

Оглавление

Вступление.....	10
Запуск bash.....	12
Управление версиями.....	13
Hello World.....	14
Условные обозначения.....	14
Использование исходного кода примеров.....	15
Благодарности	17
От издательства	19
Глава 1. Идиома «большого» if	20
«Большой» if.....	20
Или ELSE.....	22
Выполняем несколько команд.....	23
Еще о случае нескольких команд.....	25
Так делать не нужно!.....	25
В заключение: стиль и удобочитаемость.....	27
Глава 2. Язык циклов	29
Циклические конструкции.....	29
Явные значения.....	31
Почти как в Python.....	34
Кавычки и пробелы.....	35
Разработка и тестирование циклов for.....	37
Циклы while и until.....	39
В заключение: стиль и удобочитаемость.....	39

Глава 3. На всякий случай: оператор Case	42
Сделайте свой выбор.....	42
Применение на практике	44
Задача.....	45
Наш сценарий	45
Сценарии-обертки.....	47
Еще один важный момент	54
В заключение: стиль и удобочитаемость.....	55
Глава 4. Язык переменных	57
Ссылка на переменную.....	57
Дополнительные параметры.....	59
Сокращенный вариант команды basename	59
Удаление пути или префикса	60
Сокращенный вариант команды dirname или удаление суффикса	61
Другие модификаторы	62
Условные подстановки	66
Значения по умолчанию.....	66
Списки значений, разделенных запятыми.....	67
Изменение значения.....	68
\$RANDOM	68
Подстановка команд	69
В заключение: стиль и удобочитаемость.....	71
Глава 5. Выражения и арифметика.....	72
Арифметика	73
Круглые скобки не нужны.....	76
Составные команды	77
В заключение: стиль и удобочитаемость.....	80
Глава 6. Функции.....	82
Вызов функций	82
Определение функций.....	83
Параметры функций.....	83
Возвращаемые значения функций.....	85
Локальные переменные.....	86

Особые случаи	87
Функция printf	88
Вывод POSIX	89
Получение и использование даты и времени.....	90
printf для повторного использования или отладки.....	91
В заключение: стиль и удобочитаемость.....	91
Глава 7. Списки и хеши	93
Сходные черты	95
Списки	96
Хеши	101
Пример подсчета слов	106
В заключение: стиль и удобочитаемость.....	109
Глава 8. Аргументы	110
Ваш первый аргумент.....	110
Поддержка ключей.....	112
Анализ ключей.....	113
Длинные ключи	115
HELP!	118
Отладочный и подробный режимы вывода	122
Версия.....	123
В заключение: стиль и удобочитаемость.....	124
Глава 9. Файлы и не только	125
Чтение файлов	125
read.....	125
mapfile.....	126
Метод «грубой силы»	130
Изменяем \$IFS при чтении файлов	130
Имитации файлов.....	133
Настроечные каталоги.....	134
Организация библиотек.....	135
Shebang!.....	136
Строгий режим bash.....	138
Код выхода.....	139

Это ловушка!	140
Встроенные документы и строки.....	142
Код выполняется в интерактивном режиме?	143
В заключение	144
Глава 10. Помимо идиом: работа с bash.....	145
Приглашения к вводу	146
Часовой пояс в приглашении	149
Получение ввода пользователя	149
read.....	150
pause.....	152
select	152
Псевдонимы.....	153
Функции	155
Локальные переменные.....	156
Возможности Readline.....	156
Журналирование в bash.....	158
Обработка JSON с помощью jq.....	159
Поиск в списке процессов.....	160
Ротация старых файлов	161
Встроенная документация	163
Отладка в bash	169
Модульное тестирование в bash.....	172
В заключение.....	173
Глава 11. Разработка своего руководства по стилю	174
Удобочитаемость	177
Комментарии.....	179
Имена.....	180
Функции.....	182
Кавычки.....	183
Форматирование	184
Синтаксис	186
Другие рекомендации.....	187
Шаблон сценария.....	187
Другие руководства по стилю	189

Инструмент проверки оформления кода на bash.....	190
В заключение.....	191
Приложение. Руководство по стилю	192
Удобочитаемость	193
Комментарии.....	194
Имена	194
Функции.....	195
Кавычки.....	197
Форматирование.....	198
Синтаксис	198
Другие рекомендации.....	199
Шаблон сценария.....	200
Об авторах	202
Иллюстрация на обложке.....	203

Язык циклов

В `bash` имеются не только циклы `for` в стиле `C`, но также другие виды и стили организации циклического выполнения инструкций. Некоторые из них ближе программистам на `Python`, но у каждого есть свои особые задачи. Существует цикл `for` без очевидных аргументов, который удобно использовать как в сценариях, так и внутри функций. Также есть похожий на итератор цикл `for`, значения для которого могут задаваться явно или возвращаться другими командами.

Циклические конструкции

Циклические конструкции распространены в языках программирования. С момента изобретения языка `C` многие языки программирования заимствовали из него цикл `for`. Это мощная и понятная конструкция, объединяющая код инициализации, условие завершения и код, выполняемый в начале каждой итерации. В `C`, `Java` и многих других языках цикл `for` выглядит следующим образом:

```
/* HE bash */
for (i=0; i<10; i++) {
    printf("%d\n", i);
}
```

В `bash` используется тот же подход, хотя и с некоторыми отличиями в синтаксисе:

```
for ((i=0; i<10; i++)); do
    printf '%d\n' "$i"
done
```

Во-первых, обратите внимание на двойные круглые скобки. Во-вторых, для обозначения блока инструкций, составляющих тело цикла, в `bash` вместо фигурных скобок используются ключевые слова `do` и `done`. Так же как в `C/C++`, идиоматическое использование цикла `for` — это пустой бесконечный цикл (позже мы также покажем конструкцию `while true; do`):

```
for (;;); do
    printf 'forever'
done
```

Но это не единственный вид цикла `for` в `bash`. В сценариях широко используется следующая идиома:

```
for value; do
    echo "$value"
    # Что-то сделать со значением $value...
done
```

На первый взгляд, этой конструкции чего-то не хватает, не так ли? Откуда `value` получает свои значения? Если записать такой цикл в командной строке, он ничего не выведет. Однако в сценарии оболочки такой цикл `for` будет перебирать параметры командной строки. То есть он последовательно будет присваивать переменной `value` значения `$1`, `$2`, `$3` и т. д.

Поместите этот цикл `for` в файл с именем `myloop.sh` и запустите его, как показано ниже. В результате он выведет три аргумента (`-c`, `17`, `core`):

```
$ bash myloop.sh -c 17 core
-c
17
core
$
```

Такую краткую форму цикла `for` также часто можно встретить в определениях функций:

```
function Listem {
    for arg; do
        echo "arg to func: '$arg'"
    done
    echo "Inside func: \${0} is still: '$0'"
}
```

Внутри определения функции переменные \$1, \$2 и т. д. представляют собой параметры функции, а не сценария. Поэтому цикл `for` будет перебирать параметры, переданные функции.

Этот минималистский цикл `for` выполняет итерацию по подразумеваемому списку значений — параметров, передаваемых сценарию или функции. При использовании в основной части сценария он перебирает параметры, переданные сценарию; внутри функции он перебирает параметры, переданные этой функции.

Это одна из малоизвестных идиом `bash`. Но вы должны понимать, как она работает, и в следующем разделе мы вернемся к ее обсуждению (как говорят адепты Python, явное лучше неявного).

Вам может понравиться такая же простая запись цикла, но с явными значениями на наш выбор, не ограниченный параметрами. В `bash` есть все, что для этого нужно.

Явные значения

В `bash` можно передать в цикл `for` список значений, например:

```
for num in 1 2 3 4 5; do
    echo "$num"
done
```

Поскольку `bash` поддерживает работу со строками, мы не ограничены только числами:

```
for person in Sue Neil Pat Harry; do
    echo $person
done
```


Список значений может включать не только литералы, но и переменные:

```
for person in $ME $3 Pat ${RA[2]} Sue; do
    echo $person
done
```

Также источником значений для цикла `for` могут быть результаты выполнения команд — отдельных или их конвейеров:

```
for arg in $(some cmd or other | sort -u)
```

Вот еще несколько примеров:

```
for arg in $(cat /some/file)
for arg in $(< /some/file) # Faster than shelling out to cat
for pic in $(find . -name '*.jpg')
for val in $(find . -type d | LC_ALL=C sort)
```

Типичная запись цикла `for`, особенно в старых сценариях, выглядит примерно так:

```
for i in $(seq 1 10)
```

Здесь команда `seq` генерирует последовательность чисел. Эквивалент такой записи цикла:

```
for ((i = 1; i <= 10; i++))
```

Последний вариант представления цикла `for` более эффективен и, вероятно, более понятен (обратите внимание, что в этих двух формах значение `i` будет различаться после завершения цикла — 10 в первом случае и 11 во втором, хотя вне цикла это значение обычно не используется).

Существует еще один вариант, но у него есть проблемы совместимости с разными версиями `bash`, потому что поддержка фигурных скобок появилась в версии 3.0, а дополнение нулями числовых значений — в версии 4.0:

```
for i in {01..10}; do echo "$i"; done
```



Нули в начале числа

В `bash` версии 4.0 и выше, если любой из первых двух членов в выражении вида {начало..конец..шаг} начинается с нуля, то переменная цикла будет принимать значения одинаковой длины и дополняться нулями слева. То есть при использовании выражения {098..100} переменная цикла будет последовательно принимать значения 098, 099, 100, а в случае {98..0100} каждое значение будет иметь длину в 4 символа: 0098, 0099, 0100.

Конструкция в фигурных скобках особенно удобна, когда требуется, чтобы генерируемые числа были частью строки. Для этого достаточно поместить конструкцию в фигурных скобках в запись строки. Например, пять имен файлов от `log01.txt` до `log05.txt` можно сгенерировать следующим образом:

```
for filename in log{01..5}.txt ; do
    # Сделать что-то с очередным именем файла
    echo $filename
done
```



Фигурные скобки или `printf -v`?

Тот же результат можно получить с помощью числового цикла `for`, используя команду `printf -v` для конструирования имени файла из чисел, но применение фигурных скобок выглядит проще. Используйте числовой цикл `for` и `printf` для более сложных задач, чем генерация имен файлов.

Для создания последовательности чисел с плавающей точкой очень удобно использовать команду `seq`. Вы указываете начальное значение, приращение на каждом шаге и конечное значение, например:

```
for value in $(seq 2.1 0.3 3.2); do
    echo $value
done
```

Получаем следующую последовательность:

```
2.1
2.4
```

2.7
3.0

Помните, что `bash` не поддерживает арифметику с плавающей точкой. Но такие значения могут понадобиться для передачи из сценария какой-то другой программе.

Почти как в Python

Вот еще одна конструкция, часто встречающаяся в циклах `for` в `bash`:

```
for person in ${name_list[@]}; do
    echo $person
done
```

Такой цикл может произвести, например, следующий вывод:

```
Arthur
Ann
Henry
John
```

Внешне эта конструкция похожа на цикл `for` в Python, где можно перебирать значения, возвращаемые итератором. В этом примере `bash` перебирает ряд значений, но они поступают не от итератора — все имена известны до начала цикла.

`${name_list[@]}` — это конструкция перечисления всех значений в массиве, который далее мы будем называть *списком* (подробное обсуждение терминологии вы найдете во введении к главе 7; в этом примере список называется `name_list`). Подстановка производится при подготовке команды к выполнению. Содержимое массива извлекается до передачи управления оператору `for`, то есть цикл получает значения, как если бы они были введены явно:

```
for person in Arthur Ann Henry John
```

А как обстоит дело со словарями? Для коллекций, которые в Python называются «словарями», в `bash` используется термин «ассоциативные массивы», а в некоторых других языках — «пары ключ/значение» или

«хеши» (см. введение к главе 7). Для работы с парами ключ/значение можно использовать конструкцию `${hash[@]}`. Чтобы выполнить итерации только по ключам (то есть индексам) хеша, добавьте восклицательный знак и используйте конструкцию `${!hash[@]}`, как показано ниже:

```
# Объявляем хеш (то есть массив пар ключ/значение)
declare -A hash
# Записываем данные в хеш
while read key value; do
    hash[$key]=$value
done
# Показываем содержимое хеша, хотя оно может
# выводиться не в порядке записи
for key in "${!hash[@]}"; do
    echo "key $key ==> value ${hash[$key]}"
done
```

Вот еще один пример:

```
# Объявляем хеш (то есть массив пар ключ/значение)
declare -A hash
# Записываем данные в хеш: слова и количества вхождений
while read word count; do
    let hash[$word]+="$count"
done
# Показываем содержимое хеша, хотя мы не можем управлять
# порядком элементов
for key in "${!hash[@]}";do
    echo "word $key count = ${hash[$key]}"
done
```

Эта глава в основном посвящена конструированию циклов, таких как `for`, поэтому за дополнительными подробностями и примерами списков и хешей обращайтесь к главе 7.

Кавычки и пробелы

Есть еще один важный аспект, который следует учитывать при написании циклов `for`. Вы наверняка обратили внимание на использование кавычек в предыдущем примере. Причина в том, что если значения в списке имеют пробелы (например, если каждый элемент списка

содержит имя и фамилию), то мы можем получить неожиданный результат. Рассмотрим пример:

```
for person in ${namelist[@]}; do
    echo $person
done
```

Такой цикл может произвести следующий вывод:

```
Art
Smith
Ann
Arundel
Hank
Till
John
Jakes
```

Получив список с четырьмя именами и фамилиями, цикл `for` вывел восемь отдельных значений. Почему? Ответ заключается в механизме подстановки в конструкцию `${namelist[@]}` — `bash` просто помещает значения из массива на место выражения с переменной. В результате получается список из восьми слов:

```
for person in Art Smith Ann Arundel Hank Till John Jakes
```

Цикл `for` получает список слов, но не знает, откуда они взялись.

Для решения этой проблемы в `bash` предусмотрен синтаксис с кавычками: поместите выражение списка в кавычки, и тогда каждый элемент будет заключен в кавычки. Выражение:

```
for person in "${namelist[@]}"
```

будет преобразовано в:

```
for person in "Art Smith" "Ann Arundel" "Hank Till" "John Jakes"
```

и даст желаемый результат:

```
Art Smith
Ann Arundel
Hank Till
John Jakes
```

Если вы собираетесь использовать цикл `for` для перебора списка имен файлов, обязательно используйте кавычки, потому что в именах файлов могут встречаться пробелы.

И еще один момент, о котором стоит упомянуть. В синтаксисе списка можно использовать знаки `*` или `@` для перечисления всех его элементов. Конструкция `${namelist[*]}` дает тот же результат, за исключением случая, когда она заключена в кавычки: `"${namelist[*]}"` вернет все значения внутри одной строки. Например:

```
for person in "${namelist[*]}"; do
    echo $person
done
```

выведет одну строку:

```
Art Smith Ann Arundel Hank Till John Jakes
```

Иногда требуется именно вывод в одну строку, но в цикле `for` это означает выполнение только одной итерации. Мы советуем использовать символ `@`, если вы не уверены, что вам нужен именно символ `*`.

Дополнительную информацию вы найдете в разделе «Кавычки» главы 11.

Разработка и тестирование циклов for

Обработка списка в цикле чрезвычайно удобна. Два очевидных случая ее применения: запуск SSH-команд со списком серверов и переименование файлов, например `for file in *.JPEG; do mv -v $file ${file/JPEG/jpg}; done`. Но как разработать и протестировать сценарий или даже простую команду `for`? Точно так же, как разрабатывается все остальное: начните с простого и постепенно двигайтесь вперед. Для проверки используйте `echo` (см. пример 2.1). Обратите внимание, что встроенная команда `echo` поддерживает ряд интересных параметров, но не соответствует стандарту POSIX (см. раздел «Вывод POSIX» в главе 6). Наиболее интересными и часто используемыми операторами являются `-e` (выключает интерпретацию символа, следующего за обратной косой чертой) и `-n` (подавляет автоматический переход на новую строку).

Пример 2.1. Переименование файлов — тестовая версия

```
### Конструируем и проверяем команду rename, обратите внимание на echo
for file in *.JPEG; do echo mv -v $file ${file/JPEG/jpg}; done

### Выполняем команды SSH на нескольких узлах, обратите внимание на первую
### команду echo (весь код можно записать в одну строку, но мы
### отформатировали его, чтобы уместить по ширине книжной страницы)
for node in web-server{00..09}; do
    echo ssh $node 'echo -e "$HOSTNAME\t$(date +%F) $(uptime)";'
done
```

Как только код заработает должным образом, удалите первую команду `echo`. Но имейте в виду, что при использовании перенаправления в блоке `for`, возможно, придется заменить `|` на `.p.`, `>` на `.gt.` и т. д., пока не будут пройдены все этапы.

**Выполнение одной и той же команды на нескольких хостах**

Хотя этот вопрос выходит за рамки данной книги, мы бы хотели отметить, что для запуска одной и той же команды на нескольких хостах предпочтительно использовать специальные инструменты, например Ansible, Chef или Puppet. Однако если нужно быстро выполнить пару команд, то подойдет один из следующих инструментов:

clusterssh

Написан на Perl, открывает множество неуправляемых терминалов в окнах.

mssh (MultiSSH)

SSH-клиент с графическим интерфейсом на основе GTK+, выполняется в одном окне.

mussh

Сценарий командной оболочки для работы с несколькими хостами.

pconsole

Консольный тайловый оконный менеджер, создающий отдельный терминал для каждого хоста.

multixterm

Написан на Expect и Tk, управляет несколькими терминалами xterm.

PAC Manager

Графический интерфейс в стиле SecureCRT для Linux, написанный на Perl.

Циклы `while` и `until`

Мы уже упоминали циклы `while` выше. В `bash` они работают в полном соответствии с ожиданиями — тело цикла выполняется, пока код условия выхода не обнулится:

```
while <УСЛОВИЕ>; do <ТЕЛО>; done
```

Такие циклы часто используются для чтения файлов, как будет показано в главе 9, а иногда — для анализа параметров, как изложено в разделе «Анализ ключей» в главе 8.

В отличие от других языков, в `bash` цикл `until` полностью эквивалентен циклу `! while`. Он действует по принципу: «выполнять тело цикла, пока код условия выхода нулевой»:

```
until <УСЛОВИЕ>; do <ТЕЛО>; done
### То же самое:
! while <УСЛОВИЕ>; do <ТЕЛО>; done
```

Это очень удобно в случае ожидания какого-либо события, например перезагрузки узла (пример 2.2).

Пример 2.2. Ожидание перезагрузки

```
until ssh user@10.10.10.10; do sleep 3; done
```

В заключение: стиль и удобочитаемость

В начале этой главы мы кратко рассмотрели цикл `for` в стиле `C/C++`. Поскольку `bash` в большей степени ориентирован на работу со строками, он поддерживает еще несколько стилей цикла `for`, о которых следует знать. Минималистская конструкция `for variable` обеспечивает неявный (и, возможно, малопонятный) обход аргументов сценария или функции. Явная передача в цикл `for` списка строковых или иных значений обеспечивает идеальный механизм для обхода всех элементов списка или всех ключей в хеше.

Теперь вы знаете, что `${namelist[@]}` и `${namelist[*]}` извлекают все значения из списка, но если эти две конструкции заключить в двойные кавычки, то они дадут разный результат: первая вернет каждый элемент в отдельной строке, а вторая — все элементы в одной строке. То же относится к специальным переменным командной оболочки `$@` и `*$*`. Обе вызывают список всех аргументов сценария (например, `$1`, `$2` и т. д.). Однако при заключении в двойные кавычки они дают разный результат: несколько строк и одну строку. Мы вспомнили об этом, только чтобы вернуться к простейшему циклу `for`:

```
for param
```

и отметить, что он эквивалентен следующей конструкции:

```
for param in "$@"
```

Мы считаем вторую форму более удачной, потому что она явно показывает, какие значения перебирает цикл. Однако кто-то может возразить, что само имя переменной `$@` и необходимость заключать ее в кавычки относятся к специфическим особенностям `bash`, которые менее понятны для неискушенного пользователя, чем первая минималистская форма. Если вы предпочитаете первый вариант, то просто добавьте комментарий:

```
for param # Итерации по всем аргументам сценария
```

Когда итерации выполняются по последовательности целочисленных значений, наиболее читабельным и эффективным, пожалуй, является цикл `for` в стиле `C` с двойными круглыми скобками. Кстати, если эффективность имеет большое значение, объявите переменную цикла как целочисленную, добавив инструкцию `declare -i i` в начало сценария, это позволит избежать ресурсоемких преобразований из строки в число и обратно.

Каковы возможности обработки данных в циклах? Данные можно анализировать, принимая решения на основе результатов этого анализа. Здесь мы подошли к еще одной важной особенности `bash` —

сверхмощному и чрезвычайно гибкому оператору `case`, о котором мы поговорим в следующей главе.

Циклы `for` чрезвычайно полезны, но могут приводить к ошибкам. Начинайте с простой конструкции и используйте `echo`, пока не убедитесь, что ваша команда работает должным образом. И помните о «синтаксическом сахаре» `while` и `until`, помогающем улучшить читаемость кода.