Un tutoriel simple Makefile

Les makefiles sont un moyen simple d'organiser la compilation de code. Ce tutoriel ne raye même pas la surface de ce qui est possible en utilisant *make* , mais est conçu comme un guide de démarrage afin que vous puissiez créez rapidement et facilement vos propres makefiles pour les petites et moyennes entreprises projets.

Un exemple simple

Commençons par les trois fichiers suivants, hellomake.c, hellofunc.c, et hellomake.h, qui représenteraient un main typique programme, du code fonctionnel dans un fichier séparé et un fichier inclus, respectivement.

hellomake.c

```
#include <hellomake.h>

 int main() {
   // appelle une fonction dans un autre fichier
   myPrintHelloMake();

   retour(0);
 }
```

hellofunc.c

```
#include <stdio.h>
 #include <hellomake.h>

 void myPrintHelloMake(void) {

   printf("Bonjour les makefiles !\n");

   revenir;
 }
```

hellomake.h

```
/*
 exemple inclure un fichier
 */

 void myPrintHelloMake(void);
```

Normalement, vous compileriez cette collection de code en exécutant la commande suivante :

```
gcc -o hellomake hellomake.c hellofunc.c -I.
```

Cela compile les deux fichiers .c et nomme l'exécutable hellomake. La `-JE.` est inclus pour que gcc regarde dans le courant répertoire (.) pour le fichier d'inclusion hellomake.h. Sans makefile, l'approche typique du cycle de test/modification /débogage consiste à utiliser le flèche dans un terminal pour revenir à votre dernière commande de compilation afin que vous pas besoin de le taper à chaque fois, surtout une fois que vous en avez ajouté quelques-uns plus de fichiers .c au mélange.

Malheureusement, cette approche de la compilation a deux inconvénients. Première, si vous perdez la commande de compilation ou changez d'ordinateur, vous devez retaper à partir de zéro, ce qui est au mieux inefficace. Deuxièmement, si

vous êtes seulement apporter des modifications à un fichier .c, en les recompilant tous à chaque fois aussi chronophage et inefficace. Alors, il est temps de voir ce que nous pouvons faire avec un makefile.

Le makefile le plus simple que vous pourriez créer ressemblerait à ceci :

Makefile 1

```
hellomake : hellomake.c hellofunc.c
      gcc -o hellomake hellomake.c hellofunc.c -I.
```

Si vous mettez cette règle dans un fichier appelé `Makefile` ou `makefile` puis tapez `make` sur la ligne de commande exécutera la commande de compilation telle que vous l'avez écrite dans le makefile. Notez que `make` sans arguments exécute le premier règle dans le fichier. De plus, en mettant la liste des fichiers sur lesquels la commande dépend de la première ligne après le `:` , make know que la règle `hellomake` doit être exécutée si l'un de ces les fichiers changent. Immédiatement, vous avez résolu le problème #1 et pouvez éviter en utilisant la flèche vers le haut à plusieurs reprises, en recherchant votre dernière compilation commande. Cependant, le système n'est toujours pas efficace en termes de compiler uniquement les dernières modifications.

Une chose très importante à noter est qu'il y a un onglet avant le gcc commande dans le makefile. Il doit y avoir un onglet au début de chaque command, et `make` ne sera pas content si ce n'est pas là.

Afin d'être un peu plus efficace, essayons ce qui suit :

Makefile 2

```
CC=gcc
 CFLAGS=-I.

 hellomake : hellomake.o hellofunc.o
      $(CC) -o hellomake hellomake.o hellofunc.o
```

Alors maintenant, nous avons défini quelques constantes `CC` et `CFLAG` . Il s'avère que ce sont des constantes spéciales qui communiquer pour `faire` comment on veut compiler les fichiers hellomake.c et hellofunc.c. En particulier, la macro `CC` est le compilateur C à utiliser, et `CFLAGS` est la liste des drapeaux à passer à la commande de compilation. En mettant l'objet fichiers--hellomake.o et hellofunc.o--dans la liste des dépendances et dans le règle, `make` sait qu'il doit d'abord compiler les versions .c individuellement, puis créez l'exécutable hellomake.

L'utilisation de cette forme de makefile est suffisante pour la plupart des petites échelles projets. Cependant, il manque une chose : la dépendance vis-à-vis du inclure des fichiers. Si vous deviez apporter une modification à hellomake.h, pour exemple, `make` ne recompilerait pas les fichiers .c, même si ils devaient l'être. Afin de résoudre ce problème, nous devons dire à `make` que tous les fichiers .c dépendent de certains fichiers .h. Nous pouvons le faire en écrire une règle simple et l'ajouter au makefile.

Makefile 3

```
CC=gcc
 CFLAGS=-I.
 DEPS = hellomake.h

 %.o : %.c $(DEPS)
         $(CC) -c -o $@ $< $(CFLAGS)

 hellomake : hellomake.o hellofunc.o
         $(CC) -o hellomake hellomake.o hellofunc.o
```

Cet ajout crée d'abord la macro DEPS, qui est l'ensemble de .h fichiers dont dépendent les fichiers .c. Puis on définit une règle qui s'applique à tous les fichiers se terminant par le suffixe .o. La règle dit que le fichier .o dépend de la version .c du fichier et des fichiers .h inclus dans la macro DEPS. La règle dit alors que pour générer le fichier .o, `make` doit compiler le fichier .c en utilisant le compilateur défini dans la macro CC. Le drapeau -c dit de générer le fichier objet, le `-o $@` dit de mettre la sortie de la compilation dans le fichier nommé sur le côté gauche du `:` , le `$<` est le premier élément de la liste des dépendances, et la macro CFLAGS est définie comme ci-dessus.

Comme dernière simplification, utilisons les macros spéciales `$@` et `$^` , qui sont les côtés gauche et droit du `:` , respectivement, pour rendre la règle de compilation globale plus générale. Dans l'exemple ci-dessous, tous les fichiers inclus doivent être répertoriés comme faisant partie de la macro DEPS, et tous les fichiers objets doivent être répertoriés comme partie de la macro OBJ.

Makefile 4

```
CC=gcc
 CFLAGS=-I.
 DEPS = hellomake.h
 OBJ = hellomake.o hellofunc.o

 %.o : %.c $(DEPS)
         $(CC) -c -o $@ $< $(CFLAGS)

 bonjourmake: $(OBJ)
```

```
        $(CC) -o $@ $^ $(CFLAGS)
```

Alors que se passe-t-il si nous voulons commencer à mettre nos fichiers .h dans une inclusion répertoire, notre code source dans un répertoire src et quelques bibliothèques dans un répertoire lib ? Aussi, pouvons-nous cacher d'une manière ou d'une autre ces fichiers .o ennuyeux qui traînent partout? La réponse, de bien sûr, est oui. Le makefile suivant définit les chemins d'accès à l'include et lib répertoires, et place les fichiers objet dans un obj sous-répertoire dans le répertoire src. Il a également une macro définie pour toutes les bibliothèques que vous souhaitez inclure, telles que la bibliothèque mathématique `-lm` . Ce makefile doit être situé dans le src annuaire. Notez qu'il comprend également une règle pour nettoyer votre répertoires source et objet si vous tapez `make clean` . La La règle .PHONY empêche `make` de faire quelque chose avec un fichier nommé nettoyer.

Makefile 5

```
IDIR =../inclure
 CC=gcc
 CFLAGS=-I$(IDIR)

 ODIR=obj
 LDIR =../lib

 LIBS=-lm

 _DEPS = hellomake.h
 DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))

 _OBJ = hellomake.o hellofunc.o
 OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))


 $(ODIR)/%.o : %.c $(DEPS)
        $(CC) -c -o $@ $< $(CFLAGS)

 bonjourmake: $(OBJ)
        $(CC) -o $@ $^ $(CFLAGS) $(LIBS)

 .PHONY : propre

 nettoyer:
        rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~
```

Alors maintenant, vous avez un makefile parfaitement bon que vous pouvez modifier pour gérer des projets logiciels de petite et moyenne taille. Vous pouvez ajouter plusieurs règles à un makefile ; vous pouvez même créer des règles qui

appellent d'autres règles. Pour plus d'informations sur les makefiles et le `make` fonction, consultez la Manuel GNU Make , qui vous en dira plus que vous n'avez jamais voulu savoir (vraiment).