Blog do Beraldo

O comando SED do Linux

13 Comentários

A ferramenta SED, junto ao AWK, são as duas principais linguagens para manipulação de arquivos e *streams* do Unix/Linux. Ambas possuem vasta abrangência e o que uma não pode fazer, a outra provavelmente o fará. Com o SED é possível substituir e "casar" padrões, sempre por meio de Expressões Regulares, outra grande ferramenta do mundo da Computação. O SED, assim como o AWK, lê um arquivo, linha por linha, e aplica a expressão do parâmetro a cada uma delas. A sintaxe do SED é a seguinte:

```
sed 'expressão' arquivo
```

É possível filtrar a saída de outro comando, em vez de filtrar um arquivo, usando pipe. Usarei saídas do comando echo para explicar o sed. SED usa Expressões Regulares (ER's). Se você não conhece ER's, sugiro ler o Expressões Regulares – Guia de Consulta Rápida. Também recomendo o livro do mesmo autor do texto citado, Expressões Regulares – Aurélio Marinho Jargas, e Expressões Regulares Cookbook – Soluções Detalhadas Em Oito Linguagens de Programação – Steven Levithan; Jan.

NOTA: O conteúdo deste post foi aprimorado e somado a diversos outros temas ligados à Programação usando Shell Script. Tudo isso resultou no curso "Shell Script Para Programadores". Conheço melhor o curso neste link.

Sumário

1. Substituindo textos: comando s

Sintaxe: **s/ER/sub/[modificadores]**: Substitui os padrões “casados” em “ER” por “sub”. A utilidade mais conhecida do SED é a substituição de textos. Para isso, usamos o comando “s” do sed:

```
echo "Tutorial sobre sed: aprenda a trabalhar com sed" | sed 's/sed/awk/'
```

A saída será: **Tutorial sobre awk: aprenda a trabalhar com sed**. Note que a última ocorrência de “sed” não foi modificada. Isso porque o sed faz a modificação na primeira ocorrência e, em seguida, já procura pelo padrão na linha seguinte, se houver. Para que o sed execute a ação na linha toda, usamos o modificador “g”:

```
echo "Tutorial sobre sed: aprenda a trabalhar com sed" | sed 's/sed/awk/g'
```

Agora a saída é a esperada: **Tutorial sobre awk: aprenda a trabalhar com awk**. Os modificadores serão abordados com mais detalhes logo mais abaixo. Não percam! =P

1.1. Escapando o delimitador

Como estamos usando a barra (/) como delimitador, se quisermos usar esse caractere na ER ou na substituição, temos de escapá-lo, com barra invertida:

```
echo "o arquivo /proc/cpuinfo possui informações importantes" | sed 's/\/proc\/cpuinfo/\/
```

Saída: **o arquivo /etc/passwd possui informações importantes**

1.2. Usando a expressão casada em ER na substituição

Às vezes queremos acrescentar algo ao texto original. Por isso, é preciso que tenhamos o texto casado na ER. Para isso, usamos o caractere & para nos referirmos à expressão casada pela ER. Suponhamos que queremos colocar todos os caminhos de arquivos entre aspas. Para isso:

```
echo -e "o arquivo /proc/cpuinfo possui informações importantes \nEste tutorial está em /
```

Saída: o arquivo “/proc/cpuinfo” possui informações importantes Este tutorial está em “/home/beraldo/tutoriais”. Há muitos outros nesse diretório A opção “-r” usa o padrão estendido de expressões regulares – o mesmo usado por “egrep”. Isso possibilita o uso da lista negada.

1.3 Usando “retrovisores” (\1, \2…)

Muitas vezes queremos substituir partes de um texto, mas manter outras. Para isso, podemos usar os “retrovisores”, que inserem na string final os padrões casados na ER inseridos em grupos (**(** e **)**). O primeiro grupo casado é referenciado por \1, o segundo, por \2 e assim sucessivamente, até \9. Por exemplo, vamos manter apenas a primeira palavra de cada linha do texto:

```
echo -e "primeira linha\nsegunda linha\nterceira linha" | sed -r 's/^([a-z]+).*/\1/g'
```

Importante: Para usar os retrovisores, é necessário usar a opção -r do sed, que ativa o padrão estendido de ERs.

2. Modificadores

2.1. Modificador “g”: tratar linha inteira

Já vimos o modificador g anteriormente, que serve para forçar o SED a aplicar a expressão na linha inteira. É possível associar um número ao modificador “g”, para fazer com que o SED atue apenas a partir de uma determinada ocorrência do padrão. Por exemplo, vamos substituir por NULL todas as palavraas do texto:

```
echo "palavra1 palavra2 palavra3" | sed -r 's/([a-zA-Z0-9]+)/NULL/g'
```

Vamos supor que gostaríamos de remover todas as ocorrências, exceto a primeira. Para isso executaríamos o sed assim:

```
echo "palavra1 palavra2 palavra3" | sed -r 's/([a-zA-Z0-9]+)/NULL/2g'
```

O “2” faz com que a substituição ocorra apenas a partir da segunda ocorrência do padrão.

2.2. Modificador “p”: exibir na tela

Por padrão, o SED imprime na saída padrão todas as linhas lidas – e modificadas. A opção -n impede que o sed faça a impressão das linhas. É nessa situação que o modificador “p” é útil. Quando quisermos imprimir somente as linhas alteradas pelo SED, usaremos o modificador “p”.

```
echo -e "linha sem números \nLinha com números: 1 2 3 4 5 6 \nLinha com mais números 42 8
```

Serão mostradas apenas as linhas que casaram com o padrão. Como foi possível ver no exemplo acima, é possível misturar os modificadores.

3. Restringindo por linhas

E possível fazer o SED atuar somente num intervalo de linhas. Vamos mostrar os dez primeiros usuários do sistema, listando as dez primeiras linhas do arquivo /etc/passwd.

```
sed -r -n '1,10 s/([^:]+).*/\1/p' /etc/passwd
```

Note a opção -n e o modificador “p”, para mostrar somente o padrão casado. É possível usar o caractere $ (cifrão) para se referir à última linha do arquivo:

```
sed -r -n '1,$ s/([^:]+).*/\1/p' /etc/passwd
```

4. Restringindo por padrões

Além de restringir por número de linhas, podemos restringer por padrão:

```
echo -e "esta\né\numa\nlinha\nusada\npara\ntestar\no\ncomando\nsed" | sed  -e '/uma/,/coma
```

[cta id=’925′]

5. Remover linha: comando d

O comando “d” do sed remove linhas. Vamos remover as dez primeiraas linhas de /etc/passwd:

```
sed '1,10 d' /etc/passwd
```

6. Sair: comando q

O comando “q” faz o sed parar após encontrar o padrão Imprime as dez primeiras linhas de /etc/passwd

```
sed '10 q' /etc/passwd
```

Para após encontrar um número:

```
echo -e "sou uma linha\nsou outra linha\nesta linha tem número 1\nesta já não tem número"
```

7. Adicionando linha: comando “a”

É possível inserir linha após achar um padrão: Insere linha após uma linha que contenha número

```
echo -e "sou uma linha\nsou outra linha\nesta linha tem número 1\nesta já não tem número"
```

8. Adicionando linha: comando “i”

Quase igual ao comando “a”, mas insere a nova linha antes da linha casada com o padrão Insere linha antes de uma linha que contenha número

```
echo -e "sou uma linha\nsou outra linha\nesta linha tem número 1\nesta já não tem número"
```

9. Mudando uma linha: comando c

Semelhante a “a” e “i”, mas muda a linha que casou com o padrão

```
echo -e "sou uma linha\nsou outra linha\nesta linha tem número 1\nesta já não tem número"
```

10. Dicas de Livros e Cursos

Um excelente eBook, que comtempla o SED e também o AWK (outra excelente linguagem) é o *SED And AWK Pocket Reference*. Aproveite a versão digital, que pode ser lida em qualquer dispositivo usando o aplicativo *Kindle*, da Amazon

Para aprender mais sobre Comandos Linux e Shell Script, veja meu curso **Shell Script Para Programadores**.

http://rberaldo.com.br/o-comando-sed-do-linux/

Comando SED no Linux: Uso e Exemplos

O comando SED no Linux é uma ferramenta poderosa que ajuda a executar tarefas de uso geral. Entre elas: analisar e transformar textos. Hoje em dia, o SED é suportado nos principais sistemas operacionais.

A sintaxe e vários recursos úteis do SED são, em sua maioria, emprestados do editor *ed*. O mecanismo em si inclui fluxos de entrada padrões, bem como de arquivos de texto. O script base da sintaxe do SED parece um pouco difícil no começo. No entanto, muitas tarefas complexas podem ser solucionadas com apenas algumas linhas de script SED.

Conteúdo

Instalação do SED

Nos sistemas baseados em Linux, o SED é instalado por padrão. Lembre-se, antes de começar, você deve acessar seu VPS com SSH. Confira nosso tutorial PuTTY se você estiver com problemas.

O comando ***which*** pode ser usado para verificar se o sistema possui SED ou não. No GNU/Linux Debian, pode-se instalar o SED usando o apt package manager da seguinte forma:

[user]: ~$ sudo apt-get install sed

Para garantir que o SED esteja instalado corretamente, o seguinte comando é utilizado:

[user]: ~$ sed --version

O comando retorna o seguinte output:

sed (GNU sed) 4.2.2
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Written by Jay Fenlason, Tom Lord, Ken Pizzini,
and Paolo Bonzini.
GNU sed home page: <http://www.gnu.org/software/sed/>.
General help using GNU software: <http://www.gnu.org/gethelp/>.
E-mail bug reports to: <bug-sed@gnu.org>

SED *Workflow*

O *workflow* básico do SED é leitura, execução e exibição, como mencionado na **Figura 1.**

A leitura recebe os valores de entrada e armazena em um **buffer de padrões**. A fase de execução realiza o comando no arquivo sequencialmente. E após a execução, exibe o fluxo de saída. O buffer de padrões é liberado conforme o conteúdo é exibido na interface. O processo leitura, execução e exibição se repetem até o final do arquivo.

Exemplo do SED

Para explicar de forma simples, o SED lê um arquivo e retorna seu texto. Para criar um arquivo de texto, use o editor *vi*.

[user]: ~$ vi text.txtPrograms must be written for people to read, and only incidentally for machines to execute.” — Harold Abelson, Structure and Interpretation of Computer Programs

O comando para ler um arquivo de texto é o seguinte:

[user]: ~$ sed ‘’ text.txtPrograms must be written for people to read, and only incidentally for machines to execute.” — Harold Abelson, Structure and Interpretation of Computer Programs

O comando recebe entrada do arquivo “**text.txt**”. Antes de fazer o uploading de entrada do arquivo, escreva um argumento de linha de comando entre um par de aspas simples. Isso “falará” para a linha do comando executar o SED.

O SED lê o arquivo de entrada “**text.txt**” e armazena os dados em um buffer de padrões. Após isso, a operação pode ser realizada.

Neste caso, nós passamos um argumento vazio para que nenhuma operação seja executada. O SED apresenta os dados armazenados como uma saída padrão e esvazia o buffer de padrões.

[user]: ~$ sed ‘’Programs must be written for people to read, and only incidentally for machines to execute.” — Harold Abelson, Structure and Interpretation of Computer ProgramsPrograms must be written for people to read, and only incidentally for machines to execute.” — Harold Abelson, Structure and Interpretation of Computer Programs

O comando SED recebe a entrada do teclado. Ele é exibido na primeira e segunda linha. Ele exibe os dados armazenados no buffer de padrões. Para encerrar a sessão do SED, use **CTRL+D** no teclado.

Comandos básicos

Vamos aprender os comandos básicos do SED do Linux:

Comando de deletar

Para executar o comando de deletar, use ***d*** junto com um arquivo entre aspas. O comando irá deletar a primeira linha no arquivo ***text.txt***.

[user]: ~$ sed ‘1d’ text.txt

Comando de Escrita

Para executar o comando de escrita, escreva ***w***, o número da linha, e o arquivo, entre aspas. O seguinte comando lê a segunda linha e a escreve no arquivo ***text2.txt***.

[user]: ~$ sed '2~2 w text2.txt' text.txt
[user]: ~$ cat text2.txt
Harold Abelson, Structure and Interpretation of Computer Programs

Comando de Acrescentar

Use a palavra-chave e o número da linha entre aspas. Após fechar as aspas, forneça a fonte a ser acrescentada. O seguinte comando acrescentará depois da segunda linha no arquivo ***text.txt*** arquivo.

[user]: ~$ sed '2 a The Append example' text.txt[user]: ~$ cat text.txtPrograms must be written for people to read, and only incidentally for machines to execute.” — Harold Abelson, Structure and Interpretation of Computer ProgramsThe Append example'

Comando de Leitura

Use ***r*** e digite a localização do arquivo entre aspas. O seguinte comando irá ler a entrada do arquivo de texto e acrescentar depois da terceira linha no arquivo ***text2.txt***.

[user]: ~$ sed '3 r text.txt' text2.txt
[user]: ~$ cat text2.txt
Programs must be written for people to read, and only incidentally for machines to execute.” — Harold Abelson, Structure and Interpretation of Computer Programs
The Append example'
Programs must be written for people to read, and only incidentally for machines to execute.” — Harold Abelson, Structure and Interpretation of Computer Programs

Conclusão

Neste tutorial, você aprendeu sobre a instalação do SED e operações básicas. A ferramenta SED permite que o usuário utilize a linha de comando de diversas maneiras.

Você também poderá resolver problemas complexos de maneira eficiente. Além do SED, o GNU-Linux oferece muitas ferramentas úteis para realizar tarefas cotidianas.

https://www.hostinger.com.br/tutoriais/como-usar-comando-sed-linux

The Basics of Using the Sed Stream Editor to Manipulate Text in Linux

Linux BasicsLinux CommandsInteractive

- By Justin Ellingwood and Brian Hogan

  Last Validated onMay 19, 2020 Originally Published onJuly 24, 2013 576.8k views

Introduction

The `sed` command, short for **stream editor**, performs editing operations on text coming from standard input or a file. `sed` edits line-by-line and in a non-interactive way.

This means that you make all of the editing decisions as you are calling the command, and `sed` executes the directions automatically. This may seem confusing or unintuitive, but it is a very powerful and fast way to transform text, especially as part of a script or automated workflow.

This tutorial will cover some basic operations and introduce you to the syntax required to operate this editor. You will almost certainly never replace your regular text editor with `sed`, but it will probably become a welcomed addition to your text editing toolbox.

Note: This tutorial uses the GNU version of `sed` found on Ubuntu and other Linux operating systems. If you're using macOS, you'll have the BSD version which has different options and arguments. You can install the GNU version of `sed` with Homebrew using `brew install gnu-sed`.

Launch an Interactive Terminal!

Basic Usage

`sed` operates on a stream of text that it reads from either a text file or from standard input (STDIN). This means that you can send the output of another command directly into sed for editing, or you can work on a file that you've already created.

You should also be aware that `sed` outputs everything to standard out (STDOUT) by default. That means that, unless redirected, `sed` will print its output to the screen instead of saving it in a file.

The basic usage is:

```
sed [options] commands [file-to-edit]
```

In this tutorial, you'll use a copy of the BSD Software License to experiment with `sed`. On Ubuntu, execute the following commands to copy the BSD license file to your home directory so you can work with it:

```
cd
cp /usr/share/common-licenses/BSD .
```

If you don’t have a local copy of the BSD license, create one yourself with this command:

```
cat << 'EOF' > BSD
Copyright (c) The Regents of the University of California.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the University nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
EOF
```

Let’s use `sed` to view the contents of the BSD license file. `sed` sends its results to the screen by default, which means you can use it as a file reader by passing it no editing commands. Try executing the following command:

```
sed '' BSD
```

You’ll see the BSD license displayed to the screen:

```
Output
Copyright (c) The Regents of the University of California.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.
...
...
```

The single quotes contain the editing commands you pass to `sed`. In this case, you passed it nothing, so `sed` printed each line it received to standard output.

`sed` can use standard input rather than a file. Pipe the output of the `cat` command into `sed` to produce the same result:

```
cat BSD | sed ''
```

You'll see the output of the file:

```
Output
Copyright (c) The Regents of the University of California.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.
. . .
. . .
```

As you can see, you can operate on files or streams of text, like the ones produced when piping output with the pipe (`|`) character, just as easily.

Printing Lines

In the previous example, you saw that input passed into `sed` without any operations would print the results directly to standard output.

Let's explore `sed`'s explicit `print` command, which you specify by using the `p` character within single quotes.

Execute the following command:

```
sed 'p' BSD
```

You'll see each line of the `BSD` file printed twice:

```
Output
Copyright (c) The Regents of the University of California.
Copyright (c) The Regents of the University of California.
All rights reserved.
All rights reserved.


Redistribution and use in source and binary forms, with or without
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
modification, are permitted provided that the following conditions
are met:
are met:
```

```
. . .
. . .
```

`sed` automatically prints each line by default, and then you've told it to print lines explicitly with the "p" command, so you get each line printed twice.

If you examine the output closely, you'll see that it has the first line twice, followed by the second line twice, etc, which tells you that `sed` operates on data line by line. It reads a line, operates on it, and outputs the resulting text before repeating the process on the next line.

You can clean up the results by passing the `-n` option to `sed`, which suppresses the automatic printing:

```
sed -n 'p' BSD
```

```
Output
Copyright (c) The Regents of the University of California.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.
. . .
. . .
```

We now are back to printing each line once.

The examples so far can hardly be considered editing (unless you wanted to print each line twice…). Next you'll explore how `sed` can modify the output by targeting specific sections of the text data.

Using Address Ranges

Addresses let you target specific parts of a text stream. You can specify a specific line or even a range of lines.

Let's have `sed` print the first line of the file. Execute the following command:

```
sed -n '1p' BSD
```

The first line prints to the screen:

```
Output
Copyright (c) The Regents of the University of California.
```

By placing the number `1` before the print command, you told `sed` the line number to operate on. You can just as easily print five lines (don't forget the "-n"):

```
sed -n '1,5p' BSD
```

You'll see this output:

```
Output
Copyright (c) The Regents of the University of California.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
```

You've just given an address range to `sed`. If you give `sed` an address, it will only perform the commands that follow on those lines. In this example, you've told sed to print line 1 through line 5. You could have specified this in a different way by giving the first address and then using an offset to tell sed how many additional lines to travel, like this:

```
sed -n '1,+4p' BSD
```

This will result in the same output, because you told `sed` to start at line 1 and then operate on the next 4 lines as well.

If you want to print every other line, specify the interval after the `~` character. The following command prints every other line in the `BSD` file, starting with line 1:

```
sed -n '1~2p' BSD
```

Here's the output you'll see:

```
Output
Copyright (c) The Regents of the University of California.

modification, are permitted provided that the following conditions
1. Redistributions of source code must retain the above copyright
2. Redistributions in binary form must reproduce the above copyright
    documentation and/or other materials provided with the distribution.
    may be used to endorse or promote products derived from this software
. . .
. . .
```

You can use `sed` to delete text from the output as well.

Deleting Text

You can perform text deletion where you previously were specifying text printing by changing the `p` command to the `d` command.

In this case, you no longer need the `-n` command because `sed` will print everything that is not deleted. This will help you see what's going on.

Modify the last command from the previous section to make it
delete every other line starting with the first:

```
sed '1~2d' BSD
```

The result is that you see every line you were **not** given last time:

```
Output
All rights reserved.
Redistribution and use in source and binary forms, with or without
are met:
    notice, this list of conditions and the following disclaimer.
    notice, this list of conditions and the following disclaimer in the
3. Neither the name of the University nor the names of its contributors
    without specific prior written permission.
. . .
. . .
```

It is important to note here that our source file is not being affected. It is still intact. The edits are output to our screen.

If we want to save our edits, we can redirect standard output to a file like so:

```
sed '1~2d' BSD > everyother.txt
```

Now open the file with `cat`:

```
cat everyother.txt
```

You see the same output that you saw onscreen previously:

```
Output
All rights reserved.
Redistribution and use in source and binary forms, with or without
are met:
    notice, this list of conditions and the following disclaimer.
    notice, this list of conditions and the following disclaimer in the
3. Neither the name of the University nor the names of its contributors
    without specific prior written permission.
. . .
. . .
```

The `sed` command does not edit the source file by default, but you can change this behavior by passing the `-i` option, which means “perform edits in-place.” This will alter the source file.

Warning: Using the `-i` switch will overwrite the original file, so you should use this with care. Perform the operations without the `-i` switch first and then run the command again with `-i` once you have what you want, create a backup of the original file, or redirect the output to a file. It’s very easy to accidentally alter the original file with the `-i` switch.

Let’s try it by editing the `everyother.txt` file you just created, in-place. Let’s further reduce the file by deleting every other line
again:

```
sed -i '1~2d' everyother.txt
```

If you use `cat` to display the file with `cat everyother.txt`, you'll see that the file has been edited.

The `-i` option can be **dangerous**. Thankfully, `sed` gives you the ability to create a backup file prior to editing.

To create a backup file prior to editing, add the backup extension directly after the "-i" option:

```
sed -i.bak '1~2d' everyother.txt
```

This creates a backup file with the `.bak` extension, and then edits the original file in-place.

Next you'll look at how to use `sed` to perform search and replace operations.

Substituting Text

Perhaps the most well-known use for `sed` is substituting text. `sed` can search for text patterns using regular expressions, and then replace the found text with something else.

You can learn more about regular expressions by following the Using Grep Regular Expressions to Search for Text Patterns in Linux.

In its most basic form, you can change one word to another word using the following syntax:

```
's/old_word/new_word/'
```

The `s` is the substitute command. The three slashes (`/`) are used to separate the different text fields. You can use other characters to delimit the fields if it would be more helpful.

For instance, if you were trying to change a website name, using another delimiter would be helpful since URLs contain slashes.

Execute the following command to print a URL with `echo` and modify it with `sed`, using the underscore (_) character as the delimiter:

```
echo "http://www.example.com/index.html" | sed 's_com/index_org/home_'
```

This replaces `com/index` with `org/home`. The output shows the modifed URL:

```
Output
http://www.example.org/home.html
```

Do not forget the final delimiter, or `sed` will complain. If you ran this command:

```
echo "http://www.example.com/index.html" | sed 's_com/index_org/home'
```

You'd see this output:

```
Output
```

```
sed: -e expression #1, char 20: unterminated `s' command
```

Let’s create a new file to practice some substitutions. Execute the following command to create a new text file called `song.txt`:

```
echo "this is the song that never ends
yes, it goes on and on, my friend
some people started singing it
not knowing what it was
and they'll continue singing it forever
just because..." > song.txt
```

Now let’s substitute the expression `on` with `forward`. Use the following command:

```
sed 's/on/forward/' song.txt
```

The output looks like this:

```
Output
this is the sforwardg that never ends
yes, it goes forward and on, my friend
some people started singing it
not knowing what it was
and they'll cforwardtinue singing it forever
just because...
```

You can see a few notable things here. First, is that `sed` replaced patterns, not words. The `on` within `song` is changed to `forward`.

The other thing to notice is that on line 2, the second `on` was not changed to `forward`.

This is because by default, the `s` command operates on the first match in a line and then moves to the next line. To make `sed` replace every instance of `on` instead of just the first on each line, you must pass an optional flag to the substitute command.

Provide the `g` flag to the substitute command by placing it after the substitution set:

```
sed 's/on/forward/g' song.txt
```

You’ll see this output:

```
Output
this is the sforwardg that never ends
yes, it goes forward and forward, my friend
some people started singing it
not knowing what it was
and they'll cforwardtinue singing it forever
just because...
```

Now the substitute command changes every instance.

If you *only* wanted to change the second instance of “on” that sed finds on each line, then you would use the number `2` instead of the `g`:

```
sed 's/on/forward/2' song.txt
```

This time the other lines are unchanged, as they don’t have a second occurrence:

```
Output
this is the song that never ends
yes, it goes on and forward, my friend
some people started singing it
not knowing what it was
and they'll continue singing it forever
just because...
```

If you only want to see which lines were substituted, use the `-n` option again to suppress automatic printing.

You can then pass the `p` option to the substitute command to print lines where substitution took place.

```
sed -n 's/on/forward/2p' song.txt
```

The line that changed prints to the screen:

```
Output
yes, it goes on and forward, my friend
```

As you can see, you can combine the flags at the end of the command.

If you want the search process to ignore case, you can pass it the “i” flag.

```
sed 's/SINGING/saying/i' song.txt
```

Here’s the output you’ll see:

```
Output
this is the song that never ends
yes, it goes on and on, my friend
some people started saying it
not knowing what it was
and they'll continue saying it forever
just because...
```

Replacing and Referencing Matched Text

If you want to find more complex patterns with regular expressions, you have a number of different methods of referencing the matched pattern in the replacement text.

For instance, to match from the beginning of the line to `at`, use the following command:

```
sed 's/^.*at/REPLACED/' song.txt
```

You’ll see this output:

```
Output
REPLACED never ends
yes, it goes on and on, my friend
some people started singing it
REPLACED it was
and they'll continue singing it forever
just because...
```

You can see that the wildcard expression matches from the beginning of the line to the last instance of `at`.

Since you don’t know the exact phrase that will match in the search string, you can use the `&` character to represent the matched text in the replacement string.

Let’s put parentheses around the matched text:

```
sed 's/^.*at/(&)/' song.txt
```

You’ll see this output:

```
Output
(this is the song that) never ends
yes, it goes on and on, my friend
some people started singing it
(not knowing what) it was
and they'll continue singing it forever
just because...
```

A more flexible way of referencing matched text is to use escaped parentheses to group sections of matched text.

Every group of search text marked with parentheses can be referenced by an escaped reference number. For instance, the first parentheses group can be referenced with `\1`, the second with `\2` and so on.

In this example, we’ll switch the first two words of each line:

```
sed 's/\([a-zA-Z0-9][a-zA-Z0-9]*\) \([a-zA-Z0-9][a-zA-Z0-9]*\)/\2 \1/' song.txt
```

You’ll see this output:

```
Output
is this the song that never ends
yes, goes it on and on, my friend
people some started singing it
knowing not what it was
they and'll continue singing it forever
because just...
```

As you can see, the results are not perfect. For instance, the second line skips the first word because it has a character not listed in our character set. Similarly, it treated `they'll` as two words in the fifth line.

Let’s improve the regular expression to be more accurate:

```
sed 's/\([^ ][^ ]*\) \([^ ][^ ]*\)/\2 \1/' song.txt
```

You'll see this output:

```
Output
is this the song that never ends
it yes, goes on and on, my friend
people some started singing it
knowing not what it was
they'll and continue singing it forever
because... just
```

This is much better than last time. This groups punctuation with the associated word.

Notice how we repeat the expression inside the parentheses (once without the `*` character, and then once with it). This is because the `*` character matches the character set that comes before it zero or more times. This means that the match with the wildcard would be considered a "match" even if the pattern is not found.

To ensure that `sed` finds the text at least once, you must match it once without the wildcard before employing the wildcard.

Conclusion

In this tutorial you explored the `sed` command. You printed specific lines from the file, searched for text, deleted lines, overwrote the original file, and used regular expressions to replace text. You should be able to see already how you can quickly transform a text document using properly constructed sed commands.

In the next article in this series, you will explore some more advanced features.

Next in series: Intermediate Sed: Manipulating Streams of Text in a Linux Environment

About the authors

Justin Ellingwood

Senior Technical Writer @DigitalOcean

Brian Hogan

I manage the Write for DOnations program, write and edit community articles, and make things on the Internet.

Tutorial Series

Using Sed

In this interactive tutorial series, you'll explore basic and advanced operations with `sed` , the command line stream editor found in Linux. You will almost certainly never replace your regular text editor with `sed`, but it will certainly become a welcomed addition to your text editing toolbox.

Next in series: Intermediate Sed: Manipulating Streams of Text in a Linux Environment
https://www.digitalocean.com/community/tutorials/the-basics-of-using-the-sed-stream-editor-to-manipulate-text-in-linux

Intermediate Sed: Manipulating Streams of Text in a Linux Environment

Linux BasicsLinux CommandsInteractive

-
-
- By Justin Ellingwood and Brian Hogan

  Last Validated onMay 19, 2020 Originally Published onJuly 25, 2013 60.6k views

Introduction

The `sed` stream editor is a powerful editing tool that can make sweeping changes with very little input. In the previous article on `sed`, you explored the basics of using sed to edit text.

This article will continue your introduction by examining some more advanced topics.

Note: This tutorial uses the GNU version of `sed` found on Ubuntu and other Linux operating systems. If you're using macOS, you'll have the BSD version which has different options and arguments. You can install the GNU version of `sed` with Homebrew using `brew install gnu-sed`.

Launch an Interactive Terminal!

To complete this tutorial, you'll need some files to manipulate, which you should have from the the first tutorial. If you don't have them, you can recreate them with the following commands:

```
cd
cp /usr/share/common-licenses/BSD .
echo "this is the song that never ends
yes, it goes on and on, my friend
some people started singing it
not knowing what it was
and they'll continue singing it forever
just because..." > song.txt
```

In addition, you'll use the GPL 3 license in this tutorial, so copy that file as well:

```
cp /usr/share/common-licenses/GPL-3 .
```

If you don't have it, you can download it with `curl`:

```
curl -o GPL-3 https://www.gnu.org/licenses/gpl-3.0.txt
```

Now that you have the files, you'll explore using `sed` with multiple commands.

Supplying Multiple Editing Sequences

There are quite a few instances where you might wish to pass multiple commands to `sed` simultaneously. There are a few ways that this can be accomplished.

Since `sed` operates through standard input and output, you can string different calls to `sed` together through a pipeline. Execute this command to replace the word `and` with an apersand (`&`), and the word `people` with `horses`:

```
sed 's/and/\&/' song.txt | sed 's/people/horses/'
```

Note that you need to escape the “&” since it means “the complete matched pattern” to `sed`):

You’ll see the following output:

```
Output
this is the song that never ends
yes, it goes on & on, my friend
some horses started singing it
not knowing what it was
& they'll continue singing it forever
just because...
```

This works, but it creates unnecessary overhead with multiple calls to `sed`, requires more typing, and does not take advantage of `sed`’s built-in capabilities.

You can string various commands to `sed` by using the `-e` option before each command. This is how you would rewrite the previous command:

```
sed -e 's/and/\&/' -e 's/people/horses/' song.txt
```

Another approach to stringing commands together is using a semi-colon character (`;`) to separate distinct commands. This works the same as the previous example, but the “`-e`” is not required:

```
sed 's/and/\&/;s/people/horses/' song.txt
```

Note how when using the `-e` construct, you need separate single-quotation groups for the different commands. However, when separating commands with a semi-colon, all commands are placed within just one single-quoted command string. Although these two ways of expressing multiple commands is useful, there are times when the previous piping technique is still required.

Consider the `=` operator. This operator inserts a line-number on a new line between each existing line. The output looks like this:

```
sed '=' song.txt
```

Here’s the ouput you’ll see:

```
Output
```

```
1
this is the song that never ends
2
yes, it goes on and on, my friend
3
some people started singing it
4
not knowing what it was
5
and they'll continue singing it forever
6
just because...
```

If you would like to change the formatting of the numbering by modifying the text, however, you'll see that things do not work as expected.

To demonstrate, let's look at the `G` command, which by default, enters a blank line between each line (this actually is more complex, but you'll explore that later):

```
sed 'G' song.txt
```

Here's the result:

```
Output
this is the song that never ends

yes, it goes on and on, my friend

some people started singing it

not knowing what it was

and they'll continue singing it forever

just because...
```

If you combine these two commands, you might expect a space between each regular line and line-number line:

```
sed '=;G' song.txt
```

However, you get something different:

```
Output
1
this is the song that never ends

2
yes, it goes on and on, my friend

3
some people started singing it

4
not knowing what it was

. . .
```

```
. . .
```

This happens because the = operator modifies the actual output stream directly. This means that you cannot use the results for more editing.

You can get around this by using two `sed` calls, treating the first `sed` modification as a simple stream of text for the second:

```
sed '=' song.txt | sed 'G'
```

You now see the results you were expecting:

```
Output
1

this is the song that never ends

2

yes, it goes on and on, my friend

3

some people started singing it
. . .
. . .
```

Keep in mind that some of the commands operate like this, especially if you are stringing multiple commands together and the output differs from what you were expecting.

Advanced Addressing

One of the advantages of `sed`'s addressable commands is that regular expressions can be used as selection criteria. This means that you are not limited to operating on known line values, like you saw previously:

```
sed '1,3s/.*/Hello/' song.txt
```

```
Output
Hello
Hello
Hello
not knowing what it was
and they'll continue singing it forever
just because...
```

You can, instead, use regular expressions to match only lines that contain a certain pattern. To do this, place the match pattern between two forward slashes (/) prior to giving the command strings:

```
sed '/singing/s/it/& loudly/' song.txt
```

```
Output
```

```
this is the song that never ends
yes, it goes on and on, my friend
some people started singing it loudly
not knowing what it was
and they'll continue singing it loudly forever
just because...
```

In this example, you've placed `loudly` after the first occurrance of `it` on every line that contains the string `singing`. Notice that the second and fourth line are unaltered because they do not match the pattern.

The expressions for addressing can be arbitrarily complex. This provides a great deal of flexibility in executing commands.

This is not a complex example, but it demonstrates using regular expressions to generate addresses for other commands. The following command matches any blank lines (the start of a line followed immediately by the end of the line) and passes them to the delete command:

```
sed '/^$/d' GPL-3
```

This is the output you'll see:

```
Output
                    GNU GENERAL PUBLIC LICENSE
                        Version 3, 29 June 2007
Copyright (C) 2007 Free Software Foundation, Inc.
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
                             Preamble
The GNU General Public License is a free, copyleft license for
. . .
. . .
```

Keep in mind that you can use regular expressions on either side of a range as well. For instance, you can delete lines starting at a line that only contains the word START until a line reading END,

For example, create a file called `inputfile`:

```
echo "This is an input file
START
this is the text we don't want
END
This is additional text" > inputfile
```

Now use `sed` to delete the content between START and END:

```
sed '/^START$/,/^END$/d' inputfile
```

You'll see the following output:

```
This is an input file
This is additional text
```

One thing to note though, is that this will delete everything from the first `START` to the first `END`, and then restart the deletion if it comes across another `START` marker.

If you want to invert an address (operate on any line that does *not* match a pattern), you can follow the pattern with an exclamation point (`!`).

For example, you can delete any line that is *not* blank (not terribly useful, but just an example), with the following command:

```
sed '/^$/!d' GPL-3
```

This results in a large blank output, as `sed` still prints lines by default:

```
Output
```

The address does not need to be a complex expression to be inverted. Inversion works the same on regular numbered address too.

Using the Hold Buffer

One piece of functionality that increases `sed`'s ability perform multi-line aware edits is what is called the "hold buffer". The hold buffer is an area of temporary storage that can be modified by certain commands.

The presence of this extra buffer means that you can store lines while working on other lines, and then operate on each buffer as necessary.

The following are the commands that affect the holding buffer:

- **h**: Copies the current pattern buffer (the line you're currently matched and working on) into the the holding buffer (this erases the previous contents of the hold buffer).
- **H**: Appends the current pattern buffer to the end of the current holding pattern, separated by a new-line (\n) character.
- **g**: Copies the current holding buffer into the current pattern buffer. The previous pattern buffer is erased.
- **G**: Appends the current holding pattern to the end of the current pattern buffer, separated by a new-line (\n) character.
- **x**: Swap the current pattern and holding buffers.

The contents of the holding buffer cannot be operated on until it is moved to the pattern buffer in one way or another.

Let's explore this idea with a complex example.

This is a procedural example of how to join adjacent lines (`sed` actually has a built-in command that would take care of a lot of this for us. The `N` command appends the next line to the current line. You are going to do things the hard way though for the sake of practice):

```
sed -n '1~2h;2~2{H;g;s/\n/ /;p}' song.txt
```

Here's the output you'll see:

```
Output
this is the song that never ends yes, it goes on and on, my friend
some people started singing it not knowing what it was
and they'll continue singing it forever just because...
```

This is a lot to digest, so let's break it down.

The first thing to note is that the `-n` option is used to suppress automatic printing. `sed` will only print when you specifically tell it too.

The first part of the instruction is `1\~2h`. The beginning is an address specification meaning to perform the subsequent operation on the first line, and then on every other line afterwards (each odd numbered line). The `h` part is the command to copy the matched line into the holding buffer.

The second half of the command is more complex. Again, it begins with an address specification. This time, it is referring to the even numbered lines (the opposite of the first command).

The rest of the command is enclosed in braces. This means that the rest of the commands will inherit the address that was just specified. Without the braces, only the "H" command would inherit the address, and the rest of the commands would be executed on every line.

The `H` command copies a new-line character, followed by the current pattern buffer, onto the end of the current holding pattern.

This holding pattern (an odd numbered line, followed by a new-line character, followed by an even numbered line) is then copied back into the pattern buffer (replacing the previous pattern buffer) with the `g` command.

Next, the new-line character is replaced with a space and the line is printed with the `p` command.

If you are curious, using the `N` command would shorten this considerably. The following command will produce the same results that you've just seen:

```
sed -n 'N;s/\n/ /p' song.txt
```

Using Scripts

As you begin to use more complex commands, it may be helpful to compose them in a text editor. This is also helpful if you have a large number of commands that you'd like to apply to a single target.

For example, if you like to compose messages in plain text, but you need to perform a set of standardized formatting before using the text, a `sed` script would be useful.

Instead of typing each set of `sed` calls, you can put the commands in a script and supply it as an argument to `sed`. A `sed` script is simply a list of raw `sed` commands (the part normally between the single-quote characters).

To try this out, create a new file called `sed_script` with the following contents:

```
echo "s/this/that/g
s/people/horses/g
1,5s/it/that/g" > sed_script
```

Save the file and exit the editor.

Now tell `sed` to use the file by using the `-f` switch:

```
sed -f sed_script song.txt
```

The results will look like this:

```
Output
that is the song that never ends
yes, that goes on and on, my friend
some horses started singing that
not knowing what that was
and they'll continue singing that forever
just because...
```

This allows you to put all of your edits in one file and execute it on arbitrary text files that need to conform to the format you've created.

Conclusion

Sed's commands are not always easy to understand at first, and it often takes some actual experimentation to get an idea of their utility. For this reason, it's recommended that you practice manipulating text before you actually need to. Have an end goal in mind and try to implement it only using `sed`.

Hopefully, by this point, you are beginning to understand the power that a proper mastery of `sed` can give you. The more comfortable you are with `sed`, the less work you will have to do in the long run.

https://www.digitalocean.com/community/tutorials/intermediate-sed-manipulating-streams-of-text-in-a-linux-environment