Руководство по написанию скриптов в Linux Bash

В этом разделе мы будем учиться правильно писать скрипты на bash

- Как писать bash-скрипты.
- Регулярные выражения Bash.

Как писать bash-скрипты.

Встроенные команды в среде bash (и ее аналоги sh, zsh и другие) совместимы с любым приложением, соответствующим стандартам POSIX, в операционной системе Linux. Это дает вам возможность включить в свой bash-скрипт любое совместимое приложение, расширяя ваши возможности в области автоматизации повседневных задач администрирования систем Linux, развертывания и сборки приложений, а также выполнения разнообразных пакетных операций, включая обработку аудио и видео.

Командная строка представляет собой наиболее мощный интерфейс для взаимодействия с системой на данный момент. Получить базовое понимание ее работы довольно просто. Рекомендуется изучить руководство по командам bash, для чего можно воспользоваться командой man bash.

Суть bash-скриптов заключается в записи всех ваших действий в один файл и выполнении их по мере необходимости.

В данной статье мы расскажем о создании bash-скриптов с нуля и продемонстрируем, какую пользу вы можете извлечь из их использования. Если вы планируете серьезно заняться этим, то рекомендуется иметь под рукой справочник по bash для удобства.

Какой редактор выбрать?

Для выполнения задачи вам потребуется удобный текстовый редактор. Если вы используете SSH для подключения, то у вас есть три основных варианта:

- vim
- nano
- mcedit

Если вы работаете локально, выбор полностью зависит от вас. Один из стандартных выборов для Linux – это gedit. В данной инструкции мы использовали nano при работе через SSH на удаленном сервере.

Запускаем “Hello, World!”

Обычно, первой программой, которую разрабатывают программисты, является “Hello, World!” – простой вывод этой фразы. Мы также начнем с этого. Для вывода строки в консоль используется команда echo. Просто введите echo “Hello, World!” в командной строке, и вы увидите соответствующий вывод:

```
root@linux:~ # echo "Hello, World!"
Hello, World!
```

Давайте сделаем это с помощью программы. Используя команду touch helloworld.sh, мы создадим файл с именем helloworld.sh. Затем, с помощью команды nano helloworld.sh, мы откроем этот файл для редактирования. Далее заполним файл нашей программой:

```
#!/bin/bash
echo "Hello, World!"
```

Для сохранения изменений и выхода из редактора nano, выполните следующие действия: нажмите комбинацию клавиш CTRL + O, чтобы сохранить файл (после чего нажмите Enter для подтверждения перезаписи текущего файла), а затем нажмите CTRL + X для выхода из редактора. Вы также можете выйти без сохранения, в этом случае вас спросят, действительно ли вы хотите выйти без сохранения. Если ваш ответ “да”, нажмите клавишу N для выхода без сохранения. Если вы хотите сохранить изменения, нажмите Y, и вас попросят указать путь для сохранения измененного файла; нажмите Enter, чтобы перезаписать исходный файл.

Теперь рассмотрим, что мы написали в скрипте.

Первая строка содержит #!/bin/bash, что фактически указывает на расположение интерпретатора. Это позволяет запускать скрипт без необходимости явно указывать интерпретатор. Вы можете убедиться, что ваш интерпретатор bash находится по указанному пути, используя команду `which bash`:

```
root@linux:~ # which bash
/usr/bin/bash
```

Во второй строке содержится вся наша программа. Мы уже рассмотрели, как она работает выше, и теперь перейдем к ее выполнению.

Для запуска вашего скрипта или команды существуют два способа.

Способ №1: Используйте команду `bash helloworld.sh`. Этот способ включает интерпретатор и передает имя файла для выполнения в качестве аргумента.

```
root@linux:~ # bash helloworld.sh
Hello, World!
```

Способ №2: Сначала необходимо предоставить системе разрешение на выполнение скрипта с помощью команды `chmod +x helloworld.sh`. Эта команда устанавливает флаг исполнения для файла. Теперь вы можете запустить скрипт так же, как любой другой исполняемый файл в Linux, с помощью команды `./helloworld.sh`.

```
root@linux:~ # ./helloworld.sh
Hello, World!
```

Первая программа готова; её функциональность ограничивается выводом строки в консоль.

Аргументы

Это параметры, которые можно передать программе при ее вызове. Например, программа ping ожидает IP-адрес или DNS-имя в качестве обязательного аргумента, который нужно пинговать, например: `ping google.com`. Этот механизм предоставляет простой и удобный способ взаимодействия пользователя с программой.

Давайте научим нашу программу принимать и обрабатывать аргументы. Доступ к аргументам можно получить через специальную команду $X, где X – это число. $0 всегда представляет имя исполняемого скрипта. $1 – первый аргумент, $2 – второй, и так далее. Конечно же, если вам нужно передать много аргументов вашему приложению, это может быть утомительным процессом, и для этого можно использовать цикл, чтобы перебрать все поступившие аргументы:

```
for var in "$@"; do
    echo "$var"
done
```

Подробнее о циклах мы рассмотрим в последующих разделах.

Давайте рассмотрим пример: создадим новый файл с помощью команды `touch hellousername.sh` и предоставим ему права на выполнение с помощью команды `chmod +x hellousername.sh`.

Затем откроем файл `hellousername.sh` в редакторе nano.

Вот код примера:

```
#!/bin/bash


echo "Hello, $1!"
```

Сохраняем изменения и закрываем редактор. Теперь давайте посмотрим, что у нас получилось.

```
root@linux:~ # ./hellousername.sh Dima
Hello, Dima!
```

Эта программа небольшая, но она демонстрирует, как использовать аргументы на самом базовом уровне. В данном случае она принимает один аргумент, “Dima”, и сразу же использует его без каких-либо дополнительных проверок.

```
root@linux:~ # ./hellousername.sh
Hello, !
```

В этом сценарии у нашей программы есть недоразумение: она не приветствует никого. Для устранения этой проблемы есть два способа: проверить количество аргументов или проверить содержимое аргумента.

Способ №1

```
#!/bin/bash
if [ "$#" -lt 1 ]; then
    echo "Недостаточно аргументов. Пожалуйста, передайте в качестве аргумента имя. Пример: $0
Dima"
    exit 1
fi
echo "Hello, $1!"
```

Мы более подробно изучим структуру if-then [else] fi в следующих разделах, пока не будем останавливаться на этом подробно. Важно понимать, что в данном контексте проверяется переменная $#. Она представляет собой количество аргументов, не считая имени скрипта, которое всегда равно $0.

Способ №2

```
#!/bin/bash
if [ -z "$1" ]; then
   echo "Имя пустое или не передано. Пожалуйста, передайте в качестве аргумента имя. Пример:
$0 Dima"
   exit 1
fi

echo "Hello, $1!"
```

Здесь также используется структура if-then [else] fi. Ключ -z в операторе if применяется для проверки переменной на пустую строку, а противоположный ключ -n используется для проверки того, что строка не является пустой. Несмотря на то что это не самый правильный способ для проверки входящих аргументов, он может быть полезным внутри самой программы. Например, для проверки результата выполнения приложения внутри программы и убедиться, что оно что-то вернуло.

Управляющие конструкции

При написании программ, даже на языках программирования, длиннее нескольких строк, трудно обойтись без использования условных операторов. В разных языках программирования существуют разные способы реализации условных операторов, но в большинстве случаев применяется синтаксис if-else. Этот синтаксис также присутствует и в языке программирования Bash.

Давайте рассмотрим один из вышеупомянутых примеров.

```
#!/bin/bash
if [ "$#" -lt 1 ]; then
    echo "Недостаточно аргументов. Пожалуйста, передайте в качестве аргумента имя. Пример: $0
Dima"
    exit 1
fi
echo "Hello, $1!"
```

Здесь выполняется проверка системной переменной $# на то, что она меньше 1 (оператор -lt используется для сравнения). Если это условие верно, мы переходим к блоку команд, который начинается с ключевого слова then. Вся структура условного оператора, начиная с if и заканчивая fi, должна быть правильно закрыта. Более сложная структура условных операторов может выглядеть приблизительно так:

```
if TEST-COMMAND1
then
  STATEMENTS1
elif TEST-COMMAND2
then
  STATEMENTS2
else
  STATEMENTS3
fi
```

Реальный пример:

```
#!/bin/bash
if [ "$#" -lt 1 ];
then
    echo "Недостаточно аргументов. Пожалуйста, передайте в качестве аргумента имя. Пример: $0
Dima"
    exit 1
```

```
fi
if [ "$1" = "Vasya" ]; then
        echo "Whatsupp, Dmitry?"
elif [ "$1" = "Masha" ];
then
        echo "Hey, Masha"
elif [ "$1" = "Michael" ];
then
        echo "Shalom, Michael"
else
        echo "Hi, $1"
fi
```

Вывод программы:

```
root@linux:~ # ./hellousername.sh Dima
Whatsupp, Dmitry?
root@linux:~ # ./hellousername.sh Masha
Hey, Masha
root@linux:~ # ./hellousername.sh Michael
Shalom, Michael
root@linux:~ # ./hellousername.sh Andrew
Hi, Andrew
root@Linux:~ # ./hellousername.sh
Недостаточно аргументов. Пожалуйста, передайте в качестве аргумента имя. Пример:
./hellousername.sh Dima
```

Выражение “$1” = “Dima” проверяет, совпадают ли строки между собой. Блок команд, следующий за оператором else, выполняется только в случае, если выше в коде не было обнаружено другого более подходящего блока для выполнения.

&& и ||

В предыдущей главе, вы, возможно, обратили внимание, что я использовал команду `exit 1`, чтобы завершить выполнение программы в случае неудачной проверки аргумента. Это означает, что программа завершается с ошибкой. В языке программирования Bash есть операторы `&&` и `||`, которые используются для создания последовательностей команд. Каждая команда в последовательности зависит от результата выполнения предыдущей команды.

Пример 1: `command1 && command2`. В этом случае `command2` выполнится только в том случае, если `command1` завершится с кодом возврата 0 (по умолчанию).

Пример 2: `command1 || command2`. В этом случае `command2` выполнится только, если `command1` завершится с кодом возврата, отличным от 0.

Пример 3: `command1 && command2 || command3`. Если `command1` завершится с кодом возврата 0, то будет выполнен `command2`, в противном случае выполнится `command3`.

Переменные

Как гласит один из основных принципов программирования — “Не повторяй себя” (DRY). Поэтому мы перепишем предыдущий пример, используя переменные, чтобы избежать повторного вызова команды echo каждый раз.

В языке программирования Bash переменные создаются с помощью оператора присваивания, например: `x="foo bar"` или `z=$1`. Мы присвоили переменной x строку “foo bar”, а переменной z – значение первого аргумента. Использование именованных переменных гораздо удобнее, чем использование $1, особенно когда значение аргумента нужно использовать во многих местах.

Кроме того, аргументы могут быть представлены в различном порядке. Мы также подчеркиваем важность использования осмысленных имен переменных. Это помогает избежать путаницы, особенно при дальнейшем развитии программы. Избегайте использования неинформативных имен переменных (и функций), таких как “a”, “b”, “zzzz” и т.д.

Чтобы избежать повторного вызова команды echo с разными строками, мы разбиваем строку на части. Первая часть содержит приветствие, вторая – имя. Третья часть – завершающий знак препинания, который мы не выносим в переменную.

```
#!/bin/bash

greetString="Hello"
nameString="stranger"

if [ "$#" -lt 1 ];
then
    echo "Недостаточно аргументов. Пожалуйста, передайте в качестве аргумента имя. Пример: $0
Dima"
    exit 1
fi

if [ "$1" = "Dima" ];
then
    nameString="Dmitry"
 greetString="Whatsup"
```

```
elif [ "$1" = "Masha" ];
then
 nameString="Maria"
elif [ "$1" = "Michael" ];
then
 greetString="Shalom"
 nameString="Michael"
fi

echo "$greetString, $nameString!"
```

В данном примере мы инициализируем переменные greetString и nameString значениями по умолчанию. Затем программа выводит значения этих двух переменных с использованием команды echo и форматированной строки (в двойных кавычках переменные раскрываются). После этого программа проверяет, нужно ли присвоить переменным другие значения.

Switch case

Использование конструкции if-else в нашем примере не является наилучшим вариантом. Мы всего лишь сравниваем значение переменной с определенным набором значений. В таком случае более подходящим выбором будет конструкция switch-case.

```
case "$variable" in

 "$condition1" )
 command...
 ;;

 "$condition2" )
 command...
 ;;

esac
```

Давайте перепишем нашу программу для приветствия, используя конструкцию switch-case:

```
#!/bin/bash

name=$1

case "$name" in
```

```
  "Dima" )
    nameString="Dmitry"
 greetString="Whatsup"
  ;;
  "Masha" )
 greetString="Hey"
 nameString="Maria"
  ;;
  * )
 greetString="Hello"
 nameString="stranger"
  ;;
esac

echo "$greetString, $nameString!"
```

Циклы

Как и в любом полноценном языке программирования, Bash поддерживает два типа циклов: цикл for и цикл while. Циклы используются для выполнения определенного кода заданное количество раз. Например, при анализе CSV-файла, можно использовать цикл для поочередного перебора строк и обработки каждой строки отдельно.

Цикл for

```
for var in list
do
команды
done
```

Реальный пример:

```
#!/bin/bash

for name in Maria Dima Michael stranger
do
 echo "Hello, $name!"
done
```

Вывод:

```
root@linux:~ # ./cycle.sh
Hello, Maria!
Hello, Dima!
Hello, Michael!
Hello, stranger!
```

В данной программе происходит простой перебор всех имен, разделенных пробелами, и их вывод с использованием команды echo.

Теперь давайте попробуем усовершенствовать этот пример:

```
#!/bin/bash
file=$1
for name in $(cat $file)
do
 echo "Hello, $name!"
done
```

Давайте создадим файл с именем “names”, а затем запишем в него список имен для приветствия с помощью команды “touch names”.

```
Maria
Dmitry
Ivan
Nikolai
Innokentiy
```

Вывод:

```
root@linux:~ # ./cycle.sh
^C
root@linux:~ # ./cycle.sh names
Hello, Maria!
Hello, Dmitry!
Hello, Ivan!
Hello, Nikolai!
Hello, Innokentiy!
```

Обратите внимание на символ “^C”. Это символ для прерывания выполнения программы. В данном случае мы запустили программу без аргументов, и она зациклилась, можно сказать, что “зависла”. Пришлось принудительно завершить её выполнение. Важно помнить о необходимости проверки входных данных в реальных программах. Как это делать, можно

узнать из главы о конструкциях if-else и switch-case, например.

В нашей программе есть небольшая ошибка. Давайте внесем изменения в файл с именами:

```
Erich Maria Remarque
Dmitry
Ivan
Nikolai
Innokentiy
```

Запустим программу:

```
root@geneviev:~ # ./cycle.sh names
Hello, Erich!
Hello, Maria!
Hello, Remarque!
Hello, Dmitry!
Hello, Ivan!
Hello, Nikolai!
Hello, Innokentiy!
```

Как говорится, “Кто все эти люди?”. Причина этого в том, что у нас не установлена переменная окружения IFS (Internal Field Separator), которая определяет разделители полей. В нашем цикле используются пробелы и символы новой строки как разделители. Для исправления этой ситуации, в начале скрипта (после #!/bin/bash) можно установить использование символа новой строки в качестве разделителя полей следующим образом: IFS=$’\n’.

```
root@linux:~ # ./cycle.sh names
Hello, Erich Maria Remarque!
Hello, Dmitry!
Hello, Ivan!
Hello, Nikolai!
Hello, Innokentiy!
```

В результате получится возможность оперировать целыми строками, что будет полезно, например, при парсинге CSV-файлов.

Обычно цикл for используется с использованием счетчика в стиле, подобном C, например, `for (i=0; i<10; i++) {}`. В Bash также можно использовать подобный синтаксис.

```
#!/bin/bash
for (( i=1; i <= 10; i++ ))
do
echo "number is $i"
done
```

Вывод:

```
root@linux:~ # ./cycle.sh
number is 1
number is 2
number is 3
number is 4
number is 5
number is 6
number is 7
number is 8
number is 9
number is 10
```

Цикл while

```
while команда проверки условия
do
другие команды
done
```

Способ сделать бесконечную петлю:

```
while true
 do
 echo "this is infinity loop"
done
```

Это может быть полезным, например, когда вам нужно выполнять какие-то задачи чаще, чем позволяет расписание cron (например, каждую минуту), или когда необходимо постоянно мониторить какое-то значение. Бесконечные циклы имеют множество областей применения.

Здесь используются те же выражения, что и в конструкции if:

```
#!/bin/bash
count=0
while [ $count -lt 10 ]
do
 (( count++ ))
 echo "count: $count"
done
```

Вывод:

```
root@linux:~ # ./cycle.sh
count: 1
count: 2
count: 3
count: 4
count: 5
count: 6
count: 7
count: 8
count: 9
count: 10
```

Из цикла можно выйти, используя команду “break” (она также применима к циклам “for”):

```
#!/bin/bash
count=0
while [ $count -lt 10 ]
do
 (( count++ ))
 echo "count: $count"
 if [ "$count" -gt 5 ]
 then
  break
 fi
done
```

Вывод:

```
root@linux:~ # ./cycle.sh
count: 1
```

```
count: 2
count: 3
count: 4
count: 5
count: 6
```

Заключение

Несмотря на огромную конкуренцию в сфере автоматизации от языков программирования, таких как Python, Ruby и Perl, Bash по-прежнему остается востребованным. Он прост в изучении и использовании, гибок и почти всегда доступен в большинстве дистрибутивов Linux.

В данной статье мы рассмотрели лишь основы написания программ на языке Bash. Мы надеемся, что этот материал был полезен для вас.

Регулярные выражения Bash.

В этом гайде разберем тонкости работы с регулярными выражениями Bash, которые помогут вам реализовать весь потенциал командной строки и скриптов в Linux.

e67a6d4ded8341e628931.png

Что такое регулярные выражения?

Регулярные выражения — это специальным образом записанные строки, используемые для поиска символьных шаблонов в тексте. Чем-то они похожи на групповые символы в оболочке, но их возможности куда шире. Многие утилиты для работы с текстом в Linux и языки программирования включают в себя механизм регулярных выражений. Здесь возникают проблемы: разные программы и языки оперируют различными диалектами регулярных выражений. В этой статье рассмотрим стандарт POSIX, которому соответствуют большинство утилит в Linux.

Утилита grep

Программа `grep` — это основной инструмент для работы с регулярными выражениями. Grep анализирует данные со стандартного ввода, ищет совпадения с указанным шаблоном и выводит все подходящие строки. Обычно grep предустановлен в большинстве дистрибутивов. Если хотите потренироваться в использовании регулярных выражений, можете повторять описанные команды в виртуальной машине, либо на арендованном сервере.

Grep имеет следующий синтаксис:

```
grep [параметры] регулярное_выражение [файл…]
```

Самый простой случай использования grep — поиск строк, содержащих фиксированную подстроку. В следующем примере grep вывел все строки, содержащие последовательность nologin:

```
grep nologin /etc/passwd

Output:
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
games:x:5:60:games:/usr/games:/usr/sbin/nologin
...
```

У grep имеются множество параметров. Подробно с ними ознакомиться можно в документации. Приведем ключи, полезные при работе с регулярными выражениями:

Ключ `-v` — инвертировать критерий. В этом случае grep выводит строки, не содержащие совпадений:

```
ls /bin | grep -v zip

Output:
[
411toppm
7z
7za
7zr
…
```

Ключ `-i` — игнорировать регистр символов.

Ключ `-o` — выводить не строки, а только совпадения с шаблоном:

```
ls /bin | grep -o zip

Output:
zip
zip
zip
zip
…
```

Ключ `-w` — искать только строки, содержащие все слово, которое составляет шаблон.

```
ls /bin | grep -w zip

Output:
gpg-zip
zip
```

Для сравнения та же команда без опции. В вывод также попали строки, содержащие шаблон в качестве подслова в слове.

```
ls /bin | grep zip

Output
bunzip2
bzip2
bzip2recover
funzip
...
```

Basic Regular Expressions

Ранее упоминалось, что существует множество диалектов регулярных выражений. В стандарте POSIX рассматриваются два вида реализаций. Первый — Basic Regular Expressions (BRE). Ему соответствуют практически все POSIX-совместимые программы. Второй — Extended Regular Expression (ERE). Этот вид позволяет создавать более сложные регулярные выражения, но поддерживается не всеми утилитами. Для начала рассмотрим особенности BRE.

Метасимволы и литералы

В тексте выше мы уже столкнулись с простыми регулярными выражениями. К примеру, выражение «zip» обозначает строку, соответствующую следующим критериям: в строке не меньше трех символов; в строке присутствуют символы «z», «i», «p», причем именно в таком порядке; между ними нет других символов. Символы, соответствующие сами себе (как «z», «i», «p») называются литералами. Кроме того, существуют другая категория символов, называемая метасимволами. Они применяются для составления различных критериев поиска. К метасимволам в BRE относятся:

```
^ $ . [ ] * \ -
```

Чтобы использовать метасимвол в качестве литерала, его нужно экранировать с помощью обратного слэша (`\`). Обратите внимание, что некоторые метасимволы имеют специальное значение в оболочке. Поэтому при передаче регулярного выражения в аргумент команды его следует заключать в кавычки.

Любой символ

Метасимвол «точка» (`.`) соответствует любому символу в данной позиции. Например:

```
ls /bin | grep '.zip'

Output:
bunzip2
bzip2
bzip2recover
funzip
```

```
gpg-zip
gunzip
gzip
mzip
p7zip
pbzip2
preunzip
prezip
prezip-bin
streamzip
unzip
unzipsfx
```

Здесь имеется один важный момент. Программа zip не вошла в вывод, так как метасимвол «точка» увеличил длину обязательного совпадения до четырех символов.

Якорные символы

Символ «карет» (`^`) и «доллар» (`$`) в регулярных выражениях играют роль якорей. Это означает, что в их присутствии совпадение с шаблоном возможно, только если оно будет найдено в начале строки (`^`) или в ее конце (`$`).

```
ls /bin | grep '^zip'
Output:
zip
zipcloak
zipdetails
zipgrep
…

ls /bin | grep ‘zip$’
Output:
funzip
gpg-zip
gunzip
…

ls /bin | grep ‘^zip$’
Output
zip
```

Регулярное выражение `^$` будет соответствовать пустым строкам.

Множества символов

Кроме описания совпадения с любым символом в заданной позиции (`.`) в регулярных выражениях имеется возможность описать символ из определенного множества. Делается это с помощью квадратных скобок. В следующем примере ищутся соответствия со строками bzip и gzip:

```
ls /bin | grep '[bg]zip'
Output:
bzip2
bzip2recover
gzip
```

Все метасимволы, кроме двух, теряют свое специальное значение внутри скобок.

Если сразу после открывающей квадратной скобки стоит символ карет (`^`), остальные символы множества интерпретируются как недопустимые в данной позиции. Например:

```
ls /bin | grep '[^bg]zip'
Output:
bunzip2
funzip
gpg-zip
gunzip
mzip
p7zip
preunzip
prezip
prezip-bin
streamzip
unzip
unzipsfx
```

Включив отрицание, мы получили список имен файлов, содержащих последовательность zip, которой предшествуют любой символ, кроме b или g. Обратите внимание, что имя zip не было найдено. Символ отрицания не отменяет необходимости присутствия символа в заданной позиции. Кроме того символ карет является отрицанием, только если стоит сразу же после открывающей скобки; в противном случае он теряет свое специальное значение.

С помощью дефиса (`-`) можно определять диапазоны символов. Так можно выразить любой диапазон символов и даже нескольких таких диапазонов. К примеру нам нужно найти все имена файлов, начинающиеся с буквы или цифры. Делается это так:

```
ls ~ | grep '^[A-Za-z0-9]'
Output:
backup
bin
Books
Desktop
docker
Documents
Downloads
GNS3
…
```

Классы символов POSIX

При использовании диапазонов символов существует одна проблема. Диапазоны трактуются по-разному в зависимости от настроек локали. Например, в некоторых ситуациях диапазон [A-Z] интерпретируется в лексикографическом порядке, то есть он включает все алфавитные символы, кроме символа a в нижнем регистре. Для решения этой проблемы в стандарте POSIX придумали несколько классов, описывающих разные множества символов. Некоторые из них:

- `[:alnum:]` — Алфавитно-цифровые символы; эквивалент диапазона [A-Za-z0-9] в ASCII.
- `[:alpha:]` — Алфавитные символы; эквивалент диапазона [A-Za-z] в ASCII.
- `[:digit:]` — Цифры от 0 до 9.
- `[:lower:]` и `[:upper:]` — Символы нижнего и верхнего регистра соответственно.
- `[:space:]` — Пробельные символы, включая пробел, табуляцию, возврат каретки, перевод строки, вертикальную табуляцию и перевод формата.

Наличие классов символов не дает удобного способа выражения частичных диапазонов, таких как [A-M]. Пример использования:

```
ls ~ | grep '[[:upper:]].*'
Output:
Books
Desktop
Documents
Downloads
GNS3
GOG Games
Learning
Music
...
```

Extended Regular Expressions

Особенности, рассматривавшиеся выше, поддерживаются практически всем POSIX-совместимыми приложениями и приложениями, реализующими BRE (например grep и потоковый редактор sed). Стандарт POSIX ERE позволяет создавать более выразительные регулярные выражения, однако не все программы умеют с ним работать. Диалект ERE поддерживался программой egrep, но GNU-версия grep также поддерживает расширенные регулярные выражения при вызове с ключом `-E`.

В ERE множество метасимволов расширяются следующими:

```
( ) { } ? + |
```

Чередование

Чередование позволяет выбирать совпадение с одним из нескольких выражений. Так же как выражения в квадратных скобках позволяют одному символу соответствовать множеству указанных символов, чередование позволяет находить совпадение с множеством строки или других регулярных выражений. Чередование обозначается метасимволом вертикальной черты:

```
echo "AAA" | grep -E 'AAA|BBB'
Output:
AAA

echo "BBB" | grep -E 'AAA|BBB'
Output:
BBB

echo "CCC" | grep -E 'AAA|BBB'
```

Группировка

Элементы регулярных выражений можно объединять и ссылаться на них как на один элемент. Делается это с помощью круглых скобок.

Следующее выражение будет соответствовать именам файлов, начинающихся с bz, gz или zip. Если отбросить круглые скобки, смысл регулярного выражения изменится, и ему будут соответствовать имена, начинающиеся с bz или содержащие gz, или zip.

```
ls /bin | grep -E '^(bz|gz|zip)'
Output:
bzcat
bzgrep
bzip2
bzip2recover
bzless
bzmore
gzexe
gzip
zip
zipdetails
zipgrep
zipinfo
zipsplit
```

Квантификаторы

Квантификаторы позволяют определить число совпадений с элементом. BRE поддерживают несколько способов.

Квантификатор `?` означает совпадение с элементом ноль или один раз. Иными словами совпадение с предыдущим элементом необязательно:

```
echo "tet" | grep -E 'tes?t'
Output:
tet

echo "test" | grep -E 'tes?t'
Output:
test

echo "tesst" | grep -E 'tes?t'
Output:
```

В последнем случае совпадения не найдены, так как буква «s» встретилась дважды.

Подобно метасимволу `?`, звездочка (`*`) обозначает необязательный элемент; однако, в отличие от знака вопроса, этот элемент может встречаться любое число раз, а не только единожды. Рассмотрим предыдущий пример, используя вместо знака вопроса звездочку:

```
echo "tet" | grep -E 'tes*t'
Output:
tet

echo "test" | grep -E 'tes*t'
Output:
test

echo "tesst" | grep -E 'tes*t'
Output:
tesst
```

На этот раз все три строки совпали с шаблоном.

Метасимвол `+` действует почти так же, как `*`, но требует совпадения с предыдущим элементом не менее одного раза:

```
echo "tet" | grep -E 'tes+t'
Output:

echo "test" | grep -E 'tes+t'
Output:
test

echo "tesst" | grep -E 'tes+t'
Output:
tesst
```

Теперь с шаблоном не совпала первая строка, так как метасимвол `+` требует хотя бы одного совпадения с предыдущим элементом.

В BRE существуют специальные метасимволы `{` и `}`, которые, в отличие от предыдущих квантификаторов, позволяют выразить минимальное и максимальное число обязательных совпадений. Всего существует четыре возможных способа задания числа совпадений:

- `{n}` — Совпадение, если предыдущий элемент встречается точно n раз.
- `{n,m}` — Совпадение, если предыдущий элемент встречается не менее n и не более m раз.
- `{n,}` — Совпадение, если предыдущий элемент встречается n или более раз.
- `{,m}` — Совпадение, если предыдущий элемент встречается не более m раз.

Пример:

```
echo "tet" | grep -E "tes{1,3}t"
Output:

echo "test" | grep -E "tes{1,3}t"
Output:
test

echo "tesst" | grep -E "tes{1,3}t"
Output:
tesst

echo "tessst" | grep -E "tes{1,3}t"
Output:
tessst

echo "tesssst" | grep -E "tes{1,3}t"
Output:
```

С шаблоном совпали только те строки, где буква s встречается один, два или три раза.

Регулярные выражения на практике

В качестве заключения рассмотрим пару примеров, как регулярные выражения могут использоваться на практике.

Проверка номеров телефонов

Допустим, у нас имеется список номеров телефонов. Корректный формат номера: (nnn) nnn-nnnn. В списке 10 номеров, три номера имеют некорректный формат.

```
cat phonenumbers.txt
Output:
(185) 136-1035
(95) 213-1874
```

```
(37) 207-2639
(285) 227-1602
(275) 298-1043
(107) 204-2197
(799) 240-1839
(218) 750-7390
(114) 776-2276
(7012) 219-3089
```

Задача — найти неправильные номера. Для этого можно использовать следующую команду:

```
grep -Ev '^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$' phonenumbers.txt
Output:
(95) 213-1874
(37) 207-2639
(7012) 219-3089
```

Здесь мы использовали параметр `-v`, чтобы обратить сопоставление и вывести только строки, не соответствующие указанному выражению. Поскольку круглые скобки считаются метасимволами в ERE, мы экранировали их обратными слешами, чтобы они интерпретировались как литералы.

Поиск некорректных имен файлов

Команда find поддерживает проверку, основанную на регулярном выражении. Тут важно помнить одну деталь: если grep выводит строку, содержащую совпадение с регулярным выражением, то `find` требует точного совпадения пути. Допустим нам нужно найти в директории имена файлов и каталогов, содержащие пробелы и другие, потенциально вредные символы:

```
find . -regex '.*[^-_./0-9a-zA-Z].*'
```

Последовательность `.*` в начале и конце означает любое количество любых символов. Такой прием необходим, так как `find` требует совпадения всего пути. В квадратных скобках находится отрицание множества допустимых символов в именах файлов. Таким образом любое имя файла или каталога, содержащее хотя бы один символ, не являющиеся дефисом, подчеркиванием, цифрой или латинской буквой, попадет в вывод.

Заключение

В этом гайде мы рассмотрели лишь несколько примеров практического применения регулярных выражений. В начале вам будет сложно придумывать длинные регулярки. Но со временем вы получите опыт, используя регулярные выражения для поиска в разных приложениях, поддерживающих такую возможность.