Intermediate Sed: Manipulating Streams of Text in a Linux Environment

Authored by: ***ASPHostServer Administrator*** *[asphostserver@gmail.com]*
Saved From: *http://faq.asphosthelpdesk.com/article.php?id=207*

Introduction

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

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

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.

If you don't already have the files at hand, let's recreate our environment from last time so that we have some files to manipulate:

```
cd
cp /usr/share/common-licenses/BSD .
cp /usr/share/common-licenses/GPL-3 .
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..." > annoying.txt
```

Since sed operates through standard input and output, we can, of course, just string different calls to sed together through a pipeline (remember to escape the "&" since it means "the complete matched pattern" to sed):

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

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, takes up more space, and does not take advantage of sed's built-in capabilities.

We can string various commands to sed by using the "-e" option before each command. This is how the above command would be re-written:

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

Another approach to stringing commands together is using a semi-colon character (;) to separate distinct commands. This works the same as above, but the "-e" is not required.

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

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

To demonstrate, we will introduce the "G" command, which by default, enters a blank line between each line (this actually is more complex, but we'll learn about that later).

```
sed 'G' annoying.txt

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 we combine these two commands, we might expect a space between each regular line and line-number line. However, we get something different:

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

```
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.

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

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

1
this is the song that never ends
2
yes, it goes on and on, my friend
3
some people started singing it
. . .
. . .
```

We now see the results we were expecting. 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 we are not limited to operating on known line values, like we learned previously:

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

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

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

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

```
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, we've placed "loudly" after the first "it" in 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. This 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
```

```
                    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 regular expressions can be used on either side of a range as well.

For instance, we can delete lines starting at a line that only contains the word "START" until a line reading "END" by issuing the following command:

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

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 we want to invert an address (operate on any line that does NOT match a pattern), we can follow the pattern with an exclamation point or bang (!).

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

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

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 we 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 we'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. We 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}' annoying.txt
```

```
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 we specifically tell it too.

The first part of the script 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, as we described above, would shorten this considerably. This command will produce the same results that we've just seen:

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

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...
```

Using Sed 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).

For example:

```
s/this/that/g
s/snow/rain/g
1,5s/pinecone/apricot/g
```

We could then call the file by using the following syntax:

```
sed -f sedScriptName fileToEdit
```

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

In this article, we've added some depth to our understanding of sed.

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 intelligent your usage becomes, the less work you will have to do in the long run.