Common threads: Sed by example, Part 2

How to further take advantage of the UNIX text editor

Daniel Robbins

October 01, 2000

Sed is a very powerful and compact text stream editor. In this article, the second in the series, Daniel shows you how to use sed to perform string substitution; create larger sed scripts; and use sed's append, insert, and change line commands.

Sed is a very useful (but often forgotten) UNIX stream editor. It's ideal for batch-editing files or for creating shell scripts to modify existing files in powerful ways. This article builds on my previous article introducing sed.

Substitution!

Let's look at one of sed's most useful commands, the substitution command. Using it, we can replace a particular string or matched regular expression with another string. Here's an example of the most basic use of this command:

```
$ sed -e 's/foo/bar/' myfile.txt
```

The above command will output the contents of myfile.txt to stdout, with the first occurrence of 'foo' (if any) on each line replaced with the string 'bar'. Please note that I said *first occurrence on each line*, though this is normally not what you want. Normally, when I do a string replacement, I want to perform it globally. That is, I want to replace *all* occurrences on every line, as follows:

```
$ sed -e 's/foo/bar/g' myfile.txt
```

The additional 'g' option after the last slash tells sed to perform a global replace.

Here are a few other things you should know about the 's///' substitution command. First, it is a command, and a command only; there are no addresses specified in any of the above examples. This means that the 's///' command can also be used with addresses to control what lines it will be applied to, as follows:

```
$ sed -e '1,10s/enchantment/entrapment/g' myfile2.txt
```

The above example will cause all occurrences of the phrase 'enchantment' to be replaced with the phrase 'entrapment', but only on lines one through ten, inclusive.

```
$ sed -e '/^$/,/^END/s/hills/mountains/g' myfile3.txt
```

This example will swap 'hills' for 'mountains', but only on blocks of text beginning with a blank line, and ending with a line beginning with the three characters 'END', inclusive.

Another nice thing about the 's///' command is that we have a lot of options when it comes to those '/' separators. If we're performing string substitution and the regular expression or replacement string has a lot of slashes in it, we can change the separator by specifying a different character after the 's'. For example, this will replace all occurrences of /usr/local with /usr:

```
$ sed -e 's:/usr/local:/usr:g' mylist.txt
```

In this example, we're using the colon as a separator. If you ever need to specify the separator character in the regular expression, put a backslash before it.

Regexp snafus

Up until now, we've only performed simple string substitution. While this is handy, we can also match a regular expression. For example, the following sed command will match a phrase beginning with '<' and ending with '>', and containing any number of characters inbetween. This phrase will be deleted (replaced with an empty string):

```
$ sed -e 's/<.*>//g' myfile.html
```

This is a good first attempt at a sed script that will remove HTML tags from a file, but it won't work well, due to a regular expression quirk. The reason? When sed tries to match the regular expression on a line, it finds the *longest* match on the line. This wasn't an issue in my previous sed article, because we were using the 'd' and 'p' commands, which would delete or print the entire line anyway. But when we use the 's///' command, it definitely makes a big difference, because the entire portion that the regular expression matches will be replaced with the target string, or in this case, deleted. This means that the above example will turn the following line:

```
<b>This</b> is what <b>I</b> meant.
```

into this:

```
meant.
```

rather than this, which is what we wanted to do:

```
This is what I meant.
```

Fortunately, there is an easy way to fix this. Instead of typing in a regular expression that says "a '<' character followed by any number of characters, and ending with a '>' character", we just need to type in a regexp that says "a '<' character followed by any number of non-'>' characters,

and ending with a '>' character". This will have the effect of matching the shortest possible match, rather than the longest possible one. The new command looks like this:

```
$ sed -e 's/<[^>]*>//g' myfile.html
```

In the above example, the '[^>]' specifies a "non-'>'" character, and the '*' after it completes this expression to mean "zero or more non-'>' characters". Test this command on a few sample html files, pipe them to more, and review their results.

More character matching

The '[ ]' regular expression syntax has some more additional options. To specify a range of characters, you can use a '-' as long as it isn't in the first or last position, as follows:

```
'[a-x]*'
```

This will match zero or more characters, as long as all of them are 'a','b','c'...'v','w','x'. In addition, the '[:space:]' character class is available for matching whitespace. Here's a fairly complete list of available character classes:

Character class	Description
[:alnum:]	Alphanumeric [a-z A-Z 0-9]
[:alpha:]	Alphabetic [a-z A-Z]
[:blank:]	Spaces or tabs
[:cntrl:]	Any control characters
[:digit:]	Numeric digits [0-9]
[:graph:]	Any visible characters (no whitespace)
[:lower:]	Lower-case [a-z]
[:print:]	Non-control characters
[:punct:]	Punctuation characters
[:space:]	Whitespace
[:upper:]	Upper-case [A-Z]
[:xdigit:]	hex digits [0-9 a-f A-F]

It's advantageous to use character classes whenever possible, because they adapt better to nonEnglish speaking locales (including accented characters when necessary, etc.).

Advanced substitution stuff

We've looked at how to perform simple and even reasonably complex straight substitutions, but sed can do even more. We can actually refer to either parts of or the entire matched regular expression, and use these parts to construct the replacement string. As an example, let's say you were replying to a message. The following example would prefix each line with the phrase "ralph said: ":

```
$ sed -e 's/.*/ralph said: &/' origmsg.txt
```

The output will look like this:

```
ralph said: Hiya Jim,

ralph said:

ralph said: I sure like this sed stuff!

ralph said:
```

In this example, we use the '&' character in the replacement string, which tells sed to insert the entire matched regular expression. So, whatever was matched by '.*' (the largest group of zero or more characters on the line, or the entire line) can be inserted anywhere in the replacement string, even multiple times. This is great, but sed is even more powerful.

Those wonderful backslashed parentheses

Even better than '&', the 's///' command allows us to define *regions* in our regular expression, and we can refer to these specific regions in our replacement string. As an example, let's say we have a file that contains the following text:

```
foo bar oni

eeny meeny miny

larry curly moe

jimmy the weasel
```

Now, let's say we wanted to write a sed script that would replace "eeny meeny miny" with "Victor eeny-meeny Von miny", etc. To do this, first we would write a regular expression that would match the three strings, separated by spaces:

```
'.* .* .*'
```

There. Now, we will define regions by inserting backslashed parentheses around each region of interest:

```
'\(.*\) \(.*\) \(.*\)'
```

This regular expression will work the same as our first one, except that it will define three logical regions that we can refer to in our replacement string. Here's the final script:

```
$ sed -e 's/\(.*\) \(.*\) \(.*\)/Victor \1-\2 Von \3/' myfile.txt
```

As you can see, we refer to each parentheses-delimited region by typing '\x', where x is the number of the region, starting at one. Output is as follows:

```
Victor foo-bar Von oni

Victor eeny-meeny Von miny

Victor larry-curly Von moe

Victor jimmy-the Von weasel
```

As you become more familiar with sed, you will be able to perform fairly powerful text processing with a minimum of effort. You may want to think about how you'd have approached this problem using your favorite scripting language -- could you have easily fit the solution in one line?

Mixing things up

As we begin creating more complex sed scripts, we need the ability to enter more than one command. There are several ways to do this. First, we can use semicolons between the commands. For example, this series of commands uses the '=' command, which tells sed to print the line number, as well as the 'p' command, which explicitly tells sed to print the line (since we're in '-n' mode):

```
$ sed -n -e '=;p' myfile.txt
```

Whenever two or more commands are specified, each command is applied (in order) to every line in the file. In the above example, first the '=' command is applied to line 1, and then the 'p' command is applied. Then, sed proceeds to line 2, and repeats the process. While the semicolon is handy, there are instances where it won't work. Another alternative is to use two -e options to specify two separate commands:

```
$ sed -n -e '=' -e 'p' myfile.txt
```

However, when we get to the more complex append and insert commands, even multiple '-e' options won't help us. For complex multiline scripts, the best way is to put your commands in a separate file. Then, reference this script file with the -f options:

```
$ sed -n -f mycommands.sed myfile.txt
```

This method, although arguably less convenient, will always work.

Multiple commands for one address

Sometimes, you may want to specify multiple commands that will apply to a single address. This comes in especially handy when you are performing lots of 's///' to transform words or syntax in the source file. To perform multiple commands per address, enter your sed commands in a file, and use the '{ }' characters to group commands, as follows:

```
1,20{
 s/[Ll]inux/GNU\/Linux/g
 s/samba/Samba/g
 s/posix/POSIX/g
}
```

The above example will apply three substitution commands to lines 1 through 20, inclusive. You can also use regular expression addresses, or a combination of the two:

```
1,/^END/{
        s/[Ll]inux/GNU\/Linux/g
        s/samba/Samba/g
        s/posix/POSIX/g
 p
}
```

This example will apply all the commands between '{ }' to the lines starting at 1 and up to a line beginning with the letters "END", or the end of file if "END" is not found in the source file.

Append, insert, and change line

Now that we're writing sed scripts in separate files, we can take advantage of the append, insert, and change line commands. These commands will insert a line after the current line, insert a line before the current line, or replace the current line in the pattern space. They can also be used to insert multiple lines into the output. The insert line command is used as follows:

```
i\
This line will be inserted before each line
```

If you don't specify an address for this command, it will be applied to each line and produce output that looks like this:

```
This line will be inserted before each line
line 1 here
This line will be inserted before each line
line 2 here
This line will be inserted before each line
line 3 here
This line will be inserted before each line
line 4 here
```

If you'd like to insert multiple lines before the current line, you can add additional lines by appending a backslash to the previous line, like so:

```
i\

insert this line\

and this one\

and this one\

and, uh, this one too.
```

The append command works similarly, but will insert a line or lines after the current line in the pattern space. It's used as follows:

```
a\

insert this line after each line.  Thanks! :)
```

On the other hand, the "change line" command will actually *replace* the current line in the pattern space, and is used as follows:

```
c\

You're history, original line! Muhahaha!
```

Because the append, insert, and change line commands need to be entered on multiple lines, you'll want to type them in to text sed scripts and tell sed to source them by using the '-f' option. Using the other methods to pass commands to sed will result in problems.

Next time

Next time, in the final article of this series on sed, I'll show you lots of excellent real-world examples of using sed for many different kinds of tasks. Not only will I show you what the scripts do, but *why* they do what they do. After you're done, you'll have additional excellent ideas of how to use sed in your various projects. I'll see you then!

Related topic

- If you'd like a good old-fashioned book, O'Reilly's sed & awk, 2nd Edition would be wonderful choice.