make

Make est un logiciel qui construit automatiquement des fichiers, souvent exécutables, ou des bibliothèques à partir d'éléments de base tels que du code source. Il utilise des fichiers appelés **makefile** qui spécifient comment construire les fichiers cibles. À la différence d'un simple script shell, make exécute les commandes seulement si elles sont nécessaires. Le but est d'arriver à un résultat (logiciel compilé ou installé, documentation créée, etc.) sans nécessairement refaire toutes les étapes. make est particulièrement utilisé sur les plateformes UNIX.

Sommaire [masquer]

Histoire [modifier | modifier le code]

Dans les années 1970, la compilation des programmes devient de plus en plus longue et complexe, nécessitant de nombreuses étapes interdépendantes. La plupart des systèmes alors utilisés reposent sur des script shell, nécessitant de répéter toutes les étapes lors de la moindre correction. C'est dans ce contexte que Make fut développé par le docteur Stuart Feldman (en), en 1977 alors qu'il travaillait pour Bell Labs. En gérant les dépendances entre fichiers sources et fichiers compilés, Make permet de ne compiler que ce qui est nécessaire à la suite de la modification d'un fichier source.

Depuis le développement original, le programme a connu de nombreuses variantes. Les plus connues sont celle de BSD et celle du projet GNU - utilisée par défaut sur les systèmes Linux. Ces variantes apportent de nouvelles fonctionnalités, et ne sont généralement pas compatibles entre elles. Par exemple, les scripts prévus pour GNU Make peuvent ne pas fonctionner sous BSD Make.

Par la suite, d'autres outils sont apparus permettant la génération automatique des fichiers de configuration (Makefile) utilisés par make. Ces outils permettent d'analyser les dépendances (automake) ou la configuration du système (autoconf) pour générer des Makefile complexes et spécifiquement adaptés à l'environnement dans lequel les actions de compilation sont exécutées.

Make a depuis inspiré une variété de logiciels de gestion de compilation, spécifiques à certaines plateformes (rake, ant), ou généralistes comme Ninja dans les années 2010.

En 2003, le Dr Feldman a été récompensé du prix de l'ACM pour le développement de make[1].

Fonctionnement [modifier | modifier le code]

Le processus de compilation est décomposé en règles élémentaires du type « La cible A dépend de B et C. Pour fabriquer A il faut exécuter telle suite de commandes ». L'ensemble de ces règles est placé dans un fichier communément appelé Makefile. Les commandes consistent en des actions élémentaires de compilation, d'édition de liens, de génération de code.

Les dépendances correspondent à des fichiers, ceux-ci peuvent être des fichiers source, ou des résultats intermédiaires dans le processus de construction. La cible correspond généralement à un fichier, mais peut également être abstraite.

La construction de A s'effectue par la commande :

```
make A
```

Make va alors vérifier que les fichiers B et C existent et sont à jour, c'est-à-dire qu'ils n'ont pas de dépendances modifiées après leur création, et dans le cas contraire Make va commencer par la construction récursive de B et C. Make va ensuite appliquer les commandes de création de A dès qu'aucun fichier de ce nom existe, ou que le fichier A est plus ancien que B ou C.

Il est usuel d'utiliser une cible *all* résumant l'intégralité du projet, cette cible abstraite ayant pour dépendance l'ensemble des fichiers à construire. D'autres cibles sont d'usage courant : *install* pour exécuter les commandes d'installation du projet, *clean* pour effacer tous les fichiers générés fabriqués.

Make permet l'utilisation de règles de dépendance explicites, correspondant à des noms de fichiers, ou implicites, correspondant à des motifs de fichiers ; par exemple tout fichier d'extension .o peut être construit à partir d'un fichier de même nom d'extension .c par une commande de compilation.

Make représente le projet sous forme d'arbre de dépendance, et certaines variantes du programme permettent la construction en parallèle de plusieurs cibles lorsque celles-ci n'ont pas de dépendances entre elles.

Makefile [modifier | modifier le code]

Make cherche dans le répertoire courant le *makefile* à utiliser. Par exemple, le make de GNU cherche dans l'ordre, un fichier `GNUmakefile`, `makefile`, `Makefile`, puis exécute les cibles spécifiées (ou par défaut) pour ce fichier uniquement.

Le langage utilisé dans le makefile est de la programmation déclarative. À l'inverse de la programmation impérative, cela signifie que l'ordre dans lequel les instructions doivent être exécutées n'a pas d'importance.

Les règles [modifier | modifier le code]

Un makefile est constitué de *règles*. La forme la plus simple de règle est la suivante :

```
cible [cible ...]: [composant ...]
[tabulation] commande 1
        .
        .
        .
[tabulation] commande n
```

La « cible » est le plus souvent un fichier à construire, mais elle peut aussi définir une action (effacer, compiler…). Les « composants » sont des pré-requis nécessaires à la réalisation de l'action définie par la règle. Autrement dit, les « composants » sont les cibles d'autres règles qui doivent être réalisées avant de pouvoir réaliser cette règle. La règle définit une action par une série de « commandes ». Ces commandes définissent comment utiliser les composants pour produire la cible. Chaque commande doit être précédée par un caractère de tabulation.

Les commandes sont exécutées par un shell distinct ou par un interpréteur de lignes de commande.

Voici un exemple de makefile :

```
all: cible1 cible2
        echo ''all : ok''

cible1:
        echo ''cible1 : ok''

cible2:
        echo ''cible2 : ok''
```

À l'exécution de ce ficher makefile par l'intermédiaire de la commande `make all` ou de la commande `make`, la règle 'all' est exécutée. Elle nécessite la réalisation des composants 'cible1' et 'cible2' qui sont associés aux règles 'cible1' et 'cible2'. Ces règles seront donc exécutées automatiquement *avant* la règle 'all'. En revanche, la commande `make cible1` n'exécutera que la règle 'cible1'.

Afin de résoudre l'ordre dans lequel les règles doivent être exécutées, make utilise un tri topologique.

À noter qu'une règle ne comporte pas nécessairement de commande.

Les composants ne sont pas toujours des cibles vers d'autres règles, ils peuvent aussi être des fichiers nécessaires à la construction du fichier cible :

```
sortie.txt: fichier1.txt fichier2.txt
        cat fichier1.txt fichier2.txt > sortie.txt
```

L'exemple de règle ci-dessus construit le fichier sortie.txt en utilisant les fichiers fichier1.txt et fichier2.txt. En exécutant le makefile, make vérifie si le fichier sortie.txt existe et s'il n'existe pas, il le construira à l'aide de la commande définie dans la règle.

Les lignes de commande peuvent avoir un ou plusieurs des trois préfixes suivants :

- Un signe moins (-), spécifiant que les erreurs doivent être ignorées ;
- Une arobase (@), spécifiant que la commande ne doit pas être affichée dans la sortie standard avant d'être exécutée ;
- Un signe plus (+), la commande est exécutée même si make est appelé dans un mode « ne pas exécuter ».

Règle multiple [modifier | modifier le code]

Lorsque plusieurs cibles nécessitent les mêmes composants et sont construites par les mêmes commandes, il est possible de définir une règle multiple. Par exemple :

```
all: cible1 cible2

cible1: texte1.txt
        echo texte1.txt

cible2: texte1.txt
        echo texte1.txt
```

peut être remplacé par :

```
all: cible1 cible2

cible1 cible2: texte1.txt
        echo texte1.txt
```

À noter que la cible *all* est obligatoire, sans quoi seule la cible *cible1* sera exécutée.

Pour connaître la cible concernée, il est possible d'utiliser la variable *$@*, ainsi par exemple :

```
all: cible1.txt cible2.txt

cible1.txt cible2.txt: texte1.txt
        cat texte1.txt > $@
```

créera deux fichiers, *cible1.txt* et *cible2.txt*, ayant le même contenu que *texte1.txt*.

Les macros [modifier | modifier le code]

Définition [modifier | modifier le code]

Un makefile peut contenir des définitions de macros. Les macros sont traditionnellement définies en lettres majuscules :

```
MACRO = definition
```

Les macros sont le plus souvent appelées comme variables quand elles ne contiennent que des définitions de chaîne de caractères simples, comme `CC = gcc`. Les variables d'environnement sont également disponibles sous forme de macros. Les macros dans un makefile peuvent être écrasées par les arguments passés à make. La commande est alors :

```
    make MACRO="valeur" [MACRO="valeur" ...] CIBLE [CIBLE ...]
```

Les macros peuvent être composées de commandes shell en utilisant l'accent grave (`) :

```
    DATE  = ` date `
```

Il existe aussi des 'macros internes' à make :

- `$@` : fait référence à la cible.
- `$?` : contient les noms de tous les composants plus récents que la cible.
- `$<` : contient le premier composant d'une règle.
- `$^` : contient tous les composants d'une règle.

Les macros permettent aux utilisateurs de spécifier quels programmes utiliser ou certaines options personnalisées au cours du processus de construction. Par exemple, la macro `CC` est fréquemment utilisée dans les makefiles pour spécifier un compilateur C à utiliser.

Expansion [modifier | modifier le code]

Pour utiliser une macro, il faut procéder à son expansion en l'encapsulant dans `$()`. Par exemple, pour utiliser la macro `CC`, il faudra écrire `$(CC)`. À noter qu'il est aussi possible d'utiliser `${}`. Par exemple :

```
    NOUVELLE_MACRO = $(MACRO)-${MACRO2}
```

Cette ligne crée une nouvelle macro `NOUVELLE_MACRO` en soustrayant le contenu de `MACRO2` au contenu de `MACRO`.

Les types d'affectation [modifier | modifier le code]

Il existe plusieurs manières de définir une macro :

- Le = est une affectation par référence (on parle d'*expansion récursive*)
- Le := est une affectation par valeur (on parle d'*expansion simple*)
- Le ?= est une affectation conditionnelle. Elle n'affecte la macro que si cette dernière n'est pas encore affectée.
- Le += est une affectation par concaténation. Elle suppose que la macro existe déjà.

Les règles de suffixes [modifier | modifier le code]

Ces règles permettent de créer des makefile pour un type de fichier. Il existe deux types de règles de suffixes : les doubles et les simples.

Une règle de *double suffixes* est définie par deux suffixes : un *suffixe cible* et un *suffixe source*. Cette règle reconnaitra tout fichier du type « cible ». Par exemple, si le suffixe cible est '.txt' et le suffixe source est '.html', la règle est équivalente à '%.txt : %.html' (où % signifie n'importe quel nom de fichier). Une règle de *suffixes simples* n'a besoin que d'un suffixe source.

La syntaxe pour définir une règle de *double suffixes* est :

```
.suffixe_source.suffixe_cible :
```

Attention, une règle de suffixe ne peut pas avoir de composants supplémentaires.

La syntaxe pour définir la liste des suffixes est :

```
.SUFFIXES: .suffixe_source .suffixe_cible
```

Un exemple d'utilisation des règles de suffixes :

```
    .SUFFIXES: .txt .html

    # transforme .html en .txt
    .html.txt:
            lynx -dump $<   >   $@
```

La ligne de commande suivante transformera le fichier `fichier.html` en `fichier.txt` :

```
 make -n fichier.txt
```

L'utilisation des règles de suffixes est considérée comme obsolète car trop restrictive.

Exemple de Makefile [modifier | modifier le code]

Voici un exemple de Makefile :

```
 # Indiquer quel compilateur est à utiliser
 CC      ?= gcc

 # Spécifier les options du compilateur
 CFLAGS  ?= -g
 LDFLAGS ?= -L/usr/openwin/lib
 LDLIBS  ?= -lX11 -lXext

 # Reconnaître les extensions de nom de fichier *.c et *.o comme suffixes
 SUFFIXES ?= .c .o
 .SUFFIXES: $(SUFFIXES) .

 # Nom de l'exécutable
 PROG  = life

 # Liste de fichiers objets nécessaires pour le programme final
 OBJS  = main.o window.o Board.o Life.o BoundedBoard.o

 all: $(PROG)

 # Étape de compilation et d'éditions de liens
 # ATTENTION, les lignes suivantes contenant "$(CC)" commencent par un caractère TABULATION et non pas des espaces
 $(PROG): $(OBJS)
      $(CC) $(CFLAGS) $(LDFLAGS) $(LDLIBS) -o $(PROG) $(OBJS)

 .c.o:
      $(CC) $(CFLAGS) -c $*.c
```

Dans cet exemple, `.c.o` est une règle implicite. Par défaut les cibles sont des fichiers, mais lorsque c'est la juxtaposition de deux suffixes, c'est une règle qui dérive n'importe quel fichier se terminant par le deuxième suffixe à partir d'un fichier portant le même nom mais se terminant par le premier suffixe.

Pour parvenir à cette cible, il faut exécuter l'action, la commande `$(CC) $(CFLAGS) -c $*.c`, où `$*` représente le nom du fichier sans suffixe.

En revanche, `all` est une cible qui dépend de `$(PROG)` (et donc de `life`, qui est un fichier).

`$(PROG)` - c'est-à-dire `life` - est une cible qui dépend de `$(OBJS)` (et donc des fichiers `main.o window.o Board.o Life.o` et `BoundedBoard.o`). Pour y parvenir, `make` exécute la commande `$(CC) $(CFLAGS) $(LDFLAGS) $(LDLIBS) -o $(PROG) $(OBJS)`.

La syntaxe `CC ?= gcc`, ou plus généralement `<variable> ?= <valeur>`, affecte `<valeur>` à `<variable>` seulement si `<variable>` n'est pas encore initialisée. Si `<variable>` contient déjà une valeur, cette instruction n'a aucun effet.

Limitations [modifier | modifier le code]

Les limitations de make découlent directement de la simplicité des concepts qui l'ont popularisé : fichiers et dates. Ces critères sont en effet insuffisants pour garantir à la fois l'efficacité et la fiabilité.

Le critère de date associé à des fichiers, à lui seul, cumule les deux défauts. À moins que le fichier ne réside sur un support non réinscriptible rien n'assure que la date d'un fichier soit effectivement la date de sa dernière modification.

Si pour un utilisateur non privilégié[note 1], il est assuré que les données ne peuvent être postérieures à la date indiquée, la date exacte de leur antériorité n'est pas pour autant garantie.

Ainsi au moindre changement de date d'un fichier, toute une production peut-être considérée nécessaire s'il s'agit d'un source mais pire encore considérée inutile si au contraire il s'agit d'une cible.

- Dans le premier cas il y a perte d'efficacité.
- Dans le second cas il y a perte de fiabilité.

Si date et fichier restent pour l'essentiel nécessaires à tout moteur de production voulu fiable et optimal, ils ne sont pas non plus suffisants et quelques exemples particuliers suffisent à l'illustrer :

- Make en lui-même ignore complètement la sémantique des fichiers dont il assure le traitement, il en ignore tout simplement le contenu. Par exemple, aucune distinction n'est faite entre code effectif et commentaires. Ainsi, dans le cas de l'ajout ou même seulement de la modification d'un commentaire au sein d'un fichier C make considérera que bibliothèques ou programmes cibles qui en dépendent devront être reconstruits.
- Si le résultat final dépend des données en entrée, il dépend tout autant des traitements appliqués. Ces traitements sont décrits dans le fichier makefile. Rares sont les écritures de fichiers makefile qui prévoient d'invalider toute production antérieurement réalisée dans le cas de l'évolution du fichier makefile ; en effet pour ce faire il conviendrait de mettre systématiquement le fichier makefile en dépendance de toutes les règles de production qu'il définit lui-même. S'il n'est techniquement pas difficile d'envisager que cela puisse être directement réalisé par une variante de make, cela aurait pour effet de bord de toucher toutes les cibles même si elles ne sont pas concernées par les modifications opérées dans le makefile.
- Une variante de l'exemple précédent est le cas où des variables régissant la production sont positionnées soit par des variables d'environnement ou encore via la ligne de commande. Là encore, make n'a aucun moyen de détecter si ces variables touchent ou non la production et notamment quand ces variables désignent des options et non des fichiers.

Une autre limitation de make est qu'il ne génère pas la liste des dépendances et n'est pas capable de vérifier que la liste fournie soit correcte. Ainsi, le simple exemple précédent qui repose sur la règle `.c.o` est erroné : en effet, les fichiers objets ne sont pas seulement dépendants du fichier source `.c` associé, mais également de tous les fichiers du projet inclus par le fichier `.c`. Une liste de dépendances plus réaliste serait :

```
$(PROG): $(OBJS)
      $(CC) $(CFLAGS) $(LDFLAGS) $(LDLIBS) -o $(PROG) $(OBJS)
main.o: main.c Board.h BoundedBoard.h Life.h global.h
      $(CC) $(CFLAGS) -c main.c
window.o: window.c window.h global.h
      $(CC) $(CFLAGS) -c window.c
Board.o: Board.c Board.h window.h global.h
      $(CC) $(CFLAGS) -c Board.c
Life.o: Life.c Life.h global.h
      $(CC) $(CFLAGS) -c Life.c
BoundedBoard.o: BoundedBoard.c BoundedBoard.h Board.h global.h
      $(CC) $(CFLAGS) -c BoundedBoard.c
```

Il est raisonnable de considérer que les fichiers systèmes inclus (comme `stdio.h`) ne changent pas et de ne pas les lister comme dépendances. Au prix de l'écriture de parsers capable de produire des listes dynamiques de dépendances, certaines versions de make[note 2] permettent de contourner ce problème.

Ce sont pour ces raisons que les moteurs de productions de nouvelle génération se spécialisent dans le traitement de langages particuliers (sémantique du contenu des fichiers) ou sont encore couplés à une base de données dans laquelle sont enregistrées toutes les caractéristiques effectives de production (audit de production) des cibles (traçabilité).

Alternatives [modifier | modifier le code]

Il existe plusieurs alternatives à make :

- makepp : un dérivé de (GNU) make, mais qui offre en plus un analyseur extensible de commandes et de fichiers inclus afin de reconnaître automatiquement les dépendances. Les options de commandes changées et autres influences sont reconnues. Le grand problème de make récursif peut être facilement évité, pour garantir une construction correcte[2]
- clearmake est la version intégrée à ClearCase. Fortement couplé à la gestion des révisions des fichiers, il réalise et stocke un audit de toutes les fabrications. Ces audits lui permettent de s'affranchir de la problématique des dates, des variables d'environnement, des dépendances implicites ou cachées ainsi que celles des paramètres passés en ligne de commande (Cf. limitations).
- ant : plutôt lié au monde Java.
- rake : un équivalent en Ruby.
- SCons : complètement différent de make, il inclut certaines des fonctions d'outil de compilation comme autoconf. On peut utiliser Python pour étendre l'outil.
- Speedy Make utilise XML pour les makefiles, très simple à écrire, offre plus d'automatismes que make.
- mk : un équivalent de make, conçu originellement pour le système Plan 9 ; il est beaucoup plus simple que make et la façon dont il communique avec l'interprète de commande le rend beaucoup plus propre et tout aussi puissant ; il présente cependant l'inconvénient d'être incompatible avec make.

Notes et références [modifier | modifier le code]

Notes [modifier | modifier le code]

1. ↑ On supposera qu'un utilisateur privilégié est suffisamment responsable pour ne pas rétrograder de manière inconsidérée la date d'un fichier.
2. ↑ Par exemple avec GNU Make qui facilite la construction de liste dynamique même s'il ne fournit pas de tels parsers.

Références [modifier | modifier le code]

1. ↑ Doar 2005, p. 94.
2. ↑ Makepp Home PageEsperantoEnglish [archive]

Annexes [modifier | modifier le code]

Bibliographie [modifier | modifier le code]

- [Doar 2005] (en) Matthew Doar, *Practical Development Environments*, O'Reilly Media, 2005 (ISBN 978-0-596-00796-6).

Sur les autres projets Wikimedia :
Make, sur Wikibooks

Articles connexes [modifier | modifier le code]

- Automake
- MSBuild
- CMake

Liens externes [modifier | modifier le code]

- (en) Site de l'implémentation GNU [archive]
- (fr) Introduction à make [archive] (exemples en Java)
- (fr) Compilation séparée et make [archive] (exemples en C)
- (fr) Aide mémoire – Makefile [archive]
- (fr) Compilez sous GNU/Linux ! (OpenClassrooms) [archive]

Portail de l'informatique

Catégories : Commande Unix | Moteur de production [+]