Linux Fu: Better Bash Scripting

Originalartikel

Backup

<html> It is easy to dismiss bash — the typical Linux shell program — as just a command prompt that allows scripting. Bash, however, is a full-blown programming language. I wouldn’t presume to tell you that it is as fast as a compiled C program, but that’s not why it exists. While a lot of people use shell scripts as an analog to a batch file in MSDOS, it can do so much more than that. Contrary to what you might think after a casual glance, it is entirely possible to write scripts that are reliable and robust enough to use in many embedded systems on a Raspberry Pi or similar computer. I say that because sometimes bash gets a bad reputation. For one thing, it emphasizes ease-of-use. So while it has features that can promote making a robust script, you have to know to turn those features on. Another issue is that a lot of the functionality you’ll use in writing a bash script doesn’t come from bash, it comes from Linux commands (or whatever environment you are using; I’m going to assume some Linux distribution). If those programs do bad things, that isn’t a problem specific to bash. One other limiting issue to bash is that many people (and I’m one of them) tend to write scripts using constructs that are compatible with older shells. Often times bash can do things better or neater, but we still use the older ways. For example: <pre>if [$X -gt 0]
 then ….
 fi</pre> This works in bash and a lot of other similar shells. However, bash can do better, for example, working on strings instead of integers: <pre>if 0
 then …
 fi</pre> <h2>Features</h2> Don’t think bash is a programming language? It has arrays, loops, sockets, regular expression matching, file I/O, and lots more. However, there are a few things you should know when writing scripts that you expect to work well. You might add your own items to this list, but this one is what comes to my mind:

Use “set -o errexit” to cause the script to exit if any line fails Use “set -o nounset” to cause an error if you use an empty environment variable If you don’t expect a variable to change, declare it readonly Expect variables could have spaces and quote accordingly Use traps to clean up your mess <h2>Exit on Error</h2> If you use “set -o errexit” then any line that returns a non-zero error code will stop script execution. You might object that you want to test for that condition like this: <pre>some_command
 if $
 then
    recover
 fi</pre> If you use the errexit flag, that test will never occur because once some_command throws the error, you are done. Simply rewrite like this: <pre>some_command || recover</pre> The effect is if some_command returns true (that is, zero), then bash knows the OR operator is satisfied so it doesn’t run any more commands. If it fails, then bash can’t tell if the OR is satisfied or not, so it runs recover. The exit code of the entire thing is either 0 from some_command or the exit code of recover, whatever that is. Sometimes you have a command that could return an error and you don’t care. That’s easy to fix: <pre>some_other_command || true</pre> By the way, usually, the last item in a pipeline determines the result. For example: <pre>a | b | c</pre> The exit code of that line is whatever c returns. However, you can “set -o pipefail” to cause any error code in the pipe to halt the script. Even better is the $PIPESTATUS variable which is an array with all the exit codes from the last pipeline. So whatever program b returned will be in ${PIPESTATUS[1]}, in the above example.</p> <h2>Unset Variables</h2> <p>Using “set -o nonunset” forces you to initialize all variables. For example, here’s a really bad script (don’t run it):</p> <pre>TOPDIR=tmp
 #rm -rf /${TOPDIRR}</pre> You can argue this isn’t great code, but regardless, because TOPDIR has a typo in the last line, you’ll erase your root directory if this runs without the protective comment in front of the rm.

This works for command line parameters, too, so it will protect you if you had: bad_cmd /$1 <h2>Readonly</h2> Many times you set a variable and you really need a constant. It shouldn’t change as the script executes and if it does that indicates a bug. You can declare those readonly: <pre>readonly BASEDIR=„~/testdir“
 readonly TIMEOUT_S=10</pre> <h2>Expect Spaces</h2> File systems allow spaces and people love to use them. This can lead to unfortunate things like: <pre>rm $2</pre> When $2 is something like “readme.txt” that’s fine. However, if $2 is “The quick red fox” you wind up trying to erase four files named “The,” “quick,” “red,” and “fox.” If you are lucky, none of those files exist and you get errors. If you are unlucky, you just erased the wrong file. The simple answer is to quote everything. <pre>rm „$2“</pre> If you ever use $@ to get all arguments, you should quote it to prevent problems. Consider this script: <pre>#!/bin/bash
 function p1
 {
   echo $1
 }

 p1 $@
 p1 „$@“</pre> Try running the script with a quoted argument like “the quick red fox”. The first function call will get four arguments and the second call will get only one, which is almost surely what you intended. <h2>Traps</h2> <img data-attachment-id=„263325“ data-permalink=„http://hackaday.com/2017/07/21/linux-fu-better-bash-scripting/trap/“ data-orig-file=„https://hackadaycom.files.wordpress.com/2017/06/trap.jpg“ data-orig-size=„800,289“ data-comments-opened=„1“ data-image-meta=„{"aperture":"0","credit":"","camera":"","caption":"","created_timestamp":"0","copyright":"","focal_length":"0","iso":"0","shutter_speed":"0","title":"","orientation":"0"}“ data-image-title=„trap“ data-image-description=„“ data-medium-file=„https://hackadaycom.files.wordpress.com/2017/06/trap.jpg?w=400&h=145“ data-large-file=„https://hackadaycom.files.wordpress.com/2017/06/trap.jpg?w=800“ class=„size-medium wp-image-263325 alignleft“ src=„https://hackadaycom.files.wordpress.com/2017/06/trap.jpg?w=400&h=145“ alt=„“ width=„400“ height=„145“ srcset=„https://hackadaycom.files.wordpress.com/2017/06/trap.jpg?w=400&h=145 400w,

https://hackadaycom.files.wordpress.com/2017/06/trap.jpg?w=250&h=90 250w, https://hackadaycom.files.wordpress.com/2017/06/trap.jpg?w=768&h=277 768w, https://hackadaycom.files.wordpress.com/2017/06/trap.jpg 800w“ sizes=„(max-width: 400px) 100vw, 400px“/>It isn’t uncommon for scripts to create temporary lock files and other things that need cleaning up if the script stops. That’s what the trap command is for. Suppose you are working on building a file called /tmp/output.data and you want to remove it if you don’t get a chance to complete. Easy: <pre style=„clear:both;“>trap „rm -f /tmp/output; exit“ INT TERM EXIT</pre> You can look up the trap command for more details, but this is a great way to make things happen when a script ends for any reason. The exit command in quotes, by the way, is necessary or else the script will attempt to keep running. Of course, in an embedded system, you might want that behavior, too. You probably want to remove the trap before you are done unless you really want output.data deleted, so: <pre>trap - INT TERM EXIT</pre> <h2>Wrap Up</h2> You should consider turning off features you don’t need, especially if taking input from outside your script. For example, using “set -o noglob” will prevent bash from expanding wildcards. Of course, if you need wildcards, you can’t do this — at least not for the part of the script that uses them. You can also use “shopt -s failglob” which will cause wildcards to throw an error, if you want to secure your script. Speaking of security, be very careful running user input as commands. Security is an entirely different topic, but even something that seems innocent can be maniuplated to do bad things if you are not careful. For example, suppose you secure sudo to allow a few commands and you offer the script: <pre>sudo -u protuser „$@“</pre> If sudo is set up right, what’s the harm? Well… the harm is that I can pass the argument “-u root reboot” (for example) and sudo will decide I’m root instead of protuser. Be careful! There are a lot of tricks to writing bash scripts that are portable. I don’t care about those in this context because if I’m deploying an embedded system on a Raspberry Pi, I will control the configuration so that I know where /tmp is and where bash is located and what version of different programs are available. However, if you are distributing scripts to machines you don’t control, you might consider searching the internet about bash script portability. If you want to catch a lot of potential errors in scripts (including some portability issues) you can try ShellCheck. You might also appreciate Google’s shell style guide. If you aren’t sure bash is really a programming language, this should convince you. </html>

From:
https://schnipsl.qgelm.de/ - **Qgelm**

Permanent link:
https://schnipsl.qgelm.de/doku.php?id=wallabag:linux-fu_-better-bash-scripting

Last update: **2021/12/06 15:24**