Makefile, organisation d'un projet

Michel Billaud

10 octobre 2018

Résumé

Quand un projet contient de nombreux fichiers sources, il est intéressant de produire les fichiers intermédiaires et les exécutables dans des répertoires séparés des sources.

Table des matières

Résumé. Quand un projet contient de nombreux fichiers sources, il est intéressant de produire les fichiers intermédiaires et les exécutables dans des répertoires séparés des sources.

Dernière mise à jour : 15 novembre 2021.

1 Problématique

1.1 Un Exemple de projet

On construit un projet avec deux exécutables à produire :

- `main_f`, qui appelle une fonction `foo()`
- `main_fb`, qui appelle `foo()` et `bar()`.

Source de `main_fb.c` :

```c
#include <stdio.h>

#include "foo.h"
#include "bar.h"

int main() {
  foo();
  bar();
  return 0;
}
```

Chaque fonction est compilée séparément. Nous avons donc les fichiers sources :

- `main_f.c` et `main_fb.c` contenant les fonctions `main()` des deux exécutables,
- les fichiers d'entêtes `foo.h` et `bar.h` contenant les prototypes,
- les fichiers d'entêtes `foo.c` et `bar.c` avec le code des fonctions.

1.2 Un makefile "basique"

En utilisant les règles par défaut, les dépendances implicites, et la génération automatique des dépendances, on peut écrire un `Makefile` simple :

```
CFLAGS = -MMD -MP

EXECS = main_f main_fb
all: $(EXECS)

main_f:  main_f.o   foo.o
main_fb: main_fb.o  foo.o bar.o

-include $(wildcard *.d)

clean:
    $(RM) *~ *.o *.d

mrproper: clean
    $(RM) $(EXECS)
```

dont l'exécution conduit au résultat voulu

```
$ make -f Makefile.simple
cc -MMD -MP   -c -o main_f.o main_f.c
cc -MMD -MP   -c -o foo.o foo.c
cc   main_f.o foo.o   -o main_f
cc -MMD -MP   -c -o main_fb.o main_fb.c
cc -MMD -MP   -c -o bar.o bar.c
cc   main_fb.o foo.o bar.o   -o main_fb
```

1.3 Critique

Cette solution a l'inconvénient d'envahir le répertoire avec des fichiers de travail :

```
$ ls
bar.c  bar.o  foo.h   main_fb    main_fb.o  main_f.o
bar.d  foo.c  foo.o   main_fb.c  main_f.c   Makefile
bar.h  foo.d  main_f  main_fb.d  main_f.d
```

C'est le problème auquel on essaie de remédier.

2 Utilisation de répertoires séparés

2.1 Objectif

L'objectif est de ne pas polluer le répertoire des sources. Pour cela on utilisera deux sous-répertoires

- `build` pour les fichiers intermédiaires (objets et dépendances) produits pendant le développement du projet,
- `dist` pour les exécutables "à livrer".

2.2 Construction raisonnée du `Makefile`

2.2.1 Les variables

Le `Makefile` commence par quelques définitions

- les **noms des sous-répertoires** :

```
BUILD_DIR = build
DIST_DIR  = dist
```

- la liste des noms des exécutables du projet, à produire dans le `BUILD_DIR`

```
EXECS = main_f main_fb
```

- pour chaque exécutable du projet, la liste des modules objets nécessaires

```
OBJS_MAIN_F =  main_f.o  foo.o
OBJS_MAIN_FB = main_fb.o foo.o bar.o
```

- Et bien sûr les options de compilation pour la génération automatique des dépendances

```
CFLAGS = -MMD -MP
```

2.2.2 La production des exécutables

- La règle suivante, qui est en principe la première, demande la **production de tous les exécutables** dans le répertoire `dist/`

```
all: $(addprefix $(DIST_DIR)/,$(EXECS))
```

- Viennent ensuite les dépendances pour la composition des fichiers exécutables. Les noms des fichiers objets sont donnés dans une variable, on les préfixe par le nom du du sous-répertoire où il faut les placer (`build/`) :

```
$(DIST_DIR)/main_f:  $(addprefix $(BUILD_DIR)/,$(OBJS_MAIN_F))
$(DIST_DIR)/main_fb: $(addprefix $(BUILD_DIR)/,$(OBJS_MAIN_FB))
```

- **La production des exécutables** dans `dist/` est décrite par une règle "générique"qui est dérivée de la macro classique d'édition des liens, en créant au besoin le répertoire `dist/`

```
$(DIST_DIR)/%:
    @mkdir -p $(DIST_DIR)
    $(LINK.c) -o $@ $^
```

- Une règle générique similaire décrit les **compilations** :

```
$(BUILD_DIR)/%.o: %.c
    @mkdir -p $(BUILD_DIR)
    $(COMPILE.c) -o $@ $^
```

2.2.3 Dépendances et utilitaires

- enfin, on trouve l'**inclusion des dépendances**

```
-include $(BUILD_DIR)/*.d
```

- ainsi que les règles utilitaires, dont les commandes de nettoyage

```
clean:
    $(RM) *~
    $(RM) -r (BUILD_DIR)

mrproper: clean
    $(RM) -r  $(DIST_DIR)
```

2.3 Exécution du Makefile

- L'exécution de la commande `make` produit bien les fichiers voulus

```
$ make
cc -MMD -MP   -c -o build/main_f.o main_f.c
cc -MMD -MP   -c -o build/foo.o foo.c
mkdir -p dist
cc -MMD -MP    -o dist/main_f build/main_f.o build/foo.o
cc -MMD -MP   -c -o build/main_fb.o main_fb.c
cc -MMD -MP   -c -o build/bar.o bar.c
mkdir -p dist
cc -MMD -MP    -o dist/main_fb build/main_fb.o build/foo.o build/bar.o
```

- Vue d'ensemble des fichiers :

```
$ LANG= tree -c
.
|-- foo.c
|-- foo.h
|-- bar.h
|-- main_f.c
|-- main_fb.c
|-- bar.c
|-- Makefile
```

```
|-- build
|   |-- bar.d
|   |-- bar.o
|   |-- foo.d
|   |-- foo.o
|   |-- main_f.d
|   |-- main_f.o
|   |-- main_fb.d
|   `-- main_fb.o
`-- dist
    |-- main_f
    `-- main_fb
```

- Les fichiers de dépendance sont corrects. Ils contiennent des “phony targets” pour les fichiers d'entete, à cause de l'option `-MP` dans les `CFLAGS` :

Fichier `build/main_fb.d` :

```
build/main_fb.o: main_fb.c foo.h bar.h

foo.h:

bar.h:
```

Fichier `build/foo.d`

```
build/foo.o: foo.c foo.h

foo.h:
```

3 Conclusion

Le `Makefile` ainsi construit ne contient que quelques lignes spécifiques à ce projet :

```
EXECS = main_f main_fb
...
main_f  : main_f.o  foo.o
main_fb : main_fb.o foo.o bar.o
...
$(DIST_DIR)/main_f  : $(addprefix $(BUILD_DIR)/,$(OBJS_MAIN_F))
$(DIST_DIR)/main_fb : $(addprefix $(BUILD_DIR)/,$(OBJS_MAIN_FB))
```

Pour des projets qui ont une structure similaire, c'est-à-dire

- plusieurs exécutables à produire,
- tous les sources dans le même répertoire,

il est facile de l'adapter.