Дата последней сборки документа:
5 декабря 2020 г. в 00:58

Практика использования и наиболее полезные конструкции командной оболочки `bash`

Содержание

1. Сценарии командной оболочки bash

1.1. Особенности обработки выражений, заключенных в кавычки

Строка в форме `$'...'` умеет обрабатывать управляющие последовательности

```
$ echo $'python\nfortran'
# выведет две строки, разбитые символом перевода строки
# python
# fortran
```

Строка в форме `"..."` умеет корректно обрабатывать подстановку значений переменных, арифметических выражений и подстановку команд

```
$ EMAIL = 'leor.finkelberg@yandex.ru'
$ echo $EMAIL  # leor.finkelberg@yandex.ru
$ echo "My email: '$EMAIL'"  # My email:'leor.finkelberg@yandex.ru'
$ echo "In this catalogy `ls -l | wc -l` files"  # In this catalogy 10 files
$ echo "$INT % 2 -> $(($INT % 2))"  # 8 % 2 -> 0
$ echo "result = <<`find . -name 'cheat*.tex' | xargs grep -inE 'tab.*ofcont*'`>>"
                                                    # result = <<21:\tableofcontents>>
```

1.2. Переменные, встроенные в оболочку

1.3. Формат файла сценария

Для того чтобы успешно создать и запустить сценарий командной оболочки требуется:

1. Написать сценарий,
2. Сделать сценарий исполняемым,
3. Поместить сценарий в каталог, где командная оболочка сможет найти его.

Простой пример в качестве иллюстрации

bash_test.sh

```
#!/bin/bash
echo "This is number of command line args: $#"
```

Сочетание символов `#!` – это специальная конструкция, которая называется *shebang* (произносится как «ше-бенг») и сообщает системе имя интерпретатора, который должен использоваться для выполнения следующего за ним текста сценария. Каждый сценарий командной оболочки должен включать это определение в первой строке.

Команда «точка» (`.`) является синонимом `source`, встроенной команды, которая читает указанный файл и интерпретирует его как ввод с клавиатуры [2, стр. 349].

1.4. Разрешение на выполнение

Теперь нужно сделать этот файл исполняемым

```
$ chmod +x bash_test.sh
$ ls -l bash_test.sh
# выведет
# -rwxr-xr-x 1 ADM 197121 31 май 19 02:55 bash_test.sh*
```

1.5. Местоположение файла сценария

Теперь можно вызывать этот скрипт

```
$ ./bash_test.sh 10 20
# выведет
# This is number of command line args: 2
```

Но чтобы вызвать сценарий, необходимо добавить явный путь перед его именем. Для того чтобы можно было вызывать этот сценарий из любой точки системы, следует добавить его в переменную окружения `PATH`. Как известно, система просматривает каталоги по списку всякий раз, когда требуется найти исполняемую программу, если путь к ней не указан явно. Список каталогов храниться в как раз в переменной окружения `PATH`.

Она содержит список каталогов, перечисленных через двоеточие

```
$ echo $PATH
# выведет
# /c/Users/ADM/bin:/mingw64/bin:/usr/local/bin:/usr/bin:/bin:/mingw64/bin:/usr/bin:\
# /c/Users/ADM/bin:/c/Program Files (x86)/Common Files/Oracle/Java/javapath:\
# /c/Program Files (x86)/Intel/iCLS Client:/c/Program Files/Intel/iCLS Client ... etc
```

К слову, для более удобного просмотра вывода (пути разделяются символом «`:`»), можно воспользоваться следущей процедурой с привлечением редактора `Vim`:

- Перенаправляем стандартный поток вывода в файл `paths.txt`:
  `echo $PATH > paths.txt`
- Открываем файл редактором `Vim`
  `vim paths.txt`
- Сочетанием клавиш `qa`[1] открываем журнал записи пользовательских сценариев (внизу экрана появится строка «`запись @a`»),
- Теперь можно выполнить нужную нам последовательность команд для первого символа «`:`». Например `f:xi<CR><Esc>q`, что означает следующее: найти слева направо символ «`:`», удалить символ, который будет находится под курсором (т.е. символ «`:`»), перейти в режим ВСТАВКИ (`i`), нажать клавишу `Enter` для перехода на следующую строку в режиме ВСТАВКИ, нажать клавишу `Esc` для выхода из режима ВСТАВКИ и, наконец, закончить запись (`q`)[2],
- Записав сценарий, можно воспользоваться командой `normal`[3]
  `:%normal! 100@a`

Результат будет выглядеть так

[1] Начать запись в именованный регистр `"a`

[2] Проверить содержимое регистра можно так `:reg a`

[3] Здесь задается произвольное большое число, указывающее число повторений команды. Еще очень важный момент заключается в том, что эта команда выполняется параллельно, а не последовательно

```
/c/Users/ADM/bin
/mingw64/bin
/usr/local/bin
/usr/bin
/bin
/mingw64/bin
```

В большинстве Unix-подобных операционных систем в переменную PATH включается каталог bin в *домашнем каталоге пользователя*, чтобы дать пользователям выполнять собственные программы.

То есть если создать каталог ~/bin и поместить сценарий в него, то наш сценарий можно будет запускать из любой точки системы

```
$ mkdir ~/bin
$ mv bash_test.sh ~/bin
```

Если каталог отсутствует в переменной PATH, его легко туда добавить, включив следующую строку в файл ~/.bash_profile

```
export PATH=~/bin:${PATH}
```

Команда export устанавливает *переменную окружения*, или другими словами экспортирует переменную, делая ее переменной окружения. Здесь «:» это просто символ-разделитель, а конструкция ${PATH} разворачивается в список путей, разделенных символом «:».

1.6. Выбор местоположения для сценариев

Каталог ~/bin хорошо подходит для сценария, если этот сценарий предназначен для *личного* использования. Сценарии, которые должны быть доступны *всем* пользователям в системе, лучше размещать в традиционном местоположении – в каталоге /usr/local/bin.

В большинстве случаев программное обеспечение созданное в *локальной* системе, будь то сценарий или скомпилированные программы, следует помещать в иерархию каталогов /usr/local, а не /bin или /usr/bin. Последние два каталога, как определено стандартом иерархии файловой системы Linux, предназначены только для файлов, поставляемых создателями дистрибутива Linux.

1.7. Присваивание значений переменным и константам

Командная оболочка не заботиться о типах значений, присваиваемых переменным. Она все значения интерпретирует как *строки*. Между именем переменной и оператором присваивания не должно быть пробелов. Значение может состоять из чего угодно, что можно развернуть в строку

```
a=z                     # Присвоить переменной a строку "z"
b="a string"            # Внутренние пробелы должны находиться в кавычках
c="a string and $b"     # При присваивании допускается выполнять подстановку,
                        # например, значений других переменных
d=$(ls -l foo.txt)      # Результат выполнения команды
e=$((5*7))              # Подстановка результата арифметического выражения
f="\t\ta string\n"      # Экранирование последовательности, такие как
                        # символы табуляции и перевода строки
```

При использовании подстановки имена переменных можно заключить в *необязательные фигурные скобки* `{}`. Это пригодится в том случае, когда имя переменной становится неоднозначным в окружающем контексте.

Например

```
$ filename="myfile"
$ touch $filename
$ mv $filename ${filename}1  # переименовывает файл myfile -> myfile1
```

Добавив фигурные скобки, мы гарантировали, что командная оболочка не будет интерпретировать последний символ 1 как часть имени новой (и пустой) переменной.

Существует еще один метод вывода текста, который называется *встроенным документом* или *встроенным сценарием*. Встроенный документ – это дополнительная форма перенаправления ввода/вывода, которая передает текст, встроенный в сценарий, на стандартный ввод команд.

Действует это перенаправление так:

```
команда << индикатор
текст
индикатор
```

Например

```
#!/bin/bash

TITLE="System Information Report Fort $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
<HTML>
    <HEAD>
        <TITLE>$TITLE</TITLE>
    </HEAD>
    <BODY>
        <H1>$TITLE</H1>
        <P>$TIME_STAMP</P>
    </BODY>
</HTML>
_EOF_
```

К слову, для того чтобы заменить `$name` на `${name}` в сценарии можно воспользоваться такой конструкцией

```
:%s/>\$\(.*\)</>${\1}</c
------------------------
:%s  # произвести замену во всех строках файла
   /  # разделитель
    >\$\(.*\)<  # поисковый шаблон; например, >$TITLE<
              /  # разделитель
               >${\1}<  # шаблон подстановки
                      /c  # замена с подтверждением по каждому совпадению
```

Эта конструкция ищет совпадение с шаблоном во *всех* строках сценария (`%`). Подшаблон `>\$\(.*\)<` совпадает, например, с подстрокой `>$TITLE<`, и при этом `TITLE` попадает в группу №1 (к ней можно обращаться как `\1`). Теперь можно `>$TITLE<` заменить на `>${\1}<`. Эта часть шаблона воспринимается «как есть», т.е. ничего дополнительно экранировать не нужно. Параметр `c`

требует подтверждение замены каждый раз, когда находит подстроку отвечающую поисковому шаблону.

На роль индикатора была выбрана строка `_EOF_`, и она отмечает конец встроенного текста. Строка-индикатор должна находиться в отдельной строке, одна, и за ней не должно следовать никаких пробелов.

Если заменить оператор перенаправления `<<` на `<<-`, то командная оболочка будет игнорировать начальные символы табуляции во встроенном документе. Для того чтобы блок встроенного документа работал корректно для форматирования тела блока необходимо использовать только ТАБУЛЯЦИЮ, поэтому в `Vim` придется отключить преобразование табуляции в пробелы (закомментировать строку `set expandtab`). Благодаря этому во встроенный документ можно добавить отступы для большей удобочитаемости

```
#!/bin/bash

FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz

ftp -n <<- _EOF_
    open $FTP_SERVER            # здесь строго ТАБУЛЯЦИЯ!
    user anonymous me@linuxbox  # здесь строго ТАБУЛЯЦИЯ!
    cd $FTP_PATH                # здесь строго ТАБУЛЯЦИЯ!
    hash                        # здесь строго ТАБУЛЯЦИЯ!
    get $REMOTE_FILE            # здесь строго ТАБУЛЯЦИЯ!
    bye                         # здесь строго ТАБУЛЯЦИЯ!
    _EOF_                       # здесь строго ТАБУЛЯЦИЯ!
ls -l $REMOTE_FILE
```

Еще можно перенаправлять вывод, организованный с помощью `<<-`, в файл

```
#!/bin/bash

USER="leor.finkelberg"

report_home_space () {
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_ > ./contfile.txt  # <- NB          # здесь строго ТАБУЛЯЦИЯ!
            <H2>Home Space Utilization (All Users)</H2>  # здесь строго ТАБУЛЯЦИЯ!
            <PRE>$(du -sh .)</PRE>                       # здесь строго ТАБУЛЯЦИЯ!
            _EOF_                                        # здесь строго ТАБУЛЯЦИЯ!
...
```

Конструкция `cat <<- _EOF_ > ./contfile.txt` перезаписывает файл `contfile.txt` при каждом запуске сценария. Если требуется «дозаписать» файл, то следует использовать конструкцию `cat <<- _EOF_ >> ./contfile.txt`.

Можно вывод команды `cat` направить как в стандартный поток вывода (терминал), так и в файл с помощью команды `tee`[4]

```
report_home_space () {
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_ | tee ./contfile.txt  # <- NB
                ...
                _EOF_
    ...
}
```

[4]Утилита `tee` копирует стандартный ввод на стандартный вывод, а также в один или несколько файлов [1]

Без ключей утилита `tee` переписывает файл вывода, если они существуют, и реагирует на прерывания. Ключ `-a` заставляет утилиту добавлять вывод к существующим файлам вместо их переписывания.

1.8. Функции

Функции имеют две синтаксические формы. Первая выглядит так

```
function fun_name {
    commands
    return
}
```

Вторая форма выгладит так

```
fun_name () {
    commands
    return
}
```

Обе формы эквивалентны и могут использоваться одна вместо другой. Ниже приводится сценарий, демонстрирующий использование функций командной оболочки

```
1  #!/bin/bash
2
3  function funct {
4      echo "Step 2"
5      return
6  }
7
8  echo "Step 1"  # Step 1
9  funct          # Step 2
10 echo "Step 3"  # Step 3
```

Когда командная оболочка читает сценарий, она пропускает строки с 1-ой по 7-ую, так как они содержат только определение функции. Выполнение начинается со строки 8 с команды `echo`. Строка 9 вызывает функцию `funct`, и командная оболочка выполняет функцию как любую другую команду. Управление передается в строку 4, и выполняется вторая команда `echo`.

Команда `return` в этой строке завершает выполнение функции и возвращает управление в строку, следующую за вызовом функции. После этого выполняется заключительная команда `echo`. Функции должны быть определены в сценарии до их вызова.

Имена функций подчиняются тем же правилам, что и имена переменных. Функция должна содержать хотя бы одну команду. Команда `return` (которая является необязательной) помогает удовлетворить это требование.

Примеры использования функций с *локальными переменными*

```
#!/bin/bash

foo=0  # глобальная переменная

funct_1 () {
    local foo  # переменная ‘foo‘ локальная для ‘funct_1‘
    foo=1
    echo "funct_1: foo = $foo"
}
```

```
# для вызова функции
funct_1
```

Локальные переменные объявляются добавлением слова `local` перед именем переменной. В результате создается переменная, локальная по отношению к функции, в которой она определена. Когда выполнение выйдет за пределы функции, переменная перестанет существовать.

Рассмотрим такую функцию

```
report_disk_space () {
    cat <<- _EOF_                          # здесь строго ТАБУЛЯЦИЯ!
        <H2>Disk Space Utilization</H2>    # здесь строго ТАБУЛЯЦИЯ!
        <PRE>$(df -h)</PRE>                # здесь строго ТАБУЛЯЦИЯ!
        _EOF_                              # здесь строго ТАБУЛЯЦИЯ!
    return
}

# вызов функции
report_disk_space
```

Она получает информацию о дисковом пространстве с помощью команды `df -h`.

1.9. Ветвление

Инструкция `if` имеет следующий синтаксис

```
if commands; then
    commands
[elif commands; then
    commands...]
[else
    commands]
fi
```

В командной оболочке `bash` поддерживается еще один способ ветвления. Операторы `&&` и || действуют подобно логическим операторам в составной команде `[[...]]`. Они имеют следующий синтаксис

```
command1 && command2
```

и

```
command1 || command2
```

В последовательности с оператором `&&` первая команда выполняется всегда, а вторая – только если первая завершилась успешно. В последовательности с оператором || первая команда выполняется всегда, а вторая – только если первая завершилась неудачей.

Например

```
# если каталога ‘temp‘ не существует, то его нужно создать
$ [[ -d temp ]] || mkdir temp
```

1.10. Современная версия команды test

Улучшенная версия команды `test` выглядит так

```
[[ выражение ]]
```

Команда `test` и ее формы (`[...]`, `[[...]]`) возвращают *код завершения*, показывающий, что выражение либо истинно – `true` (0), либо ложно – `false` (не 0). А вот самое `выражение` возвращает истинное (`true`) или ложное (`false`) значение.

Команда `[[...]]` очень похожа на команду `[...]`, но добавляет новое выражение для проверок строк `строка1 =~ регулярное_выражение`.

Например можно проверить отвечает ли заданное число регулярному выражению

```
#!/bin/bash

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [ $INT -eq 0 ]; then
        echo "INT is zero."
    else
        if [ $INT -lt 0 ]; then
            echo "INT is negative."
        else
            echo "INT is positive."
        fi
        if [ $((INT % 2 )) -eq 0 ]; then
            echo "INT is even."
        else
            echo "INT is odd."
        fi
    fi
else
    echo "INT is not an integer." >&2
    exit 1
fi
```

В команде `[[...]]` поддерживаются *классы символов*[5] POSIX. Ключевые слова описывают различные классы символов, такие как алфавитные, управляющие символы и пр. POSIX, например

```
$ EMAIL="leor.finkelberg@yandex.ru"
$ if [[ "$EMAIL" =~ ^[[:alpha:]]+\.[[:alpha:]]+@[[:alpha:]]+\.ru$ ]]; then echo 'OK'; fi
```

Еще одна дополнительная особенность `[[...]]`: оператор `==` поддерживает сопоставление с шаблонами по аналогии с механизмом подстановки путей. Например

```
$ FILE="foo.bar"
$ if [[ $FILE == foo.* ]]; then
> echo "$FILE matches pattern 'foo.*'"
>fi
# foo.bar matches pattern 'foo.*'
```

Или можно воспользоваться регулярным выражением

```
$ FILE="foo.bar"
$ if [[ $FILE =~ ^foo\..+$ ]]; then echo 'OK'; fi
# OK
```

В дополнение к составной команде `[[...]]` `bash` поддерживает также составную команду `((...))`, которую удобно использовать для работы с целыми числами.

[5]Класс символов POSIX состоит из ключевых слов, заключенных между `[:` и `:]`. Эти конструкции должны находится в квадратных скобках скобкового выражения. Например, `[[:alpha:]!]` соответствует любому одиночному символу или восклицательному знаку [3, стр. 98]

Команда `((...))` применяется для *проверки истинности арифметических выражений*. Арифметическое выражение считается *истинным*, если его *результат отличается от нуля*.

Пример

```
#!/bin/bash

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if (( $INT == 0 )); then
        echo "INT is zero."
    ...
    fi
    if (( $INT % 2 )); then
        echo "INT is even."
    else
        echo "INT is odd."
...
```

1.11. Объединение выражений

Команда `[[...]]` поддерживает три логические операции: И (`&&`), ИЛИ (`||`) и НЕ (`!`).

Пример

```
#!/bin/bash

MIN_VAL=1
MAX_VAL=100
INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
    if [[ INT -ge MIN_VAL && INT -le MAX_VAL ]]; then
        echo "$INT is whithin $MIN_VAL to $MAX_VAL."
    else
        echo "$INT is out of range."
    fi
else
    else "INT is not an integer." >&2
    exit 1
fi
```

Логические блоки можно заключать в круглые скобки

```
...
if [[ ($INT -lt 0) && ($INT -gt 10) ]]; then
    echo ...
```

Еще с `[[...]]` можно сочетать подстановку `$(...)` и вычисления `(...)`

```
...
if [[ ($INT -gt 0) && $(($INT % 2)) -eq 0 ]]; then
    echo ...
```

Узнать имеет ли текущий пользователь права на чтение всех домашних каталогов можно следующим образом

```
USER="leor.finkelberg"

report_home_space () {
```

```
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_                                       # здесь строго ТАБУЛЯЦИЯ!
            <H2>Home Space Utilization (All Users)</H2>     # здесь строго ТАБУЛЯЦИЯ!
            <PRE>$(du -sh .)</PRE>                          # здесь строго ТАБУЛЯЦИЯ!
            _EOF_                                           # здесь строго ТАБУЛЯЦИЯ!
    else
        cat <<- _EOF_                                       # здесь строго ТАБУЛЯЦИЯ!
            <H2>Home Space Utilization ($USER)</H2>         # здесь строго ТАБУЛЯЦИЯ!
            <PRE>$(du -sh $HOME)</PRE>                      # здесь строго ТАБУЛЯЦИЯ!
            _EOF_                                           # здесь строго ТАБУЛЯЦИЯ!
    fi
    return
}

report_home_space
```

1.12. Чтение и ввод с клавиатуры

Команда `read` используется для чтения единственной строки со стандартного ввода. Синтаксис

```
read [-parameters] [var ...]
```

Если имя переменной не указано, строка с данными сохраняется в переменную `REPLY`.

Например, для того чтобы пользователь мог задать имя файла, в который сценарий будет перенаправлять вывод `<<-`, можно воспользоваться ключом `-r`

```
#!/bin/bash

read -p "Enter filename: " filename  # <-

report_home_space () {
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_ | tee ./$filename  # <-
            ...
        _EOF_
    ...
}
```

Еще один простой пример

```
#!/bin/bash

# читает пользовательский ввод
read -p "Enter integer number: " int

if [[ "$int" =~ ^-?[[:digit:]]+$ ]]; then
    if [ $int -eq 0 ]; then
        echo "$int is zero."
    else
        if [ $int -lt 0 ]; then
            echo "$int is negative."
        else
            echo "$int is positive."
        fi
        if [ $((int % 2)) -eq 0 ]; then
            echo "$int is even."
        else
            echo "$int is odd."
```

```
        fi
    fi
else
    echo "Input value is not an integer." >&2
    exit 1
fi
```

Для того чтобы прочитать несколько значений, следует воспользоваться такой конструкцией

```
read -p "Enter one or more values > " var1 var2 var3 var4 var5

echo "var1 = '$var1'"
echo "var2 = '$var2'"
echo "var3 = '$var3'"
echo "var4 = '$var4'"
echo "var5 = '$var5'"
```

С помощью ключа `-s` можно скрывать символы при вводе, а с помощью ключа `-t` устанавливать время ожидания на ввод. Например

```
if read -t 10 -sp "Enter secret passphrase > " secret_pass; then
    echo -e "\nSecret passphrase = '$secret_pass'"
    # флаг -e нужен для интерпретации управляющей последовательности
...
```

Здесь пользователю предлагается ввести секретный пароль за отведенные 10 сек. Если в течение этого времени ввод не был завершен, сценарий завершается с кодом ошибки. Поскольку в команду включен параметр `-s`, символы пароля не выводятся на экран в процессе ввода.

1.13. Выделение полей в строке ввода с помощью IFS

Обычно командная оболочка выполняте разбиение ввода на слова перед передачей его команде `read`. Слова во вводе, разделенные одним или несколькими пробелами, становятся отдельными значениями и присваиваются командой `read` разным переменным.

Такое поведение командной оболочки регулируется переменной `IFS` (Internal Field Separator). По умолчанию переменная `IFS` хранит символы пробела, табуляции и перевода строки, каждый из которых может служить разделителем полей.

Изменяя значение переменной `IFS`, можно управлять делением ввода на поля перед передачей команде `read`. Например, пусть файл `/etc/passwd` хранит строки данных, в которых поля отделяются друг от друга двоеточием. Присвоив переменной `IFS` значение, состоящее из единственного двоеточия, можно с помощью `read` прочитать содержимое `/etc/passwd` и благополучно разделить строки на поля для присваивания разным переменным. Ниже приводится сценарий, который именно так и действует

```
#!/bin/bash

# файл должен существовать до обращения и иметь
# содержание вида Leor:100:34345:Leor Finkelberg:~:bash
FILE="./passwd.txt"

# строка-приглашение
read -p "Enter a username > " user_name  # вводим, например, leor

# благодаря флагу `-i` имя пользователя можно задавать в нижнем регистре
# подстановка $user_name отрабатывает раньше, чем поиск по регулярному выражению
```

```
file_info=$(grep -iE "^$user_name:" $FILE)  # одна единственная строка из файла ./passwd.txt

if [[ -n "$file_info" ]]; then  # если строка непустая, то...
    IFS=":" read user pw uid gid name home shell <<< "$file_info"  # <-
    echo "User = '$user'"
    echo "UID = '$uid'"
    echo "GID = '$gid'"
    echo "Full Name = '$name'"
    echo "Home Dir. = '$home'"
    echo "Shell = '$shell'"
else
    echo "No such user '$user_name'" >&2
    exit 1
fi
```

Командная оболочка позволяет выполнять в одной строке одно или несколько операций присваивания значений переменным непосредственно *перед* командной, на поведение которой эти переменные влияют. Они *изменяют окружение*, в котором выполняется команда. Действие этих операций присваивания ности *временный* характер, окружение изменяется только на время выполнения команды. В данном случае в переменной `IFS` сохраняет символ двоеточия.

То же самое можно выразить иначе

```
OLD_IFS="$IFS"
IFS=":"
read user pw uid gid name home shell <<< "$file_info"
IFS="$OLD_IFS"
```

Очевидно, что размещение операции присваивания перед командой позволяет получить более компактный код, действующий точно так же.

Оператор `<<<` отмечает *встроенную строку*. Встроенную строку (here string) простирается только до конца текущей строки кода. В данном примере строка из файла подается на стандартный ввод команды `read`.

Вот несколько любопытных примеров работы со *встроенной строкой*

```
$ string="This is a string of words."
$ read -r -a words <<< "$string"
$ echo "${words[0]}"  # This
$ echo "${words[1]}"  # is
...
```

Удобно тестировать регулярное выражение с помощью команды `grep`

```
$ grep -iE '(P|J)ython' <<< 'Jython'
$ grep -iE '[[:alpha:]]+\.[[:alpha:]]+@[[:alpha:]]+\.ru' <<< 'leor.finkelbex.ru'
```

1.14. Проверка ввода

Далее приводится пример программы, проверяющий входные данные разного вида

```
#!/bin/bash

clear  # очистить экран перед началом работы программы

invalid_input () {
    echo "Invalid input '$REPLY'" >&2
    exit 1
}
```

```
# переменная явно не указана, поэтому
# пользовательский ввод будет связан с переменной REPLY
read -p "Enter a single item > "

# пустой ввод (недопустимо!); если строка пустая, то вызывается функция ‘invalid_input‘
[[ -z $REPLY ]] && invalid_input
# если в переданной строке более одного слова, вызывать функцию ‘invalid_input‘
# ((...)) -> [true|false]
(( $(echo $REPLY | wc -w) > 1 )) && invalid_input

# введено допустимое имя файла
if [[ $REPLY =~ ^[-[:alnum:]\._]+$ ]]; then
    echo "’$REPLY’ is a valid filename."
    if [[ -e $REPLY ]]; then
        echo "And file ’$REPLY’ exists."
    else
        echo "However, file ’$REPLY’ does not exists."
    fi

    if [[ $REPLY =~ ^-?[[:digit:]]*\.[[:digit:]]+$ ]]; then
        echo "’$REPLY’ is a floating point number."
    else
        echo "’$REPLY’ is not a floating point number."
    fi

    if [[ $REPLY =~ ^-?[[:digit:]]+$ ]]; then
        echo "’$REPLY’ is an integer."
    else
        echo "’$REPLY’ is not an integer."
    fi
else
    echo "The string ’$REPLY’ is not a valid filename."
fi
```

Для проверки числа слов в пользовательском вводе с тем же результатом можно было бы использовать и конструкцию

```
[[ $(echo "$REPLY" | wc -w) -gt 1 ]] && invalid_input
```

1.15. Циклы

Синтаксис цикла с командой `while` выглядит так

```
while commands; do
    commands
done
```

Подобно `if`, команда `while` проверяет *код завершения* списка команд. Пока код завершения равен 0, она выполняет команды внутри цикла. В сценарии, приведенном выше, создается переменная `count`, и ей присваивается начальное значение 1. Команда `while` проверяет код завершения команды `test`. Пока `test` возвращает код 0, команды внутри цикла продолжают выполнятся. В конце каждого цикла повторно выполняется команда `test`. После шести итераций цикла значение переменной `count` увеличится до 6, команда `test` вернет код завершения, отличный от 0, и цикл завершится, а программа продолжит выполнение с инструкцией, следующей непосредственно за циклом.

Пример

```
#!/bin/bash

DELAY=2  # время отображения результатов на экране (в секундах)

while [[ $REPLY != 0 ]]; do
    clear
    cat <<- _EOF_                          # строго ТАБУЛЯЦИЯ
        Please Select:                     # строго ТАБУЛЯЦИЯ
                                           # строго ТАБУЛЯЦИЯ
        1. Display System Information      # строго ТАБУЛЯЦИЯ
        2. Display Disk Space              # строго ТАБУЛЯЦИЯ
        3. Display Home Space Utilization  # строго ТАБУЛЯЦИЯ
        0. Quit                            # строго ТАБУЛЯЦИЯ
                                           # строго ТАБУЛЯЦИЯ
    _EOF_                                  # строго ТАБУЛЯЦИЯ
    read -p "Enter selection [0-3] > "  # приглашение к вводу

    if [[ $REPLY =~ ^[0-3]$ ]]; then
        if [[ $REPLY == 1 ]]; then
            echo "Hostname: $HOSTNAME"
            echo uptime
            sleep $DELAY
        fi
        if [[ $REPLY == 2 ]]; then
            df -h
            sleep $DELAY
        fi
        if [[ $REPLY == 3 ]]; then
            if [[ $(id -u) -eq 0 ]]; then
                echo "Home Space Utilization (All Users)"
                du -sh
            else
                echo "Home Space Utilization ($USER)"
                du -sh $HOME/Roaming
            fi
            sleep $DELAY
        fi
    else
        echo "Invalid entry."
        sleep $DELAY
    fi
done
echo "Program terminated."
```

Можно переписать эту программу с использованием команд `continue` и `break`

```
#!/bin/bash

DELAY=2
USER="leor.finkelberg"

while true; do
    clear
    cat <<- _EOF_
        Please Select:

        1. Display System Information
        2. Display Disk Space
        3. Display Home Space Utilization
        0. Quit
```

```
    _EOF_
    read -p "Enter selection [0-3] > "

    if [[ $REPLY =~ ^[0-3]$ ]]; then
        if [[ $REPLY == 1 ]]; then
            echo "Hostname: $HOSTNAME"
            echo 'uptime'
            sleep $DELAY
            continue
        fi
        if [[ $REPLY == 2 ]]; then
            df -h
            sleep $DELAY
            continue
        fi
        if [[ $REPLY == 3 ]]; then
            if [[ $(id -u) -eq 0 ]]; then
                echo "Home Space Utilization (All Users)"
                du -sh .
            else
                echo "Home Space Utilization ($USER)"
            fi
            sleep $DELAY
            continue
        fi
        if [[ $REPLY == 0 ]]; then
            break
        fi
    else
        echo "Invalid entry."
        sleep $DELAY
    fi
done
echo "Program terminated."
```

Для того чтобы избежать ошибок, связанных с некорректными подстановками, следует читаемые переменные оборачивать кавычками, т.е. вместо `$REPLY` писать `"$REPLY"`. Кавычки следует использовать не только для предохранения от пустых строк, но и в том случае, если переменная содержит строку с несколькими словами, например имя файла со встроенными пробелами.

1.16. Трассировка

Ошибки часто становятся причиной неожиданного направления выполнения сценария. То есть фрагменты сценария могут никогда не выполняться или выполняться в неправильном порядке или в неправильные моменты. Чтобы увидеть, как в действительности протекает выполнение программы, воспользуемся приемом *трассировки*.

Один из способов трассировки заключается в размещении информативных сообщений в разных точках сценария, сообщающих, где протекает выполнение

```
#!/bin/bash

dir_name="my_test_dir"

echo "Preparing to delete files" >&2
if [[ -d $dir_name ]]; then
    if cd $dir_name; then
```

```
        echo "Deleting files" >&2
        echo rm -f *
    else
        echo "Cannot cd to '$dir_name'" >&2
        exit 1
    fi
else
    echo "No such directory: '$dir_name'" >&2
    exit 1
fi
echo "File deletion complete" >&2
```

Здесь сообщения посылаются в стандартный поток вывода ошибок (>&2), чтобы отделить их от обычного вывода. Кроме того отсутствуют отступы перед строками с сообщениями, – это упростить их поиск, когда придет время убрать эти строки.

Кроме того, bash поддерживает встроенный метод трассировки, реализованный в виде параметра -x. Этот параметр можно добавить в первую строку сценария

```
#!/bin/bash -x
...
```

Включенный режим трассировки позволяют увидеть, какой вид приобретают команды после применения подстановки. Начальные знаки «+» помогают отличить трассировочную информацию от обычного вывода. Знак «+» – это символ по умолчанию, используемый для вывода трассировки. Он хранится в переменной командной оболочки PS4 (Prompt string 4 – строка приглашения 4).

Можно изменить эту строку, добавив, например, номер выполняемой строки в сценарии. Здесь необходимо использовать одиночные кавычки – это предотвращает подстановку до момента, когда строка приглашения не будет использоваться фактически

```
$ export PS4='$LINENO: '
$ ./del_dir_test.sh
# выведет
3: dir_name=my_test_dir
5: echo 'Preparing to delete files'
Preparing to delete files
6: [[ -d my_test_dir ]]
15: echo 'No such directory: '\''my_test_dir'\'''
No such directory: 'my_test_dir'
16: exit 1
```

Выполнить трассировку только выбранного фрагмента сценария можно с помощью конструкции set -x ... set +x.

1.17. Управление потоком выполнения с помощью case

Командная оболочка bash поддерживает составную команду выбора из нескольких вариантов. Она имеет следующий синтаксис

```
case word in
    [pattern [| pattern]...) commands ;;]...
esac
```

Пример

```
#!/bin/bash

clear
echo "
Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter select [0-3] > "

case $REPLY in
    0)  echo "Program terminated."
    exit
    ;;
    1)  echo "Hostname: $HOSTNAME"
    echo 'uptime'
    exit
    ;;
    2)  df -h
    ;;
    3)  if [[ $(id -u) -eq 0 ]]; then
            echo "Home Space Utilization (All Users)"
            du -sh .
        else
            echo "Home Space Utilization ($USER)"
        fi
        ;;
    *)  echo "Invalid entry" >&2
        exit 1
        ;;
esac
```

Команда `case` берет значение *слова* – в данном примере занчение переменной `REPLY` – и затем сопоставляет его с указанными шаблонами. Найдя соответствие, она выполняет команды, связанные с найденными шаблоном. После нахождения соответствия сопоставление с нижележащими шаблонами уже не производится.

Примеры шаблонов:

- `a)` соответствует, если слово содержит `a`,
- `[[:alpha:]])` соответствует, если слово содержит единственный алфавитный символ,
- `???)` соответствует, если слово содержит ровно три символа,
- `*.txt)` соответствует, если слово заканчивается символами `.txt`,
- `*)` соответствует любому значению слова. Считается хорошей практикой включать этот шаблон в команду `case` последним, чтобы перехватывать любые значения слова, не соответствующие ни одному из предыдущих шаблонов, то есть чтобы перехватить любые недопустимые значения

Можно объединять шаблоны: `q|Q)`, что значит или `q` в нижнем регистре, или `Q` в верхнем регистре.

1.18. Позиционные параметры

К аргументам строки можно обращаться по номеру: `$1, $2` etc. Даже в отсутствие аргументов переменная `$0` всегда содержит первый элемент командной строки – путь к файлу выполняемой программы.

Замечание

В действительности, если использовать механизм подстановки параметров, можно получить доступ более чем к девяти параметрам. Чтобы указать число больше девяти, следует заключить его в фигурные скобки: например, `${10}`, `${55}` и т.д.

Командная оболочка поддерживает также переменную `$#`, хранящую число аргументов командной строки.

Пример использования позиционных аргументов

```
#!/bin/bash

PROGNAME=$(basename 0$)  # оставляет только имя файла, отрезая путь до файла

if [[ -e $1 ]]; then
    echo -e "\nFile Type: "
    file $1  # определяет тип файла
    echo -e "\nFile Status: "
    stat $1  # выводит информация по состоянию
else
    echo "$PROGNAME: usage: $PROGNAME file" >&2
    exit 1  # выход с ошибкой
fi
```

1.19. Использование позиционных параметров в функциях

Позиционные параметры используются для передачи аргументов не только в сценарии, но и в функции командной оболочки. Пример

```
#!/bin/bash

file_info () {
    if [[ -e $1 ]]; then
        echo -e "\nFile type: "
        file $1
        echo -e "\nFile status: "
        stat $1
    else
        echo "$FUNCNAME: usage: $FUNCNAME file" >&2
        return 1
    fi
}
# вызов функции
file_info ./read_file_gen.py
```

Благодаря этому появляется множество возможностей писать полезные функции для `.bashrc`.

1.20. Обработка позиционных аргументов скопом

Иногда бывает необходимо выполнить операцию сразу со всеми позиционными параметрами. Например, может понадобиться написать обертку для некоторой программы, то есть сценарий

или функцию, упрощающие запуск этой программы. Обертка принимает список непонятных для нее параметров командной строки и просто передает его обернутой программе.

Для этой цели командная оболочка предоставляет специальных параметра. Они оба замещаются полным списком позиционных параметров, но имеют некоторые тонкие отличия.

Специальные параметры:

- `$*`: замещается списком позиционных параметров, начиная с `$1`. Если имя параметра `$*` заключить в двойные кавычки, позиционные параметры будут перечислены в списке через первый символ в переменной IFS (по умолчанию пробел), а сам список будет размещен в одной строке и заключен в кавычки,
- `$@`: замещается списком позиционных параметров, начиная с `$1`. Если имя параметра `$@` заключить в двойные кавычки, механизм подстановки заменит его списком позиционных параметров, заключенных в кавычки *по отдельности*.

В большинстве ситуаций предпочтительнее использовать прием с `"$@"`, потому что он сохраняет целостность каждого позиционного параметра.

Пример

```
#!/bin/bash

USER="Leor Finkelberg"
PROGNAME=$(basename $0)
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
    cat <<- _EOF_
        <H2>System Uptime</H2>
        <PRE>$(echo "uptime")</PRE>
        _EOF_
    return
}

report_disk_space () {
    cat <<- _EOF_
        <H2>Disk Space Utilization</H2>
        <PRE>$(df -h)</PRE>
        _EOF_
    return
}

report_home_space () {
    if [[ $(id -u) -eq 0 ]]; then
        cat <<- _EOF_
            <H2>Home Space Utilization (All Users)</H2>
            <PRE>$(du -sh .)</PRE>
            _EOF_
    else
        cat <<- _EOF_
            <H2>Home Space Utilization ($USER)</H2>
            <PRE>$(du -sh .)</PRE>
            _EOF_
    fi
    return
}
```

```
usage () {
    echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
    return
}

write_html_page () {
    cat <<- _EOF_
        <HTML>
            <HEAD>
                <TITLE>$TITLE</TITLE>
            </HEAD>
            <BODY>
                <H1>$TITLE</H1>
                <P>$TIME_STAMP</P>
                $(report_uptime)      # подстановка того, что возвращает функция по POSIX
                $(report_disk_space)  # подстановка того, что возвращает функция по POSIX
                $(report_home_space)  # подстановка того, что возвращает функция по POSIX
            </BODY>
        </HTML>
        _EOF_
    return
}

interactive=
filename=

# цикл продолжается, пока позиционный параметр $1 не получит пустое значение
while [[ -n $1 ]]; do  # сейчас $1 это, например, -f
    case $1 in
        -f | --file)        shift  # теперь $2 становится $1
                            filename=$1  # здесь $1 это имя файла
                            ;;
        -i | --interactive) interactive=1
                            ;;
        -h | --help)        usage  # вызов функции
                            exit
                            ;;
        *)                  usage >&2
                            exit 1  # выход с ошибкой
                            ;;
    esac
    shift  # чтобы гарантировать завершение цикла
done

# интерактивный режим
if [[ -n $interactive ]]; then
    while true; do
        read -p "Enter name of output file: " filename
        if [[ -e $filename ]]; then
            read -p "'$filename' exists. Overwrite? [y/n/q] > "
            case $REPLY in
                Y|y)    break
                        ;;
                Q|q)    echo "Program terminated."
                        exit
                        ;;
                *)      continue
                        ;;
            esac
        fi
```

```
        break  # если файл не существует, выбросить из цикла
    done
fi

if [[ -n $filename ]]; then
    if touch $filename && [[ -f $filename ]]; then
        write_html_page > $filename  # перенаправить вывод в заданный файл
    else
        echo "$PROGNAME: Cannot write file '$filename'" >&2
        exit 1
    fi
else
    write_html_page  # вывести в терминал
fi
```

1.21. Цикл for

Команду `for` удобно использовать в командной строке

```
$ for i in A B C D; do echo $i; done
# выведет
A
B
C
D
```

По-настоящему мощной особенностью `for` является разнообразие способов формирования списка слов. Например, можно использовать подстановку в фигурных скобках

```
$ for i in {A..D}; do echo $i; done
```

Или подстановку имен файлов

```
$ for i in read*.sh; do echo $i; done
# выведет
read-ifs.sh
read-integer.sh
read-multiple.sh
read-secret.sh
read-validate.sh
```

или подстановку команд

```
...
for file in $(ls); do
    printf "File: %s;\t\tCount: %s\n" $file $(echo $file | wc -c)
done
```

Здесь `$(ls)` возвращает список файлов, по которому можно «пробежаться» переменной цикла `file`.

Если необязательный компонент *коллекции* в команде `for` отсутствует, она по умолчанию обрабатывает позиционные аргументы, переданные при вызове сценария

simple_for.sh

```
#!/bin/bash

# здесь переменная цикла file поочередно принимает имена файлов, переданные при вызове сценария
for file; do
```

```
    if [[ -e $file ]]; then  # если файл существует, то ...
        line=$(grep -inE 'done' $file)
        echo $line
    fi
done
```

Этот сценарий можно было бы вызвать так

```
$ ./simple_for.sh file_name1.txt file_name2.txt
```

1.22. Цикл for в стиле языка C

Пример

```
#!/bin/bash

for (( i=0; i<5; i+=1 )); do
    echo $i
done
```

Здесь `i=0` инициализирует переменную, `i<5` указывает условие останова и `i=i+1` обеспечивает изменчивость переменной.

1.23. Подстановка параметров

Некоторые формы подстановки параметров помогают решать проблемы с несуществующими, или пустыми, переменными: `${parameter:-default_value}`. Если параметр неопределен (то есть отсутствует) или содержит пустое значение, механизм подстановки вернет значение указанного слова. Если параметр непустой, механизм подстановки вернет значение параметра

```
$ foo=  # пустое значение
$ echo ${foo:-"default_value"}  # default_value
$ foo=bar
$ echo ${foo:-"default_value"}  # bar
```

Вот еще один вариант подстановки, где вместо дефиса используется знак «=»: `${parameter:=value}`. Если параметр не определен или содержит пустое значение, механизм подстановки вернет значение указанного слова и дополнительно присвоит его параметру. Если параметр непустой, механизм подстановки вернет значение параметра

```
$ foo=
$ echo ${foo:="value"}  # value
$ foo=bar
$ echo ${foo:="value"}  # bar
$ echo $foo  # bar
```

1.24. Получение имен переменных

Командная оболочка может возвращать *имена переменных*: `${!prefix*}`, `${!prefix@}`. Эти две формы подстановки возвращают имена существующих переменных, начинающиеся с указанного префикса. Согласно документации `bash`, обе формы работают совершенно одинаково.

Следующая команда выводит список всех переменных окружения с именами, начинающимися с `BASH`

```
$ echo ${!BASH*}  # BASH BASHOPTS BASHPID BASH_ALIASES BASH_ARGC...
$ echo ${!USER*}  # USERDOMAIN USERNAME USERPROFILE
```

1.25. Операции со строками

Существует множество форм подстановки, которые можно использовать для работы со строками. Многие из них хорошо подходят для операций с путями. Форма `${#parameter}` вернет длину строки, содержащуюся в указанном параметре. Обычно роль параметра играет строка, но если передать `@` или `*`, то механизм подстановки вернет число позиционных параметров

```
$ foo="This string is long."
$ echo "'$foo' is ${#foo} character long."  # 'This string is long.' is 20 characters long.
```

Следующая форма подстановки

```
${parameter:shift}
${parameter:shift:length}
```

используется для извлечения фрагмента строки, содержащейся в параметре. Извлечение начинается с указанного смещения от начала строки и продолжается до конца строки, если не указано длина.

Если указать отрицательное число, его отсчет начнется с конца строки вместо начала. Обратите внимание, что отрицательному значению должен предшествовать пробел, чтобы предотвратить путаницу с формой `${parameter:-value}`.

Следующие две формы

```
${parameter#pattern}
${parameter##pattern}
```

возвращают значение параметра, удаляя из него начальную часть, определяемую указанным шаблоном. В шаблоне допускается использовать групповые символы: например, те, что используются в подстановке путей. Эти две формы отличаются тем, что форма `#` удаляет кратчайшее совпадение, тогда как форма `##` удаляет самое длинное совпадение

```
$ foo="file.tar.gz"
$ echo ${foo#*\.}  # tar.gz
$ echo ${foo##*\.}  # gz
```

Следующие две формы

```
${parameter%pattern}
${parameter%%pattern}
```

действуют так же, как формы `#` и `##`, представленные выше, но удаляют текст с конца строки, содержащейся в параметре

```
$ foo="file.tar.gz"
$ echo ${foo%\.*}  # file.tar
$ echo ${foo%%\.*}  # file
```

Следующие формы

```
${parameter/pattern/string}
${parameter//pattern/string}
${parameter/#pattern/string}
${parameter/%pattern/string}
```

выполняют поиск с заменой в содержимом указанного параметра. Если в параметре будет найдено совпадение с шаблоном, который может содержать групповые символы, это совпадение будет заменено содержимым указанной строки. Первая форма заменит только первое совпадение с шаблоном. Форма `//` заменит все найденные совпадения. Форма `/#` выполняет замену, только если совпадение с шаблоном найдено в самом начале строки, а формы `/%` выполняет замену, только если совпадение найдено в конце строки. Часть `/string` можно опустить, и тогда совпавший фрагмент будет удален.

Пример

```
#!/bin/bash

for i; do
    if [[ -r $i ]]; then
        max_word=
        max_len=
        for j in $(strings $i); do
            len=${#j}  # <<---
            if (( len > max_len )); then
                max_len=$len
                max_word=$j
            fi
            ...
```

Если, к пример, требуется убрать нули в начале имени файла, то

```
$ file="00test_file.txt"
$ echo ${file/#00}  # test_file.txt
```

Пример использования составного оператора (++)

```
for ((i=0; i<=20; ++i)); do
    if [[ $((i % 5)) -eq 0 ]]; then
        printf "<%d> " $i
    else
        printf "%d " $i
    fi
done
```

Бывает полезным тернарный оператор

```
$ a=0
$ echo $((a<1?++a:--a))  # 1
$ echo $((a<1?++a:--a))  # 0
```

Обратите внимание, что прямое присваивание в этом операторе считается недопустимой операцией. Эту проблему можно решить, заключив выражения присваивания в круглые скобки

```
$ a=0
$ echo $(( a<1?(a+=1):(a-=1) ))
```

1.26. Массивы

Значения элементам массивов можно присваивать одним из двух способов. Присваивание одиночных значений осуществляется с использованием следующего синтаксиса

```
name[index]=value
```

Присвоить сразу несколько значений можно с использованием следующего синтаксиса

```
name=(value1 value2 ...)
```

Обратиться к элементу массива можно так

```
$ echo ${array[4]}
```

Вывести содержимое всего массива можно, используя индексы `*` и `@`

```
$ read -r -a words <<< "This is a simple string."
$ echo ${words[*]}  # This is a simple string.
$ echo ${words[@]}  # This is a simple string.
```

Для того чтобы вывести массив по элементам, разделенным не по пробелам по кавычкам, следует воспользоваться конструкцией

```
$ animals=("a dog" "a cat" "a fish")
$ for i in "${animals[@]}"; do
>    echo $i
>done
# выведет
a dog
a cat
a fish
```

В общем случае конструкция `"${array[@]}"` гораздо полезнее.

Определить число элементов в массиве можно так

```
$ cae_packages=("Ansys MAPDL" "Nastran" "Abaqus")
$ echo ${#cae_packages[@]}  # 3
```

Удалить массив можно командой `unset`

```
$ unset cae_packages  # удалить массив
$ unset 'cae_packages[2]'  # удалить один элемент
```

При удалении одного элемента массива нужно элемент массива заключить в кавычки, чтобы предотвратить подстановку путей оболочки.

1.27. Группы команд и подоболочки

Группы команд оформляются с помощью `{...}`, а подоболочки – с помощью `(...)`. Все команды, входящие в группу, выполняются в текущей оболочке, подоболочка (как можно догадаться из названия) выполняет свои команды в дочерней копии текущей командной оболочке. Это означает, что в момент запуска подоболочки создается *копия текущей оболочки* и передается новому экземпляру оболочки. Когда подоболочка завершается, ее копия окружения уничтожается, соответственно теряются любые изменения в окружении подоболочки (включая значения переменных).

Поэтому если нет прямой необходимости в использовании подоболочки, предпочтительнее использовать *группы команд*. Группы команд выполняются быстрее и требуют меньше памяти.

Замечание

Конвейеры команд всегда выполняются в *подоболочке*!

Командная оболочка поддерживает экзотическую форму подстановки, которая называется *подстановкой процессов*. Подстановка процессов оформляется двумя способами: для процессов, отправляющих результаты в *стандартный вывод*

```
<(list)
```

и для процессов, принимающих данные через стандартный ввод

```
>(list)
```

Подстановка процессов позволяет интерпретировать вывод подоболочки как обычный файл и осуществлять его перенаправление. Так как это форма подстановки, всегда можно узнать действительное подставляемое значение

```
$ echo <(echo "foo")  # /dev/fd/63
```

1.28. Ловушки

В больших и сложных программах процедура обработки сигналов может оказаться весьма кстати. Проектируя большие сценарии, важно предусмотреть их реакцию на неожиданный выход пользователя из системы или выключение компьютера во время их выполнения. Программы, представляющие эти процессы, могут выполнять некие действия, гарантирующие корректное завершение с сохранением необходимых данных.

Пример

```
#!/bin/bash

# ловушка, которая реагирует на
trap "echo 'I am ignoring you.'" SIGINT SIGTERM

for i in {1..5}; do
    echo "Iteration $i of 5"
    sleep 5
done
```

Этот сценарий определяет ловушку, которая будет выполнять команду `echo` в ответ на сигналы `SIGINT` и `SIGTERM`, получаемые сценарием во время выполнения.

При попытке прервать работу программы в терминал будет выводится сообщение «`I am ignoring you.`».

Иногда бывает непросто сформировать строку с требуемой последовательностью команд, поэтому на практике в качестве команды часто использует функции. Следующий пример демонстрирует применение разных функций для обработки разных сигналов

```
#!/bin/bash

exit_on_signal_SIGINT () {
    echo "Script interrupted." 2>&1
    exit 0  # << для того чтобы прекратить выполнение программы!
}

exit_on_signal_SIGTERM () {
    echo "Script terminated." 2>&1
    exit 0  # << для того чтобы прекратить выполнение программы!
}

# ловушки
trap exit_on_signal_SIGINT SIGINT
trap exit_on_signal_SIGTERM SIGTERM
```

```
for i in {1..5}; do
    echo "Iteration $i of 5"
    sleep 5
done
```

Здесь в каждой ловушке используется своя функция, которая будет вызвана для обработки конкретного сигнала. Обратите внимание на включение команды `exit` в обе функции обработки сигналов. Без этого сценарий продолжил бы выполняться после завершения функция.

2. Язык обработки шаблонов awk. Базовые концепции

Программа на `awk` (программа, набранная в командной строке или в файле программы) состоит из одной и более строк, в которых содержится *шаблон* и/или *действие* в следующем формате: `шаблон { действие }`.

Шаблон выбирает строки из ввода. Утилита `awk` выполняет действие над всеми строками, выбранными шаблоном. Фигурные скобки, в которые заключено действие, позволяют `awk` отличить его от шаблона.

Если программная строка не содержит шаблона, `awk` выбирает из ввода все строки. Если программная строка не содержит действия, `awk` копирует выбранные строки на стандартный вывод.

Два уникальных шаблона, `BEGIN` и `END`, выполняют команды перед тем, как утилита `awk` приступит к обработке ввода, и после того, как она завершит эту обработку. До начала обработки всего ввода утилита `awk` выполняет *действия*, связанные с шаблоном `BEGIN`, а по окончании обработки – *действия*, связанные с шаблоном `END`.

`awk` поддерживает несколько специальных переменных (`FILENAME`, `NR`, `FS`, `NF` и пр.). Рассмотрим для примера переменную `NR`, смысл которой зависит от контекста: если переменная `NR` используется в контексте обычного действия, то она содержит номер записи (строки), принадлежащей текущей строке. Однако, если эта переменная используется в контексте шаблона `END`, то она связывается с номером последней строки.

Значение разделителей полей в выводе можно изменить, присвоив его значение переменной `OFS`

```
$ cat ofs_demo
BEGIN { OFS = "\t" }
    {
    if ($1 ~ /ply/) $1 = "plymouth"
    if ($1 ~ /chev/) $1 = "chevrolet"
    print
    }
$ awk -f ofs_demo file_name.txt
```

Пример использования конструкций `BEGIN`, `END` и `printf`

```
awk -F ',' 'BEGIN { print "### PROGRAM ###" } { sum += $3 } END { printf "result=%s", sum }'
    ram_price.csv
```

Здесь печатается строка «`### PROGRAM ###`», потому что она связана с *шаблоном* `BEGIN`. Затем вычисляется сумма 3-его столбца и наконец выполняется строка «`result=%s`», так как она связана с *шаблоном* `END`.

3. Вспомогательные конструкции Vim

3.1. Регистры Vim

Регистры Vim – это всего лишь контейнеры для хранения текста. Их можно использовать как своеобразные буферы обмена копируя текст в регистры и вставляя его из регистров, или для записи макросов, сохраняя в них последовательности нажатий на клавиши.

3.2. Макросы

Клавиша q действует как своеобразная кнопка «Запись» и одновременно как кнопкаа «Стоп». Чтобы начать запись последовательности нажатий на клавиши, необходимо ввести q{register}, указав имя регистра, где будет сохранен макрос.

Посмотреть содержимое регистра "a можно с помощью :reg

```
:reg a
```

Команда @{register} служит для выполнения содержимого указанного регистра. То есть, если макрос записывался в регистр "a, то вызывать нужно макрос из того регистра как @a.

Макрос записанный, например в регистр "a, можно выполнить заданное число раз, указав число повторений, например так 25@a.

Если требуется, чтобы макрос выполнялся для каждой строки из заданного диапазона, то следует использовать команду :normal[6]

```
:normal! 25@a
```

3.3. Базовые концептуальные конструкции

Перейти в ВИЗУАЛЬНЫЙ режим, выделить 2 «слова» (фрагмент текста) и скопировать в безымянный регистр ("")

```
v2wy
```

Вставить данные из регистра (именованного, неименованного и др.), не покидая режима вставки, можно с помощью сочетания клавиш <C-r>{registr} (т.е. Ctrl+C и имя регистра). Пусть в *регистре выделенного фрагмента* ("*) хранится какая-нибудь строка. С помощью команды A переходим в режим ВСТАВКИ в конец строки. Теперь строку и регистра можно вставить сочетанием <C-r>. После этого под курсором появится символ ". Клавиша * заменит символ " содержимым регистра "*.

Фрагмент строки можно вставить слева от курсора с помощью команды P или справа от курсора с помощью команды p.

Если же копировалась строка *целиком*, то команда p вставляет строку под текущей, а команда P – над текущей.

Удалить слово, находясь внутри слова (т.е. на любой позиции в пределах слова), можно так

```
diw
```

Аналогично можно удалить слово и перейти в режим ВСТАВКИ, находясь внутри слова

[6] Восклицательный знак для игноирования пользовательских настроек

```
ciw
```

Используя подобную конструкцию, можно удалить текст, заключенный в кавычки, и тут же перейти в режим ВСТАВКИ

```
ci"
```

Захватить текущее слово в регистр `"a`

```
"ayiw
```

Вырезать текущую строку в регистр `"b`

```
"bdd
```

Захватить последовательность символов, заключенную в кавычки, в регистр `"b`

```
"byi'
```

Захватить слово в неименованный регистр (`""`)

```
""yiw
```

Захватить последовательность символов от текущего положения курсора до конца строки в *регистр захвата* (`"0`). Все что копируется с помощью `y` попадает в регистр захвата

```
y$
```

Вставить данные из регистра захвата

```
"0p
```

Захватить в *регистр выделенного фрагмента* (`"*`) несколько слов

```
v3wey
```

Вставить данные из именованного регистра `b`

```
"bp
```

Открыть файл на заданной строке

```
$ vim +10 file_name.txt
```

Открыть файл и найти заданный шаблон

```
$ vim +/pattern file_name.txt
```

Выполнить одну или несколько команд редактора `ex` при старте можно с помощью опции `-c` (`Vim` принимает до 10 штук).

Например, чтобы скопировать несколько строк из файла в буфер, а затем закрыть можно использовать ключ `-c` несколько раз

```
$ vim -c %y -c q file_name.py
```

Запустить `Vim`, открыв два файла, с вертикальным разбиением

```
$ vim -O2 file_name1.txt file_name2.txt
```

4. Полезные конструкции оболочки bash

Найти в корневом каталоге и всех подкатлогах (/), обычные файлы (-type f), измененные за последний день (-mtime -1), за исключением тех файлов, у которых есть суффикс .o (! -name '*.o')

```
find / -type f -mtime -1 ! -name '*.o'
```

Вывести список поддиректорий (-type d) текущей директории (.), в именах которых встречается подстрока 'cheat', без учета регистра (-iname)

```
find . -type d -iname 'cheat*'
```

Вывод имен файлов и удаление файлов с именами core или junk из рабочего каталога и всех его подкаталогов (круглые скобки обязательно отделяются пробелами)

```
find . \( -name core -o -name junk \) -print -exec rm {} \;
```

Скопировать все csv-файлы из родительской директории (..) в текущую (.)

```
cp -ip ../*.csv
```

Скопировать файл из родительской директории в текущую директорию

```
cp -ip ../Cheat_sheet_Git/cheat_sheet_git.tex .
```

Скопировать одну директорию в другую

```
cp -rip ../Cheat_sheet_Git/style_packages/ .
```

Переименовать файл

```
mv cheat_sheet_git.tex cheat_sheet_bash.tex
```

Найти все файлы с расширением *.csv и выбрать из них те, в которых содержится строка 'state' (для каждого файла, отвечающего поисковому шаблону, запускается свой процесс)

```
find . -name '*.csv' -exec grep -niE 'state' {} \;
```

Вывести список файлов из текущей директории и всех поддиректорий

```
ls -l *
```

Найти среди файлов с расширением *.py те, в именах которых есть подстрока 'spark' (используется конвейер)

```
ls -l *.py | grep -iE 'spark'
```

Найти файлы с расширением *.py и к каждому из них применить команду grep, которая будет искать в файле подстроку 'argparse' без учета регистра, с выводом номера строки, на которой она нашла искомую строку по регулярному выражению 'argparse' (работает медленно, так как для каждого файла, отвечающего поисковому шаблону, запускается свой процесс)

```
find . -maxdepth 1 -name '*.py' -exec grep -inE 'argparse' {} \;
```

Альтернативный вариант с использованием xargs (работает значительно быстрее варианта с -exec). Команда xargs превращает стандартный ввод в командные строки, интерпретируя каждое строковое значение, отделенное пробельными символами в качестве отдельного аргумента.

Затем она создает командную строку из команды и ряда аргументов. Когда командная строка при добавлении еще одного аргумента достигает максимально возможной длины, `xargs` запускает созданную командную строку. Если ввод продолжается, `xargs` повторяет процесс построения командной строки и ее запуска. Этот процесс продолжается до тех пор, пока не будет прочитан весь ввод

```
find . -maxdepth 1 -name '*.py' | xargs grep -inE 'argparse'
```

Для того чтобы увидеть как `xargs` строит командую строку, можно добавить флаг `-p` (интерактивный режим), например,

```
find . -name '*.py' | xargs -p grep bash
```

Найти в файлах с расширением `*.tex` строку `'section'` без учета регистра и вывести три строки контекста

```
find . -name '*.tex' | xargs grep -iE 'section' -3
```

Вывести список пакетов, в именах которых встречается подстрока `'python'` с контекстом `'sql'`

```
conda list | grep -inE 'python.*sql'
```

Получить информацию о доступном метсе на диске

```
df -h
```

Скачать файл с тем же именем, что на удаленном репозитории

```
curl -O http://merionet.ru/yourfile.tar.gz
```

Скачать файл с удаленного репозитория с новым именем и/или путем

```
curl -o newfile.tar.gz http:// merionet.ru /yourfile.tar.gz
```

Возобновить прерванную загрузку с того места, где она остановилась

```
curl -C - -O http://merionet.ru/yourfile.tar.gz
```

Скачать несколько файлов

```
curl -O http://merionet.ru/info.html -O http://wiki.merionet.ru/about.html
```

Вывести имена пакетов, имеющих отношение к `Python`. Ключ `-n` для подавления автоматического вывода всех строк

```
conda list | sed -n '/python/p'
```

Вывести первые десять строк из списка пакетов, который возвращает `conda`

```
conda list python | sed -n '1,10p'
```

Заменить в сообщении фиксации `git` «Python» на «Fortran»

```
git show -s HEAD | sed '1,$s/Python/Fortran/g'
```

Удалить строки в сообщении фиксации, начиная с первой и до первой пустой строки (`/^$`)

```
git show -s HEAD | sed '1,/^$/d'
```

Заменить в сообщении фиксации `git` «PostgreSQL» на «MySQL» и результат записать в файл

```
git show -s HEAD~2 | sed '1,$s/cookiecutter/somthingelse/g' > output.txt
```

4.1. Переадресация ввода-вывода

Перенаправить *стандартный поток вывода* данных (дескриптор файла 1) и *стандартный поток вывода ошибок* (дескриптор файла 2), которые возвращает `conda` с захватом всех пакетов, в именах которых встречается подстрока `'python'`, в файл с именем `test_file.log` (временный поток вывода данных `&1`). Если команда вернет ошибку, то сообщение ошибки перепишет содержимое файла `test_file.log`

```
conda list | grep -inE 'python' > test_file.log 2>&1
```

Более короткий вариант рассмотренной выше конструкции

```
conda list | grep -inE 'python' &> test_file.log
```

Присоединить стандартный поток вывода данных и стандартный поток вывода ошибок к содержимому файла. Конструкция `rm df` возвращает сообщение об ошибке «`rm: cannot remove 'df': No such file or directory`», которое можно добавить в файл

```
rm df &>> test_file.txt
```

Найти в текущей директории файлы с расширением `.tex`, в именах которых встречается подстрока «`bash`», перенаправить эти файлы утилите `grep`, которая будет искать в теле файлов строки, в которых встречается подстрока «`vim`», причем стандартный поток вывода ошибок перенаправляется в «черную дыру»

```
find . -name '*bash*.tex' | xargs grep -inE 'vim' 2>/dev/null
```

Записать строку в файл и дополнительно вывести строку в терминал

```
echo 'test string' | tee ./logfile.txt
```

4.2. Информация об использовании дискового пространства

Вывести размер директорий в Мегабайтах (`M`)

```
du -BM
```

Вывести итоговый размер директории (`c`) в Мегабайтах (`M`)

```
du -cBM
```

Перевести размеры директорий в понятный человеку формат

```
du -h
```

Вывести информацию об указанной директории в дружественном формате

```
du -sh style_packages/
```

Вывести размер папок текущей директории, не погружаясь глубже корневых папок

```
du -h -d 1
```

или так

```
du --max-depth=1 -h
```

Вывести информацию об использовании дискового пространства иерархией каталога с подсчетом общего размера каталога и перенаправлением стандартного потока вывода ошибок в «черную дыру»

```
du -ch -d 1 2>/dev/null
```

4.3. Информация о файлах и операциях

Чтобы узнать используется в файле табуляция или нет достаточно прочитать его в терминал командой `cat` с ключом `-A`. Символ табуляции в тексте будет представлен парой символов `^I` (что означает «Ctrl+I»). К слову, символ `$` отмечает истинный конец строки, помогая увидеть дополнительные пробелы в конце строки

```
cat -A file_name.txt
```

Вывести список удаляемых файлов

```
$ echo rm -f *.txt
```

Вывести час создания файла

```
$ stat -c %y file_name.txt | cut -c 12-13  # 17
```

4.4. Управление выводом

Вывести элементы 3-его столбца, значения которых превышают 1000000

```
awk -F ',' '{ if ($3 < 1000000) print $3 }' file_name.csv
```

Прочитать только те строки из файла, которые имеют отношение к «python»

```
awk '/python/' file_name.txt
```

Прочитать только те строки, которые отвечают заданному шаблону, и вывести 1-ый и 2-ой столбцы

```
awk '/python/ { print $1, $3 }' file_name.txt
```

Вывести только те строки, которые начинаются с подстроки «ipy»

```
awk '$1 ~ /^ipy/' file_name.txt
```

Вывести только строки полей с номерами 4 и 5; при этом нужно вывести только те строки, в которых 4-ое поле начинается с «"Ham»

```
awk -F',' '$4 ~ /^"Ham/ { print $4, $5 }' titanic_train.csv
```

Вывести только те строки, значения которых во 2-ом поле равны 1; напечатать 4-ый и 5-ый столбцы

```
awk -F ',' '$2 == 1 { print $4, $5 }' titanic_train.csv | head -n 5
```

Выбрать строки из диапазона строк, который отвечают заданным шаблонам

```
awk -F ',' '/Graham/, /Palsson/' titanic_train.csv
```

Просуммировать все элементы 3-его столбца и вывести результат

```
awk -F ',' '{ sum += $3 } END { print sum }' file_name.csv  # 4.89428e+08
```

Вывести группу строк файла на основе их номеров строк

```
awk -F ',' 'NR == 2, NR == 4' file_name.csv
```

Выбрать только те строки, в которых встречается подстрока «Mrs.» и записать их в файл `mrs_output.txt`

```
awk -F ',' '/Mrs./ { print > "mrs_output.txt" }' titanic_train.csv
```

Найти idx-файл и отрезать в его имени часть «pack-»

```
ls -l .git/objects/pack/ | grep -iE '*.idx' | awk -F ' ' '{ print $9 }' | sed 's/.*-//'
```

Прочитать файл `master` в стандартный поток вывода, оставить только последнюю строку и вывести 1-ый столбец, а также все столбцы, начиная с 8-ого

```
cat .git/logs/refs/heads/master | tail -1 | cut -d ' ' -f1,8-
```

5. Работа с архивами

Собрать csv-файлы текущей директории в *несжатый* tar-архив

```
tar -cvf only_csv_files.tar *.csv
du -sh only_csv_files.tar # 3,8M  only_csv_files.tar
```

Собрать csv-файлы текущей директории в *сжатый* tar-архив

```
tar -czvf only_csv_files_zip.tar.gz *.csv  # 920K only_csv_files_zipp.tar.gz
```

Вывести список файлов tar-архива

```
tar -tf only_csv_files_zip.tar.gz
```

Список литературы

1. Собель М. Linux. Администрирование и системное программирование. 2-е изд. – СПб.: Питер, 2011. – 880 с.
2. Шоттс У. Командная строка Linux. Полное руководство. – СПб.: Питер, 2017. – 480 с.
3. Роббинс А., Ханна Э., Лэмб Л. Изучаем редакторы vi и Vim, 7-е издание. – СПб.: Символ-Плюс, 2013. – 512 с.