

Shell как язык программирования

Оболочка `bash` поддерживает операторы выбора `if then else` и `case`, а также операторы организации циклов `for`, `while`, `until`, благодаря чему она превращается в мощный язык программирования.

Операторы `if` и `test` (или `[]`)

Конструкция условного оператора в слегка упрощенном виде выглядит так:

```
if list1 then list2 else lists fi
```

где `list1`, `list2` и `lists`— это последовательности команд, разделенные запятыми и оканчивающиеся точкой с запятой или символом новой строки. Кроме того, эти последовательности могут быть заключены в фигурные скобки: `{list}`.

Оператор `if` проверяет значение, возвращаемое командами из `list1`. Если в этом списке несколько команд, то проверяется значение, возвращаемое последней командой списка. Если это значение равно 0, то будут выполняться команды из `list2`; если это значение не нулевое, будут выполнены команды из `lists`. Значение, возвращаемое таким составным оператором `if`, совпадает со значением, выдаваемым последней командой выполняемой последовательности.

Полный формат команды `if` имеет вид:

```
if list then list [ elif list then list ] ... [ else list ] fi
```

(здесь квадратные скобки означают только необязательность присутствия в операторе того, что в них содержится).

В качестве выражения, которое стоит сразу после `if` или `elif`, часто используется команда `test`, которая может обозначаться также квадратными скобками `[]`. Команда `test` выполняет вычисление некоторого выражения и возвращает значение 0, если выражение истинно, и 1 в противном случае. Выражение передается программе `test` как аргумент. Вместо того, чтобы писать

```
test expression
```

можно заключить выражение в квадратные скобки:

```
[ expression ] .
```

Заметьте, что `test` и `[]` — это два имени одной и той же программы, а не какое-то магическое преобразование, выполняемое оболочкой `bash` (только синтаксис `[]` требует, чтобы была поставлена закрывающая скобка). Заметьте также, что вместо `test` в конструкции `if` может быть использована любая программа. В заключение приведем пример использования оператора `if`:

```
if [ -e textmode2.htm ] ; then
Is textmode*
else
pwd
fi
```

5.8.3. Оператор `case`

Формат оператора `case` таков:

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

Команда `case` вначале производит раскрытие слова `word` и пытается сопоставить результат с каждым из образцов `pattern` поочередно. После нахождения первого совпадения дальнейшие проверки не производятся, выполняется список команд, стоящий после того образца, с которым обнаружено совпадение. Значение, возвращаемое оператором, равно 0, если совпадений с образцами

не обнаружено. В противном случае возвращается значение, выдаваемое последней командой из соответствующего списка. Следующий пример использования оператора case заимствован из системного скрипта /etc/rc.d/rc.sysinit.

```
case "$UTC" in
yes|true)
CLOCKFLAGS="$CLOCKFLAGS -u";
CLOCKDEF="$CLOCKDEF (utc)";
//
no|false)
CLOCKFLAGS="$CLOCKFLAGS —localtime";
CLOCKDEF="$CLOCKDEF (localtime)";
t i
esac
```

Если переменная принимает значение yes или true, то будет выполнена первая пара команд, а если ее значение равно no или false — вторая пара.

Оператор select

Оператор select позволяет организовать интерактивное взаимодействие с пользователем. Он имеет следующий формат:

```
select name [ in word; ] do list ; done
```

Вначале из шаблона word формируется список слов, соответствующих шаблону. Этот набор слов выводится в стандартный поток ошибок, причем каждое слово сопровождается порядковым номером. Если шаблон word пропущен, таким же образом выводятся позиционные параметры. После этого выщается стандартное приглашение P33, и оболочка ожидает ввода строки на стандартном вводе. Если введенная строка содержит число, соответствующее одному из отображенных слов, то переменной name присваивается значение, равное этому слову. Если введена пустая строка, то номера и соответствующие слова выводятся заново. Если введено любое другое значение, переменной name присваивается нулевое значение. Введенная пользователем строка запоминается в переменной REPLY. Список команд list выполняется с выбранным значением переменной name.

Вот небольшой скрипт:

```
#!/bin/sh
echo "Какую ОС Вы предпочитаете?"
142 Самоучитель Linux для пользователя
select var in "Linux" "Gnu Hurd" "Free BSD" "Other"; do
break
done
echo "Вы бы выбрали $var"
```

Если сохранить этот текст в файле, сделать файл исполняемым и запустить, на экран будет выдан следующий запрос:

Какую ОС Вы предпочитаете?

- 1) Linux
 - 2) Gnu Hurd
 - 3) Free BSD
 - 4) Other
- I?

Нажмите любую из 4 предложенных цифр (1, 2, 3, 4). Если вы, например, введете 1, то увидите сообщение:

Вы бы выбрали Linux

Оператор for

Оператор for работает немного не так, как в обычных языках программирования. Вместо того, чтобы организовывать увеличение или уменьшение на единицу значения некоторой переменной при каждом проходе цикла, он при каждом проходе цикла присваивает переменной очередное значение из заданного списка слов. В целом конструкция выглядит примерно так:

```
for name in words do list done
```

Правила построения списков команд (list) такие же, как и в операторе if.

Пример.

Следующий скрипт создает файлы foo_1, foo_2 и foo_3:

```
for a in 1 2 3 ; do  
touch foo_$a  
done
```

В общем случае оператор for имеет формат:

```
for name [ in word; ] do list ; done
```

Вначале производится раскрытие слова word в соответствии с правилами раскрытия выражений, приведенными выше. Затем переменной name поочередно присваиваются полученные значения, и каждый раз выполняется список команд list. Если in word пропущено, то список команд list выполняется один раз для каждого позиционного параметра, который задан. В Linux имеется команда seq, которая воспринимает в качестве аргументов два числа и выдает последовательность всех чисел, расположенных между заданными. С помощью этой команды можно заставить for в bash работать точно так же, как аналогичный оператор работает в обычных языках программирования. Для этого достаточно записать цикл for следующим образом:

```
for a in $( seq 1 10 ) ; do  
cat file_$a  
done
```

Эта команда выводит на экран содержимое десяти файлов: "file_1",....., "file_10".

Операторы while и until

Оператор while работает подобно if, только выполнение операторов из списка list2 циклически продолжается до тех пор, пока верно условие, и прерывается, если условие не верно. Конструкция выглядит следующим образом:

```
while list1 do list2 done
```

Пример:

```
while [ -d mydirectory ] ; do  
ls -l mydirectory >> logfile  
echo -- SEPARATOR — >> logfile  
sleep 60  
done
```

Такая программа будет протоколировать содержание каталога mydirectory ежеминутно до тех пор, пока каталог существует.

Оператор until аналогичен оператору while:

```
until list1 do list2 done.
```

Отличие заключается в том, что результат, возвращаемый при выполнении списка операторов list1, берется с отрицанием: list2 выполняется в том случае, если последняя команда в списке list1 возвращает ненулевой статус выхода.

Функции

Синтаксис

Оболочка bash позволяет пользователю создавать собственные функции. Функции ведут себя и используются точно так же, как обычные команды оболочки, т. е. мы можем сами создавать новые команды. Функции конструируются следующим образом:

```
function name () { list }
```

Причем слово function не обязательно, name определяет имя функции, по которому к ней можно обращаться, а тело функции состоит из списка команд list, находящегося между (и }. Этот список команд выполняется каждый раз, когда имя name задано как имя вызываемой команды. Отметим, что функции могут задаваться рекурсивно, так что разрешено вызывать функцию, которую мы задаем, внутри нее самой.

Функции выполняются в контексте текущей оболочки: для интерпретации функции новый процесс не запускается (в отличие от выполнения скриптов оболочки).

Аргументы

Когда функция вызывается на выполнение, аргументы функции становятся позиционными параметрами (positional parameters) на время выполнения функции. Они именуются как \$п, где п — номер аргумента, к которому мы хотим получить доступ. Нумерация аргументов начинается с 1, так что \$1 — это первый аргумент. Мы можем также получить все аргументы сразу с помощью \$*, и число аргументов с помощью \$#. Позиционный параметр 0 не изменяется. Если в теле функции встречается встроенная команда return, выполнение функции прерывается и управление передается команде, стоящей после вызова функции. Когда выполнение функции завершается, позиционным параметрам и специальному параметру # возвращаются те значения, которые они имели до начала выполнения функции.

Локальные переменные

Если мы хотим создать локальный параметр, можно использовать ключевое слово local. Синтаксис ее задания точно такой же, как и для обычных параметров, только определению предшествует ключевое слово local: local

```
name=value.
```

Вот пример задания функции, реализующей упоминавшуюся выше команду

```
seq!  
seq()  
{  
local I=$L;  
while [ $2 != $1 ] ; do  
{  
echo -n "$I ";  
!=$(( $1 + 1 ))  
Глава 5. Оболочка bash 145  
done ;  
echo $2
```

Обратите внимание на опцию -п оператора echo, она отменяет переход на новую строку. Хотя это и несущественно для тех целей, которые мы здесь имеем в виду, это может оказаться полезным для использования функции в других целях.

Функция вычисления факториала fact

Еще один пример:

```
fact ( )
{
if [ $1 = 0 ] ; then
echo 1;
else
{
echo $ ( ( $1 * $( fact $ ( ( $1 - 1 ) ) ) ) )
};
fi
```

Это функция факториала, пример рекурсивной функции. Обратите внимание на арифметическое расширение и подстановку команд.

Скрипты оболочки и команда source

Скрипт оболочки - это просто файл, содержащий последовательность команд оболочки. Подобно функциям, скрипты можно выполнять как обычные команды. Синтаксис доступа к аргументам такой же, как и для функций. 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. Ее формат:

```
source filename [arguments]
```

ИЛИ

```
. filename [arguments]
```

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. Если в каталогах, перечисленных в PATH, нужный файл не найден, его поиск производится в текущем каталоге.

Если заданы аргументы, на время выполнения скрипта они становятся позиционными параметрами. Если аргументов нет, позиционные параметры не изменяются. Значение (статус), возвращаемое командой source, совпадает со значением, возвращаемым последней командой, выполненной в скрипте. Если ни одна команда не выполнялась, или файл filename не найден, то статус выхода равен 0.