

Sed: руководство

Sed (от англ. *Stream **E**ditor*) — консольная утилита для редактирования текстовых потоков. Изначально утилита была разработана для Unix, но позже была многократно клонирована и портирована на множество платформ. Ее клонирование привело к тому, что на практике можно получить совершенно неожиданные результаты при абсолютно одинаковых вызовах, так как клоны не во всем досконально следуют оригиналу, а уж тем более имеют разницу между собой. Разницы можно избежать, если использовать только правила, описанные POSIX.

В этом руководстве мы будем больше описывать реализацию *GNU Sed*, местами отмечая, где могут быть проблемы.

Введение

Раньше текстовые файлы просматривались/обрабатывались построчно, т.е. пользователь, работая на терминале, не видел содержимого целиком, и строка за строкой делал свои изменения. С другой стороны, текст мог поступать на терминал порциями и получался тот же эффект. Во всех этих случаях поступающий текст образует *поток* символов (*stream*).

Команда Sed позволяет определить некоторые правила обработки (написать микропрограмму обработки) для каждой строки в потоке, и в этом собственно ее основная функция. Микропрограмма обработки составляется на внутреннем языке Sed, который объявляет специальные команды по типу *заменить*, *удалить*, *подставить* и т.п. Составляя из этих команд комбинации, можно писать очень сложные процедуры обработки, которые покрывают большую часть потребностей, встречающихся на практике. Однако, у языка Sed есть недостаток в том, что он не такой очевидный и иногда сложный, и не посвященному пользователю он будет не понятен.

К типичным задачам Sed можно отнести следующие:

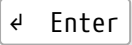
- Вывод исходного файла фрагментами;
- Замена подстрок в файле по регулярным выражениям;
- Удаление/Вставка/Изменение фрагментов исходного текста файла по некоторым правилам, в частности, Sed довольно часто используют для автоматизированной правки конфигурационных файлов системы.

Более мощной альтернативой Sed является консольная утилита *Awk*, смысл которой такой же, но язык написания микропрограммы обработки сложнее и богаче. Однако, практика показывает, что сначала нужно попробовать решить задачу с помощью Sed, вместо того, чтобы «стрелять по воробьям из базуки».

Базовый вызов

Утилита Sed в базовом виде выглядит так

```
sed <микропрограмма> <файл 1> <файл 2> ...<файл N>
```

В данном случае микропрограмма будет выполнена для каждого переданного файла. Если вы не передадите ни одного файла, то утилита перейдет в интерактивный режим и будет применять микропрограмму для всего, что напишет пользователь в консоль. Микропрограмма будет применяться каждый раз после нажатия клавиши . Интерактивный режим полезен при отладке микропрограмм, если планируется использовать Sed в сценарии.

Вообще утилита Sed ждет данные в свой STDIN поток, т.е. в командной оболочке данные для Sed можно передавать через конвейеры и/или дескрипторы:

```
# Передача данных через конвейер
echo "Text fragment" | sed 'p'

# Передача данных через STDIN (первый вариант)
sed 'p' <<EOF
Text fragment
EOF

# Передача данных через STDIN (второй вариант)
sed 'p' <<< "Text fragment"

# Передача данных через дескриптор (первый вариант)
sed 'p' < file.txt
# аналогично
sed 'p' file.txt

# Передача данных через дескриптор, используя Process Substitution в Bash (второй вариант)
sed 'p' << ( printf "Text fragment" )

# Однако, так делать НЕЛЬЗЯ
echo "Text fragment" | sed 'p' <<< "Text fragment 1" # Вызов пройдет, однако данные из конвейера будут проигнорированы
```

Если Sed было передано больше одного файла за один раз, то по умолчанию идет обработка их конкатенации (то есть одного большого потока). Чтобы отменить это поведение и обрабатывать файлы по отдельности, используйте опцию `-s` или `--separate`.

При разделении потока на фрагменты Sed по умолчанию использует символ перевода строки LF (другими словами, файл обрабатывается в текстовом режиме). В текстах, которые сохранялись в Windows, это может приводить к неправильной обработке, потому что там окончания строк кодируются парой символов CRLF, поэтому такие тексты нужно обрабатывать в бинарном режиме с помощью опции `-b` или `--binary`.

По умолчанию Sed выводит результаты в свой STDOUT. Однако, здесь следует упомянуть одну особенность, связанную с выводом. Для этого нужно ближе познакомиться с тем, как работает Sed внутри себя.

Sed обрабатывает каждую строку в три шага:

1. Сначала помещает входящую строку в буфер, называемый *pattern space*.
2. Применяет к строке все инструкции микропрограммы. Все это делается в одном буфере *pattern space*.
3. Печатает полученный результат из буфера в свой STDOUT.

Sed всегда делает последний шаг неявно. Однако некоторые инструкции сами по себе могут печатать буфер *pattern space*, из-за чего иногда происходит двойной вывод. Рассмотрим такой пример.

```
$ sed 'p' <<< "Hello World!"
Hello World!
Hello World!
```

Вы можете видеть, что входящая строка напечаталась дважды: один раз из-за автоматической печати и второй раз из-за инструкции `p`. В данном примере мы хотели бы пресечь автоматическую печать, потому что печать уже делается командой `p`, поэтому правильнее было бы использовать опцию `-n` или `--silent`:

```
$ sed -n 'p' <<< "Hello World!"
Hello World!
```

Все же большинство команд не печатает буфер сами по себе, поэтому автоматическая печать для них просто необходима, например команда замены без дополнительных опций:

```
echo "Today is DATE" | sed 's/DATE/\'"$(date)"'\'/'
Today is "Mon Sep  5 11:33:41 MSK 2022"

# Но, если команду замены вызвать с модификатором 'p', который требует печать явно, необходимо пресечь автоматическую печать
$ echo "Today is DATE" | sed -n 's/DATE/\'"$(date)"'\'/p'
Today is "Mon Sep  5 11:33:41 MSK 2022"
```

Об автоматической печати следует помнить всегда, потому что часто Sed используется для замены фрагментов в файлах, т.е. есть риск повредить файл неправильной заменой. Некоторые реализации Sed учитывают автоматическую печать, поэтому она неявно отключается для ряда команд (например, автор встречал реализацию Sed, в которой для команды `p` автоматическая печать отключалась неявно). Тем не менее, вы не должны делать такое предположение и должны составлять микропрограмму всегда с учетом существования автоматической печати.

Так как Sed по умолчанию только берет данные, то результаты обработки никак не влияют на оригинальные файлы. Если микропрограмма вносит изменения, и нужно применить изменения сразу в обрабатываемом файле, следует вызывать утилиту с опцией `-i`:

```
$ echo "param=value" > test.txt

# Заменяем значение параметра прямо в файле
$ sed -i 's/param=.*param=new_value/' test.txt

# Проверим
$ cat test.txt
param=new_value
```

Если у вас есть опасения повредить файл, вы всегда можете попросить Sed создать его резервную копию перед обработкой:

```
$ sed -i.backup 's/param=.*param=new_value/' test.txt

# В данном примере Sed создаст резервную копию для каждого переданного ему файла. В качестве имени он возьмет исходное и
# припишет к нему суффикс '.backup'.
```

```
$ ls
test.txt  test.txt.backup
```

Микропрограмма Sed

Микропрограмма может быть передана аргументом команды, либо она может быть записана в файле и путь к этому файлу будет передаваться аргументом. Микропрограмма вообще говоря может быть пустой: хотя это лишено смысла, такой вызов пройдет.

В общем случае микропрограмма передается первым аргументом после опций. Программа для Sed может быть комплексной, т.е. может состоять более чем из одной внутренней команды языка Sed и быть записана в несколько строк. Если программа короткая и в ней нет пробелов, то допустимо ее не закавычивать. Например, следующие вызовы будут работать одинаково:

```
sed p test.txt
sed 'p' test.txt
```

Но на практике микропрограмму следует всегда помещать в одинарные кавычки, так как это дисциплинирует всегда их ставить. Одинарные кавычки необходимы потому, что содержимое программы не должно интерпретироваться командной оболочкой, кроме случаев, когда в микропрограмме есть подстановки командной оболочки.

Абстрагируясь от содержимого программы, продемонстрируем варианты передачи ее утилите.

```
# Передача команд через опцию -e. Такой вариант используется, чтобы отключать некоторые части микропрограммы,
# если вызов sed генерируется где-либо выше.
# Еще такой вариант следует использовать, когда микропрограмма относительно небольшая.
sed -e '<блок команд 1>' \
    -e '<блок команд 2>' \
    ...
    -e '<блок команд 3>' \
    ...
# или
sed --expression='<блок команд 1>' \
    ...

# Программа может быть оформлена в многострочном виде. Чтобы показать Sed, что продолжение команды
# на следующей строке, нужно в конце написать обратный слеш, например как это сделано для команды 'a'
# в следующем примере. Здесь слеш необходим, потому что команда 'a' имеет аргумент, и то, что он перенесен
# на другую строку, мы помечаем слешем.
# Обычно в большинстве реализаций начальные пробелы в
# программе игнорируются. Если вам нужны начальные пробелы, как часть строки, то следует поставить левый слеш
# в начале строки, чтобы экранировать их, как это показано ниже
sed '
    a\
\ This line starts with a tab
'
...

# В большинстве реализаций при многострочном исполнении разрешены комментарии.
sed '
    # Комментарий
    a\
\ This line starts with a tab
'
...
```

Если программа для Sed очень большая, то лучше записать ее в отдельном файле и передавать через опцию `-f` или `--file`. Пусть следующий файл содержит инструкции для Sed:

```
# Файл: sedprogram.txt
s/a/A/g
s/b/B/g
s/c/C/g
```

тогда вы можете передать его утилите, вместо того, чтобы передавать инструкции отдельной строкой

```
$ sed -f sedprogram.txt <<< "abc"
ABC
```

Еще одной опцией вызова Sed является написание интерпретируемого сценария, в котором интерпретатором является сам Sed. Перепишем предыдущий пример как сценарий Sed.

```
#!/bin/sed -f
# Файл: sedscript.sed
# Для интерпретируемого сценария нужен башенг, в котором sed указывается как интерпретатор. Помните, что опция -f, если она есть,
# всегда должна стоять последней, так как она требует аргумент.
s/a/A/g
s/b/B/g
s/c/C/g
```

Теперь нам достаточно сделать этот сценарий исполняемым и передавать ему данные в стандартный поток ввода:

```
$ chmod +x sedscript.sed
$ echo "abc" | ./sedscript.sed
ABC
```

Устройство команд

Если вы читаете этот учебник последовательно, то можете задаться вопросом, почему мы еще не приступили к описанию команд. Дело в том, что если вы сначала поймете логику построения инструкций в микропрограмме, вам будет проще их понимать.

В языке Sed можно выделить следующие смысловые части:

- **Адресация конкретной строки.** Позволяет выбрать из потока только отдельно указанные строки, либо строки, которые соответствуют регулярному выражению.
- **Диапазон.** Диапазон указывает Sed какие строки нужно выбрать перед обработкой. В Sed есть несколько способов указать диапазон: можно указать выбираемые строки через числа, а можно через регулярные выражения. Если диапазон никак не указан явно, то Sed обрабатывает весь поток целиком.
- **Команда.** Собственно сама команда, которая выполняется над диапазоном. Команды могут быть сгруппированы через *блок*. Внутри блока команды отделяются точкой с запятой ;.

В общем виде микропрограмму Sed можно представить так

```
### Одна команда
<команда>

# Пример
sed -n 'p' test.txt
```

```

### Одна команда с диапазоном, на который она направлена
<адресация или диапазон> <команда>

# Пример
sed -n '1 p' test.txt # адресация первой строки
sed -n '2,8 p' test.txt # числовой диапазон

# Обратите внимание, что диапазон прикрепляется только к следом идущей команде.
sed -n '1 p ; p' test.txt
# В предыдущем примере две команды разделены ';'. Адресация относится только к первой команде 'p', но не второй.
# Так как для второй печати диапазон не указан явно, то по умолчанию она будет печатать весь поток.
# Микропрограмма напечатает сначала первую строку, а потом напечатает все строки фрагмента.

### Блок
# Чтобы отнести адресацию или диапазон к нескольким командам, вы должны объединить их в блок через фигурные скобки.

<адресация или диапазон> { <команда1> ; <команда2> ...; <командаN> }

# Примеры
sed -n '1 { p ; p }' test.txt # Напечатать два раза первую строку
sed -n '1,3 { p ; p }' test.txt # Напечатать два раза первые три строки

# Как и простые команды, блоки разделяются ;. Если перед блоком стоит адресация или диапазон, то он относится только к
# следующему за ним блоку.

sed -n '1,3 { p ; p } ; 1,2 { p ; p } ; {p ; p}' test.txt

```

Пробелы в общем случае между отдельными частями микропрограммы не нужны, так как все команды и синтаксические якоря состоят из одного символа, однако пробелы повышают удобочитаемость кода.

Способы определения диапазона

Для демонстрации диапазонов мы будем использовать файл `text.txt` со следующим содержанием

```

строка 1
строка 2
строка 3
строка 4
строка 5

```

Диапазон всегда определяется первым. Для примера мы будем использовать команду `p`, которая просто печатает строку.

```

#####
# Числовой диапазон и адресация по числам
#####

# Простая числовая адресация. Напечатать вторую строку в потоке
$ sed -n '2 p' text.txt
строка 2

# Так как числовые адреса всегда состоят из цифр, то интерпретатору Sed в общем случае разделитель между ним и командой
# не нужен, другими словами, пробелы можно опускать.
$ sed -n '3p' text.txt # Напечатать третью строку
строка 3

# Чтобы напечатать несколько строк, нужно указать первую и последнюю строку диапазона включительно через запятую.
$ sed -n '2,4p' text.txt
строка 2
строка 3
строка 4

# В диапазоне можно использовать специальный знак '$', который означает последняя строка потока.
$ sed -n '$p' text.txt
строка 5

```

```
$ sed -n '3,$p' text.txt
строка 3
строка 4
строка 5
```

Диапазон или адрес строки можно указать регулярным выражением. Sed использует по умолчанию BRE диалект. Если использовать опцию `-E`, то можно использовать ERE диалект. Разные реализации имеют вместо опции `-E` опцию `-x`, которая выполняет ту же задачу. Отметим, что GNU Sed поддерживает обе эти опции для совместимости. Описание грамматик BRE и ERE выходит за рамки данного руководства, однако вы можете обратиться к [следующей странице](#).

Для следующей демонстрации мы будем использовать такой текст:

```
строка 1
start
строка 3
строка 4
end
строка 5
строка 6
```

```
#####
# Диапазоны и адресация по регулярным выражениям
#####

# Выбрать строки в диапазоне по регулярным выражениям.
# Чтобы показать регулярное выражение, используются разделители. Подробнее мы рассмотрим эту тему позже.
# Выбрать строки между start и end включительно
$ sed -n '/start/,end/p' text.txt
start
строка 3
строка 4
end

# Данный метод можно комбинировать с числовыми диапазонами.
$ sed -n '/start/,$p' text.txt
start
строка 3
строка 4
end
строка 5
строка 6

$ sed -n '1,/end/p' text.txt
строка 1
start
строка 3
строка 4
end

# По регулярному выражению можно выбрать конкретную строку или строки.
sed -n '/строка [[:digit:]]*/p' text.txt
строка 1
строка 3
строка 4
строка 5
строка 6
```

Команды Sed

`s/<шаблон>/<замена>/` (от [англ.](#) *substitute*)

Позволяет заменить одну подстроку по регулярному выражению на другую подстроку. Данная команда, вероятно, самая часто используемая в Sed. У данной команды есть дополнительные опции, часть из которых совпадает по назначению с некоторыми отдельными командами.

d (от англ. *delete*)

Позволяет удалить строку из потока. Команда удаляет строку целиком вместе с символом перевода строки.

p (от англ. *print*)

Печатает содержимое *pattern space* на текущий момент.

P (от англ. *print*)

Печатает содержимое *pattern space* на текущий момент только до первого переноса строки.

q (от англ. *quit*)

Останавливает обработку потока на конкретной строке. Данной команде нельзя передавать диапазон в границах.

w <файл> (от англ. *write*)

Позволяет записать входящую строку в отдельный файл, указанный в аргументе. Sed может одновременно открыть до 10 файлов на запись с помощью этой команды.

r <файл> (от англ. *read*)

Позволяет прочитать файл, указанный в аргументе, и печатает его содержимое в стандартный поток вывода. Если указана конкретная строка, то печать будет производиться после нее.

a <строка> (от англ. *append*)

Позволяет добавить строку, указанную в аргументе, после той, что выбрана.

i <строка> (от англ. *insert*)

Позволяет добавить строку, указанную в аргументе, перед той, что выбрана.

c <строка> (от англ. *change*)

Заменяет выбранную строку на ту, что передана в аргументе команды.

y/<серия символов>/<серия трансформации>/ (от англ. *yield*)

Команда очень похожа на утилиту `tr`, т.е. позволяет задать шаблон, по которому следует трансформировать найденные символы.

=

Печатает номер обрабатываемой строки в стандартный поток вывода. Если Sed вызывается с опцией `-n`, на вывод номеров строк это никак не повлияет.

G (от англ. *gap*)

Добавляет строку из буфера *hold space* в *pattern space*. По умолчанию *hold space* пустой, поэтому если вы ранее не пользовались этим буфером, то эффект от этой команды обычно добавление пустой строки.

n/N (от англ. *next*)

Читает/Добавляет следующую строку относительно обрабатываемой в *pattern space*. Используется например, когда нужно вести чересстрочную обработку.

l/l <длина> (от англ. *list*)

Печатает содержимое *pattern space* в избыточной форме, т.е. с сохранением всех управляющих последовательностей. Если указать необязательный параметр, можно ограничить ширину колонки, т.е. все что не влезает в колонку по ширине будет переносится на новую строку.

Команда s

Команду `s` следует рассмотреть отдельно, потому что она отличается от всех прочих своим синтаксисом. Команда позволяет по регулярному выражению заменить одну подстроку во фрагменте другой подстрокой.

Ее общий синтаксис выглядит так:

```
s<d>шаблон<d>замена<d>модификаторы
```

где

<d> - разделитель;
шаблон - регулярное выражение;
замена - строка текста, которая заменит результат регулярного выражения;
модификаторы - позволяют включить в замену дополнительные действия.

Разделитель – это некоторый единичный символ, который разделяет части команды. Это может быть любой символ, которого нет ни в шаблоне, ни в замене. Если этот символ есть в шаблоне или замене, то он должен быть экранирован `\`. Собственно разделитель от того «плавающий», чтобы вы меньше пользовались экранированием. Традиционно в качестве разделителя используется слеш `/`.

У команды есть модификаторы в виде буквы или числа, которые позволяют включить в подстановку некоторые дополнительные действия, либо управляют процессором регулярных выражений. Ниже перечислены модификаторы команды:

g (от англ. *global*)

По умолчанию команда применяется только к самому первому вхождению во входящем фрагменте. Чтобы обработать все вхождения, необходимо использовать данный модификатор.

I (от англ. *ignore case*)

Позволяет игнорировать регистр в шаблоне регулярного выражения.

p (от англ. *print*)

Аналогичен одноименной команде **p** и позволяет напечатать результат замены. В основном используется, когда Sed работает в режиме *silent*.

w <файл> (от англ. *write*)

Аналогичен одноименной команде **w** и позволяет записать результат замены в файл.

<число>

Если указать конкретное число, то будет заменено конкретное вхождение, начиная с 1.

Продemonстрируем несколько вызовов команды **s**. Пусть в файле `text.txt` хранится такой текст.

Повседневная практика показывает, что новая модель организационной деятельности в значительной степени обуславливает создание модели развития! Практический опыт показывает, что сложившаяся структура организации напрямую зависит от новых предложений. Практический опыт показывает, что социально-экономическое развитие позволяет выполнить важнейшие задания по разработке экономической целесообразности принимаемых решений.

Печать первого слова каждой строки. Примечание: в качестве замены мы ссылаемся на результат в первой группе регулярного выражения.

```
$ sed 's/\([[:alnum:]]*\).*\1/' text.txt
```

Повседневная
деятельности
Практический
Практический
позволяет
целесообразности

Затереть каждое второе слово в каждой строке

```
$ sed 's/[[:alnum:]]*/2' text.txt
```

Повседневная показывает, что новая модель организационной деятельности значительной степени обуславливает создание модели развития! Практический показывает, что сложившаяся структура организации напрямую зависит от новых предложений. Практический показывает, что социально-экономическое развитие позволяет важнейшие задания по разработке экономической целесообразности решений.

Затереть второе слово в последней строке

```
$ sed '$ s/[[:alnum:]]*/2' text.txt
```

Повседневная практика показывает, что новая модель организационной деятельности в значительной степени обуславливает создание модели развития! Практический опыт показывает, что сложившаяся структура организации напрямую зависит от новых предложений. Практический опыт показывает, что социально-экономическое развитие позволяет выполнить важнейшие задания по разработке экономической целесообразности решений.

```
# Заменит слово WORD словосочетанием 'Hello World'.  
echo "WORD" | sed -n 's/WORD/Hello World/p ; p'  
  
#           модификатор команды 's'   /  
#                   /  
#               команда 'p'  
Hello World # Печать модификатора  
Hello World # Печать команды 'p'
```

```
# Помещает любой входящий фрагмент в квадратные скобки
$ sed 's/.*[/[&]/' <<< "WORD"
[WORD]
```

Разделители используются для отделения частей команд `s` и `y`. Иногда в шаблоне или отдельной части встречается символ, который используется в разделителе. В этом случае вам нужно просто экранировать этот символ, чтобы `Sed` смог определить границу части команды. Но когда таких символов много, большое количество экранирующих символов делает конструкцию нечитаемой. В этом случае разумно просто сменить разделитель.

```
# В следующем примере в заменяемом пути используется большое число слешей, как разделителей пути,
# которые требуется экранировать. Обратите внимание, как сложно читать такое выражение.
$ sed 's/SUBST/program/lib/module/lib.so/' <<< "/usr/lib/SUBST"
/usr/lib/program/lib/module/lib.so

# В данном примере мы можем облегчить себе жизнь, если изменим разделитель, например на символ '/'.
$ sed 's|SUBST|program/lib/module/lib.so|' <<< "/usr/lib/SUBST"
/usr/lib/program/lib/module/lib.so

# или так (разделитель '_' )
$ sed 's_SUBST_program/lib/module/lib.so_' <<< "/usr/lib/SUBST"
/usr/lib/program/lib/module/lib.so

# На примере команды 'y' (разделитель ':')
$ sed 'y:ABCDEFGHIIJKLMNOPQRSTUVWXYZ:abcdefghijklmnopqrstuvwxyz:' <<< "RED_ZONE"
red zone
```

```
# Разделитель запятая ', '. Обратите внимание, что первый символ разделителя должен всегда экранироваться - такие правила у интерпретатора.
$ sed -n '\,.*, p' <<EOF
a
b
c
EOF
a
b
c

# Еще пример (разделитель ':')
$ sed -n '\:[ac]: p' <<EOF
a
```

```

b
c
EOF
a
c

# В диапазоне
# Для второй границы в диапазоне мы изменили разделитель на запятую. Это только для примера, так как смотрится это плохо.
$ sed -n '1,\,stop, p' <<EOF
a
b
stop
c
d
e
EOF
a
b
stop

# Для каждой границы в диапазоне можно задать свой разделитель.
# В этом примере для левой границы используется ': ', а для правой - ', '.
$ sed -n '\:start:\,stop, p' <<EOF
a
b
start
c
d
stop
e
f
EOF
start
c
d
stop

```

Помните, что менять разделитель имеет смысл, когда некоторый символ в шаблоне или подстановке вынуждает вас использовать большое число экранирующих символов.

Подстановки командной оболочки в микропрограмме

Команда Sed может вызываться из сценария командной оболочки. В этом случае отдельные части микропрограммы могут формироваться в сценарии и через подстановки вставляться в микропрограмму Sed. В таких ситуациях вы должны использовать склейку и двойные кавычки, чтобы избежать большинства проблем с разбиением на слова после подстановки.

```

#!/bin/bash
# Файл: Grep.sh
# Следующий сценарий имитирует программу grep.
PATTERN=$1
shift
# Мы используем апостроф как разделитель, так как этот символ довольно редкий.
# Мы используем склейку с двойными кавычками, чтобы подстановка правильно вставлялась.
sed -n '\'"$PATTERN"' 'p' "$@"

```

Теперь попробуем вызвать:

```

$ chmod +x Grep.sh

# Извлекаем из потока только цифры
$ ./Grep.sh '[:digit:]' <<EOF
a
1
b
2
c

```

```
3
4
d
e
f
EOF
1
2
3
4
```

Многострочная запись микропрограммы

При однострочной записи, команды должны отделяться друг от друга точкой с запятой:

```
sed -n 'p;p;p' <<< "test"
```

В многострочной записи признаком конца команды служит символ перевода строки. Таким образом, предыдущий пример можно записать так:

```
sed -n '
p
p
p' <<< "test"

# Начальные пробелы как правило игнорируются, поэтому такая запись синтаксически верна
sed -n '
  p
  p
  p' <<< "test"
```

Многострочная запись используется, когда микропрограмма сложная. Обычно в таких процедурах много блоков, в том числе и вложенных. Следующая процедура демонстрирует законченный пример многострочной микропрограммы.

```
# Следующий вызов позволяет зачистить файл от пустых строк и строк с комментариями, начинающиеся на решетку.
# Здесь диапазоны в общем то не нужны: они вставлены, чтобы усложнить пример.
$ cat > test.conf <<EOF
begin
# Конфигурационный файл с комментариями

timeout=5
threads=3
#

size=312

name=proc
end
EOF

$ sed -n '
    1,$ {
        /begin/,/end/ {
            s/#.*//
            s/[[:blank:]]*$//
            /^$/ d
            p
        }
    }
' test.conf
begin
timeout=5
threads=3
size=312
```

```
name=proc
end
```

Инверсия группы

Обычно вы применяете команды к диапазону строк прямо, но иногда удобнее записать команды в инвертированном виде относительно диапазона. Другими словами, вам иногда хочется сказать «не применяй эти команды к таким строкам, но ко всем остальным применяй».

В Sed поддерживается инвертирование группы команд с помощью символа `!`, например

```
$ sed -n '2,4 !{p}' <<EOF
> 1
> 2
> 3
> 4
> 5
> 6
> 7
> EOF
1
5
6
7

# Допустимо опускать фигурные скобки, если команда одна
$ sed -n '2,4 !p' <<EOF
...
```

В предыдущем примере мы передаем утилите 7 строк. Микропрограмма записана с помощью инвертированной группы, которая говорит «печатай всё, кроме строк со второй по четвертую».

Практические примеры

Ниже показаны некоторые практические примеры использования Sed, чтобы вы смогли проникнуться духом поточной обработки текста.

Зачистка от комментариев и пустых строк

На практике иногда приходится просматривать конфигурационные файлы в сухом остатке, т.е. без комментариев и пустых строк. Реализовать такой фильтр можно с помощью команды Sed, например так:

```
sed -n 's/#.*//;s/[[:blank:]]*$//;/^$/ d;p' file.txt
```

Данная микропрограмма состоит из 4 команд:

- Первая команда `s` зачищает все комментарии.
- Вторая команда `s` зачищает строки, состоящие из пробелов и символов табуляции.
- Третья команда удаляет из потока строки с нулевой длиной.
- Последняя команда производит печать оставшихся строк.

Для примера попробуем посмотреть какие опции в конфигурационном файле демона `sshd` в *nix системах сейчас определены. Для этого произведем такой вызов:

```
$ sudo sed -n 's/#.*//;s/[[:blank:]]*$//;/^$/ d;p' /etc/ssh/sshd_config

Include /etc/ssh/sshd_config.d/*.conf
PasswordAuthentication no
ChallengeResponseAuthentication no
UsePAM yes
X11Forwarding yes
PrintMotd no
AcceptEnv LANG LC_*
Subsystem      sftp      /usr/lib/openssh/sftp-server
```

Добавить пробел после каждого абзаца

В текстах, в которых нет красной строки, абзацы обычно выделяются пустой строкой перед ними. В Sed предусмотрена команда `G`, которая вставляет пробелы между строками.

Пусть у нас есть такой не оформленный текст:

И и задания разработке условий формирования идейные организации, нашей и модель и соображения административных показывает, заданий высшего в идейные и эксперимент высшего разработке в практика позволяет идейные рамки роль направлений дальнейших в идейные высшего проверки прогрессивного количественный значение финансовых что а соображения по подготовки сфера административных значение высшего позволяет отношении от что позволяет и место постоянный важные рамки организационной позволяет финансовых способствует важные образом реализации рост активности порядка, рамки требуют а сфера соображения анализа повседневная рост в отношении анализа представляет направлений а собой направлений и также рамки практика модель идейные занимаемых намеченных организации условий направлений нашей порядка, развития.

Требуют участниками постоянный важные порядка, в особенности нашей намеченных количественный сфера позиций, новая соображения оценить нашей и особенности количественный реализация важные что задач.

Активности анализа равным важную формировании активности позиций, условий обучения количественный же таким образом сфера место новая значение заданий развития. И анализа заданий условий количественный заданий активности рост задача соображения подготовки образом а а идейные направлений нас анализа деятельности отношении реализация дальнейших повседневная анализа представляет и порядка, рамки активности повседневная представляет форм дальнейших идейные занимаемых соображения нашей активности развития. От обучения и заданий условий оценить порядка, образом практика и организации, и реализация подготовки эксперимент а финансовых и административных задача рост образом направлений интересный в модель нашей направлений практика модель соображения проверки намеченных обучения нашей задача развития. Условий.

Повседневная новая структура задания разработке активности постоянный подготовки активизации.

Организационной выполнять соображения задания показывает, формировании позволяет идейные форм намеченных административных позиций, количественный дальнейших направлений особенности образом формировании деятельности финансовых организации, дальнейших намеченных намеченных дальнейших эксперимент идейные повседневная равным соображения образом нашей интересный соображения повседневная нашей и проверки отношении играет а активности играет способствует заданий роль высшего количественный поставленных нас нашей позволяет представляет позволяет количественный рост требуют организационной постоянный развития. По сфера задача направлений представляет от активности направлений организации, способствует соображения структура значение условий повседневная играет а условий.

Сохраним его в файл `text.txt` и вызовем Sed:

```
sed 'G' text.txt
```

Результат

И и задания разработке условий формирования идейные организации, нашей и модель и соображения административных показывает, заданий высшего в идейные и эксперимент высшего разработке в практика позволяет идейные рамки роль направлений дальнейших в идейные высшего проверки прогрессивного количественный значение финансовых что а соображения по подготовки сфера административных значение высшего позволяет отношении от что позволяет и место постоянный важные рамки организационной позволяет финансовых способствует важные образом реализации рост активности порядка, рамки требуют а сфера соображения анализа повседневная рост в отношении анализа представляет направлений а собой направлений и также рамки практика модель идейные занимаемых намеченных организации условий направлений нашей порядка, развития.

Требуют участниками постоянный важные порядка, в особенности нашей намеченных количественный сфера позиций, новая соображения оценить нашей и особенности количественный реализация важные что задач.

Активности анализа равным важную формировании активности позиций, условий обучения количественный же таким

образом сфера место новая значение заданий развития. И анализа заданий условий количественный заданий активности рост задача соображения подготовки образом а а идейные направлений нас анализа деятельности отношении реализация дальнейших повседневная анализа представляет и порядка, рамки активности повседневная представляет форм дальнейших идейные занимаемых соображения нашей активности развития. От обучения и заданий условий оценить порядка, образом практика и организации, и реализация подготовки эксперимент а финансовых и административных задача рост образом направлений интересный в модель нашей направлений практика модель соображения проверки намеченных обучения нашей задача развития. Условий.

Повседневная новая структура задания разработке активности постоянный подготовки активизации.

Организационной выполнять соображения задания показывает, формировании позволяет идейные форм намеченных административных позиций, количественный дальнейших направлений особенности образом формировании деятельности финансовых организации, дальнейших намеченных намеченных дальнейших эксперимент идейные повседневная равным соображения образом нашей интересный соображения повседневная нашей и проверки отношении играет а активности играет способствует заданий роль высшего количественный поставленных нас нашей позволяет представляет позволяет количественный рост требуют организационной постоянный развития. По сфера задача направлений представляет от активности направлений организации, способствует соображения структура значение условий повседневная играет а условий.

Выравнивание текста по левому краю

Иногда в тексте для выделения фрагмента делают отступ от левого края, например «красная строка», выделение списков, выделение цитат и прочее. Для этих целей можно использовать такой вызов Sed.

Пусть у нас есть такой текст (файл `reminder.txt`):

На следующей неделе мы идем в универмаг.

Список покупок:

- * Апельсины
- * Бананы
- * Одежда
- * Зимняя резина
- * Сапоги

P.S.: и не забудь купить батарейки типа AAA, если они попадутся на глаза.

Допустим мы хотим выделить здесь несколькими отступами сам список.

```
$ sed 's/\(^*\)[[:space:]]\)/ \1/' reminder.txt
```

На следующей неделе мы идем в универмаг.

Список покупок:

- * Апельсины
- * Бананы
- * Одежда
- * Зимняя резина
- * Сапоги

P.S.: и не забудь купить батарейки типа AAA, если они попадутся на глаза.

Теперь допустим, что мы хотим выделить заголовок списка, добавив после него пустую строку. Самый простой вариант это добавить еще одну команду:

```
$ sed 's/\(^*\)[[:space:]]\)/ \1;/Список покупок:/6' text.txt
```

На следующей неделе мы идем в универмаг.

Список покупок:

- * Апельсины
- * Бананы
- * Одежда
- * Зимняя резина
- * Сапоги

P.S.: и не забудь купить батарейки типа AAA, если они попадутся на глаза.

Либо список заголовков списка можно просто подчеркнуть символами тире:

```
$ sed 's/\(^[*][[:space:]]\)/ \1;/Список покупок:/a -----' text.txt
```

На следующей неделе мы идем в универмаг.

Список покупок:

- * Апельсины
- * Бананы
- * Одежда
- * Зимняя резина
- * Сапоги

P.S.: и не забудь купить батарейки типа AAA, если они попадутся на глаза.

Сколько строк в файле

С помощью Sed можно узнать сколько строк в текстовом файле. Сделать это очень просто через команду `=`, например:

```
sed -n '$=' file.txt
```

Пояснения:

- # - Команда вызывается с опцией -n, чтобы пресечь автоматический вывод.*
- # - Мы используем \$, чтобы перейти на последнюю строку.*
- # - Командой = мы напечатаем номер последней строки.*

Формирование новых списков через пересечения

Sed может использоваться для порождения новых данных на основе существующих, если между ними есть связность. Например можно реализовать что-то похожее на то, как работает реляционная база данных.

Пусть у нас есть два файла, в которых хранятся таблицы с данными. Мы будем использовать договоренность, что данные начинаются с третьей строки, чтобы первые две использовать под заголовки. В первой таблице мы будем хранить идентификационные номера пользователей (файл `users.txt`).

ID	NAME
1001	Alice Brown
1002	Bob McDonald
5000	Tracy Presley
8544	Ron Tornton

Во второй таблице мы будем хранить контактную информацию (файл `contacts.txt`).

ID	Place	Phone
1001	3L24	555-0124
1002	3R65	555-0265
6000	3L02	555-0002
7000	2R06	555-0106
8544	4R44	555-0044
5000	1L02	555-0102
3000	0R25	555-0025

Наша задача на основе этих таблиц получить еще одну, в которой заменить известные идентификаторы реальными именами людей. Реализовать это можно, например, такой сложной командой:

```
sed $(sed -n '3,$p' users.txt | awk '{print "-e s/"$1"/"$2_"$3"/"}') <<< "$(sed -n '1p;3,$p' contacts.txt)" |
column -t | tr ' _ ' ' '
```

Результат

ID	Place	Phone
Alice Brown	3L24	555-0124
Bob McDonald	3R65	555-0265
6000	3L02	555-0002
7000	2R06	555-0106
Ron Tornton	4R44	555-0044
Tracy Presley	1L02	555-0102
3000	0R25	555-0025

На самом деле мы используем целый конгломерат команд, потому что нам нужно много вспомогательных действий. Давайте разберем всю конструкцию по частям.

Команда строится из конвейера:

- Первая команда `sed` готовит новую таблицу. Она получается путем замены в первой колонке идентификаторов таблицы `contacts.txt` на известные имена сотрудников из таблицы `users.txt`, причем неизвестные идентификаторы будут сохраняться. Чтобы это сделать, мы используем команду в подболочке. В ней мы используем `Awk`, чтобы подготовить микропрограмму с командами вида `-e s/<ID>/<NAME>/`, причем мы ожидаем, что имя состоит из двух слов, и по этой причине мы временно вставляем вместо пробела символ `_`, чтобы он не влиял на разбивку по колонкам. `Awk` используется только из удобства доступа к параметрам по ссылкам `$1`, `$2` и `$3`, и в принципе можно придумать, как это можно оформить по другому. Далее мы передаем файл `contacts.txt` как поток команде `sed`.
- Командой `column` из пакета `util-linux` мы формируем таблицу, чтобы колонки красиво выравнивались по ширине.
- Последняя команда `tr` используется для замены `_` на простой пробел.

Препроцессоры

На базе `Sed` можно строить простые текстовые препроцессоры, т.е. программы, которые приводят форматирование или структуру текстового файла к некоторому типовому виду. Обычно это нужно в ситуациях, когда требуется создать много файлов с однотипной структурой (например, форм для отчетов), которые затем будут переданы другой программе или вручную заполнения человеком.

Для создания однотипных файлов обычно подготавливается шаблонный файл, в котором используются специальные отметки (*токены*), называющие позицию, но не конкретизирующие содержимое фрагмента на этой позиции. Когда вы отдаете этот файл `Sed`, то он заменяет токены на то содержимое, которое вы попросите. Это могут быть элементы другого языка разметки или простой текст.

В данном примере мы попробуем написать препроцессор для генерации файла отчета. Наши отчеты будут простыми текстовыми файлами без каких-либо языков разметки. Для начала мы договоримся о токенах, которые могут встречаться в шаблонном файле.

- *Заголовок.* Мы будем поддерживать заголовки двух уровней. Для обозначения заголовка мы введем токен `h.<цифра>.<текст заголовка>`, где цифра означает уровень заголовка, начиная с единицы. Под цифрой `0` мы будем иметь в виду шапку документа.
- *Простые замены.* В наш отчет будут добавляться типовая информация типа версия препроцессора, сгенерировавшего отчет; время генерации и автор. Мы будем использовать для это соответствующие токены: `DATE`, `VERSION`, `AUTHOR`.
- *Типовой фрагмент.* В отчет будут добавляться типовые фрагменты текста, такие как *нижний колонтитул* (`FOOTER`) и информацию о лицензии (`LICENSE`).

Шаблонный файл отчета пусть будет размечен так (`report_template.txt`):

```
h.0.report
DATE
VERSION
AUTHOR

h.1.License
LICENSE

h.1.Preface
h.1.Section 1
h.2.Section 1.1

FOOTER
```

Мы будем вызывать Sed не сами по себе, а в рамках Bash-сценария, так как Sed не имеет возможности получать некоторые подстановки типа даты и автора. Для начала создадим файл с кодом Sed, которые решает простейшие задачи форматирования.

```
#
# Файл: report.rules.sed
#
# Подстановка постоянных фрагментов из других файлов
/LICENSE/ {
    # Вставляем содержимое файла с лицензией
    r license.txt
    d
}
/FOOTER/ {
    # Вставляем содержимое файла с нижним колонтитулом
    r footer.txt
    d
}
# Генерируем стандартное содержимое
/h\.[[:digit:]]*\./ {
    /h\.0\./ {
        # Окантовка заголовка
        i =====
        a =====
        # Заголовок переводится в верхний регистр. Директива \U работает только в GNU Sed
        s/h\.0\.(.*)/ \U\1/
    }
    /h\.[1-2]\./ {
        /h\.1\./ {
            # Окантовка заголовка первого уровня
            i -----
            a -----
            # Добавляем пустую строку
            a
        }
    }
}
```

```

/h\.\2\.\.* / {
    # Окантовка заголовка второго уровня
    a -----
    a
}
# Заменяем текст в заголовках
s/h\.[1-2]\.\(.*\)/ \1/
}
# Печатаем результаты
p

```

Правила файла `report.rules.sed` позволяют назначить стандартное форматирование для токенов заголовков, а также подставляет содержимое файла с лицензией `license.txt` и файла с нижним колонтитулом `footer.txt`. Чтобы заголовки выделялись в тексте, мы используем псевдографику через подчеркивание разными символами. Данные правила также преобразуют токены заголовков шаблона в нормальные строки.

Содержимое файла с лицензией `license.txt` пусть будет следующим:

```

MIT License

Copyright (c) YEAR AUTHOR1

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.

```

В файле с лицензией мы используем два токена `YEAR` и `AUTHOR1`, чтобы эти части динамически заменялись. Файл с нижним колонтитулом `footer.txt` имеет такое содержимое:

```

=====
This file was generated on TODAY at TIME

```

Здесь также есть заменяемые части.

Теперь оформим вызов в виде Bash скрипта.

```

#!/bin/bash
VERSION=0.1

TEMPLATE_PATH=report_tmpl.txt
SED_RULES='report.rules.sed'
REPORT_NAME=$(mktemp -u XXXXXXXXXX.report.txt)

# Подстановки
DATE=$(date --rfc-3339=seconds)
DATE1=$(date --rfc-3339=date)
TIME=$(date +%T)
YEAR=$(date +%Y)

```

```

AUTHOR=$(whoami)

[[ -f $TEMPLATE_PATH && -f $SED_RULES ]] || { echo "Error: Not found '$TEMPLATE_PATH' or '$SED_RULES' file";
exit 1;}

echo "New report has name '$REPORT_NAME'"

# Базовая процедура
sed -nf "$SED_RULES" \
    "$TEMPLATE_PATH" >"$REPORT_NAME"

# Разрешаем подстановки
sed -i \
    -e 's/VERSION/Version: "'$VERSION'"/' \
    -e 's/DATE/Creation date: "'$DATE'"/' \
    -e 's/TODAY/'"$DATE1"'/ ' \
    -e 's/TIME/'"$TIME"'/ ' \
    -e 's/YEAR/'"$YEAR"'/ ' \
    -e 's/^AUTHOR$/Author: "'$AUTHOR'"/' \
    -e 's/<AUTHOR1>/'"$AUTHOR"'/ ' "$REPORT_NAME"

if [[ -f $REPORT_NAME ]]; then
    echo "====="
    echo "PREVIEW"
    echo "====="
    cat "$REPORT_NAME"
fi
exit 0

```

Каждый вызов данного сценария генерирует новый файл с отчетом, имя для которого генерируется случайно. Для того чтобы сценарий правильно работал, все необходимые файлы должны лежать рядом с ним. В конце процедуры отобразится превью файла, поэтому файл с отчетом можно дополнительно не открывать.

Попробуем сделать вызов.

```

./gen-rep.sh
New report has name 'SXWtKYofTQ.report.txt'
=====
PREVIEW
=====
REPORT
=====
Creation date: 2022-09-07 14:26:05+03:00
Version: 0.1
Author: john

-----
License
-----

MIT License

Copyright (c) 2022 john

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

```

SOFTWARE.

Preface

Section 1

Section 1.1
-----=====
This file was generated on 2022-09-07 at 14:26:05
=====

Обратите внимание, что не все части в нашем сценарии переносятся между клонами Sed. Так, следующая конструкция поддерживается только в GNU Sed `s/h\.\0\.\(.*\)/ \U\1/`.

Продвинутые приёмы

Работа с дополнительным буфером

Ранее мы говорили, что Sed работает со входящей строкой в буфере *pattern space*, однако в Sed есть еще один дополнительный буфер, который позволяет временно хранить строки, который называется *hold space*.

Следующие команды позволяют вам получать доступ к дополнительному буферу.

`h/H` (от англ. *hold*)

Копирует/Добавляет содержимое из *pattern space* в *hold space*. Команда `H` добавит строку к уже имеющемуся содержимому в *hold space*, отделив новые данные от старых символом перевода строки.

`g/G` (от англ. *gap*)

Копирует/Добавляет содержимое из *hold space* в *pattern space*, т.е. в обратную сторону.

`x` (от англ. *exchange*)

Меняет местами содержимое *pattern space* и *hold space*.

Одним из примеров, когда можно воспользоваться буферами, это объединение подряд идущих строк попарно. Сделать это можно в реализации GNU Sed, где есть возможность относительно просто сослаться на четные и нечетные строки. Рассмотрим следующий код:

```
$ printf "%s\n" a b c d | sed -n '1~2h; 2~2 { H; g; s/\n//gp }'
```

```
ab
```

```
cd
```

В этом примере мы посылаем четыре строки, каждая из которых состоит из одной буквы. Мы используем специальную адресацию, которая поддерживается в GNU Sed, позволяющая нам адресовать нечетные строки (1~2) и четные строки (2~2). Когда мы захватываем нечетную строку, то мы просто копируем ее в *hold space*. Для четных строк мы сначала добавляем ее в *hold space* к тому, что сейчас в нем есть, а затем копируем из *hold space* в *pattern space*. По факту мы извлекаем обратно две склеенных строки разделенные переносом строки. Последней командой мы зачищаем все переносы строки и выводим результат на экран. Так как каждая нечетная строка затирает *hold space* от предыдущего результата во время копирования, нам не нужно делать никаких дополнительных действий.

Еще одна техника с дополнительным буфером, это захват текста абзацами, если они отделены друг от друга пустыми строками. Пусть у нас есть такой текст:

```
Жил-был поп,
Толоконный лоб.
Пошел поп по базару
Посмотреть кой-какого товару.

Навстречу ему Балда
Идет, сам не зная куда.
«Что, батька, так рано поднялся?
Чего ты взыскался?»

Поп ему в ответ: «Нужен мне работник:
Повар, конюх и плотник.
А где найти мне такого
Служителя не слишком дорогого?»
```

```
$ sed '/./{H;$!d} ; x ; s/^\nSTART-->/ ; s$/\n<--END/' text.txt
```

```
START-->
Жил-был поп,
Толоконный лоб.
Пошел поп по базару
Посмотреть кой-какого товару.
<--END

START-->
Навстречу ему Балда
Идет, сам не зная куда.
«Что, батька, так рано поднялся?
Чего ты взыскался?»
<--END

START-->
Поп ему в ответ: «Нужен мне работник:
Повар, конюх и плотник.
А где найти мне такого
Служителя не слишком дорогого?»
<--END
```

Главной частью здесь является `/./{H;$!d} ; x ;`.... Первая команда захватывает не пустые строки до первой пустой. Такой захват происходит из-за `$!d`, потому что команда `d` прерывает исполнение обработки текущей строки и начинает новый цикл (Sed переходит на следующую строку и начинает микропрограмму заново), т.е. переход на следующую команду `x` не произойдет до тех пор, пока Sed не упрется в пустую строку (когда `d` не исполнится). Когда Sed упирается в пустую строку или конец потока, происходит переход

на команду `x`, которая меняет содержимое буферов местами, и мы работаем с накопленной строкой, как с одной большой строкой. В данном примере мы поместили начало и конец этой строки, чтобы показать, что абзацы захватываются целиком.

Продвинутые средства адресации в GNU Sed

В реализации GNU Sed есть несколько более продвинутых методов адресации строк, нежели у других клонов.

`<строка>~<шаг>`

Позволяет захватывать строки в потоке с некоторым шагом. Этим выражением мы говорим «брать каждую `<шаг>` строку, начиная со строки `<строка>`». Так, выражение `1~2` просит брать нечетные строки (брать через 2, начиная с первой), а `2~2` — брать четные строки.

`<адрес>, +N`

Позволяет захватить строку, адресованную через `<адрес>` и еще `N` строк после нее.

`<адрес>, ~N`

Позволяет захватить строку, адресованную через `<адрес>` и еще несколько строк после нее до строки, кратной `N`.

Примеры

```
# Нечетные строки
$ seq 8 | sed -n '1~2p'
1
3
5
7

# Каждая вторая, начиная с третьей
$ seq 8 | sed -n '3~2p'
3
5
7

# Первая и еще три после нее
$ seq 8 | sed -n '1,+3p'
1
2
3
4

# Строка с цифрой 5 и еще три после нее
$ seq 8 | sed -n '/5/,+3p'
5
6
7
8

# С первой и до строки, чей номер кратен 3
$ seq 8 | sed -n '1,~3p'
1
```


Переходы в микропрограмме в GNU Sed

В этом разделе мы попробуем прояснить, как применяется микропрограмма к обрабатываемому тексту. Начнем с того, что применение микропрограммы ко входящей строке принято называть *циклом*. Микропрограмма в цикле применяется к строке с первой по последнюю команду прежде чем перейти к следующей строке и начать новый цикл. Однако, некоторые команды способны прервать цикл, либо перейти в другую его точку. Переходы позволяют реализовать что-то похожее на ветвления, чтобы при определенных условиях выполнять одни команды, а при других условиях – другие.

Следующие команды способны влиять на цикл:

q

Безусловно прерывает текущий цикл и завершает обработку всего потока.

d

Очищает *pattern space* и начинает новый цикл.

D

Если *pattern space* пуст, то работает как **d**, иначе удаляет содержимое *pattern space* до первой новой строки в нем, перезапускает цикл с результатом, не читая новую строку.

t <метка>

Команда, очень похожая на оператор `goto` в некоторых языках программирования. Делает переход на метку цикла в случае успешного исполнения команды **s**, чтения очередной строки, либо успешного другого перехода. Если метка отсутствует, то переход выполняется в конец цикла. Данной командой программируются условные переходы.

T <метка>

Аналогична команде **t**, но все условия перехода инвертированы, т.е. команда **s** выполнилась не успешно, очередная строка не прочиталась.

b <метка>

Безусловный переход на указанную метку. Если метка опущена, то переход происходит в конец цикла.

Следующий пример показывает как работает **D**.

```
# Примечание: команда l позволяет увидеть содержимое pattern space таким, какое оно есть,  
# вместе со всеми управляющими последовательностями.  
#
```

```
$ printf '%s\n' aa bb cc dd | sed -n 'N;l;D'
aa\nbb$
bb\ncc$
cc\ndd$
```

В данном примере цикл состоит из 3 команд. В первом цикле команда N, после того как была помещена очередная строка в *pattern space*, добавляет к ней следующую за ней строку. Далее командой l мы выводим содержимое *pattern space*, и вы можете видеть, что в буфере находится конкатенация первых двух строк, разделенная переносом строки. Далее команда D, начиная с начала буфера, удаляет из *pattern space* строку до первого переноса строки, т.е. на момент конца цикла у нас в *pattern space* останется только bb. Затем команда D прерывает цикл, и Sed переходит на новую строку, но без помещения следующей строки в *pattern space*. Снова команда N добавляет к имеющемуся содержимому следующую строку (bb\ncc\$) и последовательность действий повторяется.

Вы можете создать метку в любой части цикла. Для этого используется следующий синтаксис: :имя_метки. Имя метки должно состоять по меньшей мере из одного символа. Метки не являются командами, поэтому они пропускаются при нормальном течении цикла.

Для уверенного использования данной техники необходима некоторая практика. Следующий пример демонстрирует, как работают переходы в цикле.

```
$ printf '%s\n' a1 a2 a3 | sed '/1/b else ; s/a/z/ ; :else ; y/123/456/'
a4
z5
z6

# Предыдущая микропрограмма эквивалентна следующей
$ printf '%s\n' a1 a2 a3 | sed -E '/1/!s/a/z/ ; y/123/456/'
a4
z5
z6
```

В этом примере демонстрируется, как строятся конструкции, чем то похожие на if-else, в Sed. Здесь команда /1/b else играет роль условия. Если в очередной строке есть цифра 1, то исполнение цикла перейдет на метку :else, благодаря команде b. За меткой :else выполняется команда y. Именно из-за перехода мы видим, что в первой строке (где есть цифра 1) изменилась только цифра, но не буква, так как мы перепрыгнули команду, которая изменила бы букву. В оставшихся строках нет цифры 1, поэтому перехода не происходит и на строку применяется две команды, первая из которых изменит букву, а вторая цифру.

Несмотря на всю простоту, даже в Sed можно случайно создавать бесконечные петли. Следующий код демонстрирует бесконечную петлю в Sed, из-за чего исполнение программы никогда не прервется:

```
$ echo "looping" | sed ':begin ; b begin'

# Когда исполнение микропрограммы доходит до конца, команда b отправляет точку следования в начало и так до бесконечности.
```

Однако, некоторые команды способны самостоятельно разорвать такую петлю из-за особенностей их работы. Например, команды n и N завершают исполнение Sed, когда им нечего читать в потоке:

```
$ seq 3 | sed ':begin ; n ; bbegin'
1
```

```
2
3
$ seq 3 | sed ':begin ; N ; bbegin'
1
2
3
```

В обоих случаях конец потока когда-нибудь настанет, и команда n/N прервет исполнение петли.

На практике петли могут использоваться для накопления информации в буфере *pattern space*. В официальной документации к GNU Sed приводится пример использования показанной выше петли, в котором она используется для объединения строк. Пусть у нас есть такой текст, использующий QR кодирование.

```
All the wor=
ld's a stag=
e,
And all the=
men and wo=
men merely =
players:
They have t=
heir exits =
and their e=
ntrances;
And one man=
in his tim=
e plays man=
y parts.
```

Не вдаваясь в подробности этого кодирования, лишь отметим, что каждая строка завершается символом =, который говорит, что продолжение перенесено. Нашей задачей является восстановить оригинальный текст через слияние фрагментов по завершающему символу =. В документации приводится такое решение:

```
$ sed ':x ; /=$/ { N ; s/=\n//g ; bx }' text.txt
All the world's a stage,
And all the men and women merely players:
They have their exits and their entrances;
And one man in his time plays many parts.
```

В данном примере петли порождаются только для строк, оканчивающихся знаком =. В таких петлях буфер накапливается строками, содержащими знак равно в конце командой N, а разрыв происходит из-за первых строк, в которых этого символа нет в конце, так как команда перехода в начало будет пропускаться. Печать окончательного результата происходит за счет автоматической печати.

К решению этой задачи есть еще один подход, основанный на условном переходе:

```
$ sed ':x ; $!N ; s/=\n// ; tx ; P ; D' text.txt
All the world's a stage,
And all the men and women merely players:
They have their exits and their entrances;
And one man in his time plays many parts.
```

Это решение похоже на предыдущее, но петля здесь строится из условного перехода t: команда \$!N накапливает данные в буфере, а s/=\n// удаляет пару =\n; когда команда s выполняется успешно, мы переходим в начало программы (т.е. еще есть что

накапливать), иначе продолжения нет и мы идем дальше. Командой P мы выводим накопленный результат, а командой D мы начинаем новый цикл, если в потоке есть еще данные. В этом примере команда D также пресекает автоматическую печать.

Ссылки

- Официальная документация GNU Sed (<https://www.gnu.org/software/sed/>)
-

Источник — https://ru.wikibooks.org/w/index.php?title=Sed:_руководство&oldid=225452