Outils de développement

Makefile
Apprendre à automatiser

Florent Devin

Ecole Internationale des Sciences du Traitement de l'Information

Présentation

Présentation

- Permettre de construire des fichiers
- Éventuellement à partir d'autres fichiers
- Utilisation de commandes shell

Exemple

- Compilation de programme
- Construction de fichier `ps`, `pdf` à partir de LaTeX
- Construction de librairie
- Sauvegarde de fichiers, ...

Utilisation

- Nécessite un fichier de configuration
  - Généralement appelé *makefile*
  - Description des dépendances entre fichiers
  - Règles de mises à jour, création, sauvegarde, ...
  - Très “sensible” à l’utilisation des espaces et tabulations
- Nécessite une commande
  - Génère ce qui est demandé en fonction du fichier de configuration
  - `make`
- Se base sur la date des fichiers
- Affiche les commandes avant de les exécuter

Appels possibles

- Forme expansée : `make -f fichier_configuration cible`
- Possibilité d'omettre le `fichier_configuration` si celui-ci s'appelle : `makefile` ou `Makefile`
- Possibilité d'omettre la `cible` $\Leftrightarrow$ invocation de la première cible rencontrée
- **`À l'EISTI, nom du fichier de configuration : Makefile`**
- **`Première cible doit être all`**

Structure d'un `Makefile` simple

Listing 1 – "Exemple d'un makefile très simple"

```
#ceci est un commentaire.
#le symbole ⇥ indique une tabulation
toto : toto.c
⇥gcc toto.c -o toto
```

ligne3 `toto` : cible ; `toto.c` dépendance de la cible

ligne4 Tabulation : indique que l'on est toujours dans la même cible ; Comment faire la cible à partir des dépendances

Définition d'une règle

- Une première ligne contenant
  - un nom de cible
  - :
  - une liste de fichier dont dépend la cible
- de 0 à *n* lignes de commandes contenant
  - le caractère de tabulation
  - une ou plusieurs commandes.

Exécution de la règle uniquement si les dépendances sont présentes, ou si elles sont "constructibles"

Cas particulier

- Règles sans dépendances : règles toujours exécutées
  - `clean :`
- Règles sans commandes : permet de “construire” des règles de manière arborescente
  - `all : .../...`

Makefile facile

Listing 2 – "Exemple d'un makefile simple"

```
all : toto

toto : toto.o
	gcc toto.o -o toto

toto.o : toto.c toto.h
	gcc -c toto.c -o toto.o
```

Utilisation avancée

Définition de variables

- Possibilité de définir des variables
  - `nom = valeur` ou `nom=valeur`
  - **`CC = gcc`**
  - **`FICHIER = toto.c toto.h`**
  - **`compilation = gcc -c`**
- Utilisation des variables
  - `$(nom)` ou `${nom}`
  - **`$(FICHIER)`**
  - **`${CC}`**
- Équivaut à la substitution syntaxique du pré-processeur
- Doit être défini avant l'utilisation

Variables internes

- $@ : nom de la cible
- $? : liste des dépendances plus récentes que la cible
- $^ : liste des dépendances
- $< : nom de la première dépendance
- Pour plus d'informations : `http ://www.gnu.org/software/make/manual/make.html`

Utilisation

Listing 3 – "makefile avec variables"

```
CC = gcc
PROG=toto
OBJ=toto.o
SRC=toto.c
HEAD= toto.h

all : $(PROG)
```

Utilisation

Listing 4 – "makefile avec variables"

```
$(PROG) : $(OBJ)
    $(CC) $^ -o $@

$(OBJ) : $(SRC) $(HEAD)
    $(CC) -c $(SRC) -o $@


clean :
    rm -f $(OBJ) $(PROG)
```

Règles phony

- Phony (fam.) : Faux/fausse, bidon (d'après Harraps)
- Que se passe-t-il si une cible porte le nom d'un fichier ?
  - `make` regarde si le fichier existe et est à jour : c'est le cas
  - Pas de compilation
  - Exception pour la cible `all`
- Utilisation des règles "bidons"
- Spécifie qu'il faut toujours faire la cible, sans se soucier d'autre choses
- **`.PHONY : cible`**
- Notion de directives

wildcard et substitution

- *wildcard*
  - Permet l'utilisation de caractères jokers
  - `SRC=$(wildcard *.c)`
- *substitution*
  - Permet de changer l'extension
  - `HEAD=$(SRC :.c=.o)`
  - Attention très très sensible au *non-espace*

Suffixes et caractères spéciaux

- *Suffixes*
  - Permet de construire tous les fichiers d'un certain suffixe, à partir d'autres fichiers
  - Utile pour construire un ensemble de fichiers objets avant de construire le programme
  - `.sufx1.sufx2 :`
- *Caractères spéciaux*
  - @ : pas d'affichage de la commande avant exécution
  - - : en cas d'erreur poursuite du main

Exemple récapitulatif

Listing 5 – "Maîtrise ?"

```
SRC = $(wildcard *.c) #fichier source
OBJ = $(SRC:.c=.o)    #fichier objet

toto : $(OBJ) #compilation du prog
	@gcc $^ -o $@
.c.o :          #compilation des objets
	gcc -c $^ -o $@

.PHONY : clean

clean :         #effacement de fichier
	rm -f toto $(OBJ)
```

Makefile récursif

- Supposons une arborescence classique :
  `rep1, rep2, ..., sourep11, sourep12, ..., sourep1n, sourep21, sourep22, ..., sourep2n, ...`
- Comment faire pour tout compiler ?
- Impossible de faire :

```
.PHONY : rep1 rep2
rep1 :
	cd rep1
	make sourep1
	make sourep2
```

- Deux problèmes
  - structure peut changer
  - une commande : un shell

Impossible ?

- Utilisation de l'option `-C`
- `make -C rep cible(s)`
- Peu générique : nécessite de connaître la structure à l'avance
- Utilisation du shell :
  - `$(shell cmde)`
  - **`$(shell ls *.c)`**
  - **`DIR = $(shell find . -maxdepth 1 -type d -print)`**
- Peut nécessiter un filtre : utilisation de `filter`/`filter-out`

Filtres

- *filter*
  - **$(filter motif, texte)**
  - Tous les éléments de texte qui vérifie un motif
  - **motif** : noms séparés par des espaces, utilisation possible de %
  - **texte** : idem
- *filter-out*
  - Idem que `filter`
  - Tous les éléments de texte qui ne vérifient pas l'un des motifs

Enfin !

Listing 6 – "Erreur mais pourquoi ?"

```
DIR = $(filter-out . ./Image,$(shell
    find . -maxdepth 1 -type d -print))

all: $(DIR)

$(DIR):
	$(MAKE) -C $@
```

Il en reste

- Beaucoup d'autres fonctionnalités
  - Directives d'inclusion, export, de masquage d'export, conditionnelles
  - Règles implicites (sur certaines version de `make` uniquement)
  - L'utilitaire : `makedepend`
  - Fonctions de substitution, de tri, de répétition, ...