2,Server3`.
- Двойное двоеточие `::` используется для доступа к статическим членам класса .NET Framework. Например: `[math]::Sqrt(25)`.
- Восклицательный знак `!` является кратким псевдонимом оператора логического отрицания `-not`.

ПРИЛОЖЕНИЕ 2

PowerShell в macOS и Linux

Начиная с 2017 г. PowerShell является проектом Open Source, исходный код размещен в репозитории на GitHub (**https://github.com/PowerShell/PowerShell**). Оболочка PowerShell версий 6 и 7, которая может работать не только в Windows, но и в macOS и в различных дистрибутивах Linux (Debian, Ubuntu, CentOS, Red Hat, openSUSE, Fedora), базируется на платформе .NET Core/.NET 5 — кросс-платформенной реализации Windows .NET Framework.

> ***Замечание***
>
> Вначале кросс-платформенная версия оболочки называлась PowerShell Core 6, чтобы явно указать ее зависимость от .NET Core. Начиная с седьмой версии, слово Core из названия убрано, теперь оболочка называется просто PowerShell. Версия оболочки, базирующаяся на полной платформе Windows .NET Framework, называется Windows PowerShell.

Установка и запуск оболочки

Процедура установки PowerShell зависит от типа операционной системы и ее версии. Здесь мы не будем подробно на этом останавливаться, соответствующие инструкции для всех поддерживаемых систем имеются на сайте Microsoft по адресу **https://docs.microsoft.com/ru-ru/powershell/scripting/install/installing-powershell**.

В дальнейшем в примерах будет рассматриваться PowerShell в операционной системе Linux Ubuntu 20.04, которая была установлена в подсистеме WSL на компьютере автора. В этой системе PowerShell ставится следующими командами:

```
sudo apt-get update
sudo apt-get install -y wget apt-transport-https software-properties-common
wget -q https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-
prod.deb
sudo dpkg -i packages-microsoft-prod.deb
sudo apt-get update
sudo add-apt-repository universe
sudo apt-get install -y powershell
```

Для запуска установленной оболочки PowerShell нужно выполнить в терминале команду `pwsh`. После вывода информации о версии PowerShell мы получим обычное приглашение командной строки с путем к текущему каталогу в качестве префикса.

```
andrey@DESKTOP-BU86I5T:~$ pwsh
PowerShell 7.1.4
Copyright (c) Microsoft Corporation.

https://aka.ms/powershell
Type 'help' to get help.

PS /home/andrey>
```

Информация об используемой платформе .NET и версии PowerShell хранится в переменной `$PSVersionTable`. Посмотрим на значение этой переменной в оболочке, запущенной под Linux:

```
PS /home/andrey> $PSVersionTable

Name                           Value
----                           -----
PSVersion                      7.1.4
PSEdition                      Core
GitCommitId                    7.1.4
OS                             Linux 4.19.128-microsoft-standard...
Platform                       Unix
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
WSManStackVersion              3.0
```

В поле `PSVersion` указана версия оболочки, в поле `PSEdition` — ее тип, в поле `OS` — версия используемой операционной системы. В нашем случае мы используем PowerShell версии 7.1.4 Core в Linux.

В Windows PowerShell состав и значения полей переменной `$PSVersionTable` будут другими, здесь мы запускаем десктопную версию оболочки 5.1, использующую .NET Framework 4 (поле `CLRVersion`):

```
PS C:\Users\andrv> $PSVersionTable

Name                           Value
----                           -----
PSVersion                      5.1.19041.1151
PSEdition                      Desktop
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.19041.1151
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
```

Отличия от Windows PowerShell

В .NET Core/.NET 5 реализована не вся функциональность платформы .NET Framework, поэтому в PowerShell 6/7 по умолчанию загружается меньше модулей и, следовательно, будет доступно меньше командлетов, чем в Windows PowerShell. Список доступных модулей можно посмотреть с помощью команды `Get-Module`:

```
PS /home/andrey> Get-Module -ListAvailable | Select Name

Name
----
Microsoft.PowerShell.Archive
Microsoft.PowerShell.Host
Microsoft.PowerShell.Management
Microsoft.PowerShell.Security
Microsoft.PowerShell.Utility
PackageManagement
PowerShellGet
PSDesiredStateConfiguration
PSReadLine
ThreadJob
```

Как видим, в PowerShell для Linux загружаются десять модулей, в то время как в Windows PowerShell по умолчанию доступны несколько десятков модулей. Часть из отсутствующих модулей напрямую связана с особенностями операционной системы Windows и не может быть перенесена на другие платформы, некоторые модули могут быть добавлены в следующих версиях PowerShell.

Таким образом, в PowerShell 6/7 отсутствуют определенные командлеты, имевшиеся в Windows PowerShell, в том числе `*-Service`, `*-Acl`, `*-Event`, `Set-ExecutionPolicy`. Некоторые из отсутствующих командлетов должны появиться в следующих версиях PowerShell.

Вспомним, какие провайдеры доступны в Windows PowerShell и какие виртуальные диски они поддерживают. Для этого выполним командлет `Get-PSDrive`:

```
PS C:\Script> Get-PSDrive | Select Name, Provider, Root

Name     Provider                                Root
----     --------                                ----
Alias    Microsoft.PowerShell.Core\Alias
C        Microsoft.PowerShell.Core\FileSystem    C:\
Cert     Microsoft.PowerShell.Security\Certificate \
Env      Microsoft.PowerShell.Core\Environment
Function Microsoft.PowerShell.Core\Function
HKCU     Microsoft.PowerShell.Core\Registry      HKEY_CURRENT_USER
HKLM     Microsoft.PowerShell.Core\Registry      HKEY_LOCAL_MACHINE
Variable Microsoft.PowerShell.Core\Variable
WSMan    Microsoft.WSMan.Management\WSMan
```

В PowerShell для Linux провайдеры и виртуальные диски для системного реестра и хранилища сертификатов отсутствуют:

```
PS /mnt/c/Users/andrv> Get-PSDrive | Select Name, Provider, Root

Name     Provider                                Root
----     --------                                ----
/        Microsoft.PowerShell.Core\FileSystem    /
Alias    Microsoft.PowerShell.Core\Alias
Env      Microsoft.PowerShell.Core\Environment
Function Microsoft.PowerShell.Core\Function
Temp     Microsoft.PowerShell.Core\FileSystem    /tmp/
Variable Microsoft.PowerShell.Core\Variable
```

При работе с PowerShell в Linux и macOS необходимо учитывать особенности самих операционных систем.

- *Зависимость от регистра символов.* Сама оболочка PowerShell, как и операционная система Windows, не зависит от регистра символов. Например, имена `$a` и `$A` обозначают одну и ту же переменную, название командлетов можно писать в любом регистре (`Get-Process`, `get-process` или `GET-PROCESS`), в путях к файлам и каталогам в Windows тоже можно использовать символы в разных регистрах (C:\Windows, c:\windows или C:\WINDOWS). Напротив, в операционных системах macOS и Linux имеется зависимость от регистра символов в путях к каталогам и файлам или в именах переменных среды. Поэтому для запуска сценариев, загрузки модулей и корректной работы автодополнения файловых путей в этих системах нужно писать точные имена с учетом регистра символов. Названия самих командлетов и переменных PowerShell по-прежнему от регистра не зависят.
- *Разделители в путях файловой системы.* В Windows в качестве разделителей в путях к файлам и каталогам можно использовать символы слеша / и обратного слеша \. В macOS и Linux используется только символ слеша /.
- *Псевдонимы команд.* В Windows PowerShell за некоторыми командлетами закреплены псевдонимы, соответствующие стандартным командам UNIX-подобных систем: `ls`, `rm`, `cp`, `mv`, `cat`, `ps`, `man`, `mount`. В PowerShell 6/7 данные псевдонимы не определены, их ввод приводит к выполнению "родных" команд операционной системы.
- *Запуск команд от имени суперпользователя (sudo).* Для выполнения какой-либо одной команды с привилегиями суперпользователя root в оболочке macOS или Linux используется команда `sudo`. В сеансе работы PowerShell напрямую применить `sudo` не получится, при необходимости нужно с помощью `sudo` запустить новый экземпляр оболочки, и все команды PowerShell будут выполняться от имени суперпользователя.

Отметим также, что интегрированная среда разработки PowerShell ISE доступна только в Windows PowerShell, в состав PowerShell 6/7 она не входит. Для разработки скриптов в macOS или Linux компания Microsoft рекомендует использовать редактор Visual Studio Code с расширением для поддержки языка PowerShell.

ПРИЛОЖЕНИЕ 3

Дополнительная настройка командной строки

Работа в командной строке должна приносить удовольствие. Будет очень приятно, если в приглашении терминала отображается полезная для вас информация, оболочка сама подсказывает нужные команды, а постоянно повторяемые действия можно выполнять несколькими нажатиями клавиш. Такие возможности предоставляют дополнительные модули для PowerShell, некоторые из них мы рассмотрим далее.

Модуль PSReadLine

В *главе 4* мы говорили о стандартных способах редактирования командной строки в консоли PowerShell. Модуль PSReadLine добавляет к ним новые возможности, делающие работу с оболочкой PowerShell более удобной. Перечислим некоторые из них.

- *Цветовое выделение синтаксиса при наборе команд.* Названия командлетов, параметров и переменных, ключевые конструкции и другие элементы языка PowerShell при вводе в командную строку отображаются разным цветом. Это помогает лучше понимать структуру команд и снижает вероятность ошибки при наборе.
- *Отмена (Undo) и повторное применение (Redo) изменений в командной строке.* По умолчанию для отмены изменений нужно нажать комбинацию клавиш <Ctrl>+<z>, а для повтора — комбинацию <Ctrl>+<y>.
- *Автоматическая проверка команды при вводе.* Если при наборе команды получается синтаксически неверная конструкция (например, мы забыли поставить закрывающую скобку или не написали команду после знака конвейера), то последний знак > в приглашении командной строки становится красным.
- *Многострочное редактирование команд.* Многословные команды можно располагать на нескольких строках даже без ввода знака ` разрыва строки. При этом можно свободно перемещаться по строкам команды, изменять их, добавлять новые строки выше и ниже текущей строки. Новая строка после текущей вставляется при нажатии комбинации <Shift>+<Enter>, а перед текущей строкой — при нажатии <Ctrl>+<Enter>.

- *Поиск в истории вводившихся команд.* Если вы помните только часть команды (не обязательно сначала), то по ней можно найти подходящие варианты среди тех команд, которые выполнялись в оболочке ранее. По умолчанию интерактивный поиск в таком режиме активируется нажатием клавиш <Ctrl>+<r> (поиск по истории команд назад) или <Ctrl>+<s> (поиск по истории команд вперед).
- *Вывод вариантов автодополнения в виде меню.* Для завершения ввода команды или параметра часто бывает удобно посмотреть на полный список всех возможных вариантов и выбрать в нем нужный пункт. Такое меню с вариантами автодополнения открывается при нажатии комбинации клавиш <Ctrl>+<Пробел>. Например, если после имени команды через пробел набрать знак дефиса и нажать <Ctrl>+<Пробел>, то будет выведен список всех параметров, поддерживаемых этой командой. По предложенным вариантам можно переходить с помощью клавиш со стрелками, выбор варианта — клавиша <Enter>. Для текущего выбранного свойства на экране отображается его тип.
- *Поддержка технологии автодополнения Intellisense.* При наборе команды оболочка может сразу выдавать подходящий завершенный вариант из списка вводившихся ранее команд, который можно принять одним нажатием клавиши <→>. Активируется этот режим следующей командой: `Set-PSReadLineOption -PredictionSource 'History'`.

В Windows PowerShell 5.1 для Windows 10 и в кросс-платформенных версиях PowerShell модуль PSReadLine должен быть установлен по умолчанию. Увидеть список подключенных модулей и проверить наличие в нем PSReadLine можно с помощью командлета `Get-Module`:

```
PS C:\Users\andrv> Get-Module | Select Name, Version

Name                            Version
----                            -------
Microsoft.PowerShell.Management 3.1.0.0
Microsoft.PowerShell.Security   3.0.0.0
Microsoft.PowerShell.Utility    3.1.0.0
Microsoft.WSMan.Management      3.0.0.0
PSReadline                      2.1.0
```

Подробную информацию о модуле `PSReadLine` можно посмотреть следующим образом:

```
PS C:\Users\andrv> Get-Module PSReadline | Format-List

Name            : PSReadline
Path            : C:\Program Files\WindowsPowerShell\Modules\
                  PSReadline\2.1.0\PSReadLine.psm1
Description     : Great command line editing in the PowerShell
                  console host
ModuleType      : Script
Version         : 2.1.0
NestedModules   : {Microsoft.PowerShell.PSReadLine2}
```

```
ExportedFunctions : PSConsoleHostReadLine
ExportedCmdlets   : {Get-PSReadLineKeyHandler, Get-PSReadLineOp
                    tion, Remove-PSReadLineKeyHandler, Set-PSRe
                    adLineKeyHandler...}
ExportedVariables :
ExportedAliases   :
```

Если модуль PSReadLine не обнаружен, то его нужно установить с помощью команды:

```
Install-Module PSReadLine -Force -Scope CurrentUser -SkipPublisherCheck.
```

Текущие сочетания клавиш, назначенные в PSReadLine для выполнения различных действий, можно посмотреть с помощью команды `Get-PSReadlineKeyHandler`:

```
PS C:\Users\andrv> Get-PSReadlineKeyHandler

Basic editing functions
=======================
Key              Function           Description
---              --------           -----------
Enter            AcceptLine         Accept the input or move to the
                                    next line if input is missing
                                    a closing token.
Shift+Enter      AddLine            Move the cursor to the next line
                                    without attempting to execute
                                    the input
Backspace        BackwardDeleteChar Delete the character before the
                                    cursor
Ctrl+h           BackwardDeleteChar Delete the character before the
                                    cursor
Ctrl+Home        BackwardDeleteLine Delete text from the cursor to
                                    the start of the line
Ctrl+Backspace   BackwardKillWord   Move the text from the start of
                                    the current or previous word to
                                    the cursor to the kill ring
Ctrl+w           BackwardKillWord   Move the text from the start of
                                    the current or previous word to
                                    the cursor to the kill ring
Ctrl+C           Copy               Copy selected region to the sys
                                    tem clipboard.  If no region is
                                    selected, copy the whole line
. . .
```

Каждая операция, поддерживаемая `PSReadLine`, здесь отнесена к одной из категорий:

- Basic editing functions (Основные функции редактирования);
- Cursor movement functions (Функции для перемещения курсора);
- History functions (Функции для работы с историей команд);

- Completion functions (Функции автодополнения при вводе);
- Miscellaneous functions (Различные функции).

Изменить сочетания клавиш позволяет команда `Set-PSReadlineKeyHandler`. Например, назначим клавишам <↑> и <↓> функции `HistorySearchBackward` (поиск назад по истории команд) и `HistorySearchForward` (поиск вперед по истории команд) соответственно:

```
PS C:\Users\andrv> Set-PSReadLineKeyHandler -Key UpArrow -Function
HistorySearchBackward
PS C:\Users\andrv> Set-PSReadLineKeyHandler -Key DownArrow -Function
HistorySearchForward
```

Теперь мы сможем подставлять в командную строку команды из истории, набрав в командной строке их начальные символы и нажимая клавишу <↑> или клавишу <↓>.

Чтобы не набирать при каждом запуске PowerShell команды для настройки `PSReadLine`, их нужно поместить в свой PowerShell-профиль, путь к которому хранится в переменной `$PROFILE`. В этом случае команды из профиля будут запускаться при каждом входе в оболочку. Открыть профиль для редактирования в Блокноте можно так:

```
C:\Users\andrv> notepad %PROFILE
```

Отметим также, что по умолчанию `PSReadLine` сохраняет все команды, которые выполнялись в оболочке PowerShell, в текстовом файле на диске. Посмотреть на содержимое этого файла в Блокноте можно следующей командой:

```
C:\Users\andrv> notepad (Get-PSReadLineOption).HistorySavePath
```

Интеграция с Git. Модуль posh-git

Система управления версиями Git является фактически современным стандартом для хранения и обмена кодом, ею пользуются практически все разработчики.

Для интеграции Git и PowerShell служит модуль posh-git (**https://github.com/dahlbyk/posh-git**). После установки этого модуля в приглашении командной строки будет отображаться информация о состоянии текущего Git-репозитория, а команды и параметры Git можно будет автоматически дополнять при вводе по нажатии клавиши <Tab>.

Перед тем как устанавливать posh-git, убедитесь, что система Git установлена и доступна в оболочке PowerShell. Для этого в консоли PowerShell нужно выполнить команду `git --version`, которая должна вывести информацию о версии Git:

```
C:\Users\andrv> git --version
git version 2.28.0.windows.1
```

Устанавливается `posh-git` стандартным образом с помощью командлета `Install-Module`:

```
C:\Users\andrv> Install-Module posh-git -Scope CurrentUser -Force
```

После установки нужно импортировать все команды из posh-git:

```
C:\Users\andrv> Import-Module posh-git
```

Команду для импорта нужно поместить в свой профиль (сценарий, путь которому хранится в переменной `$PROFILE`) для автоматического выполнения при каждом запуске оболочки. Для этого можно открыть сценарий профиля в Блокноте (команда `notepad $PROFILE`) и добавить туда нужную команду или просто выполнить команду `Add-PoshGitToProfile` из модуля posh-git.

Чтобы увидеть изменения в командной строке, нужно войти в каталог, являющийся Git-репозиторием. Создадим новый каталог git-test и создадим в нем новый репозиторий Git:

```
C:\Users\andrv> mkdir git-test

    Каталог: C:\Users\andrv

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        03.10.2021     10:04                git-test

C:\Users\andrv> cd git-test
C:\Users\andrv\git-test> git init
C:\Users\andrv\git-test [master]>
```

В приглашении командной строки после пути к текущему каталогу в квадратных скобках теперь выводится информация о состоянии Git-репозитория.

Пока мы видим только имя `master` текущей ветки, т. к. файлов в рабочем каталоге нет:

```
C:\Users\andrv\git-test [master]> git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Создадим три новых файла:

```
C:\Users\andrv\git-test [master]> New-Item file1.txt, file2.txt,
                                                  file3.txt -ItemType File

    Каталог: C:\Users\andrv\git-test

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----        03.10.2021     10:28              0 file1.txt
-a----        03.10.2021     10:28              0 file2.txt
-a----        03.10.2021     10:28              0 file3.txt

C:\Users\andrv\git-test [master +3 ~0 -0 !]>
```

В подсказке о состоянии репозитория появились новые данные: восклицательный знак указывает на то, что в рабочем каталоге есть файлы, которые не добавлены в индекс Git, а число после знака + показывает количество таких файлов.

Добавим файл file1.txt в индекс Git:

```
C:\Users\andrv\git-test [master +3 ~0 -0 !]> git add file1.txt
C:\Users\andrv\git-test [master +1 ~0 -0 | +2 ~0 -0 !]>
```

Теперь в подсказке указана информация о файлах в индексе Git (слева от разделителя |) и о файлах в рабочем каталоге (справа от разделителя |). В нашем случае в индексе находится один новый файл (число после знака +), а в рабочем каталоге остались два новых файла.

Изменим файл file1.txt, добавив в него строку:

```
C:\Users\andrv\git-test [master +1 ~0 -0 | +2 ~0 -0 !]> "Hello!" >> file1.txt
C:\Users\andrv\git-test [master +1 ~0 -0 | +2 ~1 -0 !]>
```

Как видим, в секции рабочего каталога после знака ~ теперь указана единица — это количество измененных файлов в рабочем каталоге.

Добавим в индекс Git файл file2.txt:

```
C:\Users\andrv\git-test [master +1 ~0 -0 | +2 ~1 -0 !]> git add file2.txt
C:\Users\andrv\git-test [master +2 ~0 -0 | +1 ~1 -0 !]>
```

Теперь в индексе два новых файла (+2), в рабочем каталоге один новый файл (+1) и один измененный файл (~1).

Удалим файл file2.txt в рабочем каталоге:

```
C:\Users\andrv\git-test [master +2 ~0 -0 | +1 ~1 -0 !]> Remove-Item file2.txt
C:\Users\andrv\git-test [master +2 ~0 -0 | +1 ~1 -1 !]>
```

Информация о файлах в индексе не изменилась (+2 ~0 -0), а в рабочем каталоге теперь один новый файл (+1), один измененный файл (~1) и один удаленный файл (-1).

Создадим *коммит* (зафиксируем изменения в репозитории), в который войдут файлы file1.txt и file2.txt из индекса:

```
C:\Users\andrv\git-test [master +2 ~0 -0 | +1 ~1 -1 !]> git commit -m "First
commit"
[master (root-commit) b22296c] First commit
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1.txt
 create mode 100644 file2.txt
C:\Users\andrv\git-test [master +1 ~1 -1 !]>
```

После коммита индекс Git очистился и в подсказке о состоянии репозитория осталась только информация о файлах в рабочем каталоге.

Добавим в индекс все файлы из рабочего каталога:

```
C:\Users\andrv\git-test [master +1 ~1 -1 !]> git add .
C:\Users\andrv\git-test [master +1 ~1 -1 ~]>
```

Индикатор состояния репозитория изменился со знака ! на знак ~, означающий, что в репозитории есть *незакоммиченные изменения*:

```
C:\Users\andrv\git-test [master +1 ~1 -1 ~]> git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.txt
        deleted:    file2.txt
        new file:   file3.txt

C:\Users\andrv\git-test [master +1 ~1 -1 ~]>
```

Таким образом, подсказка в приглашении командной строки помогает быстро понять, в каком состоянии находится Git-репозиторий. Для более наглядного представления этой информации можно изменить тему оформления консоли с помощью модуля Oh My Posh.

Оформление приглашения командной строки

Декорировать консоль PowerShell можно с помощью различных настраиваемых тем оформления, предлагаемых модулем Oh My Posh. Начиная с третьей версии, модуль Oh My Posh является кросс-платформенным, его можно применять как в операционной системе Windows, так и в macOS и Linux.

В темах Oh My Posh используются различные пиктограммы, которые отображаются с помощью шрифтов Powerline с глифами. Поэтому данные шрифты нужно предварительно установить в операционную систему и подключить в терминале.

Установка шрифтов Powerline

В темах Oh My Posh рекомендуется использовать шрифты NerdFonts, которые можно скачать, например, с сайта GitHub:

```
C:\Users\andrv> Invoke-WebRequest -Uri 'https://github.com/ryanoasis/nerd-
fonts/releases/download/v2.1.0/CascadiaCode.zip' -OutFile .\nf_fonts.zip
```

С помощью команды `Expand-Archive` распакуем загруженный архив в каталог NerdFonts:

```
C:\Users\andrv> Expand-Archive .\nf_fonts.zip -DestinationPath .\NerdFonts
```

Для установки шрифтов в систему откроем в Проводнике Windows папку NerdFonts с загруженными шрифтами, выделим в ней все ttf-файлы, щелкнем правой кнопкой мыши и в контекстном меню выберем пункт **Установить для всех пользователей**.

После этого загруженный архив и каталог с его содержимым можно удалить:

```
C:\Users\andrv> Remove-Item .\nf_fonts.zip
C:\Users\andrv> Remove-Item .\NerdFonts -Recurse
```

После установки загруженных шрифтов нужно назначить шрифт CaskaydiaCove NF в качестве шрифта для профиля Windows PowerShell в терминале Windows. Это можно сделать на вкладке **Appearance** в настройках профиля Windows PowerShell (рис. П3.1).

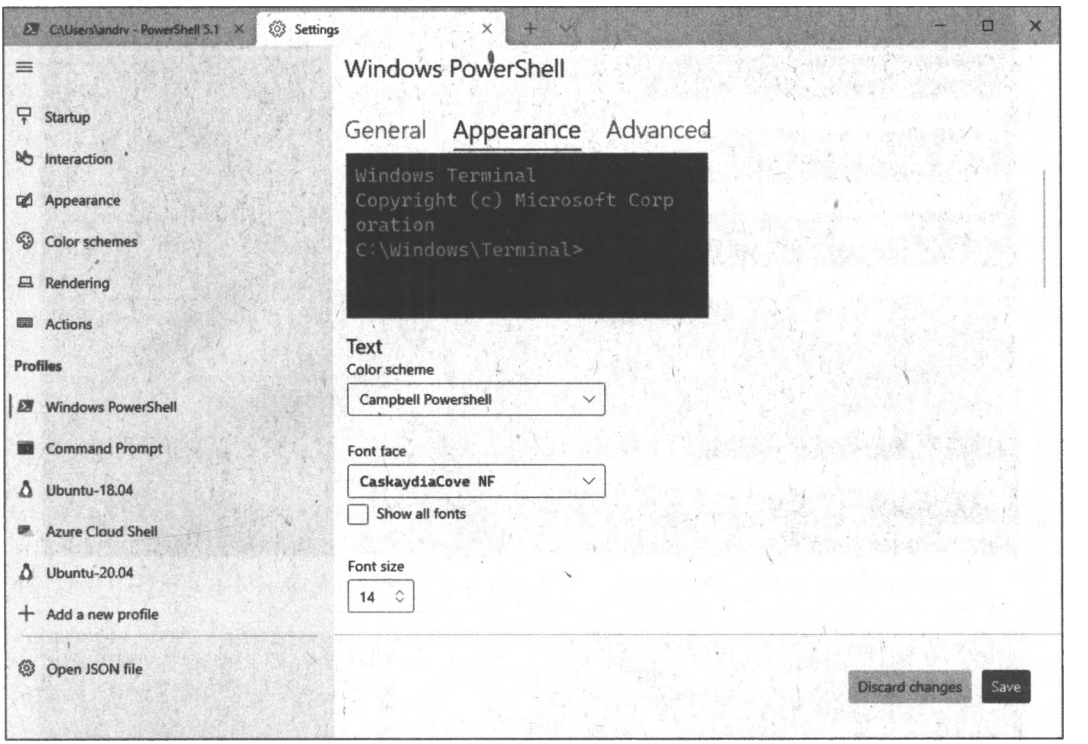


Рис. П3.1. Настройка шрифта для профиля PowerShell в терминале Windows

Модуль Oh My Posh

Модуль Oh My Posh устанавливается стандартно:

```
C:\Users\andrv> Install-Module oh-my-posh -Scope CurrentUser -Force
```

Модуль довольно объемный, установка его может занять продолжительное время.

После установки с помощью команды `Get-PoshThemes` можно посмотреть все стандартные встроенные темы оформления, которые поддерживает Oh My Posh (рис. П3.2).

Установить понравившуюся тему можно с помощью команды `Set-PoshPrompt`, указав название темы в качестве значения параметра `-Theme`. Например, выберем тему agnoster и перейдем в каталог git-test, являющийся репозиторием Git (рис. П3.3).

Теперь в приглашении командной строки отображаются имена пользователя и компьютера, путь к текущей директории от домашнего каталога и имя текущей ветки в Git-репозитории.

```
C:\Users\andrv> Get-PoshThemes
Theme: agnoster
andrv@DESKTOP-BU86I5T
Theme: agnosterplus
                                                                  14:17:23
andrv@DESKTOP-BU86I5T
Theme: aliens
andrv@DESKTOP-BU86I5T ~
Theme: avit
~
Theme: blueish
andrv@DESKTOP-BU86I5T ~
Theme: blue-owl
```

Рис. П3.2. Темы оформления модуля Oh My Posh

```
                                                powershell | 14:18:43
andrv@DESKTOP-BU86I5T ~
$
Theme: tonybaloney
~
Theme: wopian
~                                                               14:18:43
Theme: ys
# andrv @ DESKTOP-BU86I5T in ~ [14:18:43]
$
Theme: zash
@andrv ~
Themes location: C:\Users\andrv\OneDrive\Documents\WindowsPowerShell\Modules\oh-my-
posh\5.7.0\themes
To change your theme, use the Set-PoshPrompt command. Example:
  Set-PoshPrompt -Theme jandedobbeleer
C:\Users\andrv> Set-PoshPrompt -Theme agnoster
andrv@DESKTOP-BU86I5T  cd git-test
andrv@DESKTOP-BU86I5T  git-test  master
```

Рис. П3.3. Установка темы оформления agnoster

Для автоматического применения выбранной схемы при загрузке PowerShell нужно команду `Set-PoshPrompt -Theme agnoster` добавить в профиль PowerShell (напомним, что путь к профилю хранится в переменной `$PROFILE`).

По умолчанию в теме agnoster отключен вывод подробной информации о состоянии Git-репозитория, которую предоставляет модуль posh-git. Чтобы включить эту информацию, нужно в конфигурационный JSON-файл с настройками темы внести изменения, описанные в документации модуля Oh My Posh (**https://ohmyposh.dev/docs/git**).

Путь к каталогу с темами можно посмотреть в конце вывода команды `Get-PoshThemes` (рис. П3.4).

Выполнив команду `dir` для этого каталога, мы по названию схемы узнаем имя соответствующего конфигурационного файла (для нашей схемы agnoster это файл

```
C:\Users\andrv - PowerShell 5.1
andrv in ~ >
Theme: star

~
→
Theme: stelbent.minimal

                                                    powershell | 20:41:59
andrv@DESKTOP-BU86I5T  ~
$
Theme: tonybaloney

  ~
Theme: wopian

~                                                              20:41:59
→
Theme: ys

# andrv @ DESKTOP-BU86I5T in ~ [20:41:59]
$
Theme: zash

@andrv → ~
------------------------------------------------------------------------------

Themes location: C:\Users\andrv\OneDrive\Documents\WindowsPowerShell\Modules\oh-m
y-posh\5.7.0\themes

To change your theme, use the Set-PoshPrompt command. Example:
  Set-PoshPrompt -Theme jandedobbeleer

C:\Users\andrv>
```

Рис. П3.4. Путь к каталогу с темами Oh My Posh

agnoster.omp.json). Данный файл нужно открыть в текстовом редакторе и добавить в раздел, связанный с Git, параметр `"properties": { "display_status": true }`:

```
{
  "type": "git",
  "style": "powerline",
  "powerline_symbol": "\uE0B0",
  "foreground": "#193549",
  "background": "#95ffa4",
  "properties": {
    "display_status": true
  }
}
```

После сохранения конфигурационного файла и перезапуска оболочки мы в приглашении командной строки увидим информацию о статусе файлов в рабочем каталоге и индексе Git (рис. П3.5).

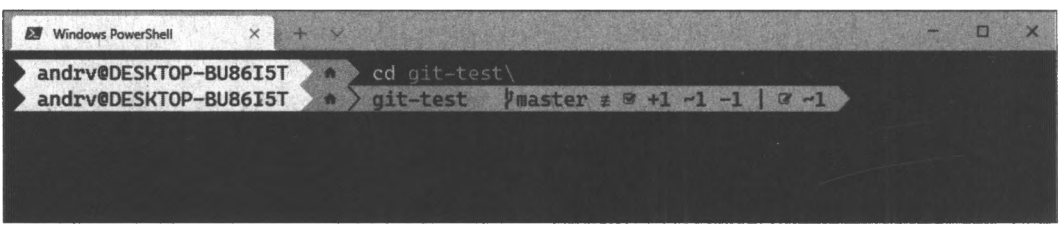


Рис. П3.5. Информация о состоянии файлов в Git-репозитории для темы agnoster

Оформление списков файлов и каталогов. Модуль Terminal-Icons

Работая в командной строке, мы часто с помощью команды `Get-ChildItem` (псевдонимы `dir` и `ls`) просматриваем списки файлов и каталогов. Модуль Terminal-Icons делает эти списки более наглядными, выделяя различные типы файлов разным цветом и добавляя к названиям файлов соответствующие иконки.

Установим и импортируем модуль Terminal-Icons:

```
C:\Users\andrv> Install-Module Terminal-Icons -Scope CurrentUser -Force
C:\Users\andrv> Import-Module Terminal-Icons
```

Теперь список файлов и каталогов выглядит намного интереснее (рис. П3.6).

Если вам понравился такой формат отображения файлов и каталогов, то команду импортирования модуля Terminal-Icons нужно добавить в сценарий-профиль `$PROFILE`.

```
Windows PowerShell
andrv@DESKTOP-BU86I5T  Get-ChildItem

    Каталог: C:\Users\andrv

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        15.02.2021     20:30                .config
d-----        19.09.2021      7:26                .dbus-keyrings
d-----        11.02.2021     16:12                .local
d-----        06.06.2021     13:26                .quokka
d-----        11.10.2020      7:48                .ssh
d-----        28.09.2020     23:28                .vscode
d-----        03.03.2021     20:04                .wallaby
d-r---        28.09.2020     16:18                3D Objects
d-r---        28.09.2020     16:18                Contacts
d-----        01.04.2021     19:41                Documents
d-r---        03.10.2021     13:20                Downloads
d-r---        28.09.2020     16:18                Favorites
d-----        03.10.2021     10:57                git-test
d-r---        28.09.2020     16:18                Links
d-r---        28.09.2020     16:18                Music
d-----        03.10.2021     14:05                NerdFonts
dar--l        03.10.2021      9:46                OneDrive
d-----        26.08.2021     22:19                otf
d-r---        28.09.2020     16:18                Saved Games
d-----        31.05.2021      7:06                script
d-r---        28.09.2020     16:18                Searches
d-----        30.09.2021      7:34                test
d-----        12.07.2021     15:13                Tracing
d-r---        04.10.2020     23:52                Videos
d-----        04.04.2021     11:21                vimfiles
-a----        18.10.2020     17:20         93     .gitconfig
-a----        03.03.2021     20:00        202     .node_repl_history
-a----        03.10.2021     13:09          2     .omp.cache
-a----        15.02.2021     20:47         14     .php_history
-a----        23.05.2021     21:07        428     1.csv
-a----        28.09.2020     23:37         23     1.js
```

Рис. П3.6. Список файлов и каталогов после подключения модуля Terminal-Icons

Предметный указатель

.

.NET-объекты и классы
◊ System.Convert 107
◊ System.DateTime 107
◊ System.Math 46, 106
◊ System.String 107
◊ System.TimeSpan 107
◊ Windows.Forms.Form 332

A

ActiveX, технология 266
API-ориентированная архитектура 21

C

camelCase 307
COM-объекты
◊ Excel.Application 268
◊ Word.Application 268
◊ WScript.Shell 243
CSV 245

D

Dot-sourcing 196

E

Escape-последовательности 128

H

HTTP 270

J

JSON 250, 279

M

Microsoft Excel, объектная модель 267
◊ объект Application 267
◊ объект Selection 267
◊ объект Workbook 267
◊ объект Workbooks 267
Microsoft Word, объектная модель
◊ объект Application 267
◊ объект Document 267
◊ объект Documents 267
◊ объект Selection 267

O

OLE, технология
◊ OLE Automation 266

P

PascalCase 306
POST 279
PowerShell 6/7 43
PowerShell ISE 285
◊ запуск кода 285
◊ отладка 292
◊ редактирование 289

V

Visual Studio Code 294
◊ командная панель 297
◊ линтер 296

◊ отладка 297
◊ расширение для PowerShell 294
◊ сниппеты 296

W

Windows Console (ConHost) 35
Windows Forms (WinForms) 331
Windows PowerShell
◊ заголовок командного окна 84
◊ приглашение командной строки 85
Windows Presentation Foundation (WPF) 335
Windows Script Host (WSH) 25
Windows Terminal 36
WMI-объекты
◊ Win32_Process 260
◊ Win32_Service 266

X

XAML 335

А

Автоматическое завершение команд 68
Ассоциативные массивы 144
Атрибуты
◊ AllowEmptyString() 180
◊ AllowNull() 180
◊ CmdletBinding() 189
◊ ValidateCount() 180
◊ ValidateLength() 136, 180
◊ ValidateNotNull() 180
◊ ValidatePattern() 180
◊ ValidateRange() 136, 180
◊ ValidateScript() 180
◊ ValidateSet() 180

Б

Библиотека типов объекта 240

Д

Диски PowerShell 57

Ж

Журнал команд *См.* Журнал сеанса
Журнал сеанса 78

К

Командлет 47
Командная оболочка 34
Командная строка
◊ вложенная 319
Командно-сценарные языки 17
Команды PowerShell
◊ Add-Content 231
◊ Add-History 79
◊ ConvertFrom-CSV 249, 250
◊ ConvertTo-CSV 249, 250
◊ ConvertTo-Html 117
◊ Copy-Item 231
◊ Disable-PSBreakPoint 322
◊ Enable-PSBreakPoint 322
◊ Export-Alias 56
◊ Export-CSV 248
◊ Find-Module 202
◊ ForEach-Object 103
◊ Format-Custom 110
◊ Format-List 110
◊ Format-Table 110
◊ Format-Wide 110
◊ Get-Alias 53
◊ Get-ChildItem 60, 223
◊ Get-Command 51
◊ Get-Content 229
◊ Get-ExecutionPolicy 89, 192
◊ Get-Help 71, 74
◊ Get-History 78
◊ Get-Host 83
◊ Get-Location 60
◊ Get-Member 93
◊ Get-Process 92, 254
◊ Get-PSBreakPoint 322, 327

Команды PowerShell *(прод.)*
◊ Get-PSDrive 57
◊ Get-PSProvider 58
◊ Get-PSRepository 201
◊ Get-Service 95, 262
◊ Get-Verb 173
◊ gm 93
◊ Group-Object 104
◊ Import-Alias 56
◊ Import-CSV 246
◊ Invoke-History 79
◊ Invoke-Item 118, 259
◊ Invoke-RestMethod 280
◊ Invoke-WebRequest 270
◊ kill 258
◊ Measure-Object 105
◊ Move-Item 236
◊ New-Item 226
◊ New-Object 241
◊ New-PSDrive 62
◊ Out-Default 113
◊ Out-File 114
◊ Out-GridView 115
◊ Out-Host 113
◊ Out-Null 115
◊ Out-Printer 114
◊ Read-Host 314
◊ Remove-Item 236
◊ Remove-PSBreakPoint 322, 329
◊ Rename-Item 235
◊ Restart-Service 264
◊ Select-Object 100
◊ Select-String 237
◊ Set-Alias 55
◊ Set-Content 230
◊ Set-ExecutionPolicy 89, 192
◊ Set-Location 60, 222
◊ Set-PSBreakPoint 322, 326
◊ Set-PSDebug 315
◊ Set-Service 265
◊ Sort-Object 98
◊ Start-Process 259
◊ Start-Service 264
◊ Start-Transcript 80
◊ Stop-Process 258
◊ Stop-Service 263
◊ Stop-Transcript 80
◊ Suspend-Service 263
◊ Test-Path 87, 130
◊ Uninstall-Module 204
◊ Update-Module 204
◊ Where-Object 95
◊ Write-Debug 314
◊ Write-Error 120
◊ Write-Host 313
◊ Write-Verbose 120
Композиция команд 91
Конвейеризация команд 23
Консоль 33

Л

Литерал 158

М

Массивы 138
◊ доступ к элементам 139
◊ оператор присваивания 142
◊ сохранение вывода командлетов 143
◊ увеличение длины 141
◊ удаление массивов 143
◊ удаление элементов 142
Метасимвол 158
Модули
◊ oh-my-posh 358
◊ posh-git 355
◊ PowerShell Gallery 199, 201
◊ PSReadLine 352
◊ Terminal-Icons 362
◊ модули-сценарии 199
◊ форматирования 109

О

Объекты в PowerShell
◊ анализ структуры 93
◊ выделение свойств 100
◊ выполнение действий в конвейере 103
◊ группировка 104
◊ измерение характеристик 105
◊ сортировка 98
◊ фильтрация 95
Операторы
◊ арифметические 149
◊ логические 97, 159
◊ присваивания 154
◊ проверки на соответствие шаблону 157
◊ сравнения 96, 155

Ошибки
◊ критические 206, 214, 215
◊ некритические 206, 214
◊ объект ErrorRecord 207

П

Параметры командлетов 48
◊ общие 50
Параметры функций 175
◊ валидация 179
◊ дополнительные атрибуты 179
◊ значения по умолчанию 178
◊ оператор Param 182
◊ переключатели 181
◊ типы 177
Переменные PowerShell
◊ $$ 131
◊ $? 131, 213, 214
◊ $^ 132
◊ $_ 132
◊ $Args 132, 173
◊ $DebugPreference 121, 132
◊ $Error 132, 210
◊ $ErrorActionPreference 121, 132, 214
◊ $Home 132
◊ $Host 132
◊ $InformationPreference 121
◊ $Input 132, 186
◊ $LASTEXITCODE 214
◊ $MaximumAliasCount 132
◊ $MaximumDriveCount 132
◊ $MaximumErrorCount 210
◊ $MaximumFunctionCount 132
◊ $MaximumHistoryCount 132
◊ $MaximumVariableCount 132
◊ $NestedPromptLevel 320
◊ $Null 130
◊ $Profile 87
◊ $PSHome 94, 132
◊ $PSVersionTable 349
◊ $StackTrace 132
◊ $Transcript 81
◊ $VerbosePreference 121, 132
◊ $WarningPreference 132
◊ дополнительные атрибуты 136
◊ константы 136
◊ область видимости 197
◊ переменные оболочки 131
Переменные окружения *См* Переменные среды Windows
Переменные среды Windows 137
Переназначение устройств ввода/вывода 22
Перенаправление вывода 44, 113, 121, 122
Платформа .NET Framework 29
Политики выполнения 88
Потоки PowerShell 120
Приведение типов 134, 135
Провайдеры PowerShell 58
Протоколирование действий 80
Профили PowerShell 87
Псевдонимы команд 53
Псевдонимы типов 134
Псевдотерминал 34

С

Сборка .NET 331
Системный путь 137
Службы 261
Сниппет 289
Сплаттинг переменных 183
Справочная система PowerShell 71
Стандартные текстовые потоки 22
Статические методы 106
Строки
◊ в двойных кавычках 127
◊ в одинарных кавычках 127
◊ типа here-string 129
Суффиксы-множители 126
Сценарии PowerShell 53, 191
◊ выход 195
◊ запуск 192
◊ передача аргументов 194
◊ создание 191

Т

Терминал 32
Точка останова 321

У

Управляющие инструкции
◊ Break 165
◊ Continue 165
◊ Exit 195
◊ If ... ElseIf ... Else 160
◊ Switch 166
◊ Trap 215
◊ try/catch/finally 218

Ф

Форматирование вывода 110
Функции 52, 172
◊ область видимости 195
◊ расширенные 189

Х

Хэш-таблица 102, 144

Ц

Циклы
◊ Do ... While 162
◊ For 162
◊ Foreach
  ▫ вне конвейера команд 163
  ▫ внутри конвейера команд 164
◊ While 161

Ш

Шаблоны
◊ с подстановочными символами 157
◊ с регулярными выражениями 158

Современный PowerShell

Попов Андрей Владимирович,
специалист ИТ-поддержки
и преподаватель с 25-летним стажем,
автор нескольких книг по технологиям
сценариев Windows, в том числе
«Введение в Windows PowerShell»
о первой версии оболочки и среды
выполнения Microsoft PowerShell.

Книга предназначена для пользователей и администраторов Windows, которые хотят узнать о возможностях оболочки Windows PowerShell и использовать их для автоматизации повседневных задач администрирования. Описаны базовые механизмы и основные конструкции языка PowerShell, приведены приемы создания и отладки сценариев. Даны примеры работы с файловой системой, службами и серверами автоматизации, с веб-ресурсами при помощи HTTP-запросов, подробно рассказывается о построении графического интерфейса для сценариев PowerShell. Представлены кросс-платформенные возможности PowerShell в macOS и Linux.

Автоматизация решения повседневных задач администратора

СИСАДМИН
СИСТЕМНЫЙ АДМИНИСТРАТОР

bhv®

КАТЕГОРИЯ: операционные системы

ISBN 978-5-9775-6874-6

9 785977 568746

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru