ЛАБОРАТОРНАЯ РАБОТА №3 ОБОЛОЧКА КОМАНДНОЙ СТРОКИ WINDOWS POWERSHELL 2.0

Цель работы – знакомство с основными возможностями оболочки командной строки Windows PowerShell 2.0

1 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Цели и задачи создания новой оболочки

Новая оболочка Windows PowerShell была задумана разработчиками Microsoft как более мощная среда для написания сценариев и работы из командной строки. Разработчики PowerShell преследовали несколько целей, главная из которых – создание среды составления сценариев, которая наилучшим образом подходила бы для современных версий ОС Windows и была бы более функциональной, расширяемой и простой в использовании, чем какой-либо аналогичный продукт для любой другой ОС. В первую очередь эта среда должна была подходить для решения задач, стоящих перед системными администраторами, 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. В новой оболочке присутствуют команды, позволяющие в интерактивном режиме работать с COM-объектами, а также с экземплярами классов, определенных в информационных схемах WMI и .NET.
- 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 новой оболочке имена всех внутренних команд (в PowerShell 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, создаваемыми пользователем. При этом они полностью интегрируются в оболочку, информация о них может быть получена из стандартной справочной системы PowerShell.
- 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 ls, то он получит ожидаемый результат: список файлов в текущем каталоге (то же самое относится к команде dir).
- Разработка полноценной встроенной справочной системы для внутренних команд. Для большинства внутренних команд в справочной системе дано подробное описание и примеры использования. В любом случае встроенная справка по любой внутренней команде будет содержать краткое описание всех ее параметров.
- 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.

Главной особенностью среды PowerShell, отличающей ее от всех других оболочек командной строки, является то, что единицей обработки и передачи информации здесь является **объект**, а не строка текста.

1.2 Отличие PowerShell от других оболочек – ориентация на объекты

При разработке любого языка программирования одним из основных является вопрос о том, какие типы данных и каким образом будут в нем представлены. При создании PowerShell разработчики решили не изобретать ничего нового и воспользоваться унифицированной объектной моделью .NET.

Рассмотрим пример. В Windows 7 есть консольная утилита tasklist.exe, которая выдает информацию о процессах, запущенных в системе: (рис.1)

```
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
```

Рис.1-Информация о процессах

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` соответствующую строку, извлечь из нее подстроку, содержащую нужное число и убрать пробелы между разрядами. В PowerShell 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 позволяет PowerShell напрямую использовать функциональность различных библиотек, являющихся частью платформы .NET. Например, чтобы узнать, каким днем недели было 9 ноября 2015 года, в PowerShell можно выполнить следующую команду:

```
(get-date "09.11.2015").dayofweek.tostring()
```

В этом случае команда get-date возвращает .NET-объект DateTime, имеющий свойство DayOfWeek, при обращении к которому вычисляется день недели для соответствующей даты.

1.3 Запуск оболочки. Выполнение команд

Для запуска оболочки следует нажать на кнопку Пуск (Start), открыть меню Все программы (All Programs), выбрать элемент Стандартные, Windows PowerShell и Windows PowerShell ISE. Другой

вариант запуска оболочки – пункт Выполнить… (Run) в меню Пуск (Start), ввести имя файла powershell_ise и нажать кнопку OK.

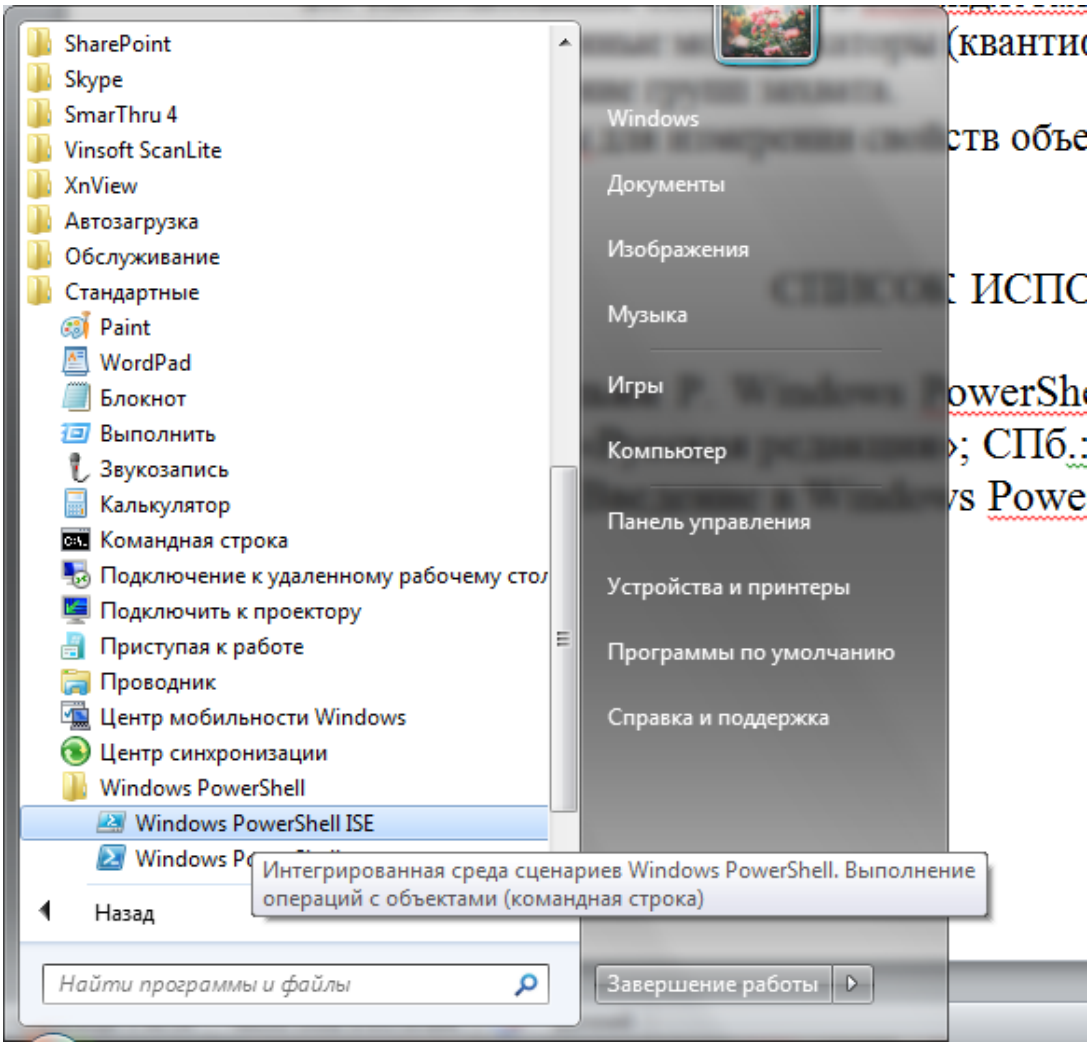

Рис. 1. Запуск PowerShell ISE с помощью меню

В результате откроется новое командное окно с приглашением вводить команды (рис. 2). В нижней части окна вводятся команды. Средняя часть окна содержит результаты выполнения введенной команды или сообщения об ошибках. Верхняя часть используется для работы с командными файлами.

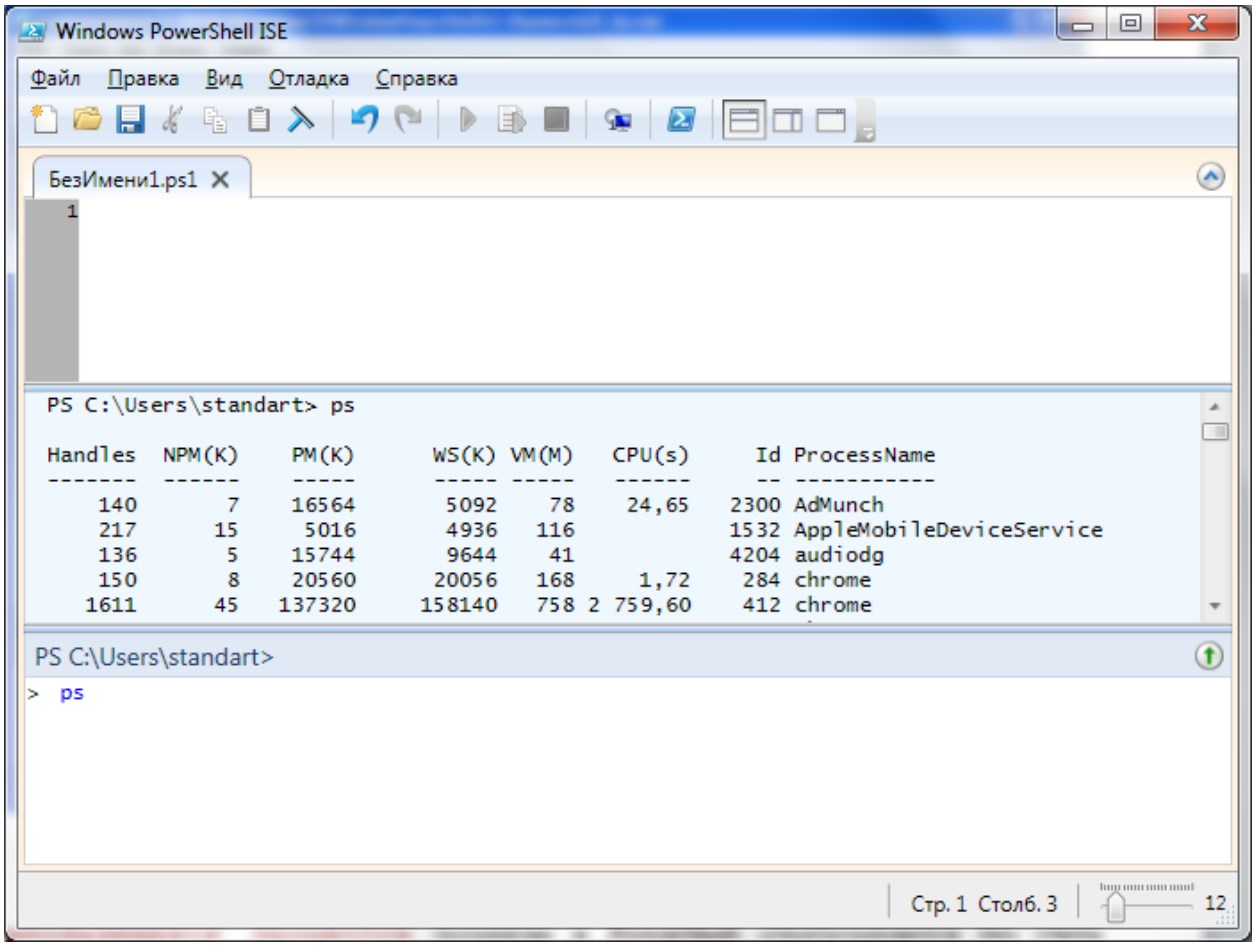

Рис. 2. Командное окно оболочки PowerShell ISE

Выполним первую команду в PowerShell - команду `ps – список выполняющихся процессов` (команды в PowerShell обрабатываются без учета регистра). На экран будет выведен список выполняющихся процессов.

Предыстория введенных команд работает также, как и в CMD.

1.4 Типы команд PowerShell

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

Первый тип – так называемые **командлеты** (cmdlet). Этот термин используется пока только внутри PowerShell. Командлет – аналог внутренней команды интерпретатора командной строки - представляет собой класс .NET, порожденный от базового класса `Cmdlet`; разрабатываются командлеты с помощью пакета PowerShell Software Developers Kit (SDK). Единый базовый класс `Cmdlet` гарантирует совместимый синтаксис всех командлетов, а также автоматизирует анализ параметров командной строки и описание синтаксиса командлетов для встроенной справки. Командлеты рассматриваются в данной работе. С

командами других типов можно ознакомиться, используя [1].

Данный тип команд компилируется в динамическую библиотеку (DLL) и подгружается к процессу PowerShell во время запуска оболочки (то есть сами по себе командлеты не могут быть запущены как приложения, но в них содержатся исполняемые объекты). Командлеты – это аналог внутренних команд традиционных оболочек.

Следующий тип команд – **функции**. Функция – это блок кода на языке PowerShell, имеющий название и находящийся в памяти до завершения текущего сеанса командной оболочки. Функции, как и командлеты, поддерживают именованные параметры. Анализ синтаксиса функции производится один раз при ее объявлении.

Сценарий – это блок кода на языке PowerShell, хранящийся во внешнем файле с расширением ps1. Анализ синтаксиса сценария производится при каждом его запуске.

Последний тип команд – **внешние исполняемые файлы**, которые выполняются обычным образом операционной системой.

1.5 Имена и синтаксис командлетов

В PowerShell аналогом внутренних команд являются командлеты. 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 (конвейеризации объектов между командлетами).

Команды Windows PowerShell следуют определенным правилам именования: Команды Windows PowerShell состоят из глагола и существительного (всегда в единственном числе), разделенных тире. Глагол задает определенное действие, а существительное определяет объект, над которым это действие будет совершено. Команды записываются на английском языке. Пример: Get-Help вызывает интерактивную справку по синтаксису Windows PowerShell.

Перед **параметрами** ставится символ «-». Например: Get-Help –Detailed.

В Windows PowerShell также включены псевдонимы многих известных команд. Это упрощает знакомство и использование Windows PowerShell. Пример: команды help (классический стиль Windows) и man (классический стиль Unix) работают так же, как и Get-Help.

Например, `Get-Process` (получить информацию о процессе), `Stop-Service` (остановить службу), `Clear-Host` (очистить экран консоли) и т.д. Чтобы просмотреть список командлетов, доступных в ходе текущего сеанса, нужно выполнить командлет `Get-Command`.

По умолчанию командлет `Get-Command` выводит сведения в трех столбцах: `CommandType`, `Name` и `Definition`. При этом в столбце

`Definition` отображается синтаксис командлетов (многоточие (…) в столбце синтаксиса указывает на то, что данные обрезаны).

Замечание. Косые черты (`/` и `\`) вместе с параметрами в оболочке Windows PowerShell не используются.

В общем случае синтаксис командлетов имеет следующую структуру:

имя_командлета –параметр1 -параметр2 аргумент1 аргумент2

Здесь `параметр1` – параметр (переключатель), не имеющий значения; `параметр2` – имя параметра, имеющего значение `аргумент1`; `аргумент2` – параметр, не имеющий имени. Например, командлет `Get-Process` имеет параметр `Name`, который определяет имя процесса, информацию о котором нужно вывести. Имя этого параметра указывать необязательно. Таким образом, для получения сведений о процессе `Far` можно ввести либо команду `Get-Process -Name Far`, либо команду `Get-Process Far`.

1.6 Автоматическое завершение команд (автозавершение ввода команд)

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

Подобное автоматическое завершение срабатывает, во-первых, для имен файлов и путей файловой системы. При нажатии клавиши <Tab> PowerShell автоматически расширит частично введенный путь файловой системы до первого найденного совпадения. При повторении нажатия клавиши <Tab> производится циклический переход по имеющимся возможностям выбора. Также в PowerShell реализована возможность автоматического завершения путей файловой системы на основе шаблонных символов (`?` и `*`). Например, если ввести команду `cd c:\pro*files` и нажать клавишу <Tab>, то в строке ввода появится команда `cd 'C:\Program Files'`.

Во-вторых, в PowerShell реализовано автозавершение 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 подходящий вариант имени выбирается путем повторного нажатия <Tab>). Аналогичным образом автозавершение срабатывает для частично введенных имен параметров командлета: нажимая клавишу <Tab>, мы будем циклически перебирать подходящие имена.

Наконец, PowerShell позволяет автоматически завершать имена используемых переменных (объектов) и имена свойств объектов.

1.7 Псевдонимы команд

Механизм псевдонимов, реализованный в оболочке PowerShell, дает возможность пользователям выполнять команды по их **альтернативным именам** (например, вместо команды `Get-Childitem` можно пользоваться псевдонимом `dir`). В PowerShell заранее определено много псевдонимов, можно также добавлять собственные псевдонимы в систему.

Псевдонимы в PowerShell делятся на два типа. Первый тип предназначен для совместимости имен с разными интерфейсами. Псевдонимы этого типа позволяют пользователям, имеющим опыт работы с другими оболочками (Cmd.exe или Unix-оболочки), использовать знакомые им имена команд для выполнения аналогичных операций в PowerShell, что упрощает освоение новой оболочки, позволяя не тратить усилий на запоминание новых 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 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 – псевдонимы `cat, chdir, clear, diff, h, history, kill, lp, ls, mount, ps, pwd, r, rm, sleep, tee, write`.

Узнать, какой именно командлет скрывается за знакомым псевдонимом, можно с помощью командлета `Get-Alias`:

```
PS C:\> Get-Alias cd
CommandType     Name        Definition
-----------     ----        ----------
Alias           cd          Set-Location
```

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, для 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. Определить собственный псевдоним можно с помощью командлета `Set-Alias`.

1.8 Справочная система PowerShell

В PowerShell предусмотрено несколько способов получения справочной информации внутри оболочки.
Краткую справку по одному командлету можно получить с помощью параметра `?` (вопросительный знак), указанного после имени этого командлета. Например:
PS C:\> get-process -?

Вместо help или man в Windows PowerShell можно также использовать команду Get-Help. Ее синтаксис описан ниже:
Get-Help выводит на экран справку об использовании справки
Get-Help * перечисляет все команды Windows PowerShell
Get-Help *команда* выводит справку по соответствующей команде
Get-Help команда -Detailed выводит подробную справку с примерами команды
Использование команды help для получения подробных сведений о команде help:
Get-Help
Get-Help -Detailed.

Комана `Get-Help` позволяет просматривать справочную информацию не только о разных командлетах, но и о синтаксисе языка PowerShell, о псевдонимах и т. д.
Например, чтобы прочитать справочную информацию об использовании массивов в PowerShell, нужно выполнить следующую команду: `Get-Help about_array`.
Командлет `Get-Help` выводит содержимое раздела справки на экран сразу целиком. Функции `man` и `help` позволяют справочную информацию выводить поэкранно (аналогично команде `MORE` интерпретатора Cmd.exe), например: `man about_array`.

1.9 Конвейеризация и управление выводом команд Windows PowerShell

Ранее было рассмотрено понятие конвейеризации (или композиции) команд интерпретатора Cmd.exe, когда выходной поток одной команды перенаправлялся во входной поток другой, объединяя тем самым две команды вместе. Подобные конвейеры команд используются в большинстве оболочек командной строки и являются средством, позволяющим передавать информацию между разными процессами. Механизм композиции команд представляет собой, вероятно, наиболее ценную концепцию, используемую в интерфейсах командной строки. Конвейеры не только снижают усилия, прилагаемые при вводе сложных

команд, но и облегчают отслеживание потока работы в командах.

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 (ниже мы подробнее рассмотрим операции фильтрации и сортировки элементов конвейера).

1.9.1 Конвейеризация объектов в PowerShell

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.

Передача данных между командами в виде объектов имеет большое преимущество над обычным обменом информацией посредством потока текста. Ведь команда, принимающая поток текста от другой утилиты, должна его проанализировать, разобрать и выделить нужную ей информацию, а это может быть непросто, так как 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 возникает новый вопрос: каким образом 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.

Пример Запустим командлет Get-Process, который выводит информацию о запущенных в системе процессах (рис.2):

```
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
```

Рис.2- Информацию о запущенных в системе

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 свойство нужно проверять у выводимых объектов?

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

1.9.2 Просмотр структуры объектов

Для анализа структуры объекта, возвращаемого определенной командой, проще всего направить этот объект по конвейеру на командлет `Get-Member` (псевдоним `gm`), например (рис.3):

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;}
. . .
```

Рис.3- объект по конвейеру на командлет `Get-Member`

Здесь мы видим имя .NET-класса, экземпляры которого возвращаются в ходе работы исследуемого командлета (в нашем примере это класс System.Diagnostic.Process), а также полный список элементов

объекта (в частности, интересующее нас свойство Responding, определяющего "зависшие" процессы). При этом на экран выводится очень много элементов, просматривать их неудобно. Командлет `Get-Member` позволяет перечислить только те элементы объекта, которые являются его **свойствами**. Для этого используется параметр `MemberType` со значением `Properties`:(рис.4)

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

```
   TypeName: System.Diagnostics.Process
Name                       MemberType Definition
----                       ---------- ----------
BasePriority               Property   System.Int32 BasePriority {get;}
EnableRaisingEvents        Property   System.Boolean EnableRaisingEvents...
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;}
. . .
Responding                 Property   System.Boolean Responding {get;}
. . .
```

Рис.4- Элементы объекта, которые являются его свойствами

Процессам ОС соответствуют объекты, имеющие очень много свойств, на экран же при работе командлета `Get-Process` выводятся лишь несколько из них (способы отображения объектов различных типов задаются конфигурационными файлами в формате XML, находящимися в каталоге, где установлен файл powershell.exe).

Рассмотрим наиболее часто используемые операции над элементами конвейера: фильтрации и сортировки.

1.9.3 Фильтрация объектов в конвейере

В PowerShell поддерживается возможность **фильтрации объектов** в конвейере, т.е. удаление из конвейера объектов, не удовлетворяющих определенному условию. Данную функциональность обеспечивает командлет `Where-Object`, позволяющий проверить каждый объект, находящийся в конвейере, и передать его дальше по конвейеру, только если объект удовлетворяет условиям проверки.

Например, для вывода информации о "зависших" процессах (объекты, возвращаемые командлетом `Get-Process`, у которых свойство `Responding` равно `False`) можно использовать следующий конвейер:

```
Get-Process | Where-Object {-not $_.Responding}
```

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

```
Get-Process | Where-Object {$_.Id -gt 1000}
```

В блоках сценариев командлета `Where-Object` для обращения к текущему объекту конвейера и извлечения нужных свойств этого объекта используется **специальная переменная** `$_`, которая создается оболочкой PowerShell автоматически. Данная переменная используется и в других командлетах, производящих обработку элементов конвейера.

Условие проверки в `Where-Object` задается в виде **блока сценария** – одной или нескольких команд PowerShell, заключенных в фигурные скобки `{}`. Результатом выполнения данного блока сценария должно быть значение логического типа: `True` (истина) или `False` (ложь). Как можно понять из примеров, в блоке сценария используются специальные операторы сравнения.

Замечание. В PowerShell для операторов сравнения не используются обычные символы `>` или `<`, так как в командной строке они обычно означают перенаправление ввода/вывода.

Основные операторы сравнения приведены в (табл. 1).

Таблица 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
-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

Операторы сравнения можно соединять друг с другом с помощью логических операторов (см. табл. 2).

Таблица 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)

1.9.4 Сортировка объектов

Сортировка элементов конвейера – еще одна операция, которая часто применяется при конвейерной обработке объектов. Данную операцию осуществляет командлет `Sort-Object`: ему передаются имена свойств, по которым нужно произвести сортировку, а он возвращает данные, упорядоченные по значениям этих свойств.

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

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

Для сортировки в обратном порядке используется параметр `Descending`:

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

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

```
Get-Process | Where-Object {$_.Id -gt 1000} | Sort-Object cpu –Descending
```

1.9.5 Использование переменных

В переменных хранятся все возможные значения, даже если они являются объектами. Имена переменных в PowerShell всегда должны начинаться с символа «$». Можно сохранить список процессов в переменной, это позволит в любое время получать доступ к списку процессов. Присвоить значение переменной легко:

```
$a = get-process | sort-object CPU
```

Вывести содержимое переменной можно, просто напечатав в командной строке $a.

1.9.6 Создание и использование массивов

Для создания и инициализации массива достаточно присвоить значения его элементам. Значения, добавляемые в массив, разделяются запятыми и отделяются от имени массива символом присваивания. Например, следующая команда создаст массив $a из трех элементов:

```
PS C:\> $a=1,5,7
```

```
PS C:\>$a
1
5
7
```

Можно создать и инициализировать массив, используя оператор диапазона (..). Например, команда

```
PS C:\> $b=10..15
```

создает и инициализирует массив $b, содержащий 6 значений 10, 11, 12, 13, 14 и 15.

Для создания массива может использоваться операция ввода значений его элементов из текстового файла:

```
PS C:\> $f = Get-Content c:\data\numb.txt –TotalCount 25
PS C:\>$f.length
25
```

В приведенном примере результат выполнения командлета Get-Content присваивается массиву $f. Необязательный параметр –TotalCount ограничивает количество прочитанных элементов величиной 25. Свойство объекта массив – length – имеет значение, равное количеству элементов массива, в примере оно равно 25 (предполагается, что в текстовом файле munb.txt по крайней мере 25 строк).

1.9.6.1 Обращение к элементам массива

Длина массива (количество элементов) хранится в свойстве Length. Для обращения к определенному элементу массива нужно указать его индекс в квадратных скобках после имени переменной. Нумерация элементов массива **всегда начинается с нуля**. В качестве индекса можно указывать и отрицательные значения, отсчет будет вестись с конца массива – индекс -1 соответствует последнему элементу массива.

1.9.6.2 Операции с массивами

По умолчанию массивы PowerShell могут содержать элементы разных типов (целые 32-х разрядные числа, строки, вещественные и другие), то есть являются полиморфными. Можно создать массив с жестко заданным типом, содержащий элементы только одного типа, указав нужный тип в квадратных скобках перед именем переменной. Например, следующая команда создаст массив 32-х разрядных целых чисел:

```
PS C:\> [int[]]$a=1,2,3
```

Массивы PowerShell базируются на .NET-массивах, имеющих фиксированную длину, поэтому обращение за предел массива фиксируется как ошибка. Имеется способ увеличения первоначально определенной 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
```

При выполнении оператора += происходит следующее:
создается новый массив, размер которого достаточен для помещения в него всех элементов;
первоначальное содержимое массива копируется в новый массив;
новые элементы копируются в конец нового массива.

Таким образом, на самом деле создается новый массив большего размера.

Можно объединить два массива, например $b и $c в один с помощью операции конкатенации +. Например:

```
PS C:\> $d=$b+$c
```

1.10 Регулярные выражения – назначение и использование

Регулярные выражения (или сокращенно “регэкспы” (regexp, regular expressions)) обладают огромной мощью, и способны сильно упростить жизнь системного администратора или программиста. В PowerShell регулярные выражения легко доступны, удобны в использовании и максимально функциональны. PowerShell использует реализацию регулярных выражений .NET.

Регулярные выражения - это специальный мини-язык, служащий для разбора (parsing) текстовых данных. С его помощью можно разделять строки на компоненты, выбирать нужные части строк для дальнейшей обработки, производить замены и т. д.

Знакомство с регулярными выражениями начнем с более простой технологии, служащей подобным целям - с **подстановочных символов.** Наверняка вы не раз выполняли команду dir, указывая ей в качестве аргумента маску файла, например *.exe. В данном случае звёздочка означает “любое количество любых символов”. Аналогично можно использовать и знак вопроса, он будет означать “один любой символ”, то есть dir ??.exe выведет все файлы с расширением .exe и именем из двух

символов. В PowerShell можно применять и еще одну конструкцию – **группы символов**. Так например [a-f] будет означать “один любой символ от a до f, то есть (a,b,c,d,e,f)”, а [smw] любую из трех букв (s, m или w). Таким образом команда get-childitem [smw]??.exe выведет файлы с расширением .exe, у которых имя состоит из трех букв, и первая буква либо s, либо m, либо w.

1.10.1 Оператор PowerShell -match

Для начала изучения мы будем использовать оператор PowerShell -match, который позволяет сравнивать текст слева от него, с регулярным выражением справа. В случае если текст подпадает под регулярное выражение, оператор выдаёт True, иначе – False.

```
PS C:\> "PowerShell" -match "Power"
True
```

При сравнении с регулярным выражением ищется лишь вхождение строки, полное совпадение текста необязательно (разумеется, это можно изменить). То есть достаточно, чтобы регулярное выражение встречалось в тексте.

```
PS C:\> "Shell" -match "Power"
False
PS C:\> "PowerShell" -match "rsh"
True
```

Еще одна тонкость: оператор -match по умолчанию не чувствителен к регистру символов (как и другие текстовые операторы в PowerShell), если же нужна чувствительность к регистру, используется -cmatch:

```
PS C:\> "PowerShell" -cmatch "rsh"
False
```

1.10.2 Использование групп символов

В регулярных выражениях можно использовать и группы символов:

```
PS C:\> Get-Process | where {$_.name -match "sy[ns]"} (рис.5)
```

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    165      11     2524       8140    79       0,30   5228 mobsync
    114      10     3436       3028    83      50,14   3404 SynTPEnh
    149      11     2356        492    93       0,06   1592 SynTPStart
    810       0      116        380     6                 4 System
```

Рис.5-Использование групп символов

И диапазоны в этих группах:

```
PS C:\> "яблоко","апельсин","груша","абрикос" -match "а[а-п]"
апельсин
абрикос
```

В левой части оператора -match находится массив строк, и оператор соответственно вывел лишь те строки, которые подошли под регулярное выражение.

Перечисления символов можно комбинировать, например группа [агдэ-я] будет означать “А или Г или Д или любой символ от Э до Я включительно”. Но гораздо интереснее использовать диапазоны для определения целых **классов символов**. Например [а-я] будет означать любую букву русского алфавита, а [a-z] английского. Аналогично можно поступать с цифрами – следующая команда выведет все процессы, в именах которых встречаются цифры:

PS C:\> Get-Process | where {$_.name -match "[0-9]"} (рис.6)

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
57	2	404	1620	16	0,05	984	ati2evxx
110	4	2540	4868	36	0,20	852	hpgs2wnd
105	3	940	3292	36	0,19	2424	hpgs2wnf
91	3	2116	3252	34	0,06	236	rundll32

Рис.6- Процессы, в именах которых встречаются цифры

Так как эта группа используется достаточно часто, для неё была выделена специальная последовательность – \d (от слова digit). По смыслу она полностью идентична [0-9], но короче.

PS C:\> Get-Process | where {$_.name -match "\d"} (рис.7)

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
93	10	1788	2336	70	1,25	548	FlashUtil10c
158	12	6500	1024	96	0,14	3336	smax4pnp
30	6	764	160	41	0,02	3920	TabTip32

Рис.7- Последовательность – \d (от слова digit).

Так же последовательность была выделена для группы “любые буквы любого алфавита, любые цифры, или символ подчеркивания” эта группа обозначается как \w (от word) она примерно эквивалентна конструкции [a-za-я_0-9] (в \w еще входят символы других алфавитов которые используются для написания слов).

Другая популярная группа: \s – “пробел, или другой пробельный символ” (например символ табуляции). Сокращение от слова space. В большинстве случаев вы можете обозначать пробел просто как пробел, но эта конструкция добавляет читабельности регулярному выражению.

Не менее популярной группой можно назвать символ . (точка). Точка в регулярных выражениях аналогична по смыслу знаку вопроса в подстановочных символах, то есть обозначает один любой символ.

Все вышеперечисленные конструкции можно использовать как отдельно, так и в составе групп, например [\s\d] будет соответствовать любой цифре или пробелу. Если вы хотите указать внутри группы символ - (тире/минус) то надо либо экранировать его символом \ (обратный слеш), либо поставить его в начале группы, чтобы он не был случайно истолкован как диапазон:

```
PS C:\> "?????","Word","123","-" -match "[-\d]"
123
-
```

1.10.3 Отрицательные группы и якоря

Рассмотрим некоторые более “продвинутые” конструкции регулярных выражений.

Предполагается, что вы уже знаете, как указать регулярному выражению, какие символы и/или их последовательности должны быть в строке для совпадения. А что если нужно указать не те символы, которые должны присутствовать, а те, которых не должно быть? То есть если нужно вывести лишь согласные буквы, вы можете их перечислить, а можете использовать и отрицательную группу с гласными, например:

```
PS C:\> "a","b","c","d","e","f","g","h" -match "[^aoueyi]"
b
c
d
f
g
h
```

"Крышка" в качестве первого символа группы символов означает именно **отрицание**. То есть на месте группы может присутствовать любой символ кроме перечисленных в ней. Для того чтобы включить отрицание в символьных группах (\d, \w, \s), не обязательно заключать их в квадратные скобки, достаточно перевести их в **верхний регистр**. Например \D будет означать "что угодно, кроме цифр", а \S "всё кроме пробелов"

```
PS C:\> "a","b","1","c","45" -match "\D"
a
b
c
PS C:\> "a","-","*","c","&" -match "\W"
-
*
&
```

Символьные группы позволяют указать лишь содержимое одной позиции, один символ, находящийся в неопределенном месте строки. А что если надо например выбрать все слова которые начинаются с буквы w? Если просто поместить эту букву в регулярное выражение, то оно совпадёт для всех строк, где w вообще встречается, и не важно – в начале, в середине или в конце строки. В таких случаях на помощь приходят "**якоря**". Они позволяют производить сравнение, начиная с определенной позиции в строке.

^ (крышка) является **якорем начала строки**, а $ (знак доллара) - обозначает конец строки.

Не запутайтесь - ^ как символ отрицания используется лишь в начале группы символов, а вне группы - этот символ является уже якорем. Авторам регулярных выражений явно не хватало специальных символов, и они по возможности использовали их более чем в одном месте.

Пример1. Вывод списка процессов, имена которых начинаются с буквы w:

PS C:\> Get-Process | where {$_.name -match "^w"} (рис.8)

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
80	10	1460	156	47	0,11	452	wininit
114	9	2732	1428	55	0,56	3508	winlogon
162	11	3660	1652	44	0,14	3620	wisptis
225	20	5076	4308	95	31,33	3800	wisptis

Рис.8- Вывод списка процессов

Эта команда вывела процессы, у которых сразу после начала имени (^) следует символ w. Иначе говоря, имя начинается на w. Для усложнения примера, и для упрощения понимания, добавим сюда “крышку” в значении отрицательной группы:

PS C:\> Get-Process | where {$_.name -match "^w[^l-z]"} (рис.9)

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
80	10	1460	156	47	0,11	452	wininit
114	9	2732	1428	55	0,56	3508	winlogon
162	11	3660	1652	44	0,14	3620	wisptis
225	20	5076	4308	95	31,50	3800	wisptis

Рис.9-(^) следует символ w

Теперь команда вывела процессы, у которых имя начинается с символа w, а следующий символ является чем угодно, только не символом из диапазона l-z.

Для закрепления опробуем второй якорь – конец строки:

```
PS C:\> "Яблоки","Груши","Дыня","Енот","Апельсины","Персик" -match "[ыи]$"
Яблоки
Груши
Апельсины
```

Это выражение вывело нам все слова в которых последняя буква И или Ы.

Если вы можете точно описать содержимое всей строки, то вы можете использовать и оба якоря одновременно:

```
PS C:\> "abc","adc","aef","bca","aeb","abec","abce" -match "^a.[cb]$"
abc
adc
aeb
```

Это регулярное выражение выводит все строки, которые начинаются с буквы А, за которой следует один любой символ (точка), затем символ С или В и затем конец строки.

Обозначения некоторых классов символов (метасимволы) приведены в (табл. 3).

Таблица 3.

Метасимволы, используемые в регулярных выражениях

Метасимвол	Описание метасимвола
.(точка)	Предполагает, что в конечном выражении на ее месте будет стоять любой символ. Продемонстрируем это на примере набора английских слов: **Исходный набор строк:** wake make machine cake maze **Регулярное выражение:** ma.e **Результат:** make maze
\w	Замещает любые символы, которые относятся к буквам, цифрам и знаку подчеркивания. Пример: **Исходный набор строк:** abc a$c

Метасимвол	Описание метасимвола
	a1c a c **Регулярное выражение:** a\wc **Результат:** abc a1c
\W	Замещает все символы, кроме букв, цифр и знака подчеркивания (то есть является обратным метасимволу \w). Пример: **Исходный набор строк:** abc a$c a1c a c **Регулярное выражение:** a\Wc **Результат:** a$c a c
\d	Замещает все цифры. Продемонстрируем его действие на том же примере: **Исходный набор строк:** abc a$c a1c a c **Регулярное выражение:** a\dc **Результат:** alc
\D	Замещает все символы, кроме цифр, например: **Исходный набор строк:** abc a$c alc a c **Регулярное выражение:** a\Dc **Результат:** abc a$c a c

1.10.4 Количественные модификаторы (квантификаторы)

Обычно регулярные выражения гораздо сложнее, чем приведенные выше, и записывать их по одному символу было бы тяжеловато. Например, нужно отобрать строки, состоящие из четырех символов, каждый из которых может быть буквой от A до F или цифрой? Регулярное выражение могло бы выглядеть примерно так:

```
PS C:\> "af12","1FE0","1fz1","B009","C1212" -match "^[a-f\d][a-f\d][a-f\d][a-f\d]$"
af12
1FE0
B009
```

Не слишком то лаконично, не правда ли? К счастью всю эту конструкцию можно значительно сократить. Для этого в регулярных выражениях существует специальная конструкция – "**количественные модификаторы**" (квантификаторы). Эти модификаторы приписываются к любой группе справа, и определяют количество вхождений этой группы. Например, количественный модификатор {4} означает 4 вхождения. Посмотрим на приведенном выше примере:

```
PS C:\> "af12","1FE0","1fz1","B009","C1212" -match "^[a-f\d]{4}$"
af12
1FE0
B009
```

Данное регулярное выражение полностью эквивалентно предыдущему – "4 раза по [a-f\d]". Но этот количественный модификатор не обязательно жестко оговаривает количество повторений. Например,можно задать количество как "от 4 до 6". Делается это указанием внутри фигурных скобок двух чисел через запятую – минимума и максимума:

```
PS C:\> "af12","1FE0","1fA999","B009","C1212","A00062","FF00FF9" -match "^[a-f\d]{4,6}$"
af12
1FE0
1fA999
B009
C1212
A00062
```

Если максимальное количество вхождений безразлично, например, нужно указать "3 вхождения или больше", то максимум можно просто опустить (оставив запятую на месте), например "строка состоящая из 3х или более цифр":

```
PS C:\> "1","12","123","1234","12345" -match "^\d{3,}$"
123
1234
```

12345

Минимальное значение опустить нельзя, но можно просто указать единицу:
PS C:\> "1","12","123","1234","12345" -match "^\d{1,3}$"
1
12
123

Как и в случае с символьными группами, для особенно популярных значений количественных модификаторов, есть короткие псевдонимы:
+ (плюс), эквивалентен {1,} то есть, "одно или больше вхождений"
* (звездочка), то же самое что и {0,} или на русском языке – "любое количество вхождений, в том числе и 0"
? (вопросительный знак), равен {0,1} – "либо одно вхождение, либо полное отсутствие вхождений".
В регулярных выражениях, количественные модификаторы **сами по себе** использоваться не могут. Для них обязателен символ или символьная группа, которые и будут определять их смысл. Вот несколько примеров:
.+ Один или более любых символов. Аналог ?* в простых подстановках (как в cmd.exe).

Следующее выражение выбирает процессы, у которых имя "начинается с буквы S, затем следует 1 или более любых символов, затем снова буква S и сразу после неё конец строки". Иначе говоря "имена которые начинаются и заканчиваются на S":
PS C:\> Get-Process | where {$_.name -match "^s.+s$"} (рис.10)

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
257	14	6540	5220	53	5,97	508	services
30	2	424	128	5	0,08	280	smss

Рис.10-"имена которые начинаются и заканчиваются на S"

\S* Любое количество символов не являющихся пробелами. Подобное выражение может совпасть и с ""(с пустой строкой), ведь под любым количеством подразумевается и ноль, то есть 0 вхождений – тоже результат.
PS C:\> "abc", "cab", "a c","ac","abdec" -match "a\S*c"
abc
ac
abdec
Заметьте, строка "ac" тоже совпала, хотя между буквами A и C вообще не было символов. Если заменить * на + то будет иначе:

```
PS C:\> "abc", "cab", "a c","ac","abdec" -match "a\S+c"
abc
abdec
```
бобры? (Это не вопрос, а регулярное выражение). Последовательность "бобр", после которой может идти символ "ы", а может и отсутствовать:
```
PS C:\> "бобр","бобры","бобрята" -match "^бобры?$"
Бобр бобры
```

1.10.5 Группы захвата и переменная $matches

Теперь, когда мы можем с помощью регулярных выражений описывать и проверять строки по достаточно сложным правилам, пора познакомится с другой не менее важной возможностью регулярных выражений – "группами захвата" (capture groups). Как следует из названия, группы можно использовать для группировки. К группам захвата, как и к символам и символьным группам, можно применять количественные модификаторы. Например, следующее выражение означает "Первая буква в строке – S, затем одна или больше групп, состоящих из “знака - (минус) и любого количества цифр за ним” до конца строки":
```
PS C:\> "S-1-5-21-1964843605-2840444903-4043112481" -match "^S(-\d+)+$"
True
```
Или:
```
PS C:\> "Ноут","Ноутбук","Лептоп" -match "Ноут(бук)?"
Ноут
Ноутбук
```

Эти примеры показывают, как можно использовать группы захвата для группировки, но это вовсе не главное их качество. Гораздо важнее то, что часть строки, подпавшая под подвыражение, находящееся внутри такой группы, помещается в специальную переменную – $matches. $Matches - это массив, и в нем может находится содержимое нескольких групп. Причем под индексом 0 туда помещается вся совпавшая строка, начиная с единицы идет содержимое групп захвата. Рассмотрим пример:
```
PS C:\> "At 17:04 Firewall service was stopped." -match "(\d\d:\d\d) (\S+)"
True
PS C:\> $matches

Name                           Value
----                           -----
2                              Firewall
1                              17:04
0                              17:04 Firewall
```

Под индексом 0 находится вся часть строки, подпавшая под регулярное выражение, под 1 находится содержимое первых скобок, и под 2 соответственно содержимое вторых скобок. К содержимому $matches

можно обращаться как к элементам любого другого массива в PowerShell:

```
PS C:\> $matches[1]
17:04
PS C:\> $matches[2]
Firewall
```

Если в строке присутствует много групп захвата, то бывает полезно дать им имена, это сильно облегчает дальнейшую работу с полученными данными:

```
PS C:\> "At 17:04 Firewall service was stopped." -match "(?<Время>\d\d:\d\d) (?<Служба>\S+)"
True

PS C:\> $matches
Name                           Value
----                           -----
Время                          17:04
Служба                         Firewall
0                              17:04 Firewall

PS C:\> $matches.Время
17:04
PS C:\> $matches["Служба"]
Firewall
```

Регулярное выражение конечно усложнилось, но зато работать с результатами гораздо приятнее. Синтаксис именования следующий:

(?<Название Группы>подвыражение)

Не перепутайте порядок, сначала следует знак вопроса. Количественные модификаторы, в том числе ? могут применяться только после группы, и следовательно в начале подвыражения – бессмысленны. Поэтому в группах знак вопроса, следующий сразу за открывающей скобкой, означает особый тип группы, в нашем примере – **именованную.**

Другой тип группы, который часто используется – **незахватывающая** группа. Она может пригодиться в тех случаях, когда не нужно захватывать содержимое группы, а надо применить её только для группировки. Например, в вышеприведённом примере с SID, такая группа была бы более уместна:

```
PS C:\> "S-1-5-21-1964843605-2840444903-4043112481" -match "^S(?:-\d+)+$"
True
PS C:\> $matches
```

```
Name                           Value
----                           -----
0                              S-1-5-21-1964843605-2840444903-4043112481
```

Синтаксис такой группы: (?:подвыражение). Группы можно и вкладывать одну в другую:

```
PS C:\> "MAC address is '00-19-D2-73-77-6F'." -match "is '([a-f\d]{2}(?:-[a-f\d]{2}){5})'"
True
PS C:\> $matches

Name                           Value
----                           -----
1                              00-19-D2-73-77-6F
0                              is '00-19-D2-73-77-6F'
```

1.11 Управляющие инструкции

1.11.1 Инструкция If …ElseIf … Else

В общем случае синтаксис инструкции If имеет вид

If (*условие1*)
  {блок_кода1}
[ElseIf (*условие2*)]
  {блок_кода2}]
[Else
  {блок_кода3}]

При выполнении инструкции If проверяется истинность условного выражения *условие1*.

Если *условие1* имеет значение $True, то выполняется блок_кода1, после чего выполнение инструкции if завершается. Если *условие1* имеет значение $False, проверяется истинность условного выражения *условие2*. Если *условие2* имеет значение $True, то выполняется блок_кода2 и выполнение инструкции if завершается. Если и *условие1,* и *условие2* имеют значение $False, то выполняется блок_кода3 и выполнение инструкции if завершается.

Пример 2. использования инструкции if в интерактивном режиме работы. Сначала переменной $a присвоим значение 10:

```
PS C:\> $a=10
```

Затем сравним значение переменной 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 приведенного примера видно также, что в оболочке PS в интерактивном режиме можно выполнять инструкции, состоящие из нескольких строк, что полезно при отладке сценариев.

1.11.2 Циклы While и Do … While

Самый простой из циклов PS – цикл While, в котором команды выполняются до тех пор, пока проверяемое условие имеет значение $True. Инструкция While имеет следующий синтаксис:
While (условие) {блок_команд}
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
```

1.11.3 Цикл For

Обычно цикл For применяется для прохождения по массиву и выполнения определенных действий с каждым из его элементов. Синтаксис инструкции For:
For (инициация; условие; повторение) {блок_команд}. Пример

```
PS C:\> For ($i=0; $i –lt 3; $i++) {$i }
0
1
2
```

1.11.4 Цикл ForEach

Инструкция ForEach позволяет последовательно перебирать элементы коллекций. Самый простой тип коллекции – массив. Особенность цикла ForEach состоит в том, что его синтаксис и выполнение 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 вид:
ForEach ($элемент in $коллекция) {блок_команд}

При выполнении цикла ForEach автоматически создается переменная $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. Приведенный ниже цикл ForEach отображает значения

элементов массива $lettArr:
PS C:\> $lettArr = “a”, “b”,”c”
PS C:\> ForEach ($lett in $lettArr) {Write-Host $lett}
a
b
c

Инструкция ForEach может также использоваться совместно с командлетами, возвращающими коллекции элементов. Например:
PS C:\> $ln = 0; ForEach ($f in Dir *.txt) {$ln +=$f.length}

В примере создается и обнуляется переменная $ln, затем в цикле ForEach с помощью командлета dir формируется коллекция файлов с расширением txt, находящихся в текущем каталоге. Инструкция ForEach перебирает все элементы этой коллекции, на каждом шаге к текущему файлу выполняется обращение с помощью переменной $f. В блоке команд цикла ForEach к текущему значению переменной $ln добавляется значение свойства Length (размер файла) переменной $f. В результате выполнения цикла в переменной $ln будет получен суммарный размер файлов в текущем каталоге, которые имеют расширение txt.

Инструкция ForEach внутри конвейера команд:

Если инструкция ForEach появляется внутри конвейера команд, то PS использует псевдоним ForEach, соответствующий командлету ForEach-Object. В этом случае фактически выполняется командлет ForEach-Object и не требуется часть инструкции ($элемент in $коллекция), так как элементы коллекции блоку команд предоставляет предыдущая команда конвейера.

Синтаксис инструкции ForEach внутри конвейера команд имеет вид:
команда | ForEach {блок_команд}

Рассмотренный выше пример подсчета суммарного размера файлов из текущего каталога для данного варианта инструкции ForEach примет следующий вид:
PS C:\> $ln = 0; dir *.txt | ForEach { $ln +=$_.Length}

В приведенном примере специальная переменная $_ используется для обращения к текущему объекту конвейера и извлечения его свойств.

1.12 Управление выводом команд в PowerShell

Рассмотрим, 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).

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. Когда объект достигает конца конвейера, PowerShell определяет его тип и ищет его в списке объектов, для которых определено 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 командному интерпретатору Cmd.exe перенаправлять эти данные в файл, на принтер или в пустое устройство.

1.12.1 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.

В оболочке PowerShell вывод форматируют только четыре специальных командлета Format (табл. 4). Это упрощает изучение, так как не нужно запоминать средства и параметры форматирования для других команд (остальные командлеты вывод не форматируют).

Таблица 4.

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): (рис.11)
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

Рис.11- таблица с тремя столбцами (Status, Name и DisplayName)

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 (рис.12.)

```
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
. . .
```

Рис.12- Список служб с помощью командлета `Format-List`

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 командлетом `Get-Service`, однако командлет `Format-Table`, используемый по умолчанию, отбрасывает их, потому что не может вывести на экран более 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 рассмотренный ранее 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-table *
```

1.12.2 Перенаправление выводимой информации

В оболочке PowerShell имеются несколько командлетов, с помощью которых можно управлять выводом данных. Эти командлеты начинаются со слова `Out`, их список можно получить с помощью команлета:

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

Name
----
Out-Default
Out-File
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 (host), которое использует это ядро. В принципе, в качестве главного может выступать любое приложение, в котором реализован ряд специальных интерфейсов, позволяющих корректно интерпретировать получаемую от PowerShell информацию. В нашем случае главным приложением является консольное окно, в котором мы работаем с оболочкой, и командлет `Out-Host` передает выводимую информацию в это консольное окно.

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

Get-Help Get-Process –Full | Out-Host –Paging

1.12.3 Сохранение данных в файл

Командлет `Out-File` позволяет направить выводимые данные вместо окна консоли в текстовый файл. Аналогичную задачу решает оператор перенаправления (>), однако командлет `Out-File` имеет несколько дополнительных параметров, с помощью которых можно более гибко управлять выводом: задавать тип кодировки файла (параметр `Encoding`), задавать длину выводимых строк в знаках (параметр `Width`), выбирать режим перезаписи файла (параметр `Append`). Например, следующая команда направит информацию о выполняющихся на компьютере процессах в файл C:\Process.txt, причем данный файл будет записан в формате ASCII:

Get-Process | Out-File -FilePath C:\Process.txt -Encoding ASCII

1.12.4 Подавление вывода

Командлет `Out-Null` служит для поглощения 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: (рис.13)

PS C:\> mkdir spo

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

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----           03.01.2015  1:01                spo
```

Рис.13-Создании каталога командой `mkdir` на экран выводится его содержимое

Если эта информация не нужна, то результат выполнения команды `mkdir` необходимо передать по конвейеру командлету `Out-Null`:

```
mkdir spo | Out-Null
```

1.12.5 Преобразование данных в формат html, сохранение в файле и просмотр результатов

Для преобразования данных в формат html служит командлет `Convertto-html`. `Параметр Property` определяет свойства объектов, включаемые в выходной документ. Например, для получения списка выполняемых процессов в формате html, включающего имя процесса и затраченное время CPU и записи результата в файл processes.html можно использовать команду

```
Get-Process | Convertto-html -Property Name, CPU > Processes.htm
```

Для просмотра содержимого файла можно использовать командлет

```
 Invoke-Item  “имя документа”
```

Например Invoke-Item “processes.htm”

1.12.6 Инвентаризация и диагностика Windows-компьютеров

Для вывода сведений о процессоре ПК служит командлет Get-wmiobject

```
Get-wmiobject -Class Win32_Processor | Format-list *
```

1.12.7 Командлеты для измерения свойств объектов

Для иэмерения времени выполнения командлетов PS служит командлет Measure-Command

В качестве примера рассмотрим получение времени выполнение командлета dir

```
(Measure-Command {dir}).TotalSeconds
```

Для получения статистических данных служит командлет Measure-Object. Для числовых массивов с его помощью можно получить максимальное, минимальное, среднее значение элементов массива и их сумму. Если имеется инициализированный массив ms, для указанной цели используется командлет

```
$ms | measure-object –maximum –minimum –average -sum
```