Aide mémoire – Makefile

- Unix/Linux
- Commande : make
- FICHIERS : makefile ou Makefile (dans cet ordre)
- www.gnu.org/software/make/manual : LA référence

Table des matières

I À quoi sert un Makefile et comment le faire marcher ?

A) Règles Explicites

```
cible(s) : dépendances          #-----  dépendances séparées par un espace
        commande1               #-----  une ‘‘tabulation’’ DOIT précéder
        commande2               #-----  les commandes
        ...
```

Dans ce cas, les commandes `commande1` et `commande2` seront exécutées si les dépendances sont plus récentes que la cible. Il est aussi possible d'écrire la règle sur une seule ligne de manière plus concise, à l'aide de ';' :

```
toto : toto.C ; g++ toto.C -o toto
```

B) Exécution

- `make` : fabrique la première cible trouvée dans le fichier Makefile
- `make xxx` : fabrique la cible `xxx` du Makefile

Les dépendances sont fabriquées récursivement si nécessaire :

```
clean :
        rm *.o

sourceclean : clean
        rm *.C
```

Ce Makefile produira, à l'exécution de `make sourceclean`, les commandes : `rm *.o`, suivie de `rm *.C`.

C) Commandes

- Sont affichées à l'écran, exemple :

```
rm *.o
rm *.C
```

⇒ Retirées de l'affichage avec '`@`', exemple : `clean: ; @ rm.o`

- Exécutées l'une après l'autre jusqu'à rencontrer une erreur.

⇒ Les erreurs peuvent être autoriseés avec '`-`', exemple : `clean: ; - rm.o`

- '`@`' et '`-`' peuvent être combinés.

D) Qu'y a-t-il dans un Makefile ?

- Règle explicite (cf. A)).
- `include fichier` (ou `_include`, si l'on ne veut pas d'erreur au cas où `fichier` n'existe pas).
- Définitions de variables.
- Parties conditonnelles.
- Commentaires : tout ce qui suit un '`#`'.

Que n'y a-t-il pas ? Règles implicites (cf III)

II Variables

A) Style SHell : = + $(...)

```
OBJ = program.c

program : $(OBJ)
        gcc -o program $(OBJ)
```

B) Deux types d'affectations : deux passes de préprocesseur

= : Substitution, pendant la deuxième passe

```
SUFF = C
prog.o : prog.$(SUFF) ; ...
```

```
foo = $(bar)
bar = $(ugh)
ugh = Hug?
all : ; echo $(foo)    #----- affiche ``Hug?''
```

```
FLAG = $(FLAG) -g -O   #----- Problème : récursivité ? boucle infinie !
```

:= : Expansion immédiate

```
x := 1
y := $(x)
x := 2                  #----- x vaut 2 et y vaut 1
```

À comparer avec :

```
x = 1
y = $(x)
x = 2                   #----- x vaut 2 et y vaut 2 aussi, maintenant
```

C) += : Ajout

Il existe une solution simple pour ajouter des éléments à une variable en évitant la récursivité et les boucles infinies, grâce au `+=` :

```
OBJ += prog3.o $(TOTO)
```

D) % : Altération et expressions régulières

```
OBJ := prog1.o prog2.o
SRC :=  $(OBJ:%.o=%.C)              #----- SRC vaut prog1.C prog2.C
```

Ici, la variable SRC, recoit la valeur de la variable OBJ dans laquelle toutes les parties valant xxxx.o sont remplacées par xxx.C ! Le caractère '%' sert à spécifier n'importe quelle partie.

```
%.o : %.c
```

Là encore on spécifie que tout fichier du type `xxx.o` dépend forcément du fichier `%.c` correspondant.

E) Override et ligne de commande

Il est possible d'affecter des variables, directement à l'appel de make :

```
> make ... "OBJ=toto.o"
```

△ Toute affectation de OBJ dans le Makefile est alors ignorée.
SAUF, si l'assignation a été déclarée "override" :

```
override CFLAGS += -g
```

Ajoute `-g` à `CFLAGS`, quelle que soit la définition de la ligne de commande.

F) Define : équivalent à =

```
define OBJ
        toto.o
endef           #----- la variable OBJ vaut toto.o
```

Cette façon de définir des variables permet d'utiliser plusieurs lignes, et donc de définir, par là même, des procédures !

```
define run-latex
        latex toto.tex
        bibtex toto
        latex toto.tex
endef           #----- run-latex, peut être utilisée comme commande
toto.dvi: toto.latex; $(run-latex)
```

G) Variables locales

- Par défaut : les variables sont “globales” (définies dans TOUT le Makefile).
- Si on veut une variable “locale” à une règle :

```
prog : CFLAGS = -O7
prog : prog.o toto.o
```

- `CFLAGS` vaut `-O7` pendant toute la génération de prog, prog.o, toto.o et de leurs dépendances.

H) Variables automatiques

$@ : correspond à la cible.
$< : correspond à la première dépendance.
$? : correspond à toutes les dépendances qui sont plus récentes que la cible.
$^ : correspond à toutes les dépendances, répétées une seule fois.
$+ : correspond à toutes les dépendances, répétées autant de fois qu'elles apparaissent dans le Makefile.
$* : correspond à la valeur commune à la cible et à la dépendance dans une expression régulière (par exemple, la partie '%' dans une expression '%.o').

Il est en outre possible de spécifier si l'on veut seulement la partie répertoire ou seulement la partie fichier :

$(@D) : correspond au répertoire de la cible (`dir` dans `dir/toto.o`).
$(@F) : correspond au fichier de la cible (`toto.o` dans `dir/toto.o`).
$(<D) : correspond au répertoire de la première dépendance.
...

Voilà par exemple comment proposer un format de compilation général pour les fichiers C :

```
% : %.c
        gcc -g  $< -o $@
```

Cette règle définit que tout exécutable `xxxx` peut être construit à partir du fichier `xxxx.c`. Pour cela make utilisera la ligne de commande en substituant les variables automatiques de la façon suivante : `gcc -g xxxx.c -o xxxx`.

I) L'utilitaire `makedepend`

Le programme `makedepend` lit chacun des fichiers source et les analyse comme un préprocesseur. Tous les #include, #define, #undef, #ifdef, #ifndef, #endif, #if, #elif et #else sont ainsi traités, éventuellement récursivement. Les dépendances ainsi générées sont ajoutées à la suite du fichier Makefile.

Le compilateur `gcc` inclut dorénavant cette fonctionnalité par la commande `gcc -M` et ses variantes `gcc -M?` (`-MM` pour ne pas inclure les headers système, `-MF` *`fichier`* pour écrire ces dépendances dans un fichier, etc.)

III Les règles implicites et leur variables associées

Un certain nombre de règles génériques, comme par exemple la compilation de programmes C, C++, sont déjà prédéfinies par make. Elles utilisent un certain nombre de variables prédéfinies par défaut. Voici par exemple les règles utilisées pour la compilation de fichiers C/C++ :

```
%.o : %.c
        $(CC) -c  $(CPPFLAGS)  $(CFLAGS) $+ -o $@

%.o : %.cpp
        $(CXX) -c  $(CPPFLAGS)  $(CXXFLAGS) $+ -o $@

% : %.o
        $(CC) $(LDFLAGS) $+ $(LOADLIBES) $(LDLIBS)  -o $@

% : %.cpp
        $(CXX) $(CPPFLAGS) $(CXXFLAGS) $(LDFLAGS) $+ $(LOADLIBES) $(LDLIBS)  -o $@
```

IV Parties conditionnelles

Il est possible de n'exécuter que certaines parties du Makefile, sous certaines conditions. Ici par exemple, certaines options de compilation ne sont passées qu'à gcc :

```
gccopt = "-O5 -funroll-all-loops"
prog : $(OBJ)
ifeq( $(CC), gcc )
        $(CC) -o prog  $^ $(gccopt)
else
        $(CC) -o prog  $^
endif
```

La syntaxe générale des structures conditionnelles est la suivante (sachant que la partie `else` peut être omise :

```
ifxxxx Test
partie-a-executer-si-la-condition-est-vérifiée
else
partie-a-executer-si-la-condition-n'est-PAS-vérifiée
endif
```

Il y a quatre tests possibles :

1. `ifeq ( arg1, arg2 )` : teste l'égalité, après substitution, des ses deux arguments.
2. `ifneq ( arg1, arg2 )` : teste l'inégalité, après substitution, des ses deux arguments.
3. `ifdef variable` : teste si la variable a une valeur non nulle.
4. `ifndef variable` : teste si la variable n'existe pas ou est vide.

V Manipulations de chaines de caractères

Nous avons déjà vu que l'on pouvait altérer des variables, par exemple avec la commande `SRC := $(OBJ:%.o=%.C)`. Cette fonctionnalité est plus générale. Il est possible de manipuler les chaînes de caractères avec des fonctions ayant la syntaxe générale suivante : `$(fonction arguments)`.

Voilà un spicilège de ces nombreuses fonctions :

- `$(subst e, a, texte)` → `taxta` : remplacement.
- `$(patsubst %ex, %ax, texte)` → `taxte` : remplacement par expressions régulières.
- `$(strip ‘‘ a e‘‘)` → `a e` : supprime les espaces inutiles.
- `$(findstring a, a b c)` → : .
- `$(filter %c %s, t.c h.s u.o)` → `t.c h.s` : sélectionne.
- `$(filter-out %c %s, t.c h.s u.o)` → `u.o` : sélection inverse.
- `$(sort b c b a)` → `a b c` : tri ensembliste alphabetique.
- `$(dir src/t.c h.s)` → `src/ ./` : sélectionne les répertoires.
- `$(notdir src/t.c h.s)` → `t.c h.s` : retire les parties répertoire.
- `$(suffix t.c h.s)` → `.c .s` : extrait les suffixes.
- `$(basename t.c h.s)` → `t h` : retire les suffixes.
- `$(addsuffix .c, t h)` → `t.c h.c` : ajoute un suffixe.
- `$(addprefix src/, t h)` → `src/t src/h` : ajoute un prefixe.
- `$(join t h, .c .s)` → `t.c h.s` : fusionne les deux listes.
- `$(word 2, a b c)` → `b` : sélectionne le i-ème élément.
- `$(wordlist 1,3, a b c d)` → `a b c` : sélectionne une liste d'éléments.
- `$(words a b c d)` → `4` : nombre d'éléments dans la liste.
- `$(firstword a b c d)` → `a` : sélectionne le premier élément.
- `$(lastword a b c d)` → `d` : sélectionne le dernier élément.
- `$(if condition,then-part[,else-part])` : si la condition contient une chaîne non vide la partie `then-part` est évaluée.
- `$(or condition1[,condition2[,condition3...]])` : retourne la première des conditions qui n'est pas la chaîne vide.
- `$(and condition1[,condition2[,condition3...]])` : renvoie la dernière des conditions ou la chaîne vide si celle-ci est l'une des conditions.
- `$(wildcard *.c)` → `t.c x.c` : utilise `*` comme dans un shell (ici les fichiers `t.c` et `x.c` étaient présent dans le répertoire courant).
- `$(foreach dir, a b c d, $(wildcard $(dir)/*.c) )` → `a/t.c d/x.c` : effectue une même commande pour chacune des instances.
- `$(origin OBJ)` → `command line` : indique où a été définie la variable. Les valeurs possibles sont : undefined (la variable n'a jamais été définie), default (si la variable a une valeur par défaut, ex : CXX=g++), environment (variable issue du shell), file (issue du Makefile), command line, override, automatic.
- `$(shell ls)` → `t.c h.s` : appelle une fonction dans le shell courant.
- `$(error mauvais appel)` → `*** mauvais appel. Arrêt.` : provoque une erreur.
- `$(warning appel incorrect)` → `appel incorrect` : affiche un message.

Mais aussi `$(call ...)`, `$(eval ...)`, `$(value ...)`, `$(flavor ...)`, etc.

VI Options de ligne de commande

Outre la définition de variables, il est possible de passer plusieurs options à make par le biais de la ligne de commande :

- `make "VAR=toto"` : la variable `VAR` vaut `toto` dans tout le Makefile (voir II . E).
- `make -n` : affiche les commandes sans les exécuter.
- `make -p` : affiche toutes les règles et variables prédéfinies.
- `make -t` : équivalent à `touch`, les cibles sont maintenant toutes à jour, sans qu'aucune commande n'ait été exécutée.
- `make -W fichier` : considère que le fichier est nouveau.
- `make -B` : considère que toutes les cibles sont à refaire, sans condition.
- `make -k` : essaie de continuer malgré les erreurs rencontrées.
- `make -j #` : spécifie le nombre de processus à lancer en parallèle ; si le nombre est omis, le plus grand nombre possible de processus est utilisé.

Voici un exemple de compilation parallèle, supposons le Makefile suivant :

```
%.o:%.c
        @echo "Compilation de $@"
        $(CC) -c $+ -o $@
        @echo "$@ a été produit"

t:h.o u.o
```

Normalement les appels sont séquentiels.

```
> make
Compilation de h.o
cc -c h.c -o h.o
h.o a été produit
Compilation de u.o
cc -c u.c -o u.o
u.o a été produit
cc     t.c h.o u.o   -o t
```

Avec l'option `-j`, les différentes compilations préliminaires indépendantes peuvent être parallèles.

```
> make -j 2
Compilation de h.o
cc -c h.c -o h.o
Compilation de u.o
cc -c u.c -o u.o
u.o a été produit
h.o a été produit
cc     t.c h.o u.o   -o t
```