tildeverse zine dist/issue-2.pdf

Contents

How Dare You!

How Very Dare You!

makefile magic

author: ben

i've used makefiles plenty of times for a variety of projects, but my understanding has not really improved beyond the most basic level.

working with pandoc for the ~club wiki and the zine inspired me to replace a janky build script with some makefile goodness.

feeling a loss for where to start, i reached out to tomasino for some guidance. he responded that i should have a look at a makefile he built for a javascript project template which he linked on his github.

let's go through some of my biggest takeaways from inspecting tomasino's example and creating several of my own makefiles from it.

Figure 1: howdareyou.jpg

anatomy of a makefile

this was something that i thought i understood, but realized i didn't fully grok.

i *think* that i understand it now, or at least know now what to search for when i get stuck the next time.

let's start with the basics: a makefile should be named Makefile (note the capital M). the basic syntax consists of a target followed by a colon (`:`) and an optional list of requirements, then a list of commands to be run (indented by literal tabs).

```
myproc: source.c
    cc -o myproc source.c
```

this example defines how to build myproc, which depends on source.c.

variables makefiles use variables in two main formats.

- recursively-expanded: evaluated every time they're referenced, even in other variables
  - defined with a single `=`
- simply-expanded: evaluated once at the time of definition
  - defined with `:=`

simply-expanded variables are generally more predictable and will likely be what you should use in most cases.

```
x := foo
```

```
y := $(x) bar
x := later
```

will become

```
y := foo bar
x := later
```

variable names are case-sensitive. you'll likely see all-caps variable names more often than not.

guided example this was the major revelation for me coming from my brittle buildscript that deleted and recreated everything on each run.

make wants to know what files you want built, as well as how to build them.

let's take the ~club wiki as an example: we want to build one html for each markdown file in the source directory. how do we tell make which files to build and how to build each one?

using a standard static target name, you'd need to add a target for each markdown file.

```
ssh.html: source/ssh.md
    pandoc -so ssh.html source/ssh.md

git.html: source/git.md
    pandoc -so git.html source/git.md
```

this already feels like a lot of duplication. let's stay DRY. here's an example from tomasino's makefile that i adapted for the wiki: github source

```
$(DST_DIR)/js/%.js: $(SRC_DIR)/ts/%.ts
    $(mkdir)
    $(tsc) $< --outFile $@
```

in this example, we're telling make that files under `DST_DIR` are built from `SRC_DIR` with a command defined in the `tsc` variable.

let's adapt this to our wiki example:

```
%.html: source/%.md
    pandoc -so $@ $<
```

here we're telling make that it can build html files from the corresponding md file in source by running pandoc. but how do we tell make which html files we want to build? let's go back to tomasino's example

```
SRC_TS_FILES != find $(SRC_DIR)/ts -name '*.ts'
DST_JS_FILES := $(SRC_TS_FILES:$(SRC_DIR)/ts/%.ts=$(DST_DIR)/js/%.js)
```

in this case, we're using the special assignment form with `!=` that uses the output of a shell executable. we now have a list all the `.ts` files that need to be compiled in `SRC_TS_FILES`. next, we assign `DST_JS_FILES` with the special replacement syntax, which changes `ts/%.ts` into `js/%.js`. so, these variables now contain a list of source and desired files, along with a definition of how to build those files.

here's how i adapted this step for the wiki:

```
SRC_MD_FILES != find source -name '*.md'
DST_HTML_FILES := $(SRC_MD_FILES:source/%.md=%.html)
```

this means that make now has a list of html files that are required based on the list of all markdown files in the source directory.

now all that's left is to tell make that we want to build all of the `DST_HTML_FILES`. let's add that list to the `all` target:

```
all: $(DST_HTML_FILES)
```

now we can generate all html files by simply calling `make`

here's the completed Makefile.

pandoc magic

author: ben

pandoc is an incredibly powerful tool for creating and converting documents.

i recently started using it for all kinds of different things, including:

- the tilde.club wiki
- my personal page on ~club (source here)
- this zine itself (see zine makefile)

let's look at some of the tricks and tips that i've learned from these.

format markdown files

this seems like it might not work, but `pandoc` will format your markdown files for you if you convert the source file from markdown *to* markdown again. for example, if you wanted to tidy up your `README.md`, you could run:

```
pandoc -f markdown -t markdown -o README.md README.md
```

note that if you want to preserve any yaml frontmatter blocks, you'll need to change the from and to types to `markdown+yaml_metadata_block`. another thing to note is that you can preserve github-formatted markdown by using `gfm`

instead of `markdown`. additionally, if you prefer to use atx-style headers, don't forget to add the `--atx-headers` switch.

a complete example:

```
pandoc \
    -f markdown+yaml_metadata_block \
    -t markdown+yaml_metadata_block \
    --atx-headers \
    -o README.md \
    README.md
```

templates

pandoc ships with a full set of templates that are used to control how things are displayed when writing to certain formats.

let's look at the templates for html and latex. to get the default template for a given format, use `-D`. the template is written to stdout, so let's save it to a file.

```
pandoc -D html > html.template
```

open up this template in your editor and you'll see what's used when you convert things to html.

important note: these templates are only used when generating standalone documents (the `-s` or `--standalone` flag).

note that the template uses lots of variable substitutions. we can set values for these in yaml frontmatter or in the command invocation.

to make some basic customizations, we can fill in the metadata values that will be automatically replaced in the document when we build.

however, some circumstances require a custom layout or additional changes to meet the requirements.

have a look at the wiki.tmpl used on the ~club wiki. some of the notable changes include:

- table of contents title
- hardcoded stylesheet
- link to author's user page

the stylesheet change would be possible on the command line by using the `-c` or `--css` option, but the other changes require a custom template.

try it out on your own project!

lua filters

sometimes the rendered page still isn't what you're looking for. maybe you need some additional tweaks (css classes, extra html items, etc).

pandoc has supported filters for a long time in the form of json passed around on pipes that allows you to modify the internal data structures before they're written to the output format.

the main downside of using these filters is that it introduces another layer of dependencies (namely the language that the filter's written in and also the library for that language to interact with pandoc's json).

as of pandoc 2.0, a lua interpreter with a class library for creating filters is built in to the pandoc executable. let's go over some basic examples.

permalinks for headers a common feature on most html created via markdown is a small link displayed next to the header so that you can deep-link directly to that section of the page. this is frequently seen on github and other documentation sites.

here's a very simple way to create the link:

```
function Header(elem)
    table.insert(elem.content, pandoc.Space())
    table.insert(elem.content, pandoc.Link("§", "#" .. elem.identifier))
    return elem
end
```

save this to a file and call it in your pandoc conversion with the `--lua-filter` option.

change header levels this is an example from this zine. i wanted to decrease the size of headers below h1 level without changing the external css. here's how i did it.

```
function Header(elem)
    if elem.level > 1 then
        elem.level = elem.level + 1
    end
    return elem
end
```

this changes all header elements to be one level smaller.

there are additional examples on pandoc.org if you'd like to see more.