Common threads: Sed by example, Part 1

Get to know the powerful UNIX editor

Daniel Robbins September 01, 2000

In this series of articles, Daniel Robbins will show you how to use the very powerful (but often forgotten) UNIX stream editor, sed. Sed is an ideal tool for batch-editing files or for creating shell scripts to modify existing files in powerful ways.

Pick an editor

In the UNIX world, we have a lot of options when it comes to editing files. Think of it -- vi, emacs, and jed come to mind, as well as many others. We all have our favorite editor (along with our favorite keybindings) that we have come to know and love. With our trusty editor, we are ready to tackle any number of UNIX-related administration or programming tasks with ease.

While interactive editors are great, they do have limitations. Though their interactive nature can be a strength, it can also be a weakness. Consider a situation where you need to perform similar types of changes on a group of files. You could instinctively fire up your favorite editor and perform a bunch of mundane, repetitive, and time-consuming edits by hand. But there's a better way.

Enter sed

It would be nice if we could automate the process of making edits to files, so that we could "batch" edit files, or even write scripts with the ability to perform sophisticated changes to existing files. Fortunately for us, for these types of situations, there is a better way -- and the better way is called "sed".

sed is a lightweight stream editor that's included with nearly all UNIX flavors, including Linux. sed has a lot of nice features. First of all, it's very lightweight, typically many times smaller than your favorite scripting language. Secondly, because sed is a *stream* editor, it can perform edits to data it receives from stdin, such as from a pipeline. So, you don't need to have the data to be edited stored in a file on disk. Because data can just as easily be piped to sed, it's very easy to use sed as part of a long, complex pipeline in a powerful shell script. Try doing that with your favorite editor.

GNU sed

Fortunately for us Linux users, one of the nicest versions of sed out there happens to be GNU sed, which is currently at version 3.02. Every Linux distribution has GNU sed, or at least should. GNU

sed is popular not only because its sources are freely distributable, but because it happens to have a lot of handy, time-saving extensions to the POSIX sed standard. GNU sed also doesn't suffer from many of the limitations that earlier and proprietary versions of sed had, such as a limited line length -- GNU sed handles lines of any length with ease.

The newest GNU sed

While researching this article, I noticed that several online sed aficionados made reference to a GNU sed 3.02a. Strangely, I couldn't find sed 3.02a on **ftp.gnu.org** (see Related topics for these links), so I had to go look for it elsewhere. I found it at **alpha.gnu.org**, in /pub/sed. I happily downloaded it, compiled it, and installed it, only to find minutes later that the most recent version of sed is 3.02.80 -- and you can find its sources right next to those for 3.02a, at **alpha.gnu.org**. After getting GNU sed 3.02.80 installed, I was finally ready to go.

alpha.gnu.org
alpha.gnu.org (see Related topics) is the home of new and experimental GNU source code. However, you'll also find a lot of nice, stable source code there, too. For some reason a lot of the GNU developers either forget to move stable sources over to ftp.gnu.org, or they have unusually long (2 years!) "beta" periods. As an example, sed 3.02a is two years old, and even 3.02.80 is one year old, but they're still (at the time this article was written, August 2000) not available on ftp.gnu.org!

The right sed

In this series, we will be using GNU sed 3.02.80. Some (but very few) of the most advanced examples you'll find in my upcoming, follow-on articles in this series will not work with GNU sed 3.02 or 3.02a. If you're using a non-GNU sed, your results may vary. Why not take some time to install GNU sed 3.02.80 now? Then, not only will you be ready for the rest of the series, but you'll also be able to use arguably the best sed in existence!

Sed examples

Sed works by performing any number of user-specified editing operations ("commands") on the input data. Sed is line-based, so the commands are performed on each line in order. And, sed writes its results to standard output (stdout); it doesn't modify any input files.

Let's look at some examples. The first several are going to be a bit weird because I'm using them to illustrate how sed works rather than to perform any useful task. However, if you're new to sed, it's very important that you understand them. Here's our first example:

```
$ sed -e 'd' /etc/services
```

If you type this command, you'll get absolutely no output. Now, what happened? In this example, we called sed with one editing command, 'd. Sed opened the /etc/services file, read a line into its pattern buffer, performed our editing command ("delete line"), and then printed the pattern buffer (which was empty). It then repeated these steps for each successive line. This produced no output, because the "d" command zapped every single line in the pattern buffer!

There are a couple of things to notice in this example. First, /etc/services was not modified at all. This is because, again, sed only reads from the file you specify on the command line, using it

as input -- it doesn't try to modify the file. The second thing to notice is that sed is line-oriented. The 'd' command didn't simply tell sed to delete all incoming data in one fell swoop. Instead, sed read each line of /etc/services one by one into its internal buffer, called the pattern buffer. Once a line was read into the pattern buffer, it performed the 'd' command and printed the contents of the pattern buffer (nothing in this example). Later, I'll show you how to use address ranges to control which lines a command is applied to -- but in the absence of addresses, a command is applied to *all lines*.

The third thing to notice is the use of single quotes to surround the 'd' command. It's a good idea to get into the habit of using single quotes to surround your sed commands, so that shell expansion is disabled.

Another sed example

Here's an example of how to use sed to remove the first line of the /etc/services file from our output stream:

```
$ sed -e '1d' /etc/services | more
```

As you can see, this command is very similar to our first 'd' command, except that it is preceded by a '1'. If you guessed that the '1' refers to line number one, you're right. While in our first example, we used 'd' by itself, this time we use the 'd' command preceded by an optional numerical address. By using addresses, you can tell sed to perform edits only on a particular line or lines.

Address ranges

Now, let's look at how to specify an address *range*. In this example, sed will delete lines 1-10 of the output:

```
$ sed -e '1,10d' /etc/services | more
```

When we separate two addresses by a comma, sed will apply the following command to the range that starts with the first address, and ends with the second address. In this example, the 'd' command was applied to lines 1-10, inclusive. All other lines were ignored.

Addresses with regular expressions

Now, it's time for a more useful example. Let's say you wanted to view the contents of your /etc/services file, but you aren't interested in viewing any of the included comments. As you know, you can place comments in your /etc/services file by starting the line with the '#' character. To avoid comments, we'd like sed to delete lines that start with a '#'. Here's how to do it:

```
$ sed -e '/^#/d' /etc/services | more
```

Try this example and see what happens. You'll notice that sed performs its desired task with flying colors. Now, let's figure out what happened.

To understand the '/^#/d' command, we first need to dissect it. First, let's remove the 'd' -- we're using the same delete line command that we've used previously. The new addition is the '/^#/' part, which is a new kind of *regular expression* address. Regular expression addresses are always surrounded by slashes. They specify a *pattern*, and the command that immediately follows a regular expression address will only be applied to a line if it happens to match this particular pattern.

So, '/^#/' is a regular expression. But what does it do? Obviously, this would be a good time for a regular expression refresher.

Regular expression refresher

We can use regular expressions to express patterns that we may find in the text. If you've ever used the '*' character on the shell command line, you've used something that's similar, but not identical to, regular expressions. Here are the special characters that you can use in regular expressions:

Character	Description
^	Matches the beginning of the line
$	Matches the end of the line
.	Matches any single character
*	Will match zero or more occurrences of the *previous* character
[ ]	Matches all the characters inside the [ ]

Probably the best way to get your feet wet with regular expressions is to see a few examples. All of these examples will be accepted by sed as valid addresses to appear on the left side of a command. Here are a few:

Regularexpression	Description
/./	Will match any line that contains at least one character
/../	Will match any line that contains at least two characters
/^#/	Will match any line that begins with a '#'
/^$/	Will match all blank lines
/}$/	Will match any lines that ends with '}' (no spaces)
/} *$/	Will match any line ending with '}' followed by *zero* or more spaces
/[abc]/	Will match any line that contains a lowercase 'a', 'b', or 'c'
/^[abc]/	Will match any line that *begins* with an 'a', 'b', or 'c'

I encourage you to try several of these examples. Take some time to get familiar with regular expressions, and try a few regular expressions of your own creation. You can use a regexp this way:

```
$ sed -e '/regexp/d' /path/to/my/test/file | more
```

This will cause sed to delete any matching lines. However, it may be easier to get familiar with regular expressions by telling sed to *print* regexp matches, and delete non-matches, rather than the other way around. This can be done with the following command:

```
$ sed -n -e '/regexp/p' /path/to/my/test/file | more
```

Note the new '-n' option, which tells sed to not print the pattern space unless explicitly commanded to do so. You'll also notice that we've replaced the 'd' command with the 'p' command, which as you might guess, explicitly commands sed to print the pattern space. Voila, now only matches will be printed.

More on addresses

Up till now, we've taken a look at line addresses, line range addresses, and regexp addresses. But there are even more possibilities. We can specify two regular expressions separated by a comma, and sed will match all lines starting from the first line that matches the first regular expression, up to and including the line that matches the second regular expression. For example, the following command will print out a block of text that begins with a line containing "BEGIN", and ending with a line that contains "END":

```
$ sed -n -e '/BEGIN/,/END/p' /my/test/file | more
```

If "BEGIN" isn't found, no data will be printed. And, if "BEGIN" is found, but no "END" is found on any line below it, all subsequent lines will be printed. This happens because of sed's stream-oriented nature -- it doesn't know whether or not an "END" will appear.

C source example

If you want to print out only the main() function in a C source file, you could type:

```
$ sed -n -e '/main[[:space:]]*(/,/^}/p' sourcefile.c | more
```

This command has two regular expressions, '/main[[:space:]]*(/' and '/^}/', and one command, 'p'. The first regular expression will match the string "main" followed by any number of spaces or tabs, followed by an open parenthesis. This should match the start of your average ANSI C main() declaration.

In this particular regular expression, we encounter the '[[:space:]]' character class. This is simply a special keyword that tells sed to match either a TAB or a space. If you wanted, instead of typing '[[:space:]]', you could have typed '[', then a literal space, then Control-V, then a literal tab and a ']' -- The Control-V tells bash that you want to insert a "real" tab rather than perform command expansion. It's clearer, especially in scripts, to use the '[[:space:]]' command class.

OK, now on to the second regexp. '/^}' will match a '}' character that appears at the beginning of a new line. If your code is formatted nicely, this will match the closing brace of your main() function. If it's not, it won't -- one of the tricky things about performing pattern matching.

The 'p' command does what it always does, explicitly telling sed to print out the line, since we are in '-n' quiet mode. Try running the command on a C source file -- it should output the entire main() { } block, including the initial "main()" and the closing '}'.

Next time

Now that we've touched on the basics, we'll be picking up the pace for the next two articles. If you're in the mood for some meatier sed material, be patient -- it's coming! In the meantime, you might want to check out the following sed and regular expression resources.

Related topics

- Read Daniel's other sed articles on *developerWorks*: Common threads: Sed by example, Part 2 and Part 3.
- If you'd like a good old-fashioned book, O'Reilly's sed & awk, 2nd Edition would be wonderful choice.
- Read David Mertz's article on "Text processing in Python" on *developerWorks*.
- See the regular expressions how-to document from Python.org.
- Refer to an overview of regular expressions from the University of Kentucky.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.