- My Homepage
- Unix/Linux Index Page
- Linux

UNIX and Linux

sed

elflord@pegasus.rutgers.edu

Tutorials

Useful information mostly written by me, the conspicuous exception being the bash manpage ...

Intro to Unix
UNIX command summary
grep tutorial powerful search tool
sed tutorial scripts to edit text files
Autofs in Linux automatically mounting removable media
procmail tutorial well known email filter
bash manpage the man page for the bash shell. Warning: this is long (~210k)

Introduction

This tutorial is meant as a brief introductory guide to sed that will help give the beginner a solid foundation regarding how sed works. It's worth noting that the tutorial also omits several commands, and will not bring you to sed enlightenment in itself. To reach sed enlightenment, your best bet is to follow the seders mailing list. to do that , send email to Al Aab <af137@freenet.toronto.on.ca>

Prerequisites

It is assumed that the reader is familiar with regular expressions. If this is not the case, read the grep tutorial which includes information on regular expressions. On this page, we just give a brief revision.

Sed regular expressions

The sed regular expressions are essentially the same as the grep regular expressions. They are summarized below.

`^`	matches the beginning of the line
`$`	matches the end of the line
`.`	Matches any single character
`(character)*`	match arbitrarily many occurences of (character)
`(character)?`	Match 0 or 1 instance of (character)
`[abcdef]`	Match any character enclosed in [] (in this instance, a b c d e or f) ranges of characters such as `[a-z]` are permitted. The behaviour of this deserves more description. See the page on grep for more details about the syntax of lists.
`[^abcdef]`	Match any character *NOT* enclosed in [] (in this instance, any character other than a b c d e or f)
`(character)\{m,n\}`	Match m-n repetitions of (character)
`(character)\{m,\}`	Match m or more repetitions of (character)
`(character)\{,n\}`	Match n or less (possibly 0) repetitions of (character)
`(character)\{n\}`	Match exactly n repetitions of (character)
`\(expression\)`	Group operator.
`\n`	Backreference - matches nth group
`expression1|expression2`	Matches expression1 or expression 2. Works with GNU sed, but this feature might not work with other forms of sed.

Special Characters

The special character in *sed* are the same as those in *grep*, with one key difference: the forward slash `/` is a special character in *sed*. The reason for this will become very clear when studying *sed* commands.

How it Works: A Brief Introduction

Sed works as follows: it reads from the standard input, one line at a time. for each line, it executes a series of editing commands, then the line is written to STDOUT. An example which shows how it works : we use the `s` sommand. `s` means "substitute" or search and replace. The format is

```
s/regular-expression/replacement text/{flags}
```

We won't discuss all the flags yet. The one we use below is `g` which means "replace all matches"

```
>cat file
I have three dogs and two cats
>sed -e 's/dog/cat/g' -e 's/cat/elephant/g' file
I have three elephants and two elephants
>
```

OK. So what happened ? Firstly, *sed* read in the line of the file and executed

```
s/dog/cat/g
```

which produced the following text:

```
I have three cats and two cats
```

and then the second command was performed on the *edited line* and the result was

```
I have three elephants and two elephants
```

We actually have a name for the "current text": it is called the *pattern space*. So a precise definition of what *sed* does is as follows :

> *sed reads the standard input into the pattern space, performs a sequence of editing commands on the pattern space, then writes the pattern space to STDOUT.*

Getting Started: Substitute and delete Commands

Firstly, the way you usually use *sed* is as follows:

```
>sed -e 'command1' -e 'command2' -e 'command3' file
>{shell command}|sed -e 'command1' -e 'command2'
>sed -f sedscript.sed file
>{shell command}|sed -f sedscript.sed
```

so *sed* can read from a file or STDIN, and the commands can be specified in a file or on the command line. Note the following :

> that if the commands are read from a file, trailing whitespace can be fatal, in particular, it will cause scripts to fail for no apparent reason. I recommend editing sed scripts with an editor such as vim which can show end of line characters so that you can "see" trailing white space at the end of line.

The Substitute Command

The format for the substitute command is as follows:

> [*address1*[,*address2*]]`s`/*pattern*/*replacement*/*[flags]*

The flags can be any of the following:

n	replace nth instance of *pattern* with *replacement*
`g`	replace all instances of *pattern* with *replacement*
`p`	write pattern space to STDOUT if a successful substitution takes place
`w` *file*	Write the pattern space to *file* if a successful substitution takes place

If no flags are specified the first match on the line is replaced. note that we will almost always use the `s` command with either the `g` flag or no flag at all.

If one address is given, then the substitution is applied to lines containing that address. An address

can be either a regular expression enclosed by forward slashes `/regex/` , or a line number . The `$` symbol can be used in place of a line number to denote the last line.

If two addresses are given separated by a comma, then the substitution is applied to all lines between the two lines that match the pattern.

This requires some clarification in the case where both addresses are patterns, as there is some ambiguity here. more precisely, the substitution is applied to all lines from the first match of *address1* to the first match of *address2* and all lines from the first match of *address1 following* the first match of *address2* to the next match of *address1* Don't worry if this seems very confusing (it is), the examples will clarify this.

The Delete Command

The delete command is very simple in its syntax: it goes like this

```
[address1[ , address2 ] ]d
```

And it deletes the content of the pattern space. All following commands are skipped (after all, there's very little you can do with an empty pattern space), and a new line is read into the pattern space.

Example 1

```
>cat file

http://www.foo.com/mypage.html

>sed -e 's@http://www.foo.com@http://www.bar.net@' file

http://www.bar.net/mypage.html
```

Note that we used a different delimiter, @ for the substitution command. Sed permits several delimiters for the s command including @%,:: these alternative delimiters are good for substitutions which include strings such as filenames, as it makes your sed code much more readable.

Example 2

```
>cat file

the black cat was chased by the brown dog

>sed -e 's/black/white/g' file

the white cat was chased by the brown dog
```

That was pretty straight forward. Now we move on to something more interesting.

Example 3

```
>cat file
```

```
the black cat was chased by the brown dog.
the black cat was not chased by the brown dog

>sed -e '/not/s/black/white/g' file

the black cat was chased by the brown dog.
the white cat was not chased by the brown dog.
```

In this instance, the substitution is only applied to lines matching the regular expression `not`. Hence it is not applied to the first line.

Example 4

```
>cat file

line 1 (one)
line 2 (two)
line 3 (three)
```

Example 4a

>sed -e '1,2d' file

line 3 (three)

Example 4b

>sed -e '3d' file
line 1 (one)
line 2 (two)

Example 4c

>sed -e '1,2s/line/LINE/' file

LINE 1 (one)
LINE 2 (two)
line 3 (three)

Example 4d

>sed -e '/^line.*one/s/line/LINE/' -e '/line/d' file

LINE 1 (one)

3a : This was pretty simple: we just deleted lines 1 to 2.
3b : This was also pretty simple. We deleted line 3.
3c : In this example, we performed a substitution on lines 1-2.
3d : now this is more interesting, and deserves some explanation. Firstly, it is clear that line 2 and 3 get deleted. But let's look closely at what happens to line 1.
First, line 1 is read into the pattern space. It matches the regular expression `^line.*one` So the substitution is carried out, and the resulting pattern space looks like this:

```
LINE 1 (one)
```

So now the second command is executed, but since the pattern space does not match the regular expression `line`, the delete command is not executed.

Example 5

```
>cat file

hello
this text is wiped out
Wiped out
hello (also wiped out)
WiPed out TOO!
goodbye
(1) This text is not deleted
(2) neither is this ... ( goodbye )
(3) neither is this
hello
but this is
and so is this
and unless we find another g**dbye
every line to the end of the file gets deleted

>sed -e '/hello/,/goodbye/d' file

(1) This text is not deleted
(2) neither is this ... ( goodbye )
(3) neither is this
```

This illustrates how the addressing works when two pattern addresses are specified. sed finds the first match of the expression "hello", deleting every line read into the pattern space until it gets to the first line after the expression "goodbye". It doesn't apply the delete command to any more addresses until it comes across the expression "hello" again. Since the expression "goodbye" is not on any subsequent line, the delete command is applied to all remaining lines.

Some More Commands

Backreferences in Sed

One of the nice things about backreferencing in sed is that you can use it not just in the search text, but in the replacement text.

The quit command

The quit or `q` command is very simple. It simply quits. No more lines are read into the pattern space and the program terminates and produces no more output.

Subroutines

We now introduce the concept of subroutines in sed:

> *In sed, curly braces, { } are used to group commands. They are used as follows:*
> *address1[,address2]{*
> *commands }*

Example: Find First Word From a List in a File

This example makes very good use of all the concepts outlined above.

For this, we use a shell script, since we need to state the one long string `x` several times (otherwise, we'd need to repeat ourselves three times with a somewhat lengthy expression). Notice that we use double quotes. This is so that `$x` is expanded to the shell variable name (which would not happen if we used single quotes). Also notice the `$1` on the end. The syntax to run this script is `script search_filename` where `script` is whatever you decided to call it and `search_filename` is the file you are trying to search. `$1` is the name the shell gives to the first command line argument.

```
#!/bin/sh
X='word1\|word2\|word3\|word4\|word5'
sed -e "
/$X/!d
/$X/{
        s/\($X\).*/\1/
        s/.*\($X\)/\1/
        q
        }" $1
```

An important note: it is tempting to think of this:

```
s/\($X\).*/\1/
s/.*\($X\)/\1/
```

as redundant, and to try and shorten it with this:

```
s/.*\($X\).*/\1/
```

This is unlikely to work. Why ? suppose we have a line

```
word1 word2 word3
```

we have no way of knowing that `$x` is going to match word1 , word2 or word3, so when we quote it (`\1`) , we don't know what we are quoting.

What has been used to make sure there are no such problems in the correct implementation is this:

> *the * operator is greedy. That is, when there is ambiguity as to what (expression)* can match, it tries to match as much as possible.*

So in the example, `s/\($X\).*/\1/` , `.*` tries to swallow as much of the line as possible. in particular, if the line looks like

```
word1 word2 word3
```

then we can be sure that `.*` matches `" word2 word3"` and hence `$X` matches `word1`.

Pattern Matching Across More than 1 Line

Yes, this is something that a lot of people want to do (whether they realise it or not) as the `s/pattern1/replacement/` does not work if the string spans more than one line.

Example

Suppose we want to replace every instance of `Microsoft Windows 95` with `Linux` (I mean, just replace the text !). Our first attempt is this:

```
s/Microsoft Windows 95/Linux/g
```

Unfortunately, the script fails if our file looks like this:

```
Microsoft
Windows 95
```

Since neither line matches the pattern `microsoft Windows 95`

So we need to do better. We need the "multiline next" or `N` command.

> *The next command N appends the next line to the pattern space.*

So our second attempt is this:

```
N
N
s/Microsoft[ \t\n]*Windows[ \t\n]*95/Linux/g
```

Now note that we have made reference to `\t` and `\n`. These are the tab and end of line characters respectively. The end of line character only appears in multiline patterns. In multiline patterns, it should also be noted that `^` and `$` match the beginning and end of the pattern space.

The above is a start, but it breaks if we have a file that looks like this:

```
Foo
Microsoft
Windows
95
```

Why does it break ? Let's look at what the script does.

1. First, it reads the line "Foo" into the pattern space.

2. It sees the N command and appends line 2 to the pattern space. The pattern space now looks like:

   ```
   Foo\nMicrosoft
   ```

3. Executing the second N command , it reads line 3 into the pattern space. At this stage, the pattern space looks like this:

   ```
   Foo\nMicrosoft\nWindows
   ```

4. Now the script runs the substitute command.

   ```
   Foo\nMicrosoft\nWindows
   ```

   This doesn't match the search pattern, so no substitution is performed.
5. Since the end of the script is reached, the contents of the pattern space are written to STDOUT , and the script starts again from the first line
6. The last line of the file "95" is read into pattern space.

   *This is the main error in the script : once the end of the script is reached, the first line that * has not been read into the pattern space already * is read. It is NOT true that the Nth iteration of the script reads from the Nth line of the file.*

   The following too N commands fail and the script exits without writing '95' to STDOUT.

So there are too things to be learned from this:

- Each line of the file is read in exactly once. After you read a line into the pattern space, you can not read it again.
- It's good practice to use $!N in place of N to avoid errors, since the N command doesn't make sense on the last line of a file.

A better version is as follows:

```
/Microsoft[ \t]*$/{
                          N
                          }
/Microsoft[ \t\n]*Windows[ \t]*$/{
                          N
                          }
s/Microsoft[ \t\n]*Windows[ \t\n]*95/Linux/g
```

This only performs the search on extra lines when necessary.

Example: removing text between matching pairs of delimiters

Suppose we want to eliminate all text enclosed by a matching pair of delimiters This is a problem that comes up frequently. For example, removing html commands from html documents. We will use <angle brackets> in this example. So the task then is to eliminate anything between matching pairs of these brackets.

Our first attempt is shown as follows:

```
s/<[^>]*>//g
```

But this might break: the angle brackets might span more than one line, or there may be nested angle brackets. Actually, the latter is unlikely to happen if the html is correct. only possible to nest angle brackets inside html comments.) But we will assume that it might happen anyway (since it makes the problem more fun) So here is the improved version.

```
:top
/<.*>/{
s/<[^<>]*>//g
t top
}
/</{
        N
        b top
        }
```

A fine point: why didn't we replace the third line of the script with

```
     s/<[^>]*>//g
```

and removing the t command that follows ? Well consider this sample file:

```
<<hello>
hello>
```

The desired output would be the empty set, since everything is enclosed in angled brackets. However, the output will look like this:

```
hello>
```

since the first line matches the expression `<[^>]*>` So the point is that we have set up the script to recursively remove the contents of the innermost matching pair of delimiters.