howto, bash, robust, shell, scripts

bash - Writing Robust Bash Shell Scripts

Local copy of: https://www.davidpashley.com/articles/writing-robust-shell-scripts/

Many people hack together shell scripts quickly to do simple tasks, but these soon take on a life of their own. Unfortunately shell scripts are full of subtle effects which result in scripts failing in unusual ways. It's possible to write scripts which minimise these problems. In this article, I explain several techniques for writing robust bash scripts.

Use set -u

How often have you written a script that broke because a variable wasn't set? I know I have, many times.

```
chroot=$1
...
rm -rf $chroot/usr/share/doc
```

If you ran the script above and accidentally forgot to give a parameter, you would have just deleted all of your system documentation rather than making a smaller chroot. So what can you do about it? Fortunately bash provides you with set -u, which will exit your script if you try to use an uninitialised variable. You can also use the slightly more readable *set -o nounset*.

```
bash /tmp/shrink-chroot.sh
```

```
/tmp/shrink-chroot.sh>
```

```
$chroot=
```

```
bash -u /tmp/shrink-chroot.sh
```

```
/tmp/shrink-chroot.sh: line 3: $1: unbound variable
```

Use set -e

Every script you write should include *set -e* at the top. This tells bash that it should exit the script if any statement returns a non-true return value. The benefit of using *-e* is that it prevents errors snowballing into serious issues when they could have been caught earlier. Again, for readability you may want to use *set -o errexit*.

Using *-e* gives you error checking for free. If you forget to check something, bash will do it or you. Unfortunately it means you can't check $? as bash will never get to the checking code if it isn't zero. There are other constructs you could use:

```
command
if [ "$?"-ne 0]; then echo "command failed"; exit 1; fi
```

could be replaced with

```
command || { echo "command failed"; exit 1; }
```

or

```
```

```
if ! command; then echo "command failed"; exit 1; fi
```

What if you have a command that returns non-zero or you are not interested in its return value? You can use command || true, or if you have a longer section of code, you can turn off the error checking, but I recommend you use this sparingly.

```
set +e
command1
command2
set -e
```

On a slightly related note, by default bash takes the error status of the last item in a pipeline, which may not be what you want. For example, *false | true* will be considered to have succeeded. If you would like this to fail, then you can use *set -o pipefail* to make it fail.

Program defensively – expect the unexpected

Your script should take into account of the unexpected, like files missing or directories not being created. There are several things you can do to prevent errors in these situations. For example, when you create a directory, if the parent directory doesn't exist, **mkdir** will return an error. If you add a -p option then **mkdir** will create all the parent directories before creating the requested directory. Another example is **rm**. If you ask **rm** to delete a non-existent file, it will complain and your script will terminate. (You are using *-e*, right?) You can fix this by using *-f*, which will silently continue if the file didn't exist.

Be prepared for spaces in filenames

Someone will always use spaces in filenames or command line arguments and you should keep this in mind when writing shell scripts. In particular you should use quotes around variables.

```
if [ $filename = "foo" ];
```

will fail if *$filename* contains a space. This can be fixed by using:

```
if [ "$filename" = "foo" ];
```

When using *$@ variable*, you should always quote it or any arguments containing a space will be expanded in to separate words.

```
foo() { for i in $@; do printf "%s\n" "$i"; done }; foo bar "baz quux"
```

```
bar
baz
quux
```

```
david% foo() { for i in "$@"; do printf "%s\n" "$i"; done }; foo bar "baz quux"
```

```
bar
baz quux
```

I can not think of a single place where you shouldn't use *"$@"* over *$@*, so when in doubt, use quotes.

If you use **find** and **xargs** together, you should use *-print0* to separate filenames with a null character rather than new lines. You then need to use *-0* with **xargs**.

```
touch "foo bar"
find | xargs ls
```

```
ls: ./foo: No such file or directory
ls: bar: No such file or directory
```

```
david% find -print0 | xargs -0 ls
```

```
./foo bar
```

Setting traps

Often you write scripts which fail and leave the filesystem in an inconsistent state; things like lock files, temporary files or you've updated one file and there is an error updating the next file. It would be nice if you could fix these problems, either by deleting the lock files or by rolling back to a known good state when your script suffers a problem. Fortunately bash provides a way to run a command or function when it receives a unix signal using the **trap** command.

```
trap command signal [signal ...]
```

There are many signals you can trap (you can get a list of them by running *kill -l*), but for cleaning up after problems there are only 3 we are interested in: **INT**, **TERM** and **EXIT**. You can also reset traps back to their default by using - as the command.

Signal	Description
INT	Interrupt – This signal is sent when someone kills the script by pressing ctrl-c.
TERM	Terminate – this signal is sent when someone sends the TERM signal using the kill command.
EXIT	Exit – this is a pseudo-signal and is triggered when your script exits, either through reaching the end of the script, an exit command or by a command failing when usingset -e.

Usually, when you write something using a lock file you would use something like:

```
if [ ! -e $lockfile ]; then
   touch $lockfile
   critical-section
   rm $lockfile
else
   echo "critical-section is already running"
fi
```

What happens if someone kills your script while *critical-section* is running? The lockfile will be left there and your script won't run again until it's been deleted. The fix is to use:

```
if [ ! -e $lockfile ]; then
   trap "rm -f $lockfile; exit" INT TERM EXIT
   touch $lockfile
   critical-section
   rm $lockfile
   trap - INT TERM EXIT
else
   echo "critical-section is already running"
fi
```

Now when you kill the script it will delete the lock file too. Notice that we explicitly exit from the script at the end of trap command, otherwise the script will resume from the point that the signal was received.

Race conditions

It's worth pointing out that there is a slight race condition in the above lock example between the time we test for the lockfile and the time we create it. A possible solution to this is to use IO redirection and bash's noclobber mode, which won't redirect to an existing file. We can use something similar to:

```
if ( set -o noclobber; echo "$$" > "$lockfile") 2> /dev/null;
then
   trap 'rm -f "$lockfile"; exit $?' INT TERM EXIT

   critical-section

   rm -f "$lockfile"
   trap - INT TERM EXIT
else
   echo "Failed to acquire lockfile: $lockfile."
   echo "Held by $(cat $lockfile)"
fi
```

A slightly more complicated problem is where you need to update a bunch of files and need the script to fail gracefully if there is a problem in the middle of the update. You want to be certain that something either happened correctly or that it appears as though it didn't happen at all.Say you had a script to add users.

```
add_to_passwd $user
cp -a /etc/skel /home/$user
chown $user /home/$user -R
```

There could be problems if you ran out of diskspace or someone killed the process. In this case you'd want the user to not exist and all their files to be removed.

```
rollback() {
   del_from_passwd $user
   if [ -e /home/$user ]; then
      rm -rf /home/$user
   fi
   exit
}

trap rollback INT TERM EXIT
add_to_passwd $user
cp -a /etc/skel /home/$user
chown $user /home/$user -R
trap - INT TERM EXIT
```

We needed to remove the trap at the end or the **rollback** function would have been called as we exited, undoing all the script's hard work.

Be atomic

Sometimes you need to update a bunch of files in a directory at once, say you need to rewrite urls form one host to another on your website. You might write:

```
for file in $(find /var/www -type f -name "*.html"); do
   perl -pi -e 's/www.example.net/www.example.com/' $file
done
```

Now if there is a problem with the script you could have half the site referring to www.example.com and the rest referring to www.example.net. You could fix this using a backup and a trap, but you also have the problem that the site will be inconsistent during the upgrade too.

The solution to this is to make the changes an (almost) atomic operation. To do this make a copy of the data, make the changes in the copy, move the original out of the way and then move the copy back into place. You need to make sure that both the old and the new directories are moved to locations that are on the same partition so you can take advantage of the property of most unix filesystems that moving directories is very fast, as they only have to update the inode for that directory.

```
cp -a /var/www /var/www-tmp
for file in $(find /var/www-tmp -type f -name "*.html"); do
```

```
   perl -pi -e 's/www.example.net/www.example.com/' $file
done
mv /var/www /var/www-old
mv /var/www-tmp /var/www
```

This means that if there is a problem with the update, the live system is not affected. Also the time where it is affected is reduced to the time between the two **mv**s, which should be very minimal, as the filesystem just has to change two entries in the inodes rather than copying all the data around.

The disadvantage of this technique is that you need to use twice as much disk space and that any process that keeps files open for a long time will still have the old files open and not the new ones, so you would have to restart those processes if this is the case. In our example this isn't a problem as apache opens the files every request. You can check for files with files open by using **lsof**. An advantage is that you now have a backup before you made your changes in case you need to revert.

~~DISCUSSION~~