

# Основные правила грамматики Bash

Bash строит свои функции поверх нескольких основных **правил грамматики**. Код, который вы видите повсюду, код, который вы используете, основан на этих правилах. Тем не менее, **это очень теоретический взгляд**, но если вам интересно, это может помочь вам понять, почему вещи выглядят так, как они выглядят.

Если вы не знаете команды, используемые в следующих примерах, просто доверьтесь объяснению.

## Простые команды

В руководстве по Bash говорится:

Простая команда - это последовательность необязательных назначений переменных, за которыми следуют слова, разделенные пробелами, и перенаправления, завершающиеся управляющим оператором. Первое слово определяет команду, которая должна быть выполнена, и передается в качестве нулевого аргумента. Остальные слова передаются в качестве аргументов вызываемой команде.

Звучит сложнее, чем есть на самом деле. Это то, что вы делаете ежедневно. Вы вводите простые команды с параметрами, и оболочка их выполняет.

Каждая сложная операция Bash может быть разделена на простые команды:


```
ls
ls > list.txt
ls -l
LC_ALL=C ls
```

Последнее может быть вам незнакомо. Это просто добавляет " LC\_ALL=C " в среду `ls` программы. Это не влияет на вашу текущую оболочку. Это также работает при вызове функций, если только Bash не запускается в режиме POSIX® (в этом случае это влияет на вашу текущую оболочку).

У каждой команды есть код выхода. Это тип статуса возврата. Оболочка может перехватить его и действовать на нем. Диапазон кодов выхода - от 0 до 255, где 0 означает успех, а остальные означают либо сбой, либо проблему, о которой нужно сообщить вызывающей программе.

Простая командная конструкция является **основой** для всех более высоких конструкций. Все, что вы выполняете, от конвейеров до функций, в конечном итоге заканчивается (многими) простыми командами. Вот почему у Bash есть только один метод для расширения и выполнения простой команды.

## Конвейеры

 **Fix Me!** Отсутствует дополнительная статья о конвейерах и конвейерной обработке

```
[time [-p]] [ ! ] command [ | command2 ... ]
```

**Пусть вас не смущает** название "конвейер". Это грамматическое название конструкции. Такой конвейер не обязательно представляет собой пару команд, где stdout / stdin соединены через реальный канал.

Конвейеры представляют собой одну или несколько **простых команд** (разделенных | символом, соединяющим их ввод и вывод), например:

```
ls /etc | wc -l
```

будет выполняться `ls /etc` и **передавать** выходные данные `wc` в, которые будут подсчитывать строки, сгенерированные командой `ls`. Результатом является количество записей каталога в `/etc`.

Последняя команда в конвейере установит код выхода для конвейера. Этот код выхода можно "перевернуть", добавив к конвейеру восклицательный знак: неудачный конвейер завершится "успешным", и наоборот. В этом примере команды в строке `if` будут выполняться, если шаблон `"^root:"` **не** найден в `/etc/passwd`:

```
если ! grep '^root:' /etc/passwd; затем  
повторите "Пользователь root не определен ... а?"  
fi
```

Да, это тоже конвейер (хотя здесь нет конвейера!), Потому что **восклицательный знак для инвертирования кода выхода** может использоваться только в конвейере. Если `grep` код выхода равен 1 (FALSE) (текст не найден), ведущий `!` "инвертирует" код выхода, и оболочка видит (и действует) код выхода 0 (TRUE) и `then` часть `if` строфы выполняется. Можно сказать, что мы проверили `"not grep "^root" /etc/passwd"`.

Параметр `set pipefail` определяет поведение того, как `bash` сообщает код завершения конвейера. Если он установлен, то код выхода ( `$?` ) является последней командой, которая завершается с ненулевым статусом, если ни один сбой не выполняется, он равен нулю. Если он не установлен, то `$?` всегда содержит код выхода последней команды (как объяснено выше).


Опция оболочки `lastpipe` будет выполнять последний элемент в конвейерной конструкции в текущей среде оболочки, т. Е. Не в подоболочке.

Существует также массив PIPESTATUS[ ] , который устанавливается после выполнения конвейера переднего плана. Каждый элемент PIPESTATUS[ ] сообщает код выхода соответствующей команды в конвейере. Примечание: (1) это только для канала переднего плана и (2) для структуры более высокого уровня, которая создается из конвейера. Like list PIPESTATUS[ ] содержит статус завершения последней выполненной команды конвейера.

Еще одна вещь, которую вы можете делать с конвейерами, - это регистрировать время их выполнения. Обратите внимание, что **time это не команда**, это часть синтаксиса конвейера:

```
# время обновления
в реальном времени 3m21.288s
пользователь 0m3.114s
система 0m4.744s
```

# Списки

 Отсутствует дополнительная статья об операторах списков

Список - это последовательность одного или нескольких **конвейеров**, разделенных одним из операторов ; , & , && , или | | , и необязательно завершающихся одним из ; , & , или <newline> .

⇒ Это группа **конвейеров**, разделенных или завершенных **токенами**, которые имеют **разные значения** для Bash.

Технически весь ваш скрипт Bash - это один большой единый список!

Оператор	Описание
<PIPELINE1> <newline> <PIPELINE2>	Новые строки полностью разделяют конвейеры. Следующий конвейер выполняется без каких-либо проверок. (Вы вводите команду и нажимаете <RETURN> !)
<PIPELINE1> ; <PIPELINE2>	Точка с запятой делает то, что <newline> делает: она разделяет конвейеры
<PIPELINE> & <PIPELINE>	Конвейер перед the & выполняется <b>асинхронно</b> ("в фоновом режиме"). Если конвейер следует этому, он выполняется сразу после запуска асинхронного конвейера
<PIPELINE1> && <PIPELINE2>	<PIPELINE1> выполняется и <b>только</b> в том случае, если его код выхода был равен 0 (TRUE), затем <PIPELINE2> выполняется (И-List)
<PIPELINE1>    <PIPELINE2>	<PIPELINE1> выполняется, и <b>только</b> если его код выхода <b>не</b> равен 0 (FALSE), затем <PIPELINE2> выполняется (ИЛИ-список)

**Примечание:** POSIX называет эту конструкцию "составными списками".

# Составные команды

Смотрите также список составных команд.

Существует две формы составных команд:

- сформируйте новый синтаксический элемент, используя список в качестве "тела"
- полностью независимые синтаксические элементы

По сути, все остальное, что не описано в этой статье. Составные команды имеют следующие характеристики:

- они **начинаются** и **заканчиваются** определенным ключевым словом или оператором (например `for ... done` )
- они могут быть перенаправлены как единое целое

Смотрите Следующую таблицу для краткого обзора (никаких подробностей - просто обзор):


Синтаксис составных команд	Описание
( <LIST> )	Выполнить <LIST> в дополнительной подболочке ⇒ статья
{ <LIST> ; }	Выполнять <LIST> как отдельную группу (но не в подболочке) ⇒ статья
(( <EXPRESSION> ))	Вычислите арифметическое выражение <EXPRESSION> ⇒ статья
[[ <EXPRESSION> ]]	Вычислите условное выражение <EXPRESSION> (оно же "новая тестовая команда") ⇒ статья
for <NAME> in <WORDS> ; do <LIST> ; done	Выполняется <LIST> при установке переменной <NAME> на одну из <WORDS> на каждой итерации (классический цикл for) ⇒ статья
for (( <EXPR1> ; <EXPR2> ; <EXPR3> )) ; do <LIST> ; done	Цикл for в стиле C (управляемый арифметическими выражениями) ⇒ статья
select <NAME> in <WORDS> ; do <LIST> ; done	Предоставляет простое меню ⇒ статья
case <WORD> in <PATTERN>) <LIST> ;; ... esac	Решения, основанные на сопоставлении шаблонов - выполнение <LIST> по совпадению ⇒ статья
if <LIST> ; then <LIST> ; else <LIST> ; fi	Предложение if: принимает решения на основе кодов выхода ⇒ статья
while <LIST1> ; do <LIST2> ; done	Выполнить <LIST2> , пока <LIST1> возвращает TRUE (код выхода) ⇒ статья

## Синтаксис составных команд

### Описание

<code>until &lt;LIST1&gt; ; do</code> <code>&lt;LIST2&gt; ; done</code>	Выполнять <LIST2> до <LIST1> тех пор, пока не будет возвращено значение TRUE (код выхода) ⇒ статья
--	--

## Определения функций оболочки

 Отсутствует дополнительная статья о функциях оболочки

Определение функции оболочки делает **составную команду** доступной через новое имя. Когда функция выполняется, у нее есть свой собственный "частный" набор позиционных параметров и дескрипторов ввода-вывода. Он действует как скрипт внутри скрипта. Проще говоря: **вы создали новую команду**.

Определение простое (одна из многих возможностей):

```
<NAME> () <COMPOUND_COMMAND> <REDIRECTIONS>
```

который обычно используется с `{...; }` составной командой и, таким образом, выглядит как:

```
print_help() { echo "Извините, помощь недоступна"; }
```

Как и выше, определение функции может иметь любую **составную команду** в качестве тела. Такие структуры, как

```
countme() for ((x=1;x<=9;x++)); выполнить echo $x; готово
```

необычны, но вполне допустимы, поскольку конструкция цикла `for` является составной командой!

Если указано **перенаправление**, перенаправление не выполняется при определении функции. Выполняется при запуске функции:

```
# это НЕ приведет к перенаправлению (во время определения)
f() { echo ok; } > файл

# ТЕПЕРЬ перенаправление будет выполнено (во время ВЫПОЛНЕНИЯ функции)
f
```

Bash допускает три эквивалентные формы определения функции:

```
NAME () <COMPOUND_COMMAND> <ПЕРЕНАПРАВЛЕНИЯ>
имя функции () <COMPOUND_COMMAND> <ПЕРЕНАПРАВЛЕНИЯ>
имя функции <COMPOUND_COMMAND> <ПЕРЕНАПРАВЛЕНИЯ>
```

Пробел между `NAME` и `()` необязателен, обычно вы видите его без пробела.

Я предлагаю использовать первую форму. Это указано в POSIX, и все оболочки, подобные Bourne, похоже, поддерживают его.

**Примечание:** До версии 2.05-alpha1 Bash распознавал определение только с помощью фигурных скобок ( name() { ... } ), другие оболочки разрешают определение с использованием **любой** команды (не только набора составных команд).

Чтобы выполнить функцию, подобную обычному сценарию оболочки, вы объединяете ее следующим образом:

```
#!/bin/bash
# Добавить shebang

myscmd()
{
    # Этот $1 принадлежит функции!
    найти / -iname "$1"
}

# Этот 1 доллар принадлежит самому скрипту!
myscmd "$ 1" # Выполнить команду сразу после определения функции

выход 0
```

### Просто информация (1):

Внутренне, для разветвления, Bash хранит определения функций в переменных среды. Переменные с содержимым "() ....".

Что-то похожее на следующее работает без "официального" объявления функции:

```
$ export testfn="() { echo test; }"
$ bash -c testfn
тест
$
```

### Просто информация (2):

Можно создавать имена функций, содержащие косые черты:

```
/bin/ls() {
    echo - ЭТО ПОДДЕЛКА
}
```

Элементы этого имени не подлежат поиску пути.

Не следует использовать странные имена функций. Цитата из сопровождающего:

- Было ошибкой разрешать такие символы в именах функций (например, ``unset`` не работает для их отмены без принудительного ввода `-f`). Мы придерживаемся их для обеспечения обратной совместимости, но я не должен поощрять их использование.

## Краткое изложение грамматики

- **простая команда** - это просто команда и ее аргументы
- **конвейер** - это одна или несколько **простых команд**, которые, вероятно, связаны в канал
- **список** - это один или несколько **конвейеров**, соединенных специальными операторами
- **составная команда** - это **список** или специальная команда, которая формирует новую мета-команду
- **определение функции** делает **составную команду** доступной под новым именем и отдельной средой

## Примеры для классификации

 Еще...

(Очень) простая команда

```
эхо "Привет, мир ..."
```

Все следующие команды являются простыми

```
x=5
```

```
>tmpfile
```

```
{x}<"$x" _=${x=<(echo moo)} <&0$(cat <"$x" >&2)
```

Обычная составная команда

```
если [ -d /data/mp3 ]; тогда
ср mymusic.mp3 /data/mp3
fi
```

- **составная команда** для `if` предложения
- **список**, который `if` **проверяет**, на самом деле содержит **простую команду** [ `-d /data/mp3` ]
- **список**, который `if` **выполняется**, содержит простую команду ( `ср mymusic.mp3 /data/mp3` )

Давайте инвертируем код выхода тестовой команды, изменится только одно:

```
если ! [ -d /data/mp3 ]; тогда
ср mymusic.mp3 /data/mp3
fi
```

- **список**, который `if` **проверяет**, теперь содержит **конвейер** (из-за `!` )

# Смотрите также

- Внутренний: список составных команд
- Внутренний: синтаксический анализ и выполнение простых команд
- Внутренний: цитирование и экранирование
- Внутренний: введение в расширения и замены
- Внутренний: несколько слов о словах ...

## Обсуждение

Джейкоб, [2010/12/28 01:43 \(\)](#), [2010/12/28 06:50 \(\)](#)

"По сути, конвейеры представляют собой одну или несколько простых команд"

На самом деле, страница руководства bash явно определяет "простые команды" и "составные команды", но не определяет (простые старые) "команды". Предположительно, они могут быть простыми командами или составными командами. Свидетель:

```
для NUM в {1..100}; выполнить echo "$NUM"; выполнено | fgrep 3 |  
во время чтения NUM; выполнить echo "${NUM/3 /three}"; выполнено
```

Раздел ПЕРЕНАПРАВЛЕНИЯ также подразумевает, что "команда" включает "составные команды" фразой "Следующие операторы перенаправления могут предшествовать или появляться в любом месте простой команды или могут следовать за командой". Например:

```
во время чтения СПИСКА; выполняйте эхо "${LIST[@]}"; готово < <(s  
ed "s/#.*//" /etc/fstab | grep .)
```

Ян Шампера, [2010/12/28 06:49 \(\)](#)



Да, я знаю, в чем здесь проблема. command определяется как

- простая команда
- составная команда
- определение функции
- (Bash) определение sorpos

Справочной странице Bash здесь немного не повезло (и этой странице тоже!).

Я сделаю несколько замечаний по этому поводу, лучшей была бы страница "расширенного описания синтаксиса", которая включает (упрощенные) синтаксические диаграммы, основанные на неполном (не до каждой строки и определения символа) EBNF, чтобы быть более точным. Это всего лишь некоторый объем работы, и поначалу в нем будут ошибки.



 syntax/basicgrammar.txt  Последнее редактирование: 2019/04/01 21:45 автор ddehbw

---

Этот сайт поддерживается Performing Databases - вашими  
экспертами по администрированию баз данных

---

Bash Hackers Wiki

---



Except where otherwise noted, content on this wiki is licensed under the following license:  
GNU Free Documentation License 1.3