eBook Gratuit

APPRENEZ makefile

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#makefile

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: makefile

It is an unofficial and free makefile ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official makefile.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec makefile

Remarques

Un *makefile* est un fichier texte qui contrôle le fonctionnement du programme `make` . Le programme `make` est généralement utilisé pour gérer la création de programmes à partir de leurs fichiers source, mais il peut généralement être utilisé pour gérer tout processus nécessitant la régénération de fichiers (ou de *cibles*) après la modification d'autres fichiers (ou *prérequis*). Le *fichier makefile* décrit la relation entre les cibles et les prérequis et spécifie également les commandes nécessaires pour mettre à jour la cible lorsqu'une ou plusieurs conditions préalables ont été modifiées. La seule façon que `make` détermine « de la date-ness » est en comparant le temps de modification des fichiers cibles et leurs conditions préalables.

Les Makefiles sont quelque peu uniques de plusieurs manières, ce qui peut être déroutant au départ.

Tout d'abord, un fichier makefile se compose de deux langages de programmation complètement différents dans le même fichier. La majeure partie du fichier est écrit dans une langue qui `make` peut comprendre: cette offre affectation de variables et de l' expansion, certaines capacités de préprocesseur (y compris d' autres fichiers, l' analyse conditionnelle des sections du fichier, etc.), ainsi que la définition des objectifs et de leur conditions préalables. De plus, chaque cible peut être associée à une *recette* qui spécifie quelles commandes doivent être invoquées pour que cette cible soit mise à jour. La recette est écrite sous forme de script shell (POSIX sh par défaut). Le programme `make` n'analyse pas ce script: il exécute un shell et transmet le script au shell à exécuter. Le fait que les recettes ne soient pas analysées par `make` , mais traitées par un processus shell distinct, est essentiel pour comprendre les makefiles.

Deuxièmement, un makefile n'est pas un langage procédural comme un script: comme `make` analyse le makefile, il construit un *graphe orienté en* interne où les cibles sont les nœuds du graphe et les relations préalables sont les arêtes. Ce n'est qu'après que tous les fichiers makefile ont été complètement analysés et que le graphe est terminé que `make` choisissez un nœud (cible) et que vous essayez de le mettre à jour. Afin de s’assurer qu’une cible est à jour, elle doit tout d’abord s’assurer que chacune des conditions préalables de cette cible est à jour, et ainsi de suite, de manière récursive.

Versions

prénom	Aussi connu sous le nom	Version initiale	Version	Date de sortie
POSIX faire		1992	IEEE Std 1003.1-2008, édition 2016	2016-09-30
NetBSD fait	bmake	1988	20160926	2016-09-26

prénom	Aussi connu sous le nom	Version initiale	Version	Date de sortie
GNU make	faire	1988	4.2.1	2016-06-10
SunPro fait	faire	2006		2015-07-13
MSVS nmake		2003	2015p3	2016-06-27

Examples

Makefile de base

Envisagez d'écrire un "bonjour le monde!" programme en c. Disons que notre code source est dans un fichier appelé source.c, maintenant pour exécuter notre programme, nous devons le compiler, généralement sous Linux (en utilisant gcc), nous aurions besoin de taper `$> gcc source.c -o output` où output est le nom de l'exécutable à générer. Pour un programme de base, cela fonctionne bien mais, à mesure que les programmes deviennent plus complexes, notre commande de compilation peut aussi devenir plus complexe. C'est là qu'un *Makefile* entre en jeu, les makefiles nous permettent d'écrire un ensemble assez complexe de règles sur la façon de compiler un programme et de le compiler simplement en tapant make sur la ligne de commande. Par exemple, voici un exemple possible Makefile pour l'exemple hello wold ci-dessus.

Makefile de base

Permet de créer un *fichier Makefile de* base et de l'enregistrer sur notre système dans le même répertoire que notre code source nommé *Makefile* . Notez que ce fichier doit être nommé Makefile, cependant le capitol M est facultatif. Cela dit, il est relativement standard d'utiliser un capitol M.

```
output: source.c
    gcc source.c -o output
```

Notez qu'il existe exactement un onglet avant la commande gcc sur la deuxième ligne (ceci est important dans les makefiles). Une fois que ce Makefile est écrit chaque fois que les types d'utilisateurs font (dans le même répertoire que le Makefile), make vérifiera si le fichier source.c a été modifié (vérifie l'horodatage) s'il a été modifié plus récemment la règle de compilation sur la ligne suivante.

Variables dans les Makefiles

Selon le projet, vous pouvez souhaiter introduire des variables dans votre fichier de création. Voici un exemple de Makefile avec des variables présentes.

```
CFLAGS = -g -Wall

output: source.c
    gcc $< $(CFLAGS) -o $@
```

Maintenant, explorons ce qui s'est passé ici. Dans la première ligne, nous avons déclaré une variable nommée CFLAGS qui contient plusieurs indicateurs communs que vous souhaiterez peut-être transmettre au compilateur. Notez que vous pouvez stocker autant de drapeaux que vous voulez dans cette variable. Ensuite, nous avons la même ligne que avant de dire à make de vérifier la source.c pour voir si elle a été modifiée plus récemment que la sortie, si oui, elle exécute la règle de compilation. Notre règle de compilation est la même que précédemment, mais elle a été raccourcie en utilisant des variables, la variable `$<` est intégrée à make (appelée variable automatique, voir *https://www.gnu.org/software/make/manual/ html_node / Automatic-Variables.html*) et il représente toujours la source. Dans ce cas, *source.c* . `$(CFLAGS)` est notre variable que nous avons définie auparavant, mais notons que nous avons dû mettre la variable entre parenthèses avec un $ devant comme ceci `$(someVariable)` . C'est la syntaxe pour dire à Make d'étendre la variable à ce que vous avez tapé auparavant. Enfin , nous avons le symbole $ @ pour , une fois de plus est une variable construite en faire, et il signifie simplement la cible de l'étape de compilation, donc dans ce cas , il est synonyme de *sortie.*

Nettoyer

Make clean est un autre concept utile pour apprendre à créer des fichiers. Permet de modifier le *Makefile* d'en haut

```
CFLAGS = -g -Wall
TARGETS = output

output: source.c
    gcc $< $(CFLAGS) -o $@

clean:
    rm $(TARGETS)
```

Comme vous pouvez le constater, nous avons simplement ajouté une règle de plus à notre *Makefile* et une variable supplémentaire contenant toutes nos cibles. C'est une règle quelque peu courante dans makefiles car elle vous permet de supprimer rapidement tous les binaires produits en tapant simplement `$> make clean` . En tapant make clean, vous indiquez au programme make d'exécuter la règle clean, puis make exécutera la commande rm pour supprimer toutes vos cibles.

J'espère que ce bref aperçu de l'utilisation de make vous aidera à accélérer votre flux de travail, les *Makefiles* peuvent devenir très complexes, mais avec ces idées, vous devriez pouvoir commencer à utiliser make et mieux comprendre ce qui se passe dans d'autres *Makefiles* . Pour plus d'informations sur l'utilisation de make, une excellente ressource est https://www.gnu.org/software/make/manual/ .

Définir des règles

Démarrage rapide

Une règle décrit quand et comment certains fichiers (**cibles de** la règle) sont créés. Il peut également servir à mettre à jour un fichier cible si l'un des fichiers requis pour sa création (**prérequis** de la cible) est plus récent que la cible.

Les règles suivent la syntaxe ci-dessous: (Notez que les *commandes* suivant une règle sont en retrait par une **tabulation**)

```
targets: prerequisites
        <commands>
```

où les *cibles* et les *prérequis* sont des noms de fichiers ou des noms réservés spéciaux et des *commandes* (le cas échéant) sont exécutés par un shell pour créer / reconstruire des *cibles* obsolètes.

Pour exécuter une règle, il vous suffit d'exécuter la commande `make` dans le terminal à partir du même répertoire que le *fichier Makefile* . L'exécution de `make` sans spécifier la cible exécute la première règle définie dans le *Makefile* . Par convention, la première règle du *Makefile* est souvent appelée *all* ou *default* , listant généralement toutes les cibles de build valides comme conditions préalables.

`make` exécute uniquement la règle si la cible est obsolète, ce qui signifie qu'elle n'existe pas ou que son heure de modification est antérieure à l'une de ses conditions préalables. Si la liste des prérequis est vide, la règle ne sera exécutée que lorsqu'elle sera appelée pour la première fois pour générer les cibles. Cependant, lorsque la règle ne crée pas de fichier et que la cible est une variable factice, la règle sera toujours exécutée.

GNU make

Règles de motif

Les règles de modèle permettent de spécifier plusieurs cibles et de créer des noms de prérequis à partir de noms de cible. Ils sont plus généraux et plus puissants que les règles ordinaires, car chaque cible peut avoir ses propres conditions préalables. Dans les règles de modèle, une relation entre une cible et une condition préalable est générée sur la base de préfixes, y compris les noms de chemin d'accès et les suffixes, ou les deux.

Imaginons que nous voulions construire les cibles `foo.o` et `bar.o` , en compilant les scripts C, respectivement `foo.c` et `bar.c` Cela pourrait être fait en utilisant les règles ordinaires ci-dessous:

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

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

où *la variable automatique* `$<` est le nom du premier prérequis et `$@` le nom de la cible (une liste complète des variables automatiques peut être trouvée ici).

Cependant, comme les cibles partagent le même suffixe, les deux règles ci-dessus peuvent désormais être remplacées par la règle de modèle suivante:

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

Règles implicites

Les *règles implicites* disent `make` comment utiliser les méthodes habituelles pour construire certains types de fichiers cibles, qui sont très souvent utilisés. `make` utilise le nom du fichier cible pour déterminer la règle implicite à appeler.

L'exemple de règle de modèle que nous avons vu dans la section précédente n'a pas besoin d'être déclaré dans un *fichier Makef,* car `make` a une règle implicite pour la compilation C. Ainsi, dans la règle suivante, les conditions préalables `foo.o` et `bar.o` seront `bar.o` en utilisant la règle implicite pour la compilation C, avant de construire `foo` .

```
foo : foo.o bar.o
    cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

Un catalogue de règles implicites et les variables utilisées par celles-ci peuvent être trouvées ici .

règle générique pour gzip un fichier

si un répertoire contient 2 fichiers:

```
$ ls
makefile
example.txt
```

et `makefile` contiennent le texte suivant

```
%.gz: %
    gzip $<
```

alors vous pouvez obtenir `example.txt.gz` en tapant dans le shell

```
$ make -f makefile example.txt.gz
```

le makefile ne contiennent qu'une seule règle qui instruisent comment *faire* pour créer un fichier dont l' extrémité nom avec *.gz* s'il y a un fichier avec le même nom mais le suffixe *.gz.*

makefile Bonjour tout le monde

C: \ makefile:

```
helloWorld :
[TAB]echo hello world
```

exécuter les résultats:

```
C:\>make
echo hello world
hello world
```

Remarque: [TAB] doit être remplacé par un onglet réel, stackoverflow remplace les tabulations par des espaces et les espaces ne sont pas utilisés de la même manière que les tabulations d'un fichier Make.

Lire Démarrer avec makefile en ligne: https://riptutorial.com/fr/makefile/topic/1793/demarrer-avec-makefile

Chapitre 2: Cible

Examples

Utiliser .PHONY pour des cibles autres que des fichiers

Utilisez `.PHONY` pour spécifier les cibles qui ne sont pas des fichiers, par exemple, `clean` ou `mrproper` .

Bon exemple

```
.PHONY: clean
clean:
    rm *.o temp
```

Mauvais exemple

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

Dans le bon exemple `make` sait que `clean` est pas un fichier, par conséquent , il ne cherchera pas si elle est ou non à jour et exécutera la recette.

Dans le mauvais exemple, `make` cherchera un fichier nommé `clean` . Si elle n'existe pas ou n'est pas à jour, elle exécutera la recette, mais si elle existe et est à jour, la recette ne sera pas exécutée.

Utiliser .PHONY pour les invocations récursives de la commande 'make'

L'utilisation récursive de make signifie l'utilisation de make en tant que commande dans un makefile. Cette technique est utile lorsqu'un grand projet contient des sous-répertoires, chacun ayant ses fichiers de configuration respectifs. L'exemple suivant aidera à comprendre l'avantage d'utiliser .PHONY avec make récursif.

```
/main
     |_ Makefile
     |_ /foo
            |_ Makefile
            |_ ... // other files
     |_ /bar
            |_ Makefile
            |_ ... // other files
     |_ /koo
            |_ Makefile
            |_ ... // other files
```

Pour exécuter le makefile du sous-répertoire depuis le makefile de main, le makefile principal comporterait une boucle comme indiqué ci-dessous (il existe d'autres moyens pour y parvenir,

mais cela est hors du sujet du sujet actuel)

```
SUBDIRS = foo bar koo

subdirs:
        for dir in $(SUBDIRS); do \
          $(MAKE) -C $$dir; \
        done
```

Cependant, cette méthode présente des pièges.

1. Toute erreur détectée dans une sous-marque est ignorée par cette règle. Elle continuera donc à générer le reste des répertoires, même en cas d'échec.
2. La capacité de make à exécuter une exécution parallèle de plusieurs cibles de build n'est pas utilisée car une seule règle est utilisée.

En déclarant les sous-répertoires en tant que cibles .PHONY (vous devez le faire car le sous-répertoire existe toujours, sinon il ne sera pas construit), ces problèmes peuvent être résolus.

```
SUBDIRS = foo bar koo

.PHONY: subdirs $(SUBDIRS)

subdirs: $(SUBDIRS)

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

Lire Cible en ligne: https://riptutorial.com/fr/makefile/topic/5542/cible

Chapitre 3: Les variables

Examples

Référencement d'une variable

Pour utiliser la valeur stockée dans une variable, utilisez le signe dollar suivi du nom de la variable entre parenthèses ou accolades.

```
x = hello
y = $(x)
# y now contains the value "hello"
y = ${x}
# parentheses and curly braces are treated exactly the same
```

Si le nom d'une variable ne comporte qu'un seul caractère, les parenthèses / accolades peuvent être omises (par exemple, `$x`). Cette pratique est utilisée pour les variables automatiques (voir ci-dessous), mais n'est pas recommandée pour les variables à usage général.

Variables simplement étendues

Les variables simplement développées se comportent comme des variables issues des langages de programmation traditionnels. L'expression du côté droit est évaluée et le résultat est stocké dans la variable. Si la partie droite contient une référence de variable, cette variable est développée avant que l'affectation ait lieu.

```
x := hello
y := $(x)
# Both $(x) and $(y) will now yield "hello"
x := world
# $(x) will now yield "world", and $(y) will yield "hello"
```

Une autre forme consiste à utiliser l'attribution à deux points:

```
x ::= hello
```

L'attribution des deux points et des deux points est équivalente. Le standard make POSIX ne mentionne que le formulaire `::=` , de sorte que les mises en œuvre avec une conformité stricte aux normes peuvent ne pas prendre en charge la version à un seul point.

Variables à extension récursive

Lorsque vous définissez une variable récursivement étendue, le contenu du côté droit est stocké tel quel. Si une référence de variable est présente, la référence elle-même est stockée (pas la valeur de la variable). Attendez de développer les références de variable jusqu'à ce que la variable soit réellement utilisée.

```
x = hello
y = $(x)
# Both $(x) and $(y) will now yield "hello"
x = world
# Both $(x) and $(y) will now yield "world"
```

Dans cet exemple, la définition de `y` est récursive. La référence à `$(x)` n'est pas développée tant que `$(y)` n'est pas développé. Cela signifie que chaque fois que la valeur de `x` change, la valeur de `y` change également.

Les variables récursivement étendues sont un outil puissant mais facilement incompris. Ils peuvent être utilisés pour créer des constructions qui ressemblent à des modèles ou des fonctions, ou même pour générer automatiquement des parties d'un fichier Make. Ils peuvent également être la source de problèmes difficiles à déboguer. Veillez à utiliser uniquement les variables récursivement étendues si nécessaire.

Variables Automatiques

Dans le contexte d'une règle individuelle, Make définit automatiquement un certain nombre de variables spéciales. Ces variables peuvent avoir une valeur différente pour chaque règle d'un fichier makefile et sont conçues pour simplifier l'écriture des règles. Ces variables ne peuvent être utilisées que dans la partie recette d'une règle.

Variable	La description
`$@`	Nom de fichier de la cible de la règle
`$%`	Le nom du membre cible, si la cible de la règle est une archive
`$<`	Nom de fichier du premier prérequis
`$^`	Liste de tous les prérequis
`$?`	Liste de tous les prérequis plus récents que la cible
`$*`	La "racine" d'une règle implicite ou de motif

L'exemple suivant utilise des variables automatiques pour générer une règle générique. Cela indique comment créer un fichier .o à partir d'un fichier .c portant le même nom. Comme nous ne connaissons pas le nom spécifique des fichiers affectés, nous utilisons `$@` comme espace réservé pour le nom du fichier de sortie et `$^` comme espace réservé pour la liste de prérequis (dans ce cas, la liste des fichiers d'entrée).

```
%.o: %.c
    cc -Wall $^ -c $@
```

Affectation variable conditionnelle

L'opérateur `?=` Est une extension qui se comporte comme `=` , sauf que l'affectation *ne* se produit que si la variable n'est pas déjà définie.

```
x = hello
x ?= world
# $(x) will yield "hello"
```

Ajout de texte à une variable existante

L'opérateur `+=` est une extension commune qui ajoute le contenu spécifié à la fin de la variable, séparés par un espace.

```
x = hello
x += world
```

Les références de variable dans la partie droite seront étendues si et seulement si la variable d'origine a été définie comme une variable simplement développée.

Lire Les variables en ligne: https://riptutorial.com/fr/makefile/topic/6191/les-variables

Chapitre 4: Makefile avancé

Examples

Construire à partir de différents dossiers sources vers différents dossiers cibles

Principales caractéristiques de ce Makefile:

- Détection automatique des sources C dans les dossiers spécifiés
- Plusieurs dossiers sources
- Plusieurs dossiers cibles correspondants pour les fichiers d'objet et de dépendance
- Génération automatique de règles pour chaque dossier cible
- Création de dossiers cibles lorsqu'ils n'existent pas
- Gestion des dépendances avec `gcc` : Construisez seulement ce qui est nécessaire
- Fonctionne sur `Unix` systèmes `Unix` et `DOS`
- Écrit pour GNU Make

Ce Makefile peut être utilisé pour construire un projet avec ce type de structure:

```
\---Project
    +---Sources
    |   +---Folder0
    |   |       main.c
    |   |
    |   +---Folder1
    |   |       file1_1.c
    |   |       file1_1.h
    |   |
    |   \---Folder2
    |           file2_1.c
    |           file2_1.h
    |           file2_2.c
    |           file2_2.h
    \---Build
        |   Makefile
        |   myApp.exe
        |
        +---Folder0
        |       main.d
        |       main.o
        |
        +---Folder1
        |       file1_1.d
        |       file1_1.o
        |
        \---Folder2
                file2_1.d
                file2_1.o
                file2_2.d
                file2_2.o
```

Makefile

```
# Set project directory one level above of Makefile directory. $(CURDIR) is a GNU make
variable containing the path to the current working directory
PROJDIR := $(realpath $(CURDIR)/..)
SOURCEDIR := $(PROJDIR)/Sources
BUILDDIR := $(PROJDIR)/Build

# Name of the final executable
TARGET = myApp.exe

# Decide whether the commands will be shwon or not
VERBOSE = TRUE

# Create the list of directories
DIRS = Folder0 Folder1 Folder2
SOURCEDIRS = $(foreach dir, $(DIRS), $(addprefix $(SOURCEDIR)/, $(dir)))
TARGETDIRS = $(foreach dir, $(DIRS), $(addprefix $(BUILDDIR)/, $(dir)))

# Generate the GCC includes parameters by adding -I before each source folder
INCLUDES = $(foreach dir, $(SOURCEDIRS), $(addprefix -I, $(dir)))

# Add this list to VPATH, the place make will look for the source files
VPATH = $(SOURCEDIRS)

# Create a list of *.c sources in DIRS
SOURCES = $(foreach dir,$(SOURCEDIRS),$(wildcard $(dir)/*.c))

# Define objects for all sources
OBJS := $(subst $(SOURCEDIR),$(BUILDDIR),$(SOURCES:.c=.o))

# Define dependencies files for all objects
DEPS = $(OBJS:.o=.d)

# Name the compiler
CC = gcc

# OS specific part
ifeq ($(OS),Windows_NT)
    RM = del /F /Q
    RMDIR = -RMDIR /S /Q
    MKDIR = -mkdir
    ERRIGNORE = 2>NUL || true
    SEP=\\
else
    RM = rm -rf
    RMDIR = rm -rf
    MKDIR = mkdir -p
    ERRIGNORE = 2>/dev/null
    SEP=/
endif

# Remove space after separator
PSEP = $(strip $(SEP))

# Hide or not the calls depending of VERBOSE
ifeq ($(VERBOSE),TRUE)
    HIDE =
else
    HIDE = @
endif
```

```
# Define the function that will generate each rule
define generateRules
$(1)/%.o: %.c
    @echo Building $$@
    $(HIDE)$(CC) -c $$(INCLUDES) -o $$(subst /,$$(PSEP),$$@) $$(subst /,$$(PSEP),$$<) -MMD
endef

.PHONY: all clean directories

all: directories $(TARGET)

$(TARGET): $(OBJS)
    $(HIDE)echo Linking $@
    $(HIDE)$(CC) $(OBJS) -o $(TARGET)

# Include dependencies
-include $(DEPS)

# Generate rules
$(foreach targetdir, $(TARGETDIRS), $(eval $(call generateRules, $(targetdir))))

directories:
    $(HIDE)$(MKDIR) $(subst /,$(PSEP),$(TARGETDIRS)) $(ERRIGNORE)

# Remove all objects, dependencies and executable files generated during the build
clean:
    $(HIDE)$(RMDIR) $(subst /,$(PSEP),$(TARGETDIRS)) $(ERRIGNORE)
    $(HIDE)$(RM) $(TARGET) $(ERRIGNORE)
    @echo Cleaning done !
```

Comment utiliser ce Makefile Pour adapter ce Makefile à votre projet, vous devez:

1. Changer la `TARGET` variable pour correspondre à votre nom cible
2. Modifier le nom des dossiers `Sources` et `Build` dans `SOURCEDIR` et `BUILDDIR`
3. Modifier le niveau de verbosité du Makefile dans le Makefile lui-même ou dans make call
4. Modifier le nom des dossiers dans `DIRS` pour correspondre à vos sources et créer des dossiers
5. Si nécessaire, changez le compilateur et les drapeaux

Listes compressées

GNU make

Cette fonction `pairmap` prend trois arguments:

1. Un nom de fonction
2. Première liste séparée par des espaces
3. Deuxième liste séparée par des espaces

Pour chaque tuple zippé dans les listes, il appellera la fonction avec les arguments suivants:

1. Tuple élément de la première liste
2. Tuple élément de la deuxième liste

Il étendra à une liste séparée par des espaces des extensions de fonction.

```
list-rem = $(wordlist 2,$(words $1),$1)
pairmap = $(and $(strip $2),$(strip $3),$(call \
    $1,$(firstword $2),$(firstword $3)) $(call \
    pairmap,$1,$(call list-rem,$2),$(call list-rem,$3)))
```

Par exemple, ceci:

```
LIST1 := foo bar baz
LIST2 := 1 2 3

func = $1-$2

all:
    @echo $(call pairmap,func,$(LIST1),$(LIST2))

.PHONY: all
```

Va imprimer `foo-1 bar-2 baz-3` .

Lire Makefile avancé en ligne: https://riptutorial.com/fr/makefile/topic/6154/makefile-avance

Chapitre 5: Règles de modèle GNU

Examples

Règle de base

Une règle de motif est indiquée par un seul caractère `%` dans la cible. Le `%` correspond à une chaîne non vide appelée la **tige** . La racine est alors substituée à chaque `%` qui apparaît dans la liste de prérequis.

Par exemple, cette règle:

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

Correspond à toute cible se terminant par `.o` . Si la cible était `foo.o` , la racine serait `foo` et compilerait `foo.c` à `foo.o` Les cibles et les prérequis sont accessibles à l'aide de variables automatiques.

Cibles correspondant à plusieurs règles de modèle

Si une cible correspond à plusieurs règles de modèle, make utilisera celle dont les conditions préalables existent ou peuvent être générées. Par exemple:

```
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
%.o: %.s
    $(AS) $(ASFLAGS) $< -o $@
```

Compilera `foo.c` pour `foo.o` ou assemblera `foo.s` à `foo.o` , en fonction de celui de `foo.c` ou `foo.s`

Si plusieurs règles ont des conditions préalables qui existent ou peuvent être générées, make utilisera la règle qui correspond à la plus courte racine. Par exemple:

```
f%r:
    @echo Stem is: $*
fo%r:
    @echo Stem is: $*
```

Utilisera la deuxième règle pour rendre la cible `foo.bar` , écho `Stem is: o.ba`

Si plusieurs règles correspondent à la plus courte racine, make utilisera la première dans le Makefile.

Répertoires dans les règles de modèle

Si le modèle cible ne contient pas de barre oblique, make supprime la partie de répertoire de la

cible qu'il tente de générer avant de la mettre en correspondance. Le répertoire sera alors placé devant la tige. Lorsque la tige est utilisée pour créer le nom cible et les conditions préalables, la partie répertoire est supprimée, la racine est substituée à la place de `%` et le répertoire est finalement placé devant la chaîne. Par exemple:

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

`lib/foobar.o` , avec:

- Stem (`$*`): `lib/bar`
- Nom de la cible (`$@`): `lib/foobar.o`
- Prérequis (`$<` , `$^`): `lib/foobar.c`

Dans cet exemple, une règle `lib/foo%.o` sur la règle `foo%.o` car elle correspond à une racine plus courte.

Règles de modèle avec plusieurs cibles

Les règles de pattern peuvent avoir plusieurs cibles mais, contrairement aux règles normales, la recette est responsable de la réalisation de toutes les cibles. Par exemple:

```
debug/%.o release/%.o: %.c
    $(CC) $(CFLAGS_DEBUG) -c $< -o debug/$*.o
    $(CC) $(CFLAGS_RELEASE) -c $< -o release/$*.o
```

Est une règle valide qui va créer des objets de débogage et de libération lorsque l'un d'eux doit être construit. Si nous écrivions quelque chose comme:

```
debug/%.o release/%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
```

Cela fonctionnerait quand un seul des programmes `debug/*.o` ou `release/*.o` est construit, mais il ne construira que la première cible (et considérera la seconde comme étant à jour) lorsque les deux doivent être construits.

Lire Règles de modèle GNU en ligne: https://riptutorial.com/fr/makefile/topic/6860/regles-de-modele-gnu

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec makefile	chen, Community, Dan , Eldar Abusalimov, kdhp, levif, MadScientist, mox, mxenoph, SketchBookGames
2	Cible	kdhp, madD7, TimF
3	Les variables	bta
4	Makefile avancé	Andrea Biondo, TimF
5	Règles de modèle GNU	Andrea Biondo, kdhp