Tip 69: Use a makefile for everything

Ben Klemens

17 February 2012

level: shell user
purpose: organize your shell scripts

A makefile is, for the most part, an organized set of shell scripts. Which is great, because you probably have a lot of silly little scripts floating around. Rather than producing a new one- or two-line script for every little task you have for your project, put them all into a makefile.

This site, for example, is generated via LaTeX. Here's the sort of makefile one could write for the process. [It's all lies: the actual process is much more involved, but the details wouldn't be relevant for you.]

Oh, and if you cut and paste, don't forget that the white space at the head of each line must be a tab, not spaces. Blame POSIX.

```
all: doc publish

doc:
pdflatex -interaction batchmode $(f).tex

publish:
scp $(f).pdf $BLOGSERVER
```

I would use it like this:

```
f=119-tip-make_everything make
```

because that sets the environment variable `f` and then calls make with the variable set. This is why `sh` won't let you you put spaces around the equal sign: the space is how it distinguishes between the assignment and the command.

But having set `f` and run make, the first target, `all`, gets called, and that depends on `doc` and `publish`, which get called in sequence. If I know it's not yet ready to copy out to the world, then I can call `make doc` and just do that step.

I'm sure you could imagine lots of other things one would do with a document (word count, spell check, write to revision control, push revision control out to a remote, make backup) that involves a nontrivial line or three of code; all of that can go into the makefile so you don't have to think about it.

How makefiles and shell scripts differ

- Shell variables have names in parens. E.g., `$(HOME)`.
- You will need to double your `$$`s for those odd cases when that's not sufficient:
  `for i in *.c; do cp $$i $${i%%.c}.bkup; done`
- *Every line runs independently, as if in a separate shell.* If you try
  `cd junkdir #do not try this at home.`
  `rm -f * #bad, bad.`
  then you will be a sad puppy. Make will first change in to the directory you are emptying, then Make is done with that subshell and returns to where you were. Then it starts a new subshell and calls `rm *`, from the directory you are in. On the plus side, Make will delete the erroneous makefile for you. Instead, do it like this: `cd junkdir && rm -f *` (where the `&&` runs commands in short-circuit sequence just line in C). Or use a backslash to continue the line, though for a case like this I wouldn't trust just a backslash.
- Did you see above how I set an environment variable and ran a command on the same line? That can come in handy now that each shell survives for a single line.
- An `@` at the head of a line means run the command but don't echo anything to the screen as it happens.
- A `-` at the head of a line means that if the command returns a nonzero value, keep going anyway. Otherwise, the script halts on the first nonzero return.

One more example target from my actual life. I use `git` here at home, but there are three subversion repositories I have to deal with. It's easy once you RTFM, but I never remember the sequence. Now I have a makefile to remember for me:

```
push:
@if [ "x$(MSG)" = 'x' ] ; then echo "MSG='whatever, dude.'" make push; fi
@test "x$(MSG)" != 'x'
git commit -a  -m "$(MSG)"
git svn fetch
git svn rebase
git svn dcommit
```

I need a message for each commit, so I do that via another environment variable set on the spot:

```
MSG="This is a commit." make
```

You can see that I have an if-then statement that prints a reminder if I forget this. Then, the makefile tests to make sure that `"x$(MSG)"` expands to something besides just `'x'`, meaning that `MSG` is not empty. This is a common shell idiom to make up for an idiotic POSIX glitch.

There you have it: makefiles aren't quite shell scripts for all these reasons, but you can summarize a lot of annoying procedural code in a makefile and never have to think about it again.