SGBLE

Zeste de savoir

Je découvre Task et le Taskfile.yaml

6 juillet 2022

Table des matières

Introduction

Simple à mettre en place et généralement disponible immédiatement, le vénérable Make (et son `Makefile`) se propose en général comme première solution quand le besoin d'un *build tool* se précise. Écrire un `Makefile` est chose plutôt aisée au début. Une cible simple peut mâcher le travail pour exécuter une ou des commandes plus pénibles à retaper que `make cible`.

L'outil n'est cependant pas pleinement satisfaisant au fur et à mesure que les règles et leur dépendances se complexifient. J'ai donc décidé de tester un nouvel outil, *Task* , qui se veut être une alternative moderne à Make et ses `Makefile`.

Je présente dans ce billet l'usage que j'en fais dans le cadre d'un projet nommé `deluge`, un système de *message queuing* écrit en Go. Cette présentation donne un aperçu de l'outil.

En conclusion, je me livre à quelques réflexions quant à Task vis-a-vis des systèmes plus sophistiqués encore.

Bref rappel sur les règles d'un `Makefile` et son vocabulaire.
Un `Makefile` se présente comme un ensemble de règles (*rules*). Une règle permet de déterminer comment créer une cible (*target*, généralement un fichier à produire) et quand (re)créer cette cible selon l'existence et la date des prérequis (des fichiers). Un prérequis peut être lui-même cible d'une autre règle, ce qui permet alors de combiner astucieusement plusieurs règles, en faisant ainsi sortir un arbre de dépendances déterminant un ordre cohérent de recettes (*recipe*) à exécuter pour arriver à une cible à partir des autres.[1]

```
# Règle
cible: prérequis1 prérequis2
        recette pour parvenir à cible
```

1. Une introduction brève mais plus complète est disponible dans le manuel de GNU Make: *What a Rule Looks Like* .

1. Task, Taskfile.yaml

1.0.1. `Taskfile.yaml`

Task travaille avec des fichiers YAML. Personnellement, je ne suis pas le plus enthousiaste en éditant des fichiers YAML. Cependant, la structure d'un `Taskfile.yaml` est peu imbriquée et reste raisonnable. Voici le premier exemple d'une tâche (*task*) appelée `build` qui compile le projet avec Go:

```yaml
version: '3'

tasks:
  build:
    cmds:
      - mkdir -p bin
      - go build -o bin/deluge -v .  # compilation et création du
          binaire bin/deluge
```

La commande suivante permet de lancer cette tâche: `task build`.

1.0.2. Dépendances et fichiers source

Jusque-là, la différence avec Make ne semble pas notable au delà de la syntaxe. Mais il est à observer dès à présent que Task ne raisonne pas en termes de règles comme Make, mais en termes de tâches, lesquelles sont plus expressives. À la place des cibles, une tâche indique précisément ce qu'on peut en attendre comme résultat après l'exécution. En l'occurrence, un nouveau fichier `bin/deluge` est généré par le compilateur de Go, cela est indiqué avec `generates`:

```yaml
tasks:
  build:
    cmds:
      - mkdir -p bin
      - go build -o bin/deluge -v .
    generates:
      - bin/deluge
```

Il est aussi possible d'indiquer les fichiers nécessaires à l'exécution de la tâche, ils sont indiqués avec `sources`:

```yaml
tasks:
  build:
    cmds:
      - mkdir -p bin
```

```
      - go build -o bin/deluge -v .
    generates:
      - bin/deluge
    sources:
      - '*.go'
      - '*/*.go'   # marche aussi avec juste '**/*.go'
```

Le projet `deluge` propose une interface gRPC . Cela implique de travailler avec des fichiers Protobuf (extension `.proto`) à partir desquels du code Go gérant la (dé)sérialisation est généré par l'outil `protoc` prévu pour cela.

Les choses deviennent intéressantes dès lors qu'il devient nécessaire de générer les fichiers Go à partir du fichier Prorobuf avant de compiler le projet dans son ensemble.

Une nouvelle tâche `grpc` est créée dans `Taskfile.yaml`, avec les déclarations de `sources` et `generates` adéquates:

```
tasks:
  build:
    cmds:
      - mkdir -p bin
      - go build -o bin/deluge -v .
    generates:
      - bin/deluge
    sources:
      - '*.go'
      - '*/*.go'   # marche aussi avec '**/*.go' seul

  grpc:
    cmds:
      - protoc --go_out=. --go-grpc_out=. service.proto
    sources:
      - service.proto
    generates:
      - grpcsvc/service.pb.go
      - grpcsvc/service_grpc.pb.go
```

Enfin, la déclaration de dépendance se fait avec une clé `deps` qu'il convient ici d'ajouter à la tâche `build` puisqu'elle a besoin d'être lancée après la tâche `grpc` qui invoquera `protoc`:

```
 build:
    deps: [grpc]  # dépendance ajoutée
    cmds:
      - mkdir -p bin
      - go build -o bin/deluge -v .
    generates:
```

```
      - bin/deluge
    sources:
      - '*.go'
      - '*/*.go'
```

Pour chaque tâche déclarée, les informations renseignées dans `generates` et `sources` permettent à Task d'effectuer les actions uniquement si nécessaire dans l'accomplissement d'une tâche[1]. Par exemple, la modification du fichier `service.proto` (déclarée comme `source` de la tâche `grpc`) entraînera en chaîne l'appel `protoc --go_out=. --go-grpc_out=. service.proto`, modifiant par là-même les fichiers Go, entraînant subséquemment l'appel `go build`. Task ne relance que les tâches nécessaires à l'accomplissement du but (en parallèle si la chaîne de dépendance le permet), comme pourrait le faire Make.

Il est intéressant de comparer avec les règles du `Makefile` qui accompagnait le projet lors de ses premières lignes:

```
grpcsvc/service.pb.go: service.proto
	protoc --go_out=. --go-grpc_out=. service.proto

bin/deluge: $(wildcard *.go) $(wildcard core/*.go)
    grpcsvc/service.pb.go grpcsvc/service_grpc.pb.go
	go build -o bin/deluge -v .
```

Avec Make, les prérequis sont généralement des fichiers. L'approche de Task me semble bien plus intuitive: une tâche doit dépendre d'autres tâches dans le sens où elles doivent s'exécuter avant, mais Task les distingue bien des fichiers source, lesquels permettent de déterminer si l'exécution des commandes doit avoir lieu.

Malgré le formatage en YAML, la version `Taskfile.yaml` m'est plus agréable et facile à la lecture. La verbosité est ici une vertu et elle rend le fichier plus explicite.

1.0.3. Préconditions

Task présente la notion de précondition dont la satisfaisabilité est nécessaire à l'exécution de la tâche, sous peine de la voir échouer. La tâche `grpc` nécessite quelques dépendances Go qu'il convient d'installer pour le bon déroulement des opérations[2].

```
tasks:
  grpc:
    cmds:
      - protoc --go_out=. --go-grpc_out=. service.proto
    sources:
      - service.proto
    generates:
      - grpcsvc/service.pb.go
```

```
      - grpcsvc/service_grpc.pb.go
    preconditions:
      - sh: test -f $GOPATH/bin/protoc-gen-go
        msg: "Please try this command: go install
            google.golang.org/protobuf/cmd/protoc-gen-go"
      - sh: test -f $GOPATH/bin/protoc-gen-go-grpc
        msg: "Please try this command: go install
            google.golang.org/grpc/cmd/protoc-gen-go-grpc"
```

Les préconditions testent si les deux dépendances sont présentes (`test -f` teste l'existence d'un fichier). La clé `msg` permet d'afficher un message à l'utilisateur en cas d'échec de précondition:

```
% task build
task: Please try this command: go install
    google.golang.org/protobuf/cmd/protoc-gen-go
task: precondition not met
```

1.0.4. Variables d'environnement

Les variables d'environnement sont fixées avec `env`:

```
version: '3'

env:
  CGO_ENABLED: '0'
  GOAMD64: v3

tasks:
   ...
   ...
   ...
```

Elles s'appliqueront alors dans l'ensemble des tâches exécutées. Il est cependant possible de les overrider uniquement pour une tâche donnée en reprécisant un `env` au sein d'une tâche:

```
version: '3'

env:
  CGO_ENABLED: '0'  # Initialement à 0
  GOAMD64: v3

tasks:
```

```
  test:
    deps: [grpc]
    env:
      CGO_ENABLED: '1'  # Fixé à 1 tel que requis pour utiliser le
          flag -race
    cmds:
      - echo Ici CGO_ENABLED = $CGO_ENABLED
      - go test -race ./...
    sources:
      - '*.go'
      - '*/*.go'
```

Cela donne bien le résultat voulu:

```
% task test
task: Task "grpc" is up to date
task: [test] echo Ici CGO_ENABLED = $CGO_ENABLED
Ici CGO_ENABLED = 1
task: [test] go test -race ./...
```

1.0.5. Autre fonctionnalités

Task propose d'autres fonctionnalités qui ne sont pas présentées ici, certaines sont plutôt originales comme `defer` (inspiré du `defer` de Go). La liste complète est ici .

Je conclus la présentation de Task en mettant en évidence un aspect qui n'a pas été abordé. Task ne se présente pas comme uniquement un *build tool*:

> Task is a **task runner / build tool** that aims to be simpler and easier to use than, for example, GNU Make.

D'autres fonctionnalités sortent du cadre du *build tool* strict pour s'apparenter en effet à de la gestion de tâches. Elles ne sont pas présentées dans le présent exposé car je n'ai pas encore eu l'occasion d'utiliser Task ainsi. Peut-être paraîtra-t-il un prochain billet là-dessus 🍊

Conclusion

Il persiste toujours un aspect fondamental et commun à Make et à Task en cela qu'ils laissent au développeur tout le soin d'écrire les recettes ou les commandes. Le développeur est donc responsable de la cohérence globale du `Makefile` ou du `Taskfile.yaml`, ainsi que de son bon

1. Task maintient, pour chaque tâche, un hash calculé à partir des sources pour déterminer si l'exécution de ses commandes est nécessaire.
2. À savoir: `google.golang.org/protobuf/cmd/protoc-gen-go` et `google.golang.org/grpc/cmd/protoc-gen-go-grpc`, cf. *Quick start | Go | gRPC* .

déroulement par la déclaration correcte des prérequis ou des dépendances, notamment dans le cas d'une gestion multi-plateformes.[1]

À cet égard, Task est-il réellement moderne au vu des alternatives plus sophistiquées qui existent? Depuis peu, un autre *build tool* prenant une approche opposée suscite mon intérêt: Bazel . La promesse d'un tel outil s'entend bien: le programmeur n'a plus, en principe, à écrire à la main le détail des commandes concrètes décrivant les étapes de *build*. Le niveau d'abstraction est placé plus haut et ces détails sont relégués au système de *build*.

L'une des premières phrases de la documentation de Bazel semble plutôt opiniâtre:

> Bazel uses an abstract, human-readable language to describe the build properties of your project at a high semantical level. Unlike other tools, Bazel operates on the concepts of libraries, binaries, scripts, and data sets, shielding you from the complexity of writing individual calls to tools such as compilers and linkers.
>
> *Intro to Bazel*

Bazel semble disposer d'un bon support pour Go (et pour Protobuf en l'occurrence). Il a donc été envisagé. Cependant, le niveau de complexité va de paire avec le niveau de sophistication. Je n'exclus aucunement avoir recours à un tel système dans le futur, d'autant plus que le projet a des chances de devenir polyglotte.[2]

En attendant, Task me semble être un bon compromis: un Make revisité, à mi-chemin entre la tradition et la modernité.

1. Et le cas de `deluge` est encore simple: l'implémentation de référence du langage Go et le support exclusif de Linux limite la disparité des contextes de *build* possibles, permettant ainsi d'alléger les commandes.
2. Si vous avez des retours d'expérience sur de tels systèmes (Bazel ou concurrent qui se place au même niveau), je suis tout ouïe!