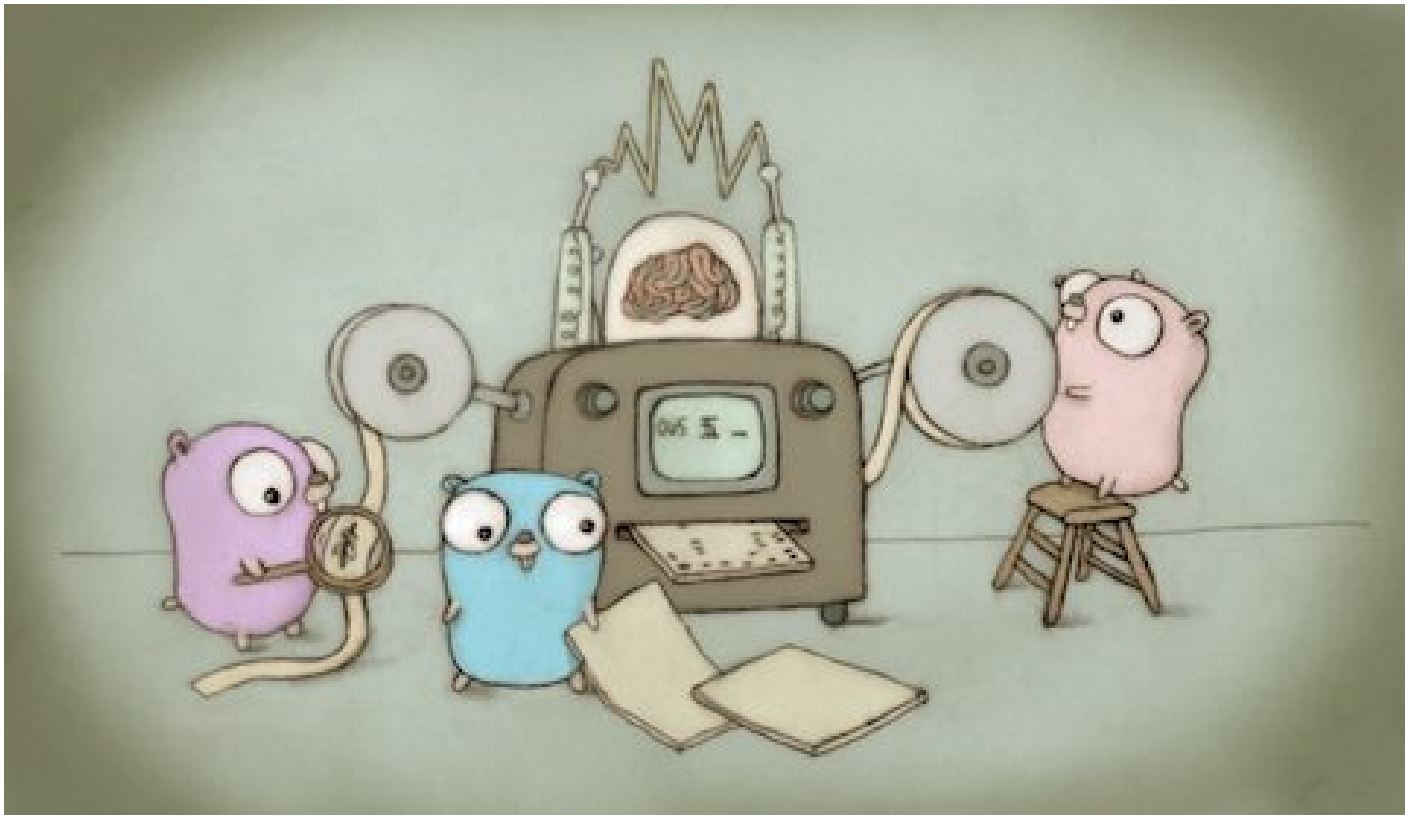
23rd Aug 2017 / Sahil Muthoo

Makefiles for Golang



Awesome Gophers (The Go gopher was designed by Renee French (http://reneefrench.blogspot.com/). The design is licensed under the Creative Commons 3.0 Attributions license.)

Go's toolchain is awesome. `make` makes the toolchain awesome-er. Go's fast compile times and internal change-tracking eliminate the need for esoteric Makefiles. This is great since we can write simple Makefiles to get the job done in style.

Quick primer on make

Introduction

`make` is a build tool from the 70s. Builds are described in files known as Makefiles. Your project's Makefile will typically live in its root directory.

Makefiles are composed of rules that look like this:

```
target: prerequisites
    recipe
```

The target is a name of a file. Prerequisites are rules to run *before* the target/file is built. A recipe describes how to build the target/file.

The combination of target, its prerequisites and recipe is known as a `rule`. Rules describe how `make` should rebuild something, should it need rebuilding. `make` detects changes to

targets by tracking the last modified time of its prerequisites. If a prerequisite has changed since the target was last built, the target will be rebuilt. Otherwise `make` takes no action and declares the target up-to-date.

One piece of esoterica to keep in mind. `make` insists that Makefiles must be indented with tabs. You'll see weird errors if your Makefile is indented with spaces or heaven's forbid, a mix of tabs and spaces. It's possible to change the default indentation character in some versions of `make` but it's best to stick with tabs.

Let's look at a simple example:

```
abc: xyz
    echo "abc" > abc

xyz:
    echo "xyz" > xyz
```

This Makefile declares 2 rules. The target `abc` depends on `xyz` and both of them have recipes that generate their respective targets. Let's run make for the first time:

```
$ make
echo "xyz" > xyz
echo "abc" > abc
```

Notice how I didn't specify which target to build. `make` chose the first target as the default. In this case `abc`. We see that `make` runs the prerequisite `xyz` before running `abc`.

Let's try running make again:

```
$ make
make: 'abc' is up to date.
```

`make` declares that there's nothing to do. The files `xyz` and `abc` already exist. All dependencies have been satisfied and there is no work to be done. `make's` dependency tracking is a powerful feature which we'll exploit in a bit.

Before that, we must understand one last thing. Variables.

make Variables

Variables in `make` are quite simple. A variable is declared using `VAR := value`. `VAR` is referenced via `$(VAR)`. All three components of a rule, the target, prerequisites and recipe may contain variables. Let's look at another simple example:

```
FILE := abc

$(FILE): xyz
    echo $(FILE) > "something"

xyz:
    echo "xyz" > xyz
```

Try running this Makefile and see what happens.There's more to variables but our primer ends here. This is all we need to know to get started with writing our first Makefile for Go. We'll pick up a few more `make` concepts as we go along.

I don't need no stinkin' Makefile

`make` shines at optimizing builds i.e. only rebuilding stale stuff. We must ask how useful this is in the land of Go. Full rebuilds finish in the order of seconds. `go install` already checks if the binary is up-to-date before installation. `go test` is all you need to kick-off tests that run at face melting speed. It's natural to ask, "Why do I need a Makefile?".

`make` is useful because it's executable documentation. It describes how to build your project, what kind of tests can be run and the external tools your project depends on and so on.

Everything else is icing on the cake. Very tasty icing. You get dependency tracking for external tools, parallel builds when you need them and dead simple CI scripts. You can go ahead and throw away all your shell scripts. That's a bonus in my book 😀

Baby's first Go Makefile

If you're still reading, I've managed to pique your interest. Hurray! The plan is that we'll start simple and slowly build up our Makefile with useful stuff. Let's get to it!

Running tests

Let's start by writing a rule to run our tests:

```
PKGS := $(shell go list ./... | grep -v /vendor)

.PHONY: test
test:
    go test $(PKGS)
```

We have one rule called `test`. The `.PHONY: test` above our rule indicates that `test` is a phony target. Recall how we said that targets are files. It's time to revisit that definition. All non-phony targets are files. Phony targets indicate that no such file will exist. Make should

run the rule every time it's invoked and skip looking for the presence of a matching file. This makes sense for a test rule where you just want to run tests. Not generate files.

We also declare a variable called `PKGS`. It contains the list of packages under test. The shell code inside `$( ... )` is evaluated once and it's output (STDOUT) is saved in `PKGS`. It's the usual list of packages without packages in vendor.

To run this rule just type `make`. We should now have running tests. woot!

Linting

Alright, now that we have running tests it's time to throw in a linter. Let's use gometalinter. It's an excellent *meta*-linter that runs a bunch of awesome linters concurrently.

Our first pass at writing a lint rule may look like this:

```
PKGS := $(shell go list ./... | grep -v /vendor)

.PHONY: test
test:
    go test $(PKGS)

.PHONY: lint
lint:
    gometalinter ./... --vendor
```

Our `lint` rule runs `gometalinter` which takes care of skipping packages in vendor via the `--vendor` flag. Once again, `lint` is a phony target because it does not generate a file. This time running lint requires us to specify the target. We can do so by running `make lint`.

Linting before testing

This is looking great already. But perhaps we'd like to lint before we test? Let's recall that we can specify prerequisites in rules. Our test rule can be adapted to look like this:

```
.PHONY: test
test: lint
    go test $(PKGS)
```

Now every time we run `test`, `lint` is run automatically. If linting fails, the test rule fails. We no longer have to remember to `lint` and then `test`. Running `make` does everything for us. Executable documentation for the win!

Dependency management for external tools

`gometalinter` is an external tool. We haven't described how to fetch it in our Makefile. If someone checks out our project and tries to run `make` it'll fail with an ugly error. Unless of course they have `gometalinter` installed.

Wouldn't it be nice if `make` could install `gometalinter` for us? Well I'm glad you asked. Let's try doing just that:

```
BIN_DIR := $(GOPATH)/bin
GOMETALINTER := $(BIN_DIR)/gometalinter

$(GOMETALINTER):
    go get -u github.com/alecthomas/gometalinter
    gometalinter --install &> /dev/null
```

The above rule describes the process of installing `gometalinter`. The `$(GOMETALINTER)` target is dynamically generated from a variable that points to the install path of `gometalinter`. Finally, we have a real target. This one's not phony 😀. Notice how we refer to the `GOPATH` in `BIN_DIR`. `GOPATH` is an environment variable that's usable in Makefiles similar to other variables.

This generated rule isn't useful on it's own. It shines when it's used as a prerequisite. Let's go ahead and add the generated rule as a prerequisite of the `lint` rule:

```
.PHONY: lint
lint: $(GOMETALINTER)
  gometalinter ./... --vendor
```

That's it. Every time lint executes, `$(GOMETALINTER)` will execute. Since `$(GOMETALINTER)` is not a phony target, it'll be rebuilt if and only if the `gometalinter` binary is not present at `$GOPATH/bin`. If the binary is present, the target is up-to-date and make will skip the rule.

Let's now look at our complete Makefile as it stands:

```
PKGS := $(shell go list ./... | grep -v /vendor)

.PHONY: test
test: lint
    go test $(PKGS)

BIN_DIR := $(GOPATH)/bin
GOMETALINTER := $(BIN_DIR)/gometalinter

$(GOMETALINTER):
    go get -u github.com/alecthomas/gometalinter
    gometalinter --install &> /dev/null
```

```
.PHONY: lint
lint: $(GOMETALINTER)
    gometalinter ./... --vendor
```

`test` depends on `lint` which depends on `$(GOMETALINTER)`. `test → lint → $(GOMETALINTER)`. That's how Make will build it's dependency graph pruning up-to-date targets along the way.

Let's examine the output of running `make` for the first time:

```
$ make
go get -u github.com/alecthomas/gometalinter
gometalinter --install &> /dev/null
gometalinter ./... --vendor
go test github.com/sahilm/yamldiff
ok      github.com/sahilm/yamldiff  0.351s
```

`make` installed `gometalinter`, then ran `lint` and finally it ran our tests. The next time we run `make`, it will skip the installation of `gometalinter` since it's already installed. Let's validate that assumption:

```
$ make
gometalinter ./... --vendor
go test github.com/sahilm/yamldiff
ok      github.com/sahilm/yamldiff  0.120s
```

Yup, Make sure does the trick! Similarly we can manage the installation of other external tools such as dep and goimports.

A nice exercise to try would be integrating `dep ensure` with the build process so that it's only run if the `vendor` directory does not exist.

Building releases in parallel

It's time to enjoy the icing. Let's use `make` to build production releases in parallel.

Easy first pass:

```
BINARY := mytool

.PHONY: windows
windows:
    mkdir -p release
    GOOS=windows GOARCH=amd64 go build -o release/$(BINARY)-v1.0.0-windows-amd64
```

```
.PHONY: linux
linux:
    mkdir -p release
    GOOS=linux GOARCH=amd64 go build -o release/$(BINARY)-v1.0.0-linux-amd64

.PHONY: darwin
darwin:
    mkdir -p release
    GOOS=darwin GOARCH=amd64 go build -o release/$(BINARY)-v1.0.0-darwin-amd64

.PHONY: release
release: windows linux darwin
```

We've defined 3 rules to build our binary on Windows, Linux and Darwin. Running them in parallel is easy with the `-j` flag to `make`. Let's run 3 jobs:

```
$ make release -j3
mkdir -p release
mkdir -p release
mkdir -p release
GOOS=linux GOARCH=amd64 go build -o release/mytool-v1.0.0-linux-amd64
GOOS=windows GOARCH=amd64 go build -o release/mytool-v1.0.0-windows-amd64
GOOS=darwin GOARCH=amd64 go build -o release/mytool-v1.0.0-darwin-amd64
```

We see that all three targets are built together. There are a couple issues we must address, the rules are near replicas of each other and the version string is hard coded. Let's tackle them one by one.

Reducing duplication

We can reduce duplication with `make`'s text functions and automatic variables.

We'll use the text function `word` and automatic variable `$@`. Let's see it in action:

```
BINARY := mytool

PLATFORMS := windows linux darwin
os = $(word 1, $@)

.PHONY: $(PLATFORMS)
$(PLATFORMS):
    mkdir -p release
    GOOS=$(os) GOARCH=amd64 go build -o release/$(BINARY)-v1.0.0-$(os)-amd64

.PHONY: release
release: windows linux darwin
```

The `$(word, 1, var)` function extracts the 1st word from `$@` which points to the name of the current target. Notice how the `os` variable is declared with a `=` and not `:=`. This means that `os` will be evaluated every time it's used as opposed to being evaluated once at declaration time.

Under the hood, `$(PLATFORMS)` expands to the targets similar to the targets we declared earlier. Within each's recipe we use the `word` function to extract the only word (windows, linux or darwin) from the name of the target. We can run `make release -j3` to confirm it's behaviour.

Injecting values at build time

Our release target is looking good except for the hard coded version number. It'll be nice if we can declare a default version string and override it from the outside at build time. To achieve this we can use variables of the form `VAR ?= value` where `VAR` defaults to `value` but can be overridden.

Let's modify our rule:

```
BINARY := mytool
VERSION ?= vlatest
PLATFORMS := windows linux darwin
os = $(word 1, $@)

.PHONY: $(PLATFORMS)
$(PLATFORMS):
    mkdir -p release
    GOOS=$(os) GOARCH=amd64 go build -o release/$(BINARY)-$(VERSION)-$(os)-amd64

.PHONY: release
release: windows linux darwin
```

We declare the `VERSION` variable using a default of `vlatest`. This can be overridden like so:

```
$ make VERSION=v2.0.0 release -j3
mkdir -p release
mkdir -p release
mkdir -p release
GOOS=linux GOARCH=amd64 go build -o release/mytool-v2.0.0-linux-amd64
GOOS=windows GOARCH=amd64 go build -o release/mytool-v2.0.0-windows-amd64
GOOS=darwin GOARCH=amd64 go build -o release/mytool-v2.0.0-darwin-amd64
```

Notice how the version changed to `v2.0.0`. Now our release rule looks good to go!

Our creation

Let's look at the Makefile we made, shall we:

```
PKGS := $(shell go list ./... | grep -v /vendor)

.PHONY: test
test: lint
    go test $(PKGS)

BIN_DIR := $(GOPATH)/bin
GOMETALINTER := $(BIN_DIR)/gometalinter

$(GOMETALINTER):
    go get -u github.com/alecthomas/gometalinter
    gometalinter --install &> /dev/null

.PHONY: lint
lint: $(GOMETALINTER)
    gometalinter ./... --vendor

BINARY := mytool
VERSION ?= vlatest
PLATFORMS := windows linux darwin
os = $(word 1, $@)

.PHONY: $(PLATFORMS)
$(PLATFORMS):
    mkdir -p release
    GOOS=$(os) GOARCH=amd64 go build -o release/$(BINARY)-$(VERSION)-$(os)-amd64

.PHONY: release
release: windows linux darwin
```

Our Makefile is capable of linting, testing and releasing our code. It installs any external tools as it goes along. It can run in parallel. I would say that's quite a handsome Makefile we have here.

Given such a Makefile, we can create CI builds in a pinch. Here's an example CI build for the popular Travis CI.

```
language: go
go:
  - 1.8
script:
  - make
before_deploy:
  - make VERSION=${TRAVIS_TAG} release -j2
deploy:
  provider: releases
```

```
  api_key:
    secure:
  file: release/*
  file_glob: true
  skip_cleanup: true
  on:
    tags: true
```

This will test every commit and make a GitHub release on every tag. Easy peasy lemon squeezy.

Fin

I believe Make is a fine build tool for Go. I hope this article encourages you to try Make on your own projects. There's no risk. You can always delete the Makefile if you don't like it 😆.

I'd love feedback from you. Please tweet at me or send me an email. Links down below. Cheers!