Linux Fu: Bash Strings

Originalartikel

Backup

<html> If you are a traditional programmer, using

```
bash
```

for scripting may seem limiting sometimes, but for certain tasks,

```
bash
```

can be very productive. It turns out, some of the limits of

```
bash
```

are really limits of older shells and people code to that to be compatible. Still other perceived issues are because some of the advanced functions in

```
bash
```

are arcane or confusing. Strings are a good example. You don’t think of

```
bash
```

as a string manipulation language, but it has many powerful ways to handle strings. In fact, it may have too many ways, since the functionality winds up in more than one place. Of course, you can also call out to programs, and sometimes it is just easier to make a call to an

```
awk
```

or Python script to do the heavy lifting. But let’s stick with

```
bash
```

-isms for handling strings. Obviously, you can put a string in an environment variable and pull it back out. I am going to assume you know how string interpolation and quoting works. In other words, this should make sense: <pre class=„brush: bash; title: ; notranslate“ title=„“>echo „Your path is $PATH and the current directory is ${PWD}“</pre><h2>The Long and the Short</h2> Suppose you want to know the length of a string. That’s a pretty basic string operation. In

```
bash
```

, you can write

```
${#var}
```

to find the length of

```
$var
```

: <pre class=„brush: bash; title: ; notranslate“ title=„“>#/bin/bashecho -n „Project Name? „read PNAMEif [1]then echo Error: Project name longer than 16 characterselse echo ${PNAME} it is!fi</pre> The “[2]do let LAST_SLASH=$LAST_SLASH+$SLASH # point at next slash SLASH=$(expr index „${FFN:$LAST_SLASH}“ /) # look for anotherdone# now LAST_SLASH points to last slashecho -n „Directory: „expr substr „$FFN“ 1 $LAST_SLASHecho -or-echo ${FFN:0:$LAST_SLASH}# Yes, I know about dirname but this is an example</pre> Enter a full path (like

```
/foo/bar/hackaday
```

) and the script will find the last slash and print the name up to and including the last slash using two different methods. This script makes use of

```
expr
```

but also uses the syntax for

```
bash
```

‘s built in substring extraction which starts at index zero. For example, if the variable FOO contains “Hackaday”: ${FOO} -> Hackaday
${FOO:1} -> ackaday
${FOO:5:3} -> day
<p>The first number is an offset and the second is a length if it is positive. You can also make either of the numbers negative, although you need a space after the colon if the offset is negative. The last character of the string is at index -1, for example. A negative length is shorthand for an absolute position from the end of the string. So:</p>
${FOO: -3} -> day
${FOO:1:-4} -> ack
${FOO: -8:-4} -> Hack Of course, either or both numbers could be variables, as you can see in the example. <h2>Less is More</h2> Sometimes you don’t want to find something, you just want to get rid of it.

```
bash
```

has lots of ways to remove substrings using fixed strings or glob-based pattern matching. There are four variations. One pair of deletions remove the longest and shortest possible substrings from the front of the string and the other pair does the same thing from the back of the string. Consider this: <pre class=„brush: bash; title: ; notranslate“ title=““>TSTR=my.first.file.txtecho ${TSTR%.*} # prints my.first.fileecho ${TSTR%%.*} # prints myecho ${TSTR#*fi} # prints rst.file.txtecho $TSTR##*fi} # prints le.txt</pre><h2>Transformation</h2> Of course, sometimes you don’t want to delete, as much as you want to replace some string with another string. You can use a single slash to replace the first instance of a search string or two slashes to replace globally. You can also fail to provide a replacement string and you’ll get another way to delete parts of strings. One other trick is to add a # or % to anchor the match to the start or end of the string, just like with a deletion. <pre class=„brush: bash; title: ; notranslate“ title=““>TSTR=my.first.file.txtecho ${TSTR/fi/Fi} # my.First.file.txtecho ${TSTR*fi/Fi} # *my.First.File.txtecho ${TSTR/#*./PREFIX-} # PREFIX-txt (note: always longest match)echo ${TSTR/%.*/.backup} # my.backup (note: always longest*

match)</pre><h2>Miscellaneous</h2> Some of the more common ways to manipulate strings in <code>bash</code> have to do with dealing with parameters. Suppose you have a script that expects a variable called <code>OTERM</code> to be set but you want to be sure: <pre class=„brush: bash; title: ; notranslate“ title=„“>REALTERM=${OTERM:-vt100}</pre> Now <code>REALTERM</code> will have the value of <code>OTERM</code> or the string “vt100” if there was nothing in <code>OTERM</code>. Sometimes you want to set <code>OTERM</code> itself so while you could assign to <code>OTERM</code> instead of <code>REALTERM</code>, there is an easier way. Use := instead of the :- sequence. If you do that, you don’t necessarily need an assignment at all, although you can use one if you like: <pre class=„brush: bash; title: ; notranslate“ title=„“>echo ${OTERM:=vt100} # now OTERM is vt100 if it was empty before</pre> You can also reverse the sense so that you replace the value only if the main value is not empty, although that’s not as generally useful: <pre class=„brush: bash; title: ; notranslate“ title=„“>echo ${DEBUG:+„Debug mode is ON“} # reverse -; no assignment</pre><p>A more drastic measure lets you print an error message to stderr and abort a non-interactive shell:</p><pre class=„brush: bash; title: ; notranslate“ title=„“>REALTERM=${OTERM:?„Error. Please set OTERM before calling this script“}</pre><h2>Just in Case</h2> Converting things to upper or lower case is fairly simple. You can provide a glob pattern that matches a single character. If you omit it, it is the same as ?, which matches any character. You can elect to change all the matching characters or just attempt to match the first character. Here are the obligatory examples: <pre class=„brush: bash; title: ; notranslate“ title=„“>NAME=„joe Hackaday“echo ${NAME^} # prints Joe Hackaday (first match of any character)echo ${NAME^^} # prints JOE HACKADAY (all of any character)echo ${NAME^^[a]} # prints joe HAckAdAy (all a characters)echo ${NAME,,] # prints joe hackaday (all characters)echo ${NAME,] # prints joe Hackaday (first character matched and didn't convert)NAME=„Joe Hackaday“echo ${NAME,,[A-H]} # prints Joe hackaday (apply pattern to all characters and convert A-H to lowercase)</pre><p>Recent versions of <code>bash</code> can also convert upper and lower case using <code>${VAR@U}</code> and <code>${VAR@L}</code> along with just the first character using <code>@u</code> and <code>@l</code>, but your mileage may vary. <h2>Pass the Test</h2> You probably realize that when you do a standard test, that actually calls a program: <pre class=„brush: bash; title: ; notranslate“ title=„“>if [$f -eq 0]then ...</pre> If you do an ls on <code>/usr/bin</code>, you’ll see an executable actually named “[” used as a shorthand for the test program. However, <code>bash</code> has its own test in the form of two brackets: <pre class=„brush: bash; title: ; notranslate“ title=„“>if $f == 0then ...</pre> That test built-in can handle regular expressions using =~ so that’s another option for matching strings: <pre class=„brush: bash; title: ; notranslate“ title=„“>if "$NAME" =~ [hH]a.k ...</pre><h2>Choose Wisely</h2> Of course, if you are doing a slew of text processing, maybe you don’t need to be using <code>bash</code>. Even if you are, don’t forget you can always leverage other programs like tr, <code>awk</code>, <code>sed</code>, and many others to do things like this. Sure, performance won’t be as good — probably — but if you are worried about performance why are you writing a script? Unless you just swear off scripting altogether, it is nice to have some of these tricks in your back pocket. Use them wisely. </html>

1)

${#PNAME} > 16

2)

” forms an arithmetic context which is why you can get away with an unquoted greater-than sign here. If you don’t mind using

```
expr
```

— which is an external program — there are at least two more ways to get

there: <pre class=„brush: bash; title: ; notranslate“ title=““>echo ${#STR}expr length „${STR}„expr match „${STR}“ '.*'</pre> Of course, if you allow yourself to call outside of

```
bash
```

, you could use

```
awk
```

or anything else to do this, too, but we’ll stick with

```
expr
```

as it is relatively lightweight. <h2>Swiss Army Knife</h2> In fact,

```
expr
```

can do a lot of string manipulations in addition to length and match. You can pull a substring from a string using

```
substr
```

. It is often handy to use

```
index
```

to find a particular character in the string first. The

```
expr
```

program uses 1 as the first character of the string. So, for example: <pre class=„brush: bash; title: ; notranslate“ title=““>#/bin/bashecho -n „Full path? „read FFNLAST_SLASH=0SLASH=$(expr index „$FFN“ /) # find first slashwhile (($SLASH != 0