Bash-scripting

Guzikova Anastasia

Содержание

1 Introduction

1.1 Shell Programming

bash(абревиатура Bourne again shell, каламбур Born again shell) — усовершенствованная вариация командной оболочки Bourne shell. Одна из наиболее популярных командных оболочек UNIX. Особенно популярна в среде Linux, где она часто используется в качестве предустановленной командной оболочки.
Bash это командный процессор с текстовым интерфейсом, который позволяет пользователю вводить команды, вызывающие определенные действия. Ключевые слова, синтаксис и другие основные особенности языка были скопированы с sh.
sh является удобным и часто используемым интерпретируемым языком программирования. Он содержит стандартные конструкции для циклов, ветвления, объявления функций и т. п. Данный язык часто используется в UNIX-подобных системах при создании различных сценариев (скриптов) работы.
Большинство принципов программирования на BASH одинаково хорошо применимы и в других командных оболочках, таких как Korn Shell (ksh), от которой Bash позаимствовал некоторые особенности, и C Shell (особенность на заточенность на C-подобный синтаксис) и его производных, и многих других оболочках.

Знание языка командной оболочки является залогом успешного решения задач администрирования системы. Даже если вы не предполагаете заниматься написанием своих сценариев, во время загрузки Linux выполняется целый ряд сценариев из /etc/rc.d, которые настраивают конфигурацию операционной системы и запускают различные сервисы, поэтому очень важно четко понимать эти скрипты и иметь достаточно знаний, чтобы вносить в них какие либо изменения.
Язык сценариев легок в изучении, в нем не так много специфических операторов и конструкций. Синтаксис языка достаточно прост и прямолинеен, он очень напоминает команды, которые приходится вводить в командной строке. Короткие скрипты практически не нуждаются в отладке, и даже отладка больших скриптов отнимает весьма незначительное время.
Есть ряд задач, которые нельзя решить с помощью shell-скриптинга:

- для ресурсоемких задач, особенно когда важна скорость исполнения
- для задач, связанных с выполнением математических вычислений, особенно это касается вычислений с плавающей запятой, вычислений с повышенной точностью, комплексных чисел
- для кросс-платформенного программирования (для этого лучше подходит язык C)
- для задач, работающих с многомерными массивами, связанными списками или деревьями
- когда необходимо предоставить графический интерфейс с пользователем (GUI)
- для проприетарных, "закрытых"программ (скрипты представляют из себя исходные тексты программ, доступные для всеобщего обозрения)

Название BASH - это аббревиатура от "Bourne-Again Shell"и игра слов от, ставшего уже классикой, "Bourne Shell"Стефена Бурна (Stephen Bourne). В последние годы BASH достиг такой популярности, что стал стандартной командной оболочкой de facto для многих разновидностей UNIX. Большинство принципов программирования на BASH одинаково хорошо применимы и в других командных оболочках, таких как Korn Shell (ksh), от которой Bash позаимствовал некоторые особенности, и C Shell и его производных.

1.2 Sha-Bang

В простейшем случае, скрипт - это ни что иное, как простой список команд системы, записанный в файл.
Создание скриптов помогает сохранить время и силы, которые тратятся на ввод последовательности команд всякий раз, когда необходимо их выполнить.

Например:

```
#!/bin/bash

gmake -j superclean
gmake -j4 vace &>../build_vace.log && \
gmake -j4 vace_anm &>../build_vace_anm.log
```

Здесь нет ничего необычного, это простая последовательность команд, которая может быть набрана в командной строке (с консоли или в xterm). Преимущество размещения последовательности команд в скрипте состоит в том, что вам не придется всякий раз набирать эту последовательность вручную. Кроме того, скрипты легко могут быть модифицированы или обобщены для разных применений.
Если файл сценария начинается с последовательности #!, которая в мире UNIX называется sha-bang, то это указывает системе какой интерпретатор следует использовать для исполнения сценария. Это двухбайтовая последовательность (или четырёхбайтовая: **некоторые разновидности UNIX** (основанные на 4.2BSD) **требуют, чтобы эта последовательность состояла из 4-х байт, за счет добавления пробела после !**, #! /bin/sh) - специальный маркер, определяющий тип сценария, в данном случае - сценарий командной оболочки bash. Более точно, sha-bang определяет интерпретатор, который вызывается для исполнения сценария, это может быть командная оболочка (shell), иной интерпретатор или утилита.

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/usr/awk -f
```

Каждая, из приведенных выше сигнатур, приводит к вызову различных интерпретаторов, будь то /bin/sh - командный интерпретатор по-умолчанию (bash для Linux-систем), либо иной.
(Можно запустить файл README с сигнатурой #!/bin/more)
При переносе сценариев с сигнатурой #!/bin/sh на другие UNIX системы, где в качестве командного интерпретатора задан другой shell, вы можете лишиться некоторых особенностей, присущих bash.
В shell-скриптах последовательность #! должна стоять самой первой и задает интерпретатор (sh или bash). Интерпретатор, в свою очередь, воспринимает эту строку как комментарий, поскольку она начинается с символа #.
Если в сценарии имеются еще такие же строки, то они воспринимаются как обычный комментарий. Т.е. вторая такая строка уже не означает запуск нового сценария.

В первой строке сценария ("sha-bang") можно указать команду env, если путь к командному интерпретатору не известен.

```
#! /usr/bin/env perl
```

env *Запускает указанную программу или сценарий с модифицированными переменными окружения (не изменяя среду системы в целом, изменения касаются только окружения запускаемой программы/сценария). Посредством [varname=xxx], устанавливает значение переменной окружения varname,*

которая будет доступна из запускаемой программы/сценария. Без параметров – просто выводит список переменных окружения с их значениями.

1.3 Invoking the script

Запустить сценарий можно командой sh scriptname или bash scriptname. (Не рекомендуется запуск сценария командой sh <scriptname>, поскольку это запрещает использование устройства стандартного ввода stdin в скрипте; вызов Bash-скрипта с помощью команды sh scriptname отключает специфичные для Bash расширения, что может привести к появлению ошибки и аварийному завершению работы сценария). Более удобный вариант - сделать файл скрипта исполняемым (а также читаемым, поскольку shell должен иметь возможность прочитать файл), командой chmod.

```
chmod +rx scriptname
./scriptname
```

Почему бы не запустить сценарий просто набрав название файла scriptname, если сценарий находится в текущем каталоге? Дело в том, что из соображений безопасности, путь к текущему каталогу "."не включен в переменную окружения $PATH. Поэтому необходимо явно указывать путь к текущему каталогу, в котором находится сценарий, т.е. ./scriptname.

2 Variables

Переменные – это одна из основ любого языка программирования. Они участвуют в арифметических операциях, в синтаксическом анализе строк и совершенно необходимы для абстрагирования каких либо величин с помощью символических имен. **Физически переменные представляют собой ни что иное как участки памяти, в которые записана некоторая информация.**
Необходимо всегда помнить о различиях между именем переменной и ее значением. Если variable1 – это имя переменной, то $variable1 – это ссылка на ее значение. "Чистые"имена переменных, без префикса $, могут использоваться только при объявлении переменный, при присваивании переменной некоторого значения, при удалении (сбросе), при экспорте и в особых случаях – когда переменная представляет собой название сигнала. Присваивание может производится с помощью символа =, инструкцией read и в заголовке цикла.

```
variable1=abacaba
echo $variable1 ${variable1}
```

Примечательно, что написание $variable фактически является упрощенной формой написания $variable. Более строгая форма записи ${variable} может с успехом использоваться в тех случаях, когда применение упрощенной формы записи порождает сообщения о синтаксических ошибках.
В отличие от большинства других языков программирования, Bash не производит разделения переменных по "типам". По сути, переменные Bash являются строковыми переменными, но, в зависимости от контекста, Bash допускает целочисленную арифметику с переменными. Определяющим фактором здесь служит содержимое переменных.

```
#!/bin/bash
a=2334                    #integer
let "a += 1"
echo "a = $a "            # a = 2335

b=${a/23/BB}
echo "b = $b"             # b = BB35
declare -i b
echo "b = $b"             # b = BB35
let "b += 1"              # BB35 + 1 =
echo "b = $b"             # b = 1

c=BB34
echo "c = $c"             # c = BB34
d=${c/BB/23}
echo "d = $d"             # d = 2334
let "d += 1"              # 2334 + 1 =
echo "d = $d"             # d = 2335

e=""
echo "e = $e"             # e =
let "e += 1"
echo "e = $e"             # e = 1

echo "f = $f"             # f =
let "f += 1"              #
echo "f = $f"             # f = 1
```

●let - встроенная команда для целочисленной арифметики.

3 Special characters

Служебные символы, используемые в текстах сценариев.

1. #

   - комментарий
   - в операциях подстановки параметров ${#var} – длина строки var
     ${#*} и ${#@} – количество входных параметров
   - в константных числовых выражениях: $((2#101011)) – число в двоичной системе счисления
   - в поиске по шаблону ${var#0} – удалить ведущий ноль из числа, если он есть

   Кавычки и обратный слэш экранируют действие символа #.

2. ; Разделитель команд

3. ;; Ограничитель в операторе выбора case

4. .

   - Команда ".". Эквивалент команды source. (Встроенная команда bash.)

     ```
     . filename
     ```

     В общем случае при запуске скрипта запускается новый процесс. Для того, чтобы выполнить скрипт внутри текущей сессии bash, необходимо использовать команду source, синонимом которой является просто точка ".". Скрипт оболочки служит просто аргументом этой команды. Эта команда читает и выполняет команды из файла с именем filename в текущем окружении и возвращает статус, определяемый последней командой из файла filename. Если filename не содержит слэша, то пути, перечисленные в переменной PATH, используются для поиска файла с именем filename. Этот файл не обязан быть исполняемым.

     ```
     . filename
     ```

   - Часть имени файла. (Если начинается - скрытый файл)

     ```
     .hidden
     ```

   - Имя каталога. (одна - текущая директория, две - уровнем выше)

     ```
     cp ../file .
     ```

   - Одиночный символ при поиске по шаблону, в регулярных выражениях.

5. " Двойные кавычки (частичные, нестрогие). В строке STRING, ограниченной двойными кавычками не выполняется интерпретация большинства специальных символов, которые могут находиться в строке. Например, символ шаблона *.

   ```
   bash$ ls [Ff]*
   File  file.txt

   bash$ ls "[Ff]*"
   ls: [Ff]*: No such file or directory
   ```

   Вообще, желательно использовать двойные кавычки при обращении к переменным. Это предотвратит интерпретацию специальных символов, которые могут содержаться в именах переменных, **за исключением $, ‘ (обратная кавычка) и (escape - обратный слэш)**. То, что символ $ попал в разряд исключений, позволяет выполнять обращение к переменным внутри строк, ограниченных двойными кавычками ("$variable"), т.е. выполнять подстановку значений переменных. Двойные кавычки могут быть использованы для предотвращения разбиения строки на слова. Заключение строки в кавычки приводит к тому, что она передается как один аргумент, даже если она содержит пробельные символы - разделители.

6. ' Одинарные кавычки (полные, строгие) - экранируют все служебные символы в строке. Это более строгая форма экранирования.
   Одиночные кавычки схожи по своему действию с двойными кавычками, только не допускают обращение к переменным, поскольку специальный символ "$"внутри одинарных кавычек воспринимается как обычный символ. Внутри одиночных кавычек, любой специальный символ, за исключением ', интерпретируется как простой символ. Одиночные кавычки ("строгие, или полные кавычки") следует рассматривать как более строгий вариант чем двойные кавычки ("нестрогие, или неполные кавычки").

   ```
   bash$ echo '$var'
   $var
   ```

7. ` Обратные кавычки. Подстановка команд. Обратные кавычки могут использоваться для записи в переменную команды `command`.

   ```
   i=`expr $i + 1`
   ```

8. : Пустая команда, всегда возвращает true.

   ```
   if condition
   then :
   else
      take-some-action
   fi

   bash$ : ${username=`whoami`}
   bash$ ${username=`whoami`}
   aguzikova: command not found

   while : # while true
    do
      commands
      if condition
      then
        break
      fi
    done
   ```

9. !

   - инверсия (или логическое отрицание) используемое в условных операторах. Оператор ! инвертирует код завершения команды, к которой он применен. Так же используется для логического отрицания в операциях сравнения, например, операция сравнения "равно"(=), при использовании оператора отрицания, преобразуется в операцию сравнения – "не равно"(!=). Символ ! является зарезервированным ключевым словом BASH.
   - В некоторых случаях символ используется для косвенного обращения к переменным с версии 2

     ```
     $${var}
     ${!var}
     ```

   - Кроме того, из командной строки оператор ! запускает механизм историй Bash. Примечательно, что этот механизм недоступен из сценариев (т.е. работает исключительно из командной строки).

10. *

- В регулярных выражениях - любое количество символов
- шаблон для подстановки в имена файлов (echo *)
- арифметический оператор умножения,возведения в степень (**)

11. **?**

- В регулярных выражениях, при подстановке в имена файлов - одиночный символ
- В выражениях с подстановкой параметра, символ ? проверяет – установлена ли переменная.
- Трехместный оператор в стиле языка C (в конструкциях с двойными скобками). ((t = a<45?7:11))

12. **$**

- $ - подстановка переменной; конец строки в регулярных выражениях
- ${} - подстановка параметра
- $*, $@ - параметры командной строки (одной строкой или каждый параметр в отдельной строке)
- $? - хранит код завершения последней выполненной команды, функции или сценария
- $$ - хранит id процесса сценария

13. **()** группа команд - исполняются в дочернем процессе subshell'e

15. **{}**

- {x,y,z} Команда интерпретируется как список команд, разделенных точкой с запятой, с вариациями, представленными в фигурных скобках. При интерпретации имен файлов (подстановка) используются параметры, заключенные в фигурные скобки.
  find *.{ko,klm}
- Блок кода (вложенный блок). Фактически создаёт анонимную функцию, но в отличие от обычных функций, переменные, создаваемые во вложенных блоках кода, доступны объёмлющему сценарию.
  Может выполнять перенаправление ввода-вывода.
  Не вызывает запуск дочернего процесса.
- {} \;

  pathname - полное имя файла - часто совместно с find

16. **[]**

  - команда [- test (Проверка истинности выражения, заключенного в квадратные скобки [ ]. Примечательно, что [является частью встроенной команды test (и ее синонимом), И не имеет никакого отношения к "внешней"утилите /usr/bin/test.)

```
[ -d "$dir" ] || echo "directory $dir not found."
[ "$var1" -ne "$var2" ] && echo "$var1 is not equal to $var2"
```

- **[[ ]]** - Проверка истинности выражения, заключенного между [[ ]] (зарезервированное слово интерпретатора), расширенный вариант команды test. Появился с версии 2.02.
  Конструкция [[...]] более предпочтительна, нежели [...], поскольку поможет избежать некоторых логических ошибок. Например, операторы &&, ||, < и > внутри [[ ]] вполне допустимы, в то время как внутри [ ] порождают сообщения об ошибках.
- диапазон символов

17. (()) Вычисляется целочисленное выражение, заключенное между двойными круглыми скобками (( )). Если результатом вычислений является ноль, то возвращается 1, или "ложь". Ненулевой результат дает код возврата 0, или "истина". То есть полная противоположность инструкциям test и [ ].

    ```
    a=$(( 5 + 3 ))
    ```

18. | Конвейер. Передает вывод предыдущей команды на ввод следующей или на вход командного интерпретатора shell. Этот метод часто используется для связывания последовательности команд в единую цепочку.
    Конвейеры (еще их называют каналами) – это классический способ взаимодействия процессов, с помощью которого stdout одного процесса перенаправляется на stdin другого. Обычно используется совместно с командами вывода, такими как cat или echo, от которых поток данных поступает в "фильтр"(команда, которая на входе получает данные, преобразует их и обрабатывает).
    Конвейер исполняется в дочернем процессе, а посему – не имеет доступа к переменным сценария.
    Если одна из команд в конвейере завершается аварийно, то это приводит к аварийному завершению работы всего конвейера.

19. & Выполнение задачи в фоне.

    jobs [-l|-p]

    выводит статусы актуальных заданий текущей сессии (выполняющихся в фоне и остановленных), где
    p - показывать только id процесса
    l - показывать всю информацию по каждому заданию

    fg [job_id]

    переводит задачу выполняться в активном режиме

    bg [job_id]

    переводит задачу выполняться в фоновом режиме

20. ^ Начало строки в регулярных выражениях

21. >, < Перенаправление ввода/вывода
    В системе по умолчанию всегда открыты три файла (клавиатура, экран, вывод сообщений об ошибках на экран). Эти и любые другие открытые файлы могут быть перенаправлены. С каждым открытым файлом связан дескриптор файла.

    ```
    0 - stdin
    1 - stdout
    2 - stderr

    [1]>filename
    [1]>>filename
    2>filename
    2>>filename
    &>filename
    i>&j
    < filename
    [j]<>filename
    n<&-
    ```

n>&–
0<&–, <&–
1>&–, >&–

```
: > filename
```

Операция > усекает файл "filename"до нулевой длины. Если до выполнения операции файла не существовало, то создается новый файл с нулевой длиной (тот же эффект дает команда 'touch'). Символ : выступает здесь в роли местозаполнителя, не выводя ничего.
Дочерние процессы наследуют дескрипторы открытых файлов. По этой причине и работают конвейеры. Чтобы предотвратить наследование дескрипторов – закройте их перед запуском дочернего процесса.

```
exec 3>&1
ls -l 2>&1 >&3 3>&- | grep bad 3>&-
exec 3>&-
```

– В конвейер передается только stderr.
Команда exec <filename перенаправляет ввод со stdin на файл. С этого момента весь ввод, вместо stdin (обычно это клавиатура), будет производиться из этого файла.

```
exec 6<&0
exec < data-file
exec <&6 6<&-
```

– Связать дескр. 6 со стандартным вводом (stdin). Сохраняя stdin.
– stdin заменяется файлом "data-file"
– Восстанавливается stdin из дескриптора 6, где он был предварительно сохранён, и дескриптор 6 закрывается, освобождая его для других процессов.

```
exec 6>&1
exec > $LOGFILE
exec 1>&6 6>&-
```

Встроенный документ (here document) является **специальной формой перенаправления** ввода/вывода, которая позволяет передать список команд интерактивной программе или команде (ftp, telnet).
Конец встроенного документа выделяется "строкой-ограничителем которая задается с помощью специальной последовательности символов <<.
В качестве строки-ограничителя должна выбираться такая последовательность символов, которая не будет встречаться в теле "встроенного документа".
Использование встроенных документов может иногда с успехом применяться и при работе с неинтерактивными командами и утилитами.
Пример. Вывод многострочных сообщений с помощью cat (например, для встроенной справки к сценарию)

```
cat <<End-of-message
        ==========
        Text: $text
        ==========
End-of-message
```

Символ '-' начинающий строку-ограничитель встроенного документа подавляет вывод символов табуляции, которые могут встречаться в теле документа, но не пробелов.
Подстановка параметров не производится, если строка ограничитель заключена в кавычки или экранирована.
Блочный комментарий:

```
: << COMMENTBLOCK
&*@!!++=
COMMENTBLOCK
```

4 Assigning Values to Variables

= оператор присваивания.
Помнить: пробельные символы до и после оператора недопустимы
Виды присваиваний:

1. простое

   ```
   a=78901
   ```

2. с помощью ключевого слова let

   ```
   let a=19+123
   ```

3. неявное присваивание в заголовке цикла

   ```
   for a in 7 8 9 11
   do
     echo -n "$a "
   done
   ```

4. при использовании инструкции read

   ```
   read a
   ```

5. подстановка команд

   ```
   a=`ls -l`
   echo $a
   echo "$a"
   ```

   Без кавычек - удаляются лишние пробелы и пустые строки
   С кавычками - сохраняются

6. использование конструкции $() вместо обратных кавычек

   ```
   arch=$(uname -m)
   ```

5 Special Variable Types

1. Локальные - переменные, область видимости которых ограничена блоком кода или телом функции

2. Переменные окружения - переменные, которые затрагивают командную оболочку и порядок взаимодействия с пользователем
   В более общем контексте, каждый процесс имеет некоторое "окружение"(среду исполнения), т.е. набор переменных, к которым процесс может обращаться за получением определенной информации. В этом смысле командная оболочка подобна любому другому процессу.
   Каждый раз, когда запускается командный интерпретатор, для него создаются переменные, соответствующие переменным окружения. Изменение переменных или добавление новых переменных окружения заставляет оболочку обновить свои переменные, и все дочерние процессы (и команды, исполняемые ею) наследуют это окружение.
   Если сценарий изменяет переменные окружения, то они должны "экспортироваться т.е передаваться окружению, локальному по отношению к сценарию. Эта функция возложена на команду export.
   Сценарий может экспортировать переменные только дочернему процессу, т.е. командам и процессам запускаемым из данного сценария. Сценарий, запускаемый из командной строки не может экспортировать переменные "наверх"командной оболочке. Дочерний процесс не может экспортировать переменные родительскому процессу.

3. Позиционные параметры - аргументы, передаваемые скрипту из командной строки

   $0 , $1 , $2 , $3 . . .

   где $0 - это название файла сценария, $1 – это первый аргумент, $2 – второй, $3 – третий и так далее. Аргументы, следующие за $9, должны заключаться в фигурные скобки, например:

   ${10}, ${11}, ${12}

   Количество аргументов:

   ```
   args=$#
   ```

   Последний с помощью косвенной адресации:

   ```
   lastarg=${!args}
   ```

   Для перебора всех аргументов командной строки (так же, как и входных аргументов функции) можно пользоваться командой shift.

   ```
   echo "$@"     # 1 2 3 4 5
   shift
   echo "$@"     # 2 3 4 5
   shift
   echo "$@"     # 3 4 5
   ```

Пример.
Код script.sh:

```
#!/bin/bash
############################
# script.sh
############################
change_name() {
        NAME="$1"
```

```
}
print_name() {
        echo "`basename $0`: $NAME"
}

print_name $NAME # ""
change_name name2
print_name $NAME # name2

change_name name3
print_name $NAME # name3
./script2.sh     # "", name4
print_name $NAME # name3

export NAME=name5
./script2.sh # name5, name4
print_name $NAME # name5
```

Код script2.sh:

```
#!/bin/bash
############################
# script2.sh
############################
echo "`basename $0`: $NAME"
NAME=name4
echo "`basename $0`: $NAME"
```

Вывод работы ./script.sh:

```
script.sh:
script.sh: name2
script.sh: name3
script2.sh:
script2.sh: name4
script.sh: name3
script2.sh: name5
script2.sh: name4
script.sh: name5
```

6 Debugging Bash scripts

Отладка скриптов Bash

6.1 Debugging on the entire script

Отладка сразу всего скрипта
Когда дела идут не так, как планировалось, необходимо определить, из-за чего в скрипте возникли проблемы. В Bash для отладки предоставляются широкие возможности. Наиболее распространенным способом является запуск подоболочки с параметром -x, благодаря которому весь скрипт будет запущен в отладочном режиме. После того, как для каждой команды будут выполнены все необходимые подстановки и замены, но перед тем, как команда будет выполнена, в стандартный выходной поток будет выдана трассировка команды и все ее аргументы.

```
#!/bin/bash
echo 'The script is starting...'
echo "Hello, $USERNAME!"
echo "List of connected users:"
w
echo 'I'\''m setting variable now.'
variable=variable_value
echo "variable: $variable"
echo -e "Enter new value for variable: \c"
read variable
echo "variable: $variable"
```

Вывод:

```
bash$ /bin/bash -x script1.sh
+ echo 'The script is starting...'
The script is starting...
+ echo 'Hello, aguzikova!'
Hello, aguzikova!
+ echo 'List of connected users:'
List of connected users:
+ w
 14:52:51 up 28 days,  2:00,  4 users,  load average: 0.88, 0.77, 0.61
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU WHAT
aguzikov pts/0    :1               Mon20   18:36m  1.10s  0.57s ssh -o ConnectT
aguzikov pts/1    :1               Mon20   42:40   0.76s  0.76s bash
aguzikov pts/2    :1.0             14:14   26:28  24.96s 24.44s texmaker
aguzikov pts/3    :1.0             14:41    1.00s  3.45s  0.04s w
+ echo 'I'\''m setting variable now.'
I'm setting variable now.
+ variable=variable_value
+ echo 'variable: variable_value'
variable: variable_value
+ echo -e 'Enter new value for variable: \c'
Enter new value for variable: + read variable
new
+ echo 'variable: new'
variable: new
```

6.2 Debugging on part(s) of the script

Отладка скрипта по частям
С помощью встроенной команды set, имеющейся в Bash, вы можете запускать в обычном режиме те части скрипта, относительно которых вы уверены, что они работают без ошибок, и отображать отладочную информацию только там, где есть подозрение на неправильную работу. Скажем, мы не уверены, что в примере commented-script1.sh будет делать команда w, поэтому мы можем окружить эту команду следующими отладочными командами:

```
set -x # activate debugging from here
w
set +x # stop debugging from here

The script is starting...
Hello, aguzikova!
List of connected users:
+ w
 14:58:16 up 28 days,  2:05,  4 users,  load average: 1.96, 1.34, 0.89
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
aguzikov pts/0    :1               Mon20   18:41m  1.10s  0.57s ssh -o ConnectT
aguzikov pts/1    :1               Mon20   48:05   0.76s  0.76s bash
aguzikov pts/2    :1.0             14:14   31:53  36.19s 35.67s texmaker
aguzikov pts/3    :1.0             14:41    0.00s  3.52s  0.04s w
+ set +x
I'm setting variable now.
variable: variable_value
Enter new value for variable: new
variable: new
```

Использование других опций:

```
bash$ ls
script1.sh
bash$ set -v
bash$ ls
ls
script1.sh
bash$ set +v
set +v
bash$ ls *
script1.sh
bash$ set -f
bash$ ls *
ls: cannot access *: No such file or directory
bash$ touch *
bash$ ls
*  script1.sh
bash$ rm *
bash$ ls
script1.sh
bash$ set +f
```

При опции -f: отключается генерация имени файла с помощью метасимволов (подстановка).
При активированной опции -v: командная оболочка печатает входные строки сразу, как они считываются.

```
#!/bin/bash -xv
```

7 Tests

Оператор if (условного перехода).

```
if [ condition1 ]
then
    command1
    command2
    command3
elif [ condition2 ] # else if
then
    command4
    command5
else
    default–command
fi
```

Строго говоря, скобки [...] необязательны:
Инструкция "if COMMAND"возвращает код возврата команды COMMAND.

```
if cd "$dir" 2>/dev/null; then
  echo "changed directory to $dir"
else
  echo "can't change directory to $dir."
fi
```

"2>/dev/null"подавление вывода сообщений об ошибках.

Рассмотрим, что есть истина для конструкции if/then:

```
if [ 0 ]          # true
if [ 1 ]          # false
if [ -1 ]         # true
if [ ]            # false (NULL)
if [ abc ]        # true  (string)
if [ $x ]         # true if x is defined
if [ -n "$x" ] # true if x is not empty
if [ "false" ] # true  (just string)
```

Операции проверки файлов

Возвращает **true** если...
-e файл существует
-f обычный файл (не каталог и не файл устройства)
-s ненулевой размер файла
-d файл является каталогом
-b файл является блочным устройством (floppy, cdrom и т.п.)
-c файл является символьным устройством (клавиатура, модем, звуковая карта и т.п.)
-p файл является каналом
-h файл является символической ссылкой
-L файл является символической ссылкой
-S файл является сокетом
-t файл (дескриптор) связан с терминальным устройством
Этот ключ может использоваться для проверки – является ли файл стандартным устройством ввода stdin ([-t 0]) или стандартным устройством вывода stdout ([-t 1]).
-r, -w, -x файл доступен для чтения/записи/исполнения (пользователю, запустившему сценарий)
-O вы являетесь владельцем файла

-G вы принадлежите к той же группе, что и файл
-N файл был модифицирован с момента последнего чтения
f1 -nt f2 файл f1 более новый, чем f2
f1 -ot f2 файл f1 более старый, чем f2
f1 -ef f2 файлы f1 и f2 являются "жесткими"ссылками на один и тот же файл
! "НЕ логическое отрицание (инверсия) результатов всех вышеприведенных проверок (возвращается true если условие отсутствует).

Пример.

Проверка, является ли файл битой ссылкой:

```
[ -h "$file" -a ! -e "$file" ] && echo "broken"
```

Операции сравнения

1. сравнение целых чисел

   -eq , -ne , -gt , -ge , -lt , -le
   `if [ "$a" -eq "$b" ]`

   или внутри двойных круглых скобок

   =, !=, >, >=, <, <=

2. сравнение строк =, ==, !=

   ```
   if [[ "$a" < "$b" ]]
   if [ "$a" \< "$b" ]
   ```

   <, > - в смысле величины ASCII-кодов, нужно экранировать в [...]

   `[[ $a == z* ]]`

   - истина, если $a начинается с символа "z"(сравнение по шаблону)

   `[[ $a == "z*" ]]`

   - истина, если $a равна строго содержимому

   `[ $a == z* ]`

   - имеют место подстановка имен файлов и разбиение на слова

   `[ "$a" == "z*" ]`

   - истина, если $a равна z

   ```
   if [ -z "$a" ] # empty
   if [ -n "$a" ] # not empty
   ```

   Всегда заключайте проверяемую строку в кавычки.

Построение сложных условий проверки:

```
if [ "$exp1" -a "$exp2" ]
if [ "$exp1" -o "$exp2" ]
[[ condition1 && condition2 ]]
[[ condition1 || condition2 ]]
```

8 Strings

Длина строки:

```
${#string}
expr length $string
expr "$string" : ".*"
expr match "$string" "$substring"
expr "$string" : "$substring"
```

Извлечение подстроки с position до конца

```
${string:position}
```

с position длиной length символов:

```
${string:position:length}
expr substr $string $position $length
```

последние length символов:

```
${string: -length}
```

Найти и извлечь первое совпадение $substring в $string:

```
expr match "$string" '\($substring\)'
expr "$string" : '\($substring\)'
```

Найти и извлечь первое совпадение $substring в $string (поиск с конца):

```
expr match "$string" '.*\($substring\)'
expr "$string" : '.*\($substring\)'
```

Удаление части строки самой короткой из найдённых с начала:

```
${string#substring}
```

самой длинной из найденных с начала:

```
${string##substring}
```

самой короткой из найдённых с конца:

```
${string%substring}
```

самой длинной из найденных с конца:

```
${string%%substring}
```

Замена подстроки первое вхождение

```
${string/substring/replacement}
```

все вхождения

```
${string//substring/replacement}
```

Если в переменной var найдено совпадение с Pattern, причем совпадающая подстрока расположена в начале строки (префикс), то оно заменяется на Replacement. Поиск ведется с начала строки:

```
${var/#Pattern/Replacement}
```

Если в переменной var найдено совпадение с Pattern, причем совпадающая подстрока расположена в конце строки (суффикс), то оно заменяется на Replacement. Поиск ведется с конца строки

```
${var/%Pattern/Replacement}
```

Пример:

```
a="12345"; echo "${a}"; echo "${a:3}"; echo "${a#12}"; echo "${a/12/21}"
```

Расширение файла:

```
${filename##*.}
```

9 Parameter substitution

Подстановка параметров

`${parameter}`

тоже самое, что значение переменной parameter. Иногда можно использовать только этот вариант.

`${parameter-default}, ${parameter:-default}`

если параметр отсутствует, то используется значение по умолчанию.

`echo ${username:-`whoami`}`

Дополнительный символ : имеет значение только тогда, когда parameter определен, но имеет "пустое"(null) значение (используется дефолтное).

`${parameter=default}, ${parameter:=default}`

если параметр отсутствует, то параметр принимает значение по умолчанию.

`echo ${username:=`whoami`}`

Дополнительный символ : имеет значение только тогда, когда parameter определен, но имеет "пустое"(null) значение (принимает дефолтное).

`${parameter+alt_value}, ${parameter:+alt_value}`

если параметр имеет какое-либо значение, то используется альтернативное, иначе пустая строка.

`${parameter?err_msg}, ${parameter:?err_msg}`

если параметр инициализирован, то используется его значение, в противном случае выводится сообщение об ошибке.

10 Loops

10.1 for

```
for arg in [list]
do
 command(s)...
done
```

Вывести от одного до пяти:

```
for i in 1 2 3 4 5; do
        echo $i
done
for i in `seq 5`; do
        echo $i
done
```

Пройти по списку файлов и вывести, что файл не существует, если он не существует, или статистику по файлу, если он существует:

```
files="text.txt
script.sh
output.log"
for file in $files
do
  if [ ! -e $file ]; then
        echo "$file doesn't exist"
        continue
  fi
  stat $file
done
```

если взять список в кавычки - примет за один аргумент.

C-style for loop (C-подобной нотация)

```
for ((a=1, b=LIMIT; a <= LIMIT ; a++, b--))
do
  echo "a=$a; b=$b "
done
```

Bash, version 3+.

```
for a in {1..10}
do
  echo -n "$a "
done
```

Распечатать параметры командной строки:

```
for params
do
        echo $params
done
```

Пример. Список символических ссылок в каталоге

```
OUTFILE=symlinks.list
directory=${1-`pwd`}
```

```
for file in "$( find $directory -type l )"
do
  echo "$file"
done | sort >> "$OUTFILE"
```

10.2 while

```
while [ condition ]
do
 command(s)...
done

var0=0
LIMIT=10

while [ "$var0" -lt "$LIMIT" ]
#       ^                    ^
# Spaces, because these are "test-brackets" . . .
do
  echo -n "$var0 "          # -n suppresses newline.
  #                ^          Space, to separate printed out numbers.

  var0=`expr $var0 + 1`     # var0=$(($var0+1))  also works.
                            # var0=$((var0 + 1)) also works.
                            # let "var0 += 1"    also works.
done                        # Various other methods also work.

while read line
do
echo $line
done < files.txt
```

C-style syntax in a while loop

```
LIMIT=10                    # 10 iterations.
a=1

while [ "$a" -le $LIMIT ]
do
  echo -n "$a "
  let "a+=1"
done                        # No surprises, so far.

echo; echo

# +=================================================================+

# Now, we'll repeat with C-like syntax.

((a = 1, LIMIT=10))         # a=1
# Double parentheses permit space when setting a variable, as in C.

while (( a <= LIMIT ))      #  Double parentheses,
do                          #+ and no "$" preceding variables.
  echo -n "$a "
```

```
  ((a += 1))              # let "a+=1"
  # Yes, indeed.
  # Double parentheses permit incrementing a variable with C-like syntax.
done

while [ $# -ne 0 ]; do
        echo "$1"
        shift
done
```

10.3 until

```
until [condition-is-true]
do
 command...
done

until [ "$var1" = end ]
do
  read var1
done
```

Для управления ходом выполнения цикла служат команды break и continue и точно соответствуют своим аналогам в других языках программирования. Команда break прерывает исполнение цикла, в то время как continue передает управление в начало цикло, минуя все последующие команды в теле цикла.
Команде break может быть передан необязательный параметр. Команда break без параметра прерывает тот цикл, в который она вставлена, а break N прерывает цикл, стоящий на N уровней выше (1 - текущий).
Команда continue, как и команда break, может иметь необязательный параметр. В простейшем случае, команда continue передает управление в начало текущего цикла, а команда continue N прерывает исполнение текущего цикла и передает управление в начало внешнего цикла, отстоящего от текущего на N уровней. (1 - текущий)

```
for i in `seq 0 9`; do
 for j in `seq 10`; do
        if [ `expr $i \* 10 + $j` -ge "$LIM" ]; then
                break 2
        else
                echo $i,$j
        fi
 done
done

for outer in I II III IV V ; do
  echo -n "Group␣$outer:␣"
  for inner in 1 2 3 4 5 6 7 8 9 10 ; do
    if [[ "$inner" -eq 7 && "$outer" = "III" ]]
    then
      continue 2
    fi
    echo -n "$inner␣"  # 7 8 9 10 will not echo on "Group III."
  done
done
```

11 Case and select

Операторы выбора

1. case

```
case "$variable" in

 "$condition1" )
 command...
 ;;

 "$condition2" )
 command...
 ;;
esac
```

```
echo "Do you wish to proceed? [y..n] \c"
while read ANS; do
        case $ANS in
        [yY]*)
                break;
        ;;
        [nN]*|exit)
                exit 0
        ;;
        *)
                echo "try again: yes, no, exit"
        esac
done

echo "Hello, $USER!"
```

Заключать переменные в кавычки необязательно, поскольку здесь не производится разбиения на отдельные слова.
Каждая строка с условием должна завершаться правой (закрывающей) круглой скобкой).
Каждый блок команд, отрабатывающих по заданному условию, должен завершаться двумя символами точка-с-запятой ;;.
Блок case должен завершаться ключевым словом esac (case записанное в обратном порядке).

2. select

```
select variable [in list]
do
 command...
 break
done
```

Можно использовать для создания меню:

```
select color in yellow white black red
do
echo "Your color - $color"
break
done
```

Вывод:

```
1) yellow
2) white
3) black
4) red
#? 3
Your color - black
```

12 Functions

Интерпретатор команд shell позволяет группировать наборы команд или конструкций, создавая повторно используемые блоки. Подобные блоки называются shell-функциями.
Функция состоит из двух частей: метки и тела.
В качестве метки выступает имя функции; тело функции создаёт набор команд, составляющих саму функцию. Имя функции должно быть уникальным.

```
[function] function_name() {
        command(s)
}
```

Функции могут быть размещены в том же самом файле, что и сценарии, либо в отдельном файле.

```
bash$ cat functions.file
hello() {
        echo "Hello, $USERNAME!"
}
square() {
        _number=`echo $1 | sed -n '/^[0-9]*$/p'`
        if [ -z "$_number" ]; then
                echo "`basename $0`: parameter must be integer"
                return 1
        fi
        expr $_number \* $_number
}
```

Подключение файла функций:

```
. functions.file
```

Чтобы вызвать функцию hello, нужно просто написать её имя.
Порядок передачи параметров функции аналогичен передаче параметров обычному сценарию.
Возврат значения функции:

1. дождаться пока функция не завершится сама с последующей передачей управления той части сценария, которая вызвала данную функцию

2. воспользоваться ключевым словом "return в результате будет осуществлена передача управления конструкции, которая расположена за оператором вызова функции. Можно указать необязательный числовой параметр (0 - означает отсутствие ошибок, 1 - наличие). Действие этого параметра аналогично действию кода завершения последней команды. При этом return без параметра использует код завершения последней команды для проверки состояния.

```
bash$ hello
Hello, aguzikova!
bash$ square
bash: parameter must be integer
bash$ square 12
144
```

Проверка загруженных функций:

```
bash$ set | grep "square ()"
square ()
```

Удаление shell-функций:

```
unset square
```

Есть готовые библиотеки функций для ряда задач.

Псевдонимы/Aliases

alias — встроенная команда интерпретаторов командной строки, позволяющее определять имена (сокращения) для команд, и их последовательностей. Также возможно переопределение команд и подстановка в них параметров. Обычно назначенные имена сохраняются только в течение сессии.

```
alias timestamp="date +\"%F_%H-%M-%S\""
alias ll="ls -l"
alias :q='exit'
```

Настройка конфигурации bash:

Схема настройки предусматривает наличие пары файлов

```
/etc/profile
/etc/bashrc
```

(для пользовательского шелла и просто интерактивного его экземпляра), а также соответствующих им пользовательских конфигов

```
~/.bash_profile
~/.bashrc
```

При авторизации первым в любом случае считывается •общесистемный профильный файл

```
/etc/profile
```

вслед за ним - **пользовательский профильный файл**

```
~/.bash_profile
```

после чего происходит обращение к

```
~/.bashrc
```

Файл

```
/etc/profile
```

может занимать особое положение - в него часто помещают переменные окружения (например, локальнозависимые), которые должны быть общими для всех пользователей данной системы. **Пользовательские же настройки** определяются в файлах

```
~/.bash_profile
~/.bashrc
```

Обычно в

```
~/.bash_profile
```

определяются переменные окружения, которые должны действовать для всех дочерних процессов, а в

```
~/.bashrc
```

параметры, всегда требуемые **в интерактивном режиме** (например, псевдонимы).

Мы можем добавить свои функции и алиасы в

```
~/.bashrc
```

файл и они будут доступны для каждой новой сессии bash.

13 Builtin commands

Внутренняя команда – это команда, которая **встроена непосредственно** в Bash. Команды делаются встроенными либо из соображений производительности – встроенные команды исполняются быстрее, чем внешние, которые, как правило, запускаются в дочернем процессе, либо из-за необходимости прямого доступа к внутренним структурам командного интерпретатора.
echo, read, cd, pwd, let, source - внутренние команды.

13.1 eval

Транслирует список аргументов, из списка, в команды.

```
eval arg1 [arg2] ... [argN]
y=`eval ls -l`
echo $y
```

– символы перевода строки не выводятся, поскольку имя переменной не в кавычках.

13.2 type

type [cmd]

Очень похожа на внешнюю команду which, type cmd выводит полный путь к "cmd". В отличие от which, type является внутренней командой Bash. С опцией -a не только различает ключевые слова и внутренние команды, но и определяет местоположение внешних команд с именами, идентичными внутренним.

```
bash$ type '['
[ is a shell builtin
bash$ type -a '['
[ is a shell builtin
[ is /usr/bin/[
```

13.3 getopts

Мощный инструмент, используемый для разбора аргументов, передаваемых сценарию из командной строки. Это встроенная команда Bash, но имеется и ее "внешний"аналог /usr/bin/getopt, а так же программистам, пишущим на C, хорошо знакома похожая библиотечная функция getopt. Она позволяет обрабатывать серии опций, объединенных в один аргумент и дополнительные аргументы, передаваемые сценарию.

```
scriptname -afv -c 2 /usr/local

while getopts ac:hfv opt; do
        case $opt in
        a) all=true
                echo "is ALL"
        ;;
        h) help=true
                echo "is HELP"
        ;;
        f) file=true
                echo "is FILE"
        ;;
        v) verbose=true
                echo "is VERBOSE"
        ;;
```

```
        c) count=$OPTARG
                echo "count is $count"
        ;;
        *)
                echo "`basename $0`: unknown option."
                exit 1
        ;;
        esac
done
```

13.4 $RANDOM

внутренняя функция Bash (не константа), которая возвращает псевдослучайные целые числа в диапазоне 0 - 32767.
Пример. Генерация случайного числа в диапазоне 1 - 25.

```
RANDOM=$$
rnumber=$((RANDOM%25+1))
```

в качестве начального числа выбирается PID процесса-сценария. Вполне допустимо взять в качестве начального числа результат работы команд 'time' или 'date'.

13.5 wait

Останавливает работу сценария до тех пор пока не будут завершены все фоновые задания или пока не будет завершено задание/процесс с указанным номером задания/PID процесса.
Возвращает код завершения указанного задания/процесса.
Вы можете использовать команду wait для предотвращения преждевременного завершения сценария до того, как завершит работу фоновое задание.

14 External commands

14.1 Common commands

14.1.1 cat

cat - это акроним от concatenate, выводит содержимое списка файлов на stdout. Для объединения файлов в один файл может использоваться в комбинации с операциями перенаправления (> или »).
Полезные ключи:
Ключ n, команды cat, вставляет порядковые номера строк в выходном файле. Ключ b - нумерут только непустые строки. Ключ v выводит непечатаемые символы в нотации с символом ^. Ключ s заменяет несколько пустых строк, идущих подряд, одной пустой строкой.

14.1.2 tac

- выводит содержимое файлов в обратном порядке, от последней строки к первой.

14.1.3 rev

выводит все строки файла задом наперед на stdout. Это не то же самое, что tac. Команда rev сохраняет порядок следования строк, но переворачивает каждую строку.

14.1.4 tee

Это оператор перенаправления, но с некоторыми особенностями. Подобно водопроводным трубам, "tee"позволяет "направить поток"данных в несколько файлов и на stdout одновременно, никак не влияя на сами данные. Эта команда может оказаться очень полезной при отладке.

```
cat listfile* | sort | tee check.file | uniq > result.file
```

14.1.5 find

Мощный инструмент для поиска файлов, каталогов. Рассмотрим примеры использования некоторых опций:

```
find $dirname -name "[a-z][0-9]*.txt" -o -name ".*"
find -perm 755
find . -name "bin" -prune  -o -print
find /var/adm -mtime +1
touch -t 04162140 dstamp; find -newer dstamp
find -type l

-exec COMMAND \;
find logs/ -size +100k -exec ls -lh {} \;
find logs/ ! -type d -size +100M -exec rm {} \;
find logs/ -name "*.log" -mtime +5 -ok rm {} \;
```

Для каждого найденного файла, соответствующего заданному шаблону поиска, выполняет команду COMMAND. Командная строка должна завершаться последовательностью символов (здесь символ ";"экранирован обратным слэшем, чтобы информировать командную оболочку о том, что символ ";"должен быть передан команде find как обычный символ). Если COMMAND содержит {}, то find подставляет полное имя найденного файла вместо "{}".

```
find . -name "*␣*" -exec rm -f {} \;
find build/defs/ -name "*.mk" -exec grep -Hns "x86_kernel_host" {} \;
```

14.1.6 xargs

Команда передачи аргументов указанной команде. Она разбивает поток аргументов на отдельные составляющие и поочередно передает их заданной команде для обработки. Эта команда может рассматриваться как мощная замена обратным одиничным кавычкам. Зачастую, когда команды, заключенные в обратные одиночные кавычки, завершаются с ошибкой too many arguments (слишком много аргументов), использование xargs позволяет обойти это ограничение. Обычно, xargs считывает список аргументов со стандартного устройства ввода stdin или из канала (конвейера), но может считывать информацию и из файла.
Если команда не задана, то по умолчанию выполняется echo. При передаче аргументов по конвейеру, xargs допускает наличие пробельных символов и символов перевода строки, которые затем автоматически отбрасываются.

```
bash$ ls -l | xargs
total 0 -rw-rw-r-- 1 bozo bozo 0 Jan 29 23:58 file1 -rw-rw-r-- 1 bozo bozo 0 Jan 29
```

xargs имеет очень любопытный ключ -n NN, который ограничивает количество передаваемых аргументов за один "присест"числом NN.
Пример. Вывести список файлов текущего каталога в 8 колонок.

```
ls | xargs -n 8 echo
```

Еще одна полезная опция -0, в комбинации с find -print0 или grep -lZ позволяет обрабатывать аргументы, содержащие пробелы и кавычки.

```
find / -type f -print0 | xargs -0 grep -liwZ GUI | xargs -0 rm -f
grep -rliwZ GUI / | xargs -0 rm -f
```

14.2 Time / Date commands

14.2.1 date

Команда date без параметров выводит дату и время на стандартное устройство вывода stdout. Она становится гораздо интереснее при использовании дополнительных ключей форматирования вывода.

```
date +"%m/%d/%Y %H:%M:%S"
date +"%D %T"
03/27/12 16:15:25
date +"%s" // seconds since 1970-01-01 00:00:00 UTC
BEGIN=`date +%s`; ./script; END=`date +%s`; DIFF=$((END-BEGIN)); printf "%dh %dm %d
```

14.2.2 time

Выводит подробную статистику по исполнению некоторой команды.

```
time gmake
real    0m3.076s
user    0m0.004s
sys     0m0.004s
```

14.2.3 at

Используется для запуска заданий в заданное время.

```
at 2pm January 15
at 2:30 am Friday < at-jobs.list
at 2:30 am Friday -f at-jobs.list
```

14.2.4 sleep

Приостанавливает исполнение сценария на заданное количество секунд, ничего не делая. Может использоваться для синхронизации процессов, запущенных в фоне, проверяя наступление ожидаемого события так часто, как это необходимо.

```
sleep 3 h
```

Команда sleep по-умолчанию принимает количество секунд, но ей можно передать и количество часов и минут и даже дней.

14.3 Text Processing Commands

14.3.1 sort

сортирует поток текста

```
cat file | sort -u
```

14.3.2 cut

предназначена для извлечения отдельных полей из текстовых файлов. Напоминает команду print $N в awk, но более ограничена в своих возможностях. В простейших случаях может быть неплохой заменой awk в сценариях. Особую значимость, для команды cut, представляют ключи -d (разделитель полей) и -f (номер(а) поля(ей)).
Список смонтированных файловых систем:

```
cat /etc/mtab | cut -d ' ' -f1,2
```

Получение версии ОС и ядра:

```
uname -a | cut -d"␣" -f1,3,11,12
```

14.3.3 paste

- для объединения нескольких файлов в один многоколоночный файл

```
paste names.txt results.txt
```

14.3.4 join

- позволяет объединять два файла по общему полю, что представляет собой упрощенную версию реляционной базы данных, оперирует только двумя файлами и объедияет только те строки, которые имеют общее поле (обычно числовое), результат объединения выводится на stdout. Объединяемые файлы должны быть отсортированы по ключевому полю.

14.3.5 head

- выводит начальные строки(символы) из файла на стандартный вывод (по-умолчанию - 10 строк, но это число можно задать иным).

```
head -20 file
head -c4 file

# -c4 – первые 4 байта.
# -20 – 20 строк
```

14.3.6 tail

- выводит последние строки из файла на stdout (по-умолчанию – 10 строк). Обычно используется для мониторинга системных журналов. Ключ -f, позволяет вести непрерывное наблюдение за добавляемыми строками в файл.

```
tail -1 file
tail -f
tailf
```

14.3.7 watch

Периодически запускает указанную программу с заданным интервалом времени.
По-умолчанию интервал между запусками принимается равным 2 секундам, но может быть изменен ключом -n.

```
watch -n 5 tail file.log
```

Выводит последние 10 строк из файла file.log, каждые пять секунд.

14.3.8 grep

многоцелевая поисковая утилита, использующая регулярные выражения.

```
grep pattern [file...]
```

```
grep -Hns $pattern $filename_pattern
```

-H - выводит имена файлов
-n - выводит номера строк
-s - silent mode - игнорирует ошибки
Также интересные опции:
-l - вывод только имен файлов, в которых найдены участки, совпадающие с заданным образцом/шаблоном, без вывода совпадающих строк.
-v (или –invert-match) – выводит только строки, не содержащие совпадений.
-r - (рекурсивный поиск) поиск выполняется в текущем каталоге и всех вложенных подкаталогах.
-A - выводит указанное количество строк до строки с совпадением -B - выводит указанное количество строк после строки с совпадением

```
grep -A2 -B2 "$pattern" file
```

```
ps aux | grep "$named" | grep -v "grep"
```

egrep - то же самое, что и grep -E. Эта команда использует несколько отличающийся, расширенный набор регулярных выражений, что позволяет выполнять поиск более гибко.

```
egrep "(reboot|shutdown)s?" *
```

fgrep - то же самое, что и grep -F. Эта команда выполняет поиск строк символов (не регулярных выражений), что несколько увеличивает скорость поиска.

14.3.9 wc

- счетчик слов в файле или в потоке

```
bash $ wc /usr/doc/sed-3.02/README
20      127      838 /usr/doc/sed-3.02/README
```

[20 строк 127 слов 838 символов]
wc -w подсчитывает только слова.
wc -l подсчитывает только строки.
wc -c подсчитывает только символы.
wc -L возвращает длину наибольшей строки.

14.3.10 tr

- замена одних символов на другие Вывести в нижнем регистре весь файл:

```
tr "A–Z" "a–z" < text.txt
```

Удалить все цифровые символы при выводе:

```
tr –d 0–9 <filename
```

Удалить повторяющиеся пробелы:

```
echo "a␣␣␣␣b␣c" | tr ––squeeze–repeats ' '
a b c
```

tr корректно распознает символьные классы POSIX.
Преобразование символов в верхний регистр:

```
tr '[:lower:]' '[:upper:]' < file
```

14.4 sed

sed - неинтерактивный редактор текстовых файлов.
awk - язык обработки шаблонов с C-подобным синтаксисом.
При всех своих различиях, эти awk и sed обладают похожим синтаксисом, они обе умеют работать с регулярными выражениями, обе, по-умолчанию, читают данные с устройства stdin и обе выводят результат обработки на устройство stdout. Обе являются утилитами UNIX-систем, и прекрасно могут взаимодействовать между собой. Вывод от одной может быть перенаправлен, по конвейеру, на вход другой.
Одно важное отличие состоит в том, что в случае с sed, сценарий легко может передавать дополнительные аргументы этой утилите, в то время, как в случае с awk, это более сложная задача.
sed принимает текст либо с устройства stdin, либо из текстового файла, выполняет некоторые операции над строками и затем выводит результат на устройство stdout или в файл. Как правило, в сценариях, sed используется в конвейерной обработке данных, совместно с другими командами и утилитами.
Из всего разнообразия операций, мы остановимся на трех, используемых наиболее часто. Это p – печать (на stdout), d – удаление и s – замена.
Встроенные операции sed
Печать:
[диапазон строк]/p
Удаление:
[диапазон строк]/d
Замена:
s/pattern1/pattern2/ – Заменить первое встреченное соответствие шаблону pattern1, в строке, на pattern2
[диапазон строк]/s/pattern1/pattern2/ – Заменить первое встреченное соответствие шаблону pattern1, на pattern2, в указанном диапазоне строк
g - global - Операция выполняется над всеми найденными соответствиями внутри каждой из заданных строк
Примеры.
Вывести содержимое файла, удаляя строки, содержащие $pattern.

```
sed "/$pattern/d" "$filename"
```

Вывести только строки, в которых встречается паттерн:

```
sed -n "/$pattern/p" $filename
```

Ключ -n - не выводить все строки.
Удалить все пустые строки.

```
/^$/d
```

Удалить все строки до первой пустой строки, включительно.

```
1,/^$/d
```

Удалить все пробелы в конце каждой строки.

```
s/ *$//
```

Удалить все найденные "$pattern оставляя остальную часть строки без изменений

```
s/$pattern//g
```

Указание диапазона строк, предшествующее одной, или более, инструкции может потребовать заключения инструкций в фигурные скобки, с соответствующими символами перевода строки.
Если команд несколько нужно использовать опцию e"перед каждой командой.

14.5 awk

Awk - это полноценный язык обработки текстовой информации с синтаксисом, напоминающим синтаксис языка C. Он обладает довольно широким набором возможностей, однако, мы рассмотрим лишь некоторые из них - наиболее употребимые в сценариях командной оболочки.
Awk "разбивает"каждую строку на отдельные поля. По-умолчанию, поля - это последовательности символов, отделенные друг от друга пробелами, однако имеется возможность назначения других символов, в качестве разделителя полей. Awk анализирует и обрабатывает каждое поле в отдельности. Это делает его идеальным инструментом для работы со структурированными текстовыми файлами, осбенно с таблицами.
Внутри сценариев командной оболочки, код awk, заключается в **"строгие"(одиночные)** кавычки и **фигурные скобки**.

– Разделитель полей можно не задавать (по умолчанию - пробел).
– Опция -f говорит о том, что инструкции awk содержатся в файле сценария (**должен быть исполняемым и в первой строке awk должен быть указан как интерпретатор команд)**
Сценарий awk - набор инструкций, состоящих из шаблонов(команд) и связанных с ними процедур.
Считывая awk разбивает текст на строки - записи, те в свою очередь разбиваются на поля согласно разделителю полей.
Шаблонная часть инструкции уточняет, когда следует выполнять инструкцию или к каким данным она должна применяться. Процедура определяет порядок обработки данных.
Шаблоном может служить любая условная или составная конструкция или регулярное выражение.
Существует два специальных шаблона: BEGIN и END.
BEGIN - применяется для инициализации переменных и создания заголовков отчёта, связанная с ним процедура выполняется перед началом обработки входного файла.
END - для вывода итоговых данных и выполнении процедур по завершении обработки файла.
Если шаблон не указан - процедура применяется к каждой записи из входного файла.
Тело процедуры заключается в фигурные скобки.
Если процедура не задана - на экран выводится всё содержимое записи, соответствующей шаблону.

```
awk '{print $3}' $filename
```

– Выводит содержимое 3-го поля из файла $filename на устройство stdout.

```
awk '{print $1 $5 $6}' $filename
```

– Выводит содержимое 1-го, 5-го и 6-го полей из файла $filename.
Только что, мы рассмотрели действие команды print. Еще, на чем мы остановимся – это переменные. Awk работает с переменными подобно сценариям командной оболочки, но более гибко.

```
{ total += ${column_number} }
```

Эта команда добавит содержимое поля с заданным номером к переменной total. Чтобы, в завершение вывести total, можно использовать команду END, которая открывает блок кода, отрабатывающий после того, как будут обработаны все входные данные.

```
END { print total }
```

Команде END, соответствует команда BEGIN, которая открывает блок кода, отрабатывающий перед началом обработки входных данных.

```
awk 'BEGIN{print "COUNT\tNAME"}; {print $2 "\t" $1} {sum+=$2} END{print "TOTAL:\t"s
```

```
awk -F": " '/model name/ {print $2}' < /proc/cpuinfo
```

```
awk '{ for(i = 1; i <= NF; i=i+1) if ($i < 0) $i = -$i; print }' matrix.txt
```

<– каждый отрицательный элемент матрицы заменить модулем этого элемента
Конструкцию, используемую для вывода строк соответствующих заданной маске:

```
awk '{if ($0 ~ /pattern/) print $0}'
```

можно сократить до

```
awk '/pattern/'
```

Условие в awk может быть задано вне скобок, т.е. получаем:

```
awk '$0 ~ /pattern/ {print $0}'
```

По умолчанию, действия производятся со всей строкой, $0 можно не указывать:

```
awk '/pattern/ {print}'
```

print - является действием по умолчанию, его тоже можно не указывать.

```
awk '/pattern/'
```

Для вывода значения первого столбца строки, в которой присутствует маска pattern:

```
awk '/pattern/ {print $1}'
```

Для вывода значения первого столбца строки, во втором столбце которой присутствует маска pattern:

```
awk '$2 ~ /pattern/ {print $1}'
```

Для замены слова pattern1 на pattern2 и вывода только измененных строк можно использовать:

```
awk '{if(sub(/pattern1/,"pattern2")){print}}'
```

Но есть нужно выводить все строки (как sed 's/pattern1/pattern2/'), конструкцию можно упростить (1 - true для всех строк):

```
awk '{sub(/pattern1/,"pattern2")}1'
```

Вывести все строки, за исключением каждой шестой:

```
awk 'NR % 6'
```

Вывести строки, начиная с 6:

```
awk 'NR > 5'
tail -n +6
sed '1,5d'
```

Вывести строки, в которых значение второго столбца равно foo:

```
awk '$2 == "foo"'
```

Вывести строки, в которых 6 и более столбцов:

```
awk 'NF >= 6'
```

Вывести строки, в которых есть слова foo и bar:

```
awk '/foo/ && /bar/'
```

Вывести строки, в которых есть слово foo, но нет bar:

```
awk '/foo/ && !/bar/'
```

Вывести строки, в которых есть слова foo или bar (как grep -e 'foo' -e 'bar'):

```
awk '/foo/ || /bar/'
```

Вывести все непустые строки:

```
awk 'NF'
```

Вывести все строки, удалив содержимое последнего столбца:

```
awk 'NF--'
```

Вывести номера строк перед содержимым:

```
awk '$0 = NR" "$0'
```

Заменим команды (пропускаем 1 строку, фильтруем строки с foo и заменяем foo на bar, затем переводим в верхний регистр и выводим значение второго столбца)

```
cat test.txt | head -n +1 | grep foo | sed 's/foo/bar/' | tr '[a-z]' '[A-Z]' | cut
```

аналогичной конструкцией на awk:

```
cat test.txt | awk 'NR>1 && /foo/{sub(/foo/,"bar"); print toupper($2)}'
```