Concevoir un Makefile

Vincent Loechner
(original : Nicolas Zin, http://www.linux.efrei.fr/)

5 novembre 2001

Table des matières

Chapitre 1

Introduction

Un makefile est un fichier utilisé par le programme make pour arriver à un but, ce but étant lui-même dépendant d'autres buts, et ce de manière récursive. On parle alors plutôt de cibles plus que de but.

Make est principalement utilisé pour créer de gros programmes, où la cible principale est un exécutable. Pour que cet exécutable puisse être compilé, il faut pouvoir rassembler les différents objets (.o) qui le composent. Ces objets dépendent à leur tour de fichiers sources ou d'autres objets.

La différence entre make et un script classique est que make sait si une cible donnée est à jour ou non ; donc il sait si cette cible doit être reconstruite, ou peut être conservée. Lorsqu'on demandera à make de refaire une cible, il regardera quelles sont les sous-cibles qui ne se sont pas à jour, les refera et refera ensuite la cible principale.

Make peut également être utilisé pour d'autres choses que pour la génération d'exécutables, comme pour l'administration système, la génération de documents latex, ...

Chapitre 2

Création d'un makefile

Lorsqu'on appelle la commande make, le programme va essayer de trouver un fichier makefile ou Makefile. Si aucun argument n'est donné à make, la première cible trouvée dans le fichier makefile est considéré comme cible principale ; sinon il considère l'argument passé en paramètre comme cible principale.

2.1 Simple Makefile

Un simple makefile est composé de règles ayant l'apect suivant :

```
CIBLE: DEPENDANCES
<TAB>  COMMANDE
<TAB>  COMMANDE
<TAB>  COMMANDE
```

La cible est généralement le nom du fichier que l'on veut générer. Les dépendances sont les fichiers utilisés pour générer cette cible. Les commandes sont les actions à mener pour générer la cible à partir des dépendances.

Quand make va essayer de créer la cible, il va regarder la (les) dépendance(s). En fait make considère qu'on travaille avec des fichiers et que la cible et les dépendances sont des noms de fichers. Pour déterminer si une cible ou une dépendance est à jour, il va regarder la date de modification du fichier portant le nom de la dépendance ou de la cible.

Il va regarder si les dépendances sont à jour, c'est à dire si elles ne dépendent pas elle non plus d'autres règles. Si c'est le cas, il va exécuter ces règles.

Ensuite on va voir si la cible a besoin d'être refaite. Si la date de modification d'une dépendance est plus jeune que la date de modification de la cible, il considère la cible comme n'étant pas à jour, et il va alors exécuter les commandes.

Si make ne trouve pas le fichier corespondant à une dépendance et qu'il n'arrive pas à créer ce fichier, il génère une erreur. Si le fichier cible n'existe pas, il considère la cible comme non à jour.

2.2 Règles d'écritures

Dans un makefile, il faut faire attention à commencer une ligne de commande par une tabulation. Cela permet à make de savoir qu'est ce qu'il doit considérer comme commande.

Si on veut mettre des expressions (dépendance ou quoique ce soit) sur plusieurs lignes, on peut le faire en terminant la ligne par un "\". Quand make voit \ il sait que l'expression continue sur la ligne suivante.

On peut formuler des règles qui n'ont pas de commandes. En particulier il est utile de placer une règle au début du makefile, souvent nommée all, qui va dépendre de la cible principale. Ainsi, si make est appelé sans argument, il va considérer la première cible qu'il voit comme la cible principale. Par ce stratagème, il va considérer comme règle principale, la bonne règle.

On peut aussi écrire des règles qui n'ont pas de dépendances. Par exemple, on fournit souvent une règle nommée clean qui efface tout les fichers générés par le make. Cette règle permet de faire le ménage.

2.3 Exemple

```
all: edit

edit : main.o command.o display.o insert.o search.o files.o utils.o
	cc -o edit main.o command.o display.o \
	        insert.o search.o files.o utils.o

main.o : main.c defs.h
	cc -c main.c
command.o : command.c defs.h command.h
	cc -c command.c
display.o : display.c defs.h buffer.h
	cc -c display.c
insert.o : insert.c defs.h buffer.h
	cc -c insert.c
search.o : search.c defs.h buffer.h
	cc -c search.c
files.o : files.c defs.h buffer.h command.h
	cc -c files.c
utils.o : utils.c defs.h
	cc -c utils.c
clean :
	rm edit main.o command.o display.o \
	   insert.o search.o files.o utils.o
```

Note : lorsque make exécute une commande il affiche les commandes exécutées. Si vous ne voulez pas que la ligne de commande soit affichée placez un '@' devant la commande.

Chapitre 3

Automatisation de tâches

Make facilite l'écriture de règles grâce à l'utilisation de variables, de caractères génériques et de règles génériques.

3.1 Utilisation de variables

Il est possible de définir des variables dans un makefile. Le but est de simplifier la modification du makefile. Si on définit une variable recevant des options de compilation, si on a à changer ces options, il est plus facile de changer une définition que tout le makefile.

Pour définir une variable il faut mettre sur une nouvelle ligne le nom de la variable suivi d'un "=" suivi de la valeur de la variable. En règle générale, on met le nom de la variable en majuscule, ça permet de la différentier des commandes ou des noms de fichiers qui sont souvent en minuscules.

Pour faire référence à une variable dans un makefile, il faut utiliser le caractère "$" suivi d'une parenthèse ouvrante, suivi du nom de la variable, suivi d'une parenthèse fermante.

Exemple :

```
CC=gcc
CFLAGS=-O2 -pipe -Wall
LDFLAGS=-O2 -L/usr/local/include -lqt

prog: a.o b.o
	$(CC) -o prog $(LDLAGS) a.o b.o
a.o: a.c
	$(CC) $(CFLAGS) -c a.c
b.o: b.c
	$(CC) $(CFLAGS) -c a.c
```

Une variable spéciale VPATH permet de dire à make dans quels répertoires il doit chercher les dépendances. Exemple : VPATH = src :../headers

3.2 Variables génériques

Ces variables sont utilisés dans la partie commande.

Symbole	Signification
$?	représente les dépendances qui ont été modifiées (les dépendances qui n'étaient pas à jour)
$^	représente toutes les dépendances. Ici, on tient compte du chemin vers le fichier. Ceci permet de référencer des fichiers qui ne sont pas dans le répertoire courant (cf. variable VPATH)
$+	même chose que $^ mais les doublons sont supprimés.
$@	représente la cible courante.
$<	représente la première dépendance. Est utile quand la première dépendance a une fonction précise.

Il est possible de définir une règle pour un ensemble de fichier, par exemple pour tous les fichiers se terminant par un .c. Pour cela on peut utiliser le caractère `%`. Quand make voit ce caractère dans un champ cible, il essaye d'y attribuer des noms de fichiers. `%.c` correspond à tous les fichiers se terminant par .c. Dans le champ des dépendances `%` est remplacé par la valeur trouvée dans la cible. Exemple :

```
%.o: %.c
    gcc -c $^
```

On peut utiliser une variable générique pour récupérer la valeur de `%` dans les commandes grâce à $*. $* donne la valeur de `%`. Exemple :

```
%.o: %.c
    gcc -c $*.c
```

3.3 Règles génériques

On peut aussi établir des règles génériques avec make. Ces règles vont être attachées à un suffixe, et seront appelées quand make tombera sur un fichier se terminant par cette extension.

Il faut d'abord donner les suffixes qui sont traités par des règles, grâce au mot clé .SUFFIXES suivi de ' :' puis des extensions (par exemple .c .cpp).

Plus loin dans le makefile on doit trouver les règles pour ces suffixes :

```
CC=g++
CFLAGS= -pipe -Wall
#
# Implicit rules
#
.SUFFIXES: .cpp
.cpp.o:
        $(CC) -c $(CFLAGS) $<
```

Comme on peut le voir, on définit une règle ayant une définition spéciale : la cible est composée de l'extension source et de l'extension destination de la cible à traiter. On trouve en dessous les commandes à exécuter pour transformer le fichier d'extension d'origine en fichier d'extension cible.

3.4 Fonctions

Il est parfois tentant de vouloir utiliser '*' comme on pourrait le faire sous un shell pour désigner tous les fichiers se terminant par .o par exemple. On peut en effet mettre une règle nommée clean qui efface tous les fichiers objets. Pour ça il faut taper :

```
clean:
        rm *.o
```

Mais on ne peut pas toujours utiliser '*'. En effet dans ce cas ça marche car toute commande est en fait lancée par un script. Donc *.o est interprété par un script. Mais il n'est pas possible de mettre '*.o' dans une variable, make mettrait la chaine "*.o" et non pas le nom de tous les fichiers se terminant par '.o'.

Pour mettre le nom des fichers se terminant par '.o' il faut passer par une fonction : $(wildcard *.o). Les fonctions commencent par '$(' et se terminent par ')' comme des réfrérence de variables. La fonction wildcard va faire ce que l'on cherche : elle va regarder les fichiers qui correspondent au motif (ici *.o) que l'on passe en paramètre.

D'autres fonctions existent, par exemple la fonction filter permet de filtrer une liste de nom via un filtre. $(filter %.o,$(noms)) correspond à tous les noms contenus dans $(noms) répondant à %.o .

La fonction $(subst FROM,TO,TEXT) permet de faire une substitution :

```
comma:= ,
empty:=
space:= $(empty) $(empty)
foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
# bar is now 'a,b,c'.
```

La fonction $(patsubst PATTERN,REPLACEMENT,TEXT) permet de faire une substitution dans une liste de mots (des fichiers par exemple) :

```
SRC:= $(wildcard *.c)
OBJ:= $(patsubst %.c,%.o,$OBJ)
```

Chapitre 4

Gestion des makefiles

4.1 Lancer make à partir d'un make

Il est possible à partir d'un makefile d'appeler d'autres makefiles. En particulier on peut être amené à compiler un programme réparti dans plusieurs répertoires. Pour ça deux possibilités :

- make -C <sous-répertoire> demande d'aller dans le sous- répertoire et de relancer un make.
- (cd <sous-répertoire>; make) : faire attention à bien mettre des parenthèses. En effet cette commande est transmise au shell, et si les parenthèses ne sont pas mises, le shell va considérer les deux instructions comme indépendantes, et va effectuer le cd <sous-répertoire>, il va rendre la main (donc ressortir du sous-répertoire) et lancer la commande suivante qui est make (donc qui sera exécuté dans le répertoire courant).

4.2 Inclure un fichier

Il est possible à l'intérieur d'un makefile d'inclure un fichier, grâce à la commande include <nom fichier> au début d'une nouvelle ligne. Ca peut être intéressant, en particulier si on a un projet réparti dans plusieurs répertoires. Il faut alors un makefile par répertoire. Tous ces makefiles peuvent inclure un makefile global contenant le programme de compilation, les règles implicites de compilation, et différentes variables. Ainsi s'il faut changer un paramètre il suffit de changer le fichier inclu par tous les makefiles. Exemple :

```
-----------Makefile.global----------
CC=g++
CFLAGS= -pipe -Wall -O2
LDFLAGS=-g
MAKE=/usr/bin/make
FIND=/usr/bin/find
```

```
#
# Implicit rules
#
.SUFFIXES: .cpp
.cpp.o:
        $(CC) -c $(CFLAGS) $<

------------Makefile------------
include Makefile.global
OBJ= main.o

all: $(OBJ)
        for i in $(SUBDIRS); do (cd $$i; $(MAKE) all); done
        $(MAKE) ciblePrincipale

clean:
        rm -f $(OBJ) core *~

ciblePrincipale:
        @echo 'Ce que l'on veut'
```

4.3 Créer des dépendances

Plus le projet est gros, plus on est tenté d'utiliser des règles implicites. Mais le problème des règles implicites est qu'elles considèrent que la cible doit être reconstruite que si la dépendance principale a été changée. Or un fichier C doit être recompilé si effectivement le fichier C a été modifié, mais aussi si un des fichiers que le fichier C inclut a été modifié.

La commande makedepend permet de déterminer les dépendances d'un fichier C ou C++. Sa syntaxe est makedepend <fichier.c>. Ce programme va lire les dépendances et va les ajouter au makefile. On peut aussi passer des arguments à makedepend, comme les répertoires utilisés par le compilateur C pour chercher les headers standards. En fait il suffit de passer à makedepend les arguments de compilation, comme on les passerait au compilateur. Il se charge de trouver ce qui l'intéresse pour établir les dépendances. La syntaxe à utiliser est : makedepend −− <options de compilation> −− <fichiers.c>. Par exemple :

```
makedepend -- $(CFLAGS) -- $(SRC)
```

Généralement on crée une règle dans le makefile qui va créer les dépendances, avant de compiler. Exemple :

```
SUBDIR= core ihm annexes

CC=gcc
CFLAGS=-I/usr/local/include -Wall -pipe
LDFLAGS=-lMesaGL -L/usr/local/lib
```

```
SRC=main.c install.c lancement.c
OBJ=$(subst .c,.o,$(SRC))

depend:
	makedepend -- $(CFLAGS) -- $(SRC)
```

4.4 Squelette d'un makefile

Si on devait faire un squelette d'un makefile d'un programme C il devrait ressembler à quelque chose comme :

```
CC=gcc
CFLAGS=-I/usr/local/include -Wall -pipe
LDFLAGS=-lMesaGL -L/usr/local/lib
RM=/bin/rm
MAKE=/usr/bin/make
MAKEDEPEND=/usr/X11R6/bin/makedepend

SRC= a.c \
     b.c \
     c.c
OBJ=$(subst .c,.o,$(SRC))

SUBDIR= paf pof


.SUFFIXES: .c
.c.o:
	$(CC) -c $(CFLAGS) $<

all:
	for i in $(SUBDIRS); do (cd $$i; $(MAKE) all); done
	$(MAKE) monProgramme

monProgramme: $(OBJ)
	$(CC) -o $@ $(LDFLAGS) $^

clean:
	$(RM) -f $(OBJ) core *~
	for i in $(SUBDIRS); do (cd $$i; $(MAKE) clean); done

depend:
	$(MAKEDEPEND) -- $(CFLAGS) -- $(SRC)
	for i in $(SUBDIRS); do (cd $$i; $(MAKE) depend); done
```

Chapitre 5

Conclusion

J'ai présenté les fonctionnalités les plus couramment utilisées de make. Beaucoup de subtilités et de possibilités existent encore, mais ne sont pas d'usage courant.

D'autre part, j'ai parlé du make GNU, car c'est celui par défaut sous Linux. D'autres make existent, ils sont un peu différent de celui de chez GNU, en particulier, le make GNU a plus de possibilités que les autres, donc il se peut que certains fonctionnalités ne fonctionnent pas avec d'autres make, même si les principales fonctionnalités sont les mêmes.