Скрипты Bash

Овсянников А.П., Овсянникова Т.В.

4 октября 2018 г.

1 Bash. Скрипты

Для работы с Unix в консольном режиме используются команды операционной системы.

Например,

```
ls -l
cp myfile youfile
cd /home/anywere
rm -r unused_catalog
```

и другие.

Если необходимо несколько раз выполнять похожий или сложный набор команд, легче записать эту последовательность в скрипт и выполнять скрипт как запускаемую программу.

Скрипт и есть программа. Только все записи в этой программе выполняеются специальным интрепретатором команд. В нашем случае это интерпретатор *bash*.

1.1 Переменные, параметры

Bash использует различные переменные.

Называя переменные следует придерживаться правила, что имя должно состоять из строчных или заглавных латинских символов и знака подчеркивания.

```
myperem=3
other_Perem="stroka␣long␣long"
PATH=$PATH:/home/secretdir
echo $myperen$other
```

В данном примере видно, что переменные нетипизированы как это, например, в языке C. Им могут быть присвоены любые значения. Эти значения трактуются как текстовые,

если не используются специальные конструкции для того чтобы интерпретировать их иначе.

Для доступа к содержимому переменной необходимо вначале записать знак доллара.

В двух последних строках видно, что строки «склеиваются». В результате значением переменной $PATH$ становится строка с присоединенной частью *:/home/secretdir* к предыдущему значению $PATH$, а *echo* напечатает строку, состсоящую из значений *myperem* и *other*.

Если необходимо выполнить арифметическое выражение над строками, которые записаны как числа, то используется следующая конструкция:

```
$a=89
$b=100
c=$(($a+$b))
echo $c
```

Каждой запускаемой программе, в том числе и скрипту может быть передан набор параметров (после имени запускаемой программы). Например,

```
./myprog param1 param2 123
```

Операционная система помещает их в специальные переменные, и эти переменные становятся доступны скрипту или программе. Это переменная @, $1, $2 ...

Пример скрипта, использующего параметры:

```
#!/bin/bash

echo $# #количество параметров
echo $@ #массив параметров
# прверка: ввели аргументы или нет
if [ $# -lt 1 ]
then
   echo Аргументы!
   exit 1 # закрыть сеанс shell
fi

shift 2 # вынули и выбросили
```

Задача bash.1. Написать скрипт, который проверяет количество введенных численных пармеров. Если из 3, то печатает их среднее арифметическое.

Задача bash.2. Написать скрипт, который получает имена файло первым и вторым параметром. Затем соединяет содержимое этих файлов в результирующий, имя которого сотосит из имен введенных параметров.

1.2 Работа с файлами.

Чаще всего скрипты используются, чтобы автоматизировать работу с файлами. Командами *bash* можно копировать, перемещеть, создавать, удалять файлы. Можно анализировать имена файлов. И многое другое .

Пример,

```
#!/bin/bash
if [ $# -lt 2 ]
then
   echo Аргументы!
   exit 1 # закрыть сеанс shell
fi

# Предполагаем, что два первых аргумента - имена файлов
# Проверяем, а существуют ли первый

if [ -e $1 ]
then
  echo Есть $1
fi
# Кто хозяин файла

master=`stat $1 --print=%U`
echo Хозяин - $master

# Время изменения
time=`stat $1 --print=%x`
echo Время - $time

# Проверяем, а существуют ли оба сразу

if [ -e $1 -a -e $2 ]
then
  echo Оба есть: $1 и $2
fi

# Проверяем, может это каталог

if [ -d $1 ]
then
```

```
  echo $1 - каталог
# А если не каталог, то не пустой ли он
else
  if [ -s $1 ]
  then
    echo Там что-то есть
  fi
  if [ -f $1 ]
  then
    echo Обычный файл
  fi
fi

# Некоторые сравнения двух файлов

if [ $1 -ot $2 ]
 then
   echo $1 создали раньше $2
fi

differ=`cmp -b $1 $2`

if [ -z "$differ" ]
then
  echo $1 и $2 одинаковые
else
  echo $1 и $2 разные
fi
```

Задача bash.3. Написать скрипт, который получает два параметра: первый - имя каталога, второй - имя файла. Если каталог не существует, скрипт создает его. Если файл существует, и его владелец и владелец каталога совпадают, файл копируется в каталог. Если не хватает скриптов, файл или каталог не существуют, владельцы не совпадают: обо всем скрипт должен оповещать.

Задача bash.4. Написать скрит, кторый получает два параметра (имена файлов), проверяет, являются ли эти файлы обычными файлами. Проверяет, какой из них новее. Если новый файл отличается от старого и его размер не 0, замещает старый новым.

1.3 Преобразование имен файлов

Bash позволяет гибко работать со строковыми переменными, списками строк. Например со списками имен файлов. Можно искать различные записаи по шаблону, заменять записи в соответствии с шаблонами и др.

Пример, работы со списками и отдельными строчками.

```
#!/bin/bash
if [ $# -lt 1 ]
then
   echo Аргументы!
   exit 1 # закрыть сеанс shell
fi

# grep позволяет отфильтровать записи  по шаблону (здесь .txt)
ls $1 |grep .txt

#  Переменной frst присваивается результат выполнения команды
frst=‘ls $1 | grep .txt‘

# В frst оказались слова, разделенные пробелами - получился список
echo $frst

# Цикл for выбирает по-очереди все записи из списка,
# присваивает каждую переменной name (может как угодно называться),
# и использует это значение для работы в текущей итерации цикла
for name in $frst
do
  echo $name
done

# Необходимо получить список названий файлов, которые
# будут использоваться при копировании или переименовании
#  Для этого воспользуемся консольным редактором sed
# sed  получает на вход текст, который может анализировать
# и изменять
# Здесь текст получается как результат работы echo - получаем
# значение переменной $frst.
# Далее sed производит замену.
# Команды sed указываются в одинарных кавычках:
# s - замена
```

```
# s/<что ищем для замены>/<на что заменяем>/g
# g означает, что замена будет во всем полученном sed тексте
# То есть все записи с txt  заменяются на tmp,
# и результат (новая строка) присваивается scnd
scnd=‘echo $frst|sed 's/txt/tmp/g’‘

# Копирование всех файлов, чьи имена оказались в frst
# в новые файлы, чьи имена получены в scnd
for f1 in $frst
do
 for f2 in $scnd
 do
   cp $f1 $f2
 done
done
ls $1
```

Задача bash.5. Написать скрипт, который получает в качестве параметров два имени каталогов. Проверяет каталоги ли это. Далее, в каждом каталоге сравнивает файлы с одинаковыми именами. Если таковые есть, то:

1. если есть одинаковые файлы, создает каталог BKP (разумно проверяя его наличие),
2. если более свежий файл не пустой, то, старый копируется в BKP, а новый на место старого
3. файл в BKP должен заканчиваться на *.bkp.* < дата > - дата копирования
4. в обоих директориях файлы с одинаковыми именами должныи оказаться новыми, а старые в BKP

2 Установка и использование программных пакетов

2.1 Раздельная компиляция программ

Обычно большие программные продукты разбиваются на разные модули, которые нужно будет соединить в одну работающую программу.

Различные модули пишутся и отлаживаются отдельно, но часто используют одни и те же функции. Чтобы не переписывать один и тот же код обычно используют раздельную

компиляцию файлов с реализацией необходимых функций и файлов-программ, использующих эти функции.

Для связи реализации и использования необходим «заголовочный» файл. В языке C и C++ это файлы с .h.

Например.

```
//______add.h_______
/* Описание интерфейса функций и
   общих структур данных
*/
typedef int* Iptr;
int add(int, int);
int print(Iptr, int);

//______Файл реализации (add.c)_________

#include "add.h"
int add(int a, int b){
  return a+b;
};

//_______Использование функции add (test.c):__________

#include <stdio.h>
#include <stdlib.h>
// включение "несистемного" заголовочного файла
#include "add.h"

int main(){

// Использование функции add()
   printf("%d",add(5,6));
};
```

Все эти три файла - это уже **проект**. Чтобы получить из них исполняемый код, нужно скомпилировать и собрать их. Строка, содержащая # **include "add.h"** вставляется в текст и файла реализации, и файла, использующего данную функцию.

```
> gcc -o test test.c add.c
```

Заметим, что заголовочный файл *add.h* в строку компилятора не включается. Компилятор пытается найти его самостоятельно.

Чтобы не компилировать каждый раз файл add.c (код отлажен, и все функции исправно работают) можно получить **объектный код** этого файла. Тогда в строку компиляции включается только объектный код и происходит просто его сборка (линковка) в исполняемый файл.

```
gcc -c add.c
```

В результате получится файл с объектным кодом *add.o*. Тогда строка компиляции будет выглядеть так:

```
> gcc -o test test.c add.o
```

Компилироваться будет только *test.c*. А все остальное будет обрабатывать линковщик.

Если *add.h* находится в текущем каталоге, то компилятор его непременно найдет.

Однако, заголовочные файлы принято сохранять в отдельный каталог, например ***include***. Тогда, необходимо сообщить компилятору место где искать заголовочные файлы. Для этого существует ключ компилятора *gcc* **-I**:

```
> mkdir include
> mv add.h include/
> gcc test.c add.o -I./include -o test
```

2.2 Статические библиотеки.

Заметим, что включать все объектные файлы в строку компилятора-линковщика очень неудобно. Не каждый знает к котором из них находится нужная функция. Гораздо лучше создать библиотечные файлы, куда включаются множество различных уже скомпилированных функций и указать линковщику. чтобы он их включал в исполняемый код.

Статическая библиотека - это файл-архив, который содержит объектный код функций. При этом функции в нем проиндексированы, чтобы облегчить их поиск при загрузке. При создании исполняемого кода весь код статической библиотеки включается в исполняемый файл. Однако нет необходимости еще раз компилировать файл с нужными функциями и помнить в каком именно объектном файле они содержатся.

Интерфейсы всех используемых функций описываются в заголовочных файлах, которые включаются в текст программы. Сами библиотечные файлы также должны быть помещены в отдельные каталоги, например *lib*.

2.2.1 Создание статической библиотеки

1. Написание и отладка программного кода.

   Файл *add.c* уже написан.

2. Компиляция и создание объектного кода

```
> gcc -c add.o
```

3. Создание библиотечного файла и помещение туда файла с объектным кодом

```
> ar rc libadd.a add.o
> ranlib libadd.a
```

4. Перемещение билиотечного файла в папку *lib*

```
> mv libadd.a lib/
```

Удостоверимся, что библиотека находится в каталоге *lib* и попробуем использовать ее функции.

Для указания места поиска библиотеки существует ключ компилятора *gcc* **-L**, а для указания названия библиотеки ключ **-l**. Название библиотеки указывается без префикса lib и без «расширения» *.a*.

```
> gcc test.c -I./include -L./lib -ladd -o test
> ./test
```

2.2.2 Создание и использование динамической(shared) библиотеки

Динамическая библиотека - это созданная специальным образом библиотека, которая присоединяется к результирующей программе в два этапа. На первом этапе линковщик встраивает в программу описания требуемых функций и переменных, которые присутствуют в библиотеке. Сами объектные файлы из библиотеки не присоединяются к программе. Присоединение этих объектных файлов(кодов функций) осуществляет системный динамический загрузчик во время запуска программы. Загрузчик проверяет все библиотеки просоединенные (слинкованные) к программе на наличие требуемых объектных файлов, затем загружает их в память и присоединяет к программе, находящейся в памяти.

Исполняемый файл при этом получается меньше чем при линкове с динамической библиотекой. Одна и та же копия загруженной библиотеки может использоваться несколькими программами. При использовании статической библиотеки каждая загружаемая программа имеет свою копию.

При загрузке в память программы использующей динамическую библиотеку требуется больше времени. Сама копия библиотеки, загруженная в память так и остается в памяти (если программист не принял специальных мер по ее выгрузке).

Динамическая библиотека содержит полноценный исполняемый код, поэтому для ее создания используется компилятор.

1. Написание и отладка программного кода.

   Файл *add.c* уже написан.

2. Компиляция и создание объектного кода независимого от адресов, такая технология получила название PIC - Position Independent Code. В компиляторе gcc данная возможность включается ключом -fPIC

   ```
   > gcc -fPIC -c add.c
   ```

3. Создание библиотечного файла из уже готовых объектных файлов с PIC-кодом. Это делается с использованием компилятора, так как динамическая библиотека - настоящая загружаемая программа

   ```
   > gcc -shared -o libadd.so add.o
   ```

4. Перемещение билиотечного файла в папку *lib*

   ```
   > mv libadd.a lib/
   ```

После того как динамическая библиотека создана и помещена в нужный каталог можно использовать ее функции.

```
> gcc -c test.c -I./include
```

Созданный таким образом объектный файл будет содержать информацию об использованных функциях, но ссылок на сами функции в нем еще нет. Эти ссылки прописывает линковщик.

```
> gcc -o test -L./lib -ladd
```

Однако при попытке запустить это приложение мы получим сообщение, что необходимая динамическая библиотека не найдена. можно посмотреть какие динамические библиотеки требует это приложение:

```
> ldd test
      linux-gate.so.1 (0xffffe000)
      libadd.so => not found
      libc.so.6 => /lib/libc.so.6 (0xb7627000)
      /lib/ld-linux.so.2 (0xb77d8000)
```

Как видно, для найденных библиотек указаны адреса их расположения в памяти. Наша же библиотека libadd.so не найдена.

Динамическая библиотека загружается в память динамическим линковщиком как только она потребовалась какому-то приложению. После этого она остается в памяти. Динамический линковщик должен знать где искать файл динамической библиотеки для загрузки.

Чтобы указать динамическому линковщику где искать библиотечный файл используется переменная среды $LD\_LIBRARY\_PATH$ - в ней прописываются пути поиска всех динамических библиотек, которые не установлены как системные (локальная установка).

```
>LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/home/ov_1111/dim_lib/lib
>export LD_LIBRARY_PATH
```

После этого динамический линковщик при загрузке приложения найдет библиотеку и загрузит ее в память.

```
>./test
>12
```

Задача bash.6. В папке */home/ovsannikova/zdan*1лежат файлы: *mas_print.c, mass.h, mas_get.c, mass_check.c.*

Разместить *mass.h* в каталоге для заголовочных файлов. Создать динамическую библиотеку **libmass.so**, поместить ее в каталог библиотек.
Прописать переменную $LD\_LIBRARY\_PATH$ в *.profile*. Скомпилировать и запустить *mass_check.c*

2.3 Makefile

Понятно, что компиляция и сборка проекта из множества файлов будет сложной. При наличии исходных текстов некоторые из них должны быть помещены в библиотеки статические или днамические, некоторые должны быть использованы для создания запускаемых приложений и т.д. Описать все это словами для человека, который не участвовал в разработке проекта, практически невозможно. Вероятность того, что что-нибудь будет забыто, а что-то перепутано очень велика. И приложение не соберется.

Для решения этой проблемы существует утилита **make** (она существует и для WINDOWS, и для Linux). Эта утилита работает с makefile, в котором специальном образом прописаны правила сборки проекта, зависимости и т.д.

Если просто запустить *make*, то эта программа постарается найти файл *makefile* и выполнить инструкции, которые записаны в нем. Можно указать другой файл с инструкциями:

```
>make -f makefile.drugoy
```

В предыдущих разделах мы уже рассматривали как получается готовое приложение: создавали объектные файлы, собирали библиотечные файлы и потом из всего этого получали готорое приложение.

Желательно, чтобы все это выполнялось автоматически.

Для начала заметим, что в каталоге **src** лежит запакованный файл *myprog.tar*. Этот файл получен с помощью архиватора **tar**. В этот файл были помещены каталоги *include*, *lib*, и тестовый файл *test.c*.

```
>tar -zcvf myprog.tar ./lib ./include test.c
```

Чтобы распаковать его, нужно сделать следующее:

```
>cd src
>tar -zxf myprog.tar
```

Теперь имеются все необходимые файлы для создания проекта. Осталось их собрать.

Напишем инструкции для *makefile*, чтобы сборка прошла автоматически.

Основные части *makefile*:

```
>[цель]:зависимости
>[tabulator]команда
```

Обычно, цель (target) представляет собой имя файла, который генерируется в процессе работы утилиты make. Примером могут служить объектные и исполняемый файлы собираемой программы. Цель также может быть именем некоторого действия, которое нужно выполнить (например, 'clean' - очистить) - абстрактная цель.

- **Цель**, это файл, который мы хотим, в конечном счете, получить после работы **make**.
- **Зависимости (преквизиты)** - это файлы, которые используются как исходные данные для получения цели.
- **Команда** - это действие, выполняемое утилитой make. В правиле может содержаться несколько команд - каждая на свое собственной строке. Важное замечание: строки, содержащие команды обязательно должны начинаться с символа табуляции!

Если бы в текущем каталоге у нас находились файлы: *test.c, add.h, add.c*, то инструкции для *makefile* были бы очень просты:

```
all:
    gcc -o test test.c add.c
```

Здесь цель по умолчанию *all* - она указывается, если никакая другая не указана явно. В этом случае даже если изменения коснулись только test.c будут перекомпилироваться все указанные файлы. Для маленьких проектов это несущественно, а для больших - занимает изрядное количество времени. Поэтому можно записать инструкции по-другому:

```
all: test

test: test.o add.o
   gcc -o test test.o add.o

test.o:test.c
   gcc -c test.c

add.o: add.c
   gcc -c add.c
```

В этом случае объектные файлы получаются отдельно. Если *add.c* не менялся, он не будет перекомпилироваться. Компиляции будут подвергаться только те файлы, которые изменились.

Однако у нас проект более сложный. Заголовочные файлы должны быть помещены в каталог *include*, библиотеки - созданы и помещены в каталог *lib*. Причем, желательно, чтобы эти каталоги не оказались в каком-то подозрительном каталоге src. Будем считать, что их необходимо поместить в домашний каталог пользователя. *makefile* ничего «не знает» домашнем каталоге пользователя. Для указания *makefile* различных параметров: типа компилятора, ключей компиляции, места размещения файлов и т.д. используются переменные *makefile*. Некторые переменные являются встроенными:

CC	компилятор, используемый для сборки (по умолчанию cc)
CFLAGS	ключи компиляции

Можно вводить и собственные переменные. Введем, например, переменную, в которой скажем где размещать каталог с заголовочными файлами.

```
TARG_INC=../include/
all: tt

tt: test.o add.o
        gcc -o tt test.o add.o

test.o: test.c
        gcc -c test.c
add.o: add.c
        gcc -c add.c
```

```
.PHONY: clean install

install:
        mkdir $(TARG_INC)
        cp add.h $(TARG_INC)
clean:

        rm -rf *.o
```

.PHONY - объявляет данную цель абстрактной, то есть результатом ее выполнения является не файл, а действие (удаление ненужных файлов - clean и запись требуемых файлов в нужные каталоги - install).

Запуск очистки:

```
> make clean
```

Будут удалены все ненужные файлы (оканчивающиеся на **.o**).

Запуск установки:

```
>make install
```

Будет создана папка *include* в домашнем каталоге пользователя (без проверки ее существования) и в нее будет скопирован файл *add.h*.

Задача bash.7. В папке **/home/ovsannikova/zdan1**лежит файл: *mas.tar*. Распаковать его в папку **src**. Записать инструкции для трех различных makefile, так, чтобы с помощью **make**.

1. получался исполняемый файл $test\_mass$
2. создавалась бы динамическая библиотека **libmass.so**, и она помещалась бы в каталог библиотек ($\tilde{}/lib$), а файл *mass.h* - в ($\tilde{}/include$).
3. с учетом размещения библиотечного и заголовочного файлов компилировался $mass\_check.c$ и создавался исполняемы файл $mass\_check$

Динамическая библиотека должна быть размещена в файле в соответствии с переменной $LD\_LIBRARY\_PATH$ в *.profile*.