[UX070]

La commande make

pour plus de détails
[JMR] pages 269 à 272
man make

Syntaxe de la commande make

make [**-f** fichier_de_dépendance] [**-options** .] [cible(+)]

avec par défaut : fichier de dépendance : par défaut makefile, Makefile.
cible : la cible de la première règle du fichier de dépendances.

De quoi s'agit il ?

La commande make permet la MAJ d'une (ou plusieurs) cible (s) en se basant sur les du fichier de dépendance.

Que contient le fichier de dépendances (makefile) ?

* des commentaires : # jusqu'à la fin de ligne
* des définitions de macros :

identificateur = *chaine*

* des règles qui sont constituées par :
  - (une) ligne de dépendances :

    cible **:** [*dépendance1*] [*dépendance2...*]
    ou bien
    cible **::** [*dépendance1*] [*dépendance2...*]

  - aucune, ou une, ou plusieurs ligne(s) de commandes :

    <tab> *commande_shell* ⚠ <tab> ≠ ␣␣...

(→ des exemples de fichiers de dépendances vous sont donnés dans les pages suivantes)

Utilisation des macros

$(*identificateur*) Les parenthèses sont obligatoires, sauf lorsque l'identificateur ne comporte qu' une seule lettre.

Mise à jour d'une cible

(voir l'exemple 1 de la page suivante)

make commence par mettre à jour (récursivement) toutes les dépendances de la cible.

Ensuite, si la cible correspond à un fichier existant plus récent que ses dépendances (celles-ci devraient donc aussi être des fichiers)

alors la cible est à jour !

sinon | les commandes shell de la règles sont exécutées, et lorsque c'est fait la cible est à jour !

```
# Exemple 1 : Makefile for the RayTracer program
OBJS= ray.o  compile.o  graph.o  geom.o
ray: $(OBJS)
    @echo Edition de liens du programme ray
    cc  $(OBJS)  -o ray -lm
ray.o:  ray.c  ray.h  graph.h
    cc -c ray.c
compile.o :  ray.h compile.c
    cc -c compile.c
graph.o :  graph.c graph.h
    cc -c graph.c
geom.o : geom.c
    cc -c geom.c
```

biblio math.

Règles d'inférence

Les règles d'inférence[6] sont des règles qui sont déduites des suffixes des noms de fichiers.

Les règles d'inférence sont en général implicites. C'est dire qu'on n'a pas besoin de les définir.

Par exemple la règle **.c.o** est la suivante :

> Si un fichier **.o** (cible) est moins récent que le fichier **.c** associé (dépendance),
> alors make recompilera le fichier **.c** pour mettre à jour le fichier **.o**.

Reprenons l'exemple 1 en utilisant cette règle :

```
# Exemple 2 : avec règle implicite .c.o
OBJS= ray.o  compile.o  graph.o  geom.o
ray: $(OBJS)
    @echo Edition de liens du programme $@
    cc  $(OBJS)  -o ray -lm

ray.o: ray.h  graph.h
compile.o :  ray.h
graph.o : graph.h
```

$@ nom de la cible couvante.

---------- (On peut aussi remplacer les trois dernières lignes par les deux suivantes) ----------

```
ray.o compile.o : ray.h
ray.o graph.o : graph.h
```

[6] inférence = déduction

Règles d'inférence explicites

Lorsque son fonctionnement implicite ne convient pas, on peut expliciter (définir) une règle d'inférence :

Voici deux exemples de règles **.c.o** explicites :

```
.c.o:
    cc -c $<
```

```
.c.o:
    @echo compilation de $<
    $(CC)       $(CFLAGS)   -c    $<
```

Et voici un exemple d'utilisation de la deuxième :

```
# Exemple 3 : avec règle .c.o explicite

OBJS= ray.o  compile.o  graph.o  geom.o
ray: $(OBJS)
    @echo Edition de liens du programme $@
    @echo       (dépendances modifiées : $?)
    cc  $(OBJS)  -o ray -lm

ray.o compile.o : ray.h
ray.o graph.o : graph.h

.c.o:
    @echo Compilation de  $<  pour obtenir $@
    $(CC)   $(CFLAGS)  -c  $<
```

Récapitulatif des macros prédéfinies

$@ cible courante.

$? dépendances plus récentes que la cible.

$* cible privée de son suffixe

$< source courante.

$(CC) compilo utilisé

$(CFLAGS) opt° utilisées.

Les deux types de cibles

cibles simples (:) → ***Une seule règle***

Il peut y avoir en fait plusieurs lignes de dépendances, mais une seule est suivie de lignes de commandes (voir exemples 1 à 3, ainsi que certaines cibles de l'exemple 4 de la page suivante)

cibles multiples (::) → ***plusieurs règles sont possibles***

Chaque règle est constituée de sa ligne de dépendance et des commandes associées (voir exemple 4, la cible **xenix**).

Un autre exemple

```
# Exemple 4 :  Makefile for Joe's Editor

# Directory  to install j into
WHERE = ~/joe

# Use these two for 'cc'
CC = cc
CFLAGS = -DKEYDEF=\"$(WHERE)/keymap.j\" -O

# Use these two for 'gcc'
#CC = gcc
#CFLAGS    =    -DKEYDEF=\"$(WHERE)/keymap.j\"    -
traditional -O

foo:
 @echo Type make followed by one of the following
 @echo
 @echo bsd hpux xenix install clean

xenix:: j.o asyncxenix.o blocks.o
 $(CC) $(CFLAGS) j.o asyncxenix.o blocks.o -lx -
ltermcap -o j
 cp keymapxenix keymap.j

bsd hpux: j.o async$*.o blocks.o
 $(CC) $(CFLAGS) j.o async$*.o blocks.o -ltermcap
-o j
 cp keymapbsd keymap.j

xenix:: keymapxenix
 cp keymapxenix keymap.j

install:
 strip j
 mv j $(WHERE)
 mv keymap.j $(WHERE)
 chmod a+x $(WHERE)/j
 chmod a+r $(WHERE)/keymap.j

clean:
 rm asyncbsd.o asyncxenix.o asynchpux.o blocks.o
j.o keymap.j

asyncbsd.o asyncxenix.o asynxhpux.o : async.h

blocks.o : blocks.h

j.o : blocks.h j.h async.h
```

Et enfin un cinquième exemple

```
# Makefile simple pour compilation de programmes C - JPBlanc - IUT Informatique de Clermont I -
# (ce makefile est utilisable même si vous ne connaissez pas la commande make)

# Pour utiliser ce 'Makefile' :
# - Copiez le dans le répertoire où se trouve le programme à compiler
# - Changez ci-dessous la valeur de PROG : donnez lui le nom de votre
#       programme sans mettre .c ni aucune extension
# - Faire la commande 'make'. C'est tout!

# si vous souhaitez reconstruire entièrement votre programme (bien qu'il soit
# partiellement à jour) faire 'make rebuild' au lieu de 'make'.

# si vous souhaitez temporairement compiler un autre programme
# (par exemple 'prog.c')       faire simplement la commande 'make prog'
#                                                    (sans aucune extension ici non plus)

#---------------------------------------------------------------------------
# Définition des variables :

# Nom du programme exécutable après compilation (remplacez
# "monprog" dans la ligne suivante par le nom de votre programme)
PROG=monprog

# nom du compilateur (décommentez uniquement la ligne qui convient).
#CC=/users/tpinfo/bin/gcc
CC=cc

# Option(s) pour norme ANSI (décommentez uniquement la ligne qui convient).
#CFLAGS=                # pas de norme ANSI
#CFLAGS= -Aa -g         # norme ANSI
CFLAGS= -Ae -g          # norme ANSI étendue (avec les //)

# Option pour déboguage ou optimisation (décommentez uniquement la ligne qui convient).
DFLAGS= -g                      # débogage
#DFLAGS= -O                     # optimisation

# Quelle(s) bibliothèque(s) faut il utiliser. Par exemple '-lm' signifie
# bibliotheque  'libm.sl' c'est-à-dire biblio des fonctions mathématiques
LIBS= -lm

# Objets : Si vous avez plusieurs fichiers .c rajoutez ici les .o correspondants
#             à la suite de $(PROG).o
OBJS=$(PROG).o

#---------------------------------------------------
# A partir d'ici sont les règles de compilations interprétées par 'make' :

$(PROG): $(OBJS)                #règle pour édition de lien de votre programme
  cc  $(OBJS) -o $(PROG) $(LIBS)

.c.o:                                           #règle d'inférence pour compilation
  cc $(CFLAGS)  $(DFLAGS) -c $<

.c:                                             #règle d'inférence pour compilation avec édition de
liens
  cc $(CFLAGS)  $(DFLAGS)  $< -o $@  $(LIBS)

rebuild: clean $(PROG) #règle pour recompilation complète de votre programme

clean:                                  #règle pour nettoyage (suppression des fichiers objets).
  rm $(OBJS)
```