DevOps

Construction automatisée de logiciels: Make et Ant

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

2026

Agenda

make

Ant

Agenda

make

Ant

En 2 mots

- Le programme (GNU) `make` exécute un Makefile
- Le Makefile défini un ensemble de règles
  - Définissent les relations de dépendances entre les fichiers sources et les objets cibles (à générer).
  - Les règles sont exécutées de manière récursive pour atteindre un but.
- Différence avec un script?
  - `make` sait si une cible est à jour.
  - Permet de n'exécuter que ce qui est nécessaire.

Motivations

- Automatiser la compilation de programmes (modulaires)
  - A partir de la dernière version des sources mise à jour.
- Automatiser le processus d'installation d'un logiciel
  - Compilation
  - Installation de l'exécutable dans le répertoire approprié
  - Génération de la documentation
  - Automatisation de tests
  - Suppression de fichiers inutiles
  - ...
- Gérer des sources dans différents langages
- Simplifier le déclenchement de suites d'opérations souvent répétées sur des fichiers

La commande make

man make

- `make`
  - Applique les règles définies dans `./Makefile`
  - Exécute par défaut la première cible définie
- `make cible`
  - Exécute la règle cible définie dans `./Makefile`
- `make -f autre_fichier`
  - Exécute les règles définies dans le fichier `autre_fichier`
- `make -n`
  - Affiche les commandes qui seraient exécutées (mais ne les exécute pas)

Les règles

Une règle dans un Makefile est de la forme:

```
cible: dépendances
	commandes
```

- Les lignes de commandes commencent par une `tabulation`.
- cible représente soit un fichier à générer, soit un identifiant
- dépendances est composé d'identifiants et/ou de noms de fichiers
- commandes correspond aux actions à effectuer

Un premier exemple

Fichier Makefile:

```
hello:
    echo "ce cours est très intéressant" >remarques.txt
    cat remarques.txt
```

Un premier exemple

Fichier Makefile:

```
hello:
    echo "ce cours est très intéressant" >remarques.txt
    cat remarques.txt
```

- `make`

  ```
  $ make
  echo "ce cours est très intéressant" >remarques.txt
  cat remarques.txt
  ce cours est très intéressant
  ```

- `make hello` donne le même résultat
- `make bye` ?

  ```
  $ make bye
  make: *** No rule to make target 'bye'.  Stop.
  ```

Cibles standards (GNU)

non exhaustif[1]

- all
  - Compile l'ensemble du programme
  - Les dépendances correspondent à l'ensemble des fichiers à produire
  - Doit être la règle par défaut
- install
  - Compile le programme et copie les exécutables/librairies à l'endroit approprié
- clean
  - La commande associée supprime tous les fichiers intermédiaires
  - Pas de dépendances

[1] `www.gnu.org/prep/standards/html_node/Standard-Targets.html`

Contenu d'un Makefile

- # des lignes de commentaires
- lignes vides
- règles standards avec cible réelle
- règles standards avec pseudo-cible (identifiant)
- déclarations de variables
- règles de suffixe
- règles "spéciales"

Règles standards

```
cible : fichier_1 fichier_2 fichier_3
< tab > commande
< tab > commande
< tab > ...
```

- Les commandes seront exécutées si des dépendances ont été modifiées ultérieurement à la dernière modification de cible
- Avant d'exécuter ces commandes, `make` va éventuellement recréer les dépendances (exécution récursive)

Un nouvel exemple

```
all: hello_world

hello_world: hello_world.o main.o
    gcc hello_world.o main.o -o hello_world

hello_world.o: hello_world.c
    gcc -c hello_world.c -Wall -g

main.o: main.c hello_world.h
    gcc -c main.c -Wall -g

clean:
    rm -rf *.o hello_world
```

Les variables

Utilisation de variables

- Définition: `NOM=valeur`
- Utilisation: `$(NOM)` ou `${NOM}`
- Variables imbriquées: `VAR= $(VAR1) $(VAR2)`

Variables standards en C/C++

- `CC`: désigne le compilateur utilisé
- `CFLAGS`: regroupe les options de compilation
- `LDFLAGS`: regroupe les options d'édition de liens
- `EXEC` ou `TARGET`: regroupe les exécutables

Variables définies à la ligne de commande: `make "EXEC = hello"`

Mise à jour de l'exemple

```
CC=gcc
CFLAGS= -Wall -g
LDFLAGS=
EXEC= hello_world

all: $(EXEC)

hello_world: hello_world.o main.o
    $(CC) hello_world.o main.o -o hello_world $(LDFLAGS)

hello_world.o: hello_world.c
    $(CC) -c hello_world.c $(CFLAGS)

main.o: main.c hello_world.h
    $(CC) -c main.c $(CFLAGS)

clean:
    rm -rf *.o $(EXEC)
```

Variables automatiques

Variables par défaut recalculées pour chaque règle[1]:

- `$@`: le nom de la cible
- `$<`: le nom de la première dépendance
- `$?`: le nom de toutes les dépendances qui sont plus récentes que la cible.
- `$^`: le nom de toutes les dépendances
- `$*`: Le nom du pattern matchant la cible dans un pattern statique

[1] `www.gnu.org/software/make/manual/html_node/Automatic-Variables.html`

Mise à jour de l'exemple

```
CC=gcc
CFLAGS= -Wall -g
LDFLAGS=
EXEC= hello_world

all: $(EXEC)

hello_world: hello_world.o main.o
    $(CC) $^ -o $@ $(LDFLAGS)

hello_world.o: hello_world.c
    $(CC) -c $^ $(CFLAGS)

main.o: main.c hello_world.h
    $(CC) -c $< $(CFLAGS)

clean:
    rm -rf *.o $(EXEC)
```

Règles d'inférence

- Objectif: créer des règles génériques
- Utilisation de '%'
- Exemple de pattern:
  - %.c
  - s.%.c
- Utilisation d'un pattern dans la cible
  - Règles génériques: pattern dans la cible et dans les dépendances
  - Pattern dans les dépendances: même substitution que dans la cible
  - Possibilité de définir des dépendances supplémentaires spécifiques

Mise à jour de l'exemple

```
CC=gcc
CFLAGS= -Wall -g
LDFLAGS=
EXEC= hello_world

all: $(EXEC)

hello_world: hello_world.o main.o
    $(CC) $^ -o $@ $(LDFLAGS)

main.o: hello_world.h

%.o: %.c
    $(CC) -c $^ $(CFLAGS)

clean:
    rm -rf *.o $(EXEC)
```

Règles d'inférences (old fashion)

Considéré obsolète

Défini des règles fondées sur des doubles suffixes[1].

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

- Fonctionne car ".c" et ".o" sont dans la liste des suffixes par défaut
  - Les suffixes par défaut sont définis par la cible spéciale `.SUFFIXES`
  - La variable `$(SUFFIXES)` contient la liste des suffixes définis.
- Ajouter des suffixes à la liste

```
.SUFFIXES: .txt
```

[1] `www.gnu.org/software/make/manual/html_node/Suffix-Rules.html`

Exemple avec des règles de suffixe

Compiler du Latex

```
all: myfile.pdf

.SUFFIXES: .tex .pdf

.tex.pdf:
    pdflatex $<
```

Les fonctions

```
$(fonction arguments)
```

Quelques fonctions principales[1]

- Fonction *wildcard*
  - `SRC=$(wildcard *.c)`
  - `SRC` va contenir la liste des fichiers .c du répertoire courant
- Pattern substitution (`patsubst`)
  - `OBJS=$(patsubst %.c,%.o, $(SRC))`
- Cas spécifique de substitution de suffixes
  - `$(var:suffix=replacement)`
  - `OBJS=$(SRC:.c=.o)`

[1] `www.gnu.org/software/make/manual/html_node/Functions.html`

Mise à jour de l'exemple

```
CC=gcc
CFLAGS= -Wall -g
LDFLAGS=
EXEC= hello_world
SRC=$(wildcard *.c)
OBJ=$(SRC:.c=.o)

all: $(EXEC)

hello_world: $(OBJ)
    $(CC) $^ -o $@ $(LDFLAGS)

main.o: hello_world.h

%.o: %.c
    $(CC) -c $^ $(CFLAGS)

clean:
    rm -rf *.o $(EXEC)
```

Les règles par défaut

`make` a un nombre de règles implicites définies par défaut[1]:

- Compilation de programmes en C

```
$(CC) $(CPPFLAGS) $(CFLAGS) -c
```

[1] www.gnu.org/software/make/manual/html_node/Catalogue-of-Rules.html

Mise à jour de l'exemple

Déconseillé

```
CC=gcc
CFLAGS= -Wall -g
LDFLAGS=
EXEC= hello_world
SRC=$(wildcard *.c)
OBJ=$(SRC:.c=.o)

all: $(EXEC)

hello_world: $(OBJ)
    $(CC) $^ -o $@ $(LDFLAGS)

main.o: hello_world.h

clean:
    rm -rf *.o $(EXEC)
```

Conditions

On peut introduire des tests de condition afin de paramétrer des variables

```
ifeq "$(OS)" "linux 32"
    ARCH=linux
endif

ifeq ($(ARCH),linux)
    CC = gcc
else
    CC = cc
endif
```

Il existe aussi `ifneq`.

Hiérarchie de Makefile

Dans un projet avec plusieurs modules, on doit compiler des fichiers dans des répertoires sources différents.

Approche hiérarchique

- Un Makefile dans le répertoire principal
- Un Makefile dans chaque répertoire source
  - "`make -C`" specifies the directory to move to before reading the makefiles.
  - Quand une ligne commence par `@`, l'affichage de cette ligne est supprimé.

```
all : module1 module2

module1:
    @make -C ./module1

module2:
    @make -C ./module2
```

Exécuter plusieurs commandes dépendantes

Je dois mettre à jour la variable d'environnement `LD_LIBRARY_PATH` avant d'exécuter un test.

```
test:
    export LD_LIBRARY_PATH=path_to_my_lib:$(LD_LIBRARY_PATH)
    run_test param1 param2
```

Problème

Exécuter plusieurs commandes dépendantes

Je dois mettre à jour la variable d'environnement `LD_LIBRARY_PATH` avant d'exécuter un test.

```
test:
    export LD_LIBRARY_PATH=path_to_my_lib:$(LD_LIBRARY_PATH)
    run_test param1 param2
```

Problème

Par défaut, chaque *commande* est exécutée dans un nouveau sous-shell.

- Backslash peut être utilisé pour diviser une séquence d'instructions
- La séquence est transmise au shell avec la tabulation au début de chaque ligne en moins

```
test:
    export LD_LIBRARY_PATH=path_to_my_lib:$(LD_LIBRARY_PATH); \
    run_test param1 param2
```

Bilan

Points forts

- Règles de dépendance entre les fichiers
- Prise en compte de la date de modification pour déterminer la nécessité d'effectuer des actions

Inconvénients

- Convivialité moyenne
- Syntaxe difficile
- Utilisation d'astuces
- Si l'objectif est surtout de simplifier le déclenchement d'une suite d'actions, un script est parfois plus commode
- Hétérogénéité des commandes difficile à gérer (Linux vs Windows)

Aller plus loin

- Peu de chances que vous ayez de gros makefile à écrire
- Sur les gros projets, les Makefile sont générés à partir de fichiers de configurations.
  - CMake
  - GNU Autotools

Agenda

make

Ant

Ant

Présentation

- Automatiser le compilation et le déploiement d'applications Java
- Remplacement de make
- Peut être utilisé pour d'autres langages
- Peut être employé pour automatiser tout processus qui peut être décrit en terme de cibles et de tâches à exécuter
  - Description de graphes de dépendances entre les cibles

Ant et la portabilité

Portabilité

- Implémenté en Java
  - Les tâches sont mises en œuvre en Java
  - Indépendantes du système d'exploitation
  - Facilement extensible: définition de nouvelles classes Java
- Les fichiers de configuration sont décrits en XML
  - eXtensible Markup Language
  - Langage à balise (< ... >)
  - Extensible
- Ant est un projet open source d'Apache

Exemple de XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- address info for Ken Anderson -->
<person surname="Anderson" name="Ken">
    <address>
        <street>University of Colorado<br />
                Department of Computer Science<br />
                430 UCB</street>
        <city>Boulder</city>
        <state>CO</state>
    </address>
</person>
```

- <?...? >: processing instruction (information pour l'application lisant le document)
- <!-- ...-->: Commentaire
- <person> ... </person>: Défini un élément
  - Peut avoir des fils (arbre)
  - Peut avoir des attributs
  - Les feuilles contiennent du texte

A propos de Ant

Points forts

- Portable
- Très nombreuses tâches déjà implémentées
- Largement répandu et intégré aux IDE (Eclipse . . .)
- Syntaxe rigoureuse d'XML

Points faibles

- Verbeux
- Dépendances de taches (non temporelles)
- Java-Centric

Exécuter Ant

- Exécution à la ligne de commande
  - `ant [options] [cibles]`
  - `ant compile`
- Par défaut, exécute le fichier `build.xml`
- Exemple d'utilisation d'options
  - Définir le fichier de configuration à exécuter
  - `ant -buildfile monbuild.xml compile`

build.xml

Le `build.xml` définit l'enchaînement à suivre pour la construction d'un projet

Un projet comporte des cibles (targets)

- Correspond à des activités telles que la compilation, l'installation, l'exécution, ...

Chaque cible est composée de tâches (task)

- Exécutées lorsque la cible est exécutée
- A des dépendances avec d'autres cibles
  - Exécutées au préalable

Un projet peut aussi inclure des propriétés

- Équivalant des variables des Makefile

Project

- Le tag project définit le projet sur lequel on travaille
- Contient 3 attributs
  - name: Nom du projet
  - default: La cible à exécuter par défaut
  - basedir: Le répertoire à partir duquel on s'exécute
- On peut inclure en plus une description du projet

Exemple

```
<project name="Sample Project" default="compile" basedir=".">

  <description>
    A sample build file for this project
  </description>

</project>
```

Properties

Le fichier `build.xml` peut définir des constantes (properties) qui peuvent ensuite être utilisées dans tout le projet.

- Simplifie la maintenance de gros fichiers `build.xml`
- Un projet peut avoir un ensemble de properties.

Syntaxe

- Associe une `value` à un `name`
  `<property name="src.dir" value="./src"/>`
- Récupérer la valeur d'une propriété: `${src.dir}`

Properties

L'ordre de définition des properties est important:

- Seule la première définition est prise en compte
- Les properties définies à la ligne de commande sont plus prioritaires

Exemple de build.xml

```
<project name="Sample Project" default="compile" basedir=".">

  <description>
    A sample build file for this project
  </description>

  <!-- global properties for this build file -->
  <property name="source.dir" location="src"/>
  <property name="build.dir" location="bin"/>
  <property name="doc.dir" location="doc"/>
  <property name="apidoc.dir" value="${doc.dir}/api"/>

</project>
```

Targets

- Les `target` définissent les règles du fichier de build
- Correspondent aux étapes majeures de la construction (par ex: compiling, creating archives, testing, ...)

Définition d'une target

- name (attribut obligatoire)
- depends (optionnel)
  - Liste de targets dont la target dépend
  - Sont exécutées avant
- description (optionnel)

Exemple de build.xml

```
<project name="Sample Project" default="compile" basedir=".">

  ...

  <!-- set up some directories used by this project -->
  <target name="init" description="setup project directories">
  </target>

  <!-- Compile the java code in src dir into build dir -->
  <target name="compile" depends="init" description="compile java sources">
  </target>

  <!-- Generate javadocs for current project into docs dir -->
  <target name="doc" depends="init" description="generate documentation">
  </target>

  <!-- Delete the build & doc directories and Emacs backup (*~) files -->
  <target name="clean" description="tidy up the workspace">
  </target>

</project>
```

Ordre d'exécution des target

Exécution une et une seule fois de A puis B puis C puis D en appelant "ant D":

```
<target name="A"/>
<target name="B" depends="A"/>
<target name="C" depends="B"/>
<target name="D" depends="C,B,A"/>
```

Condition d'exécution if/unless:

```
<target name="build.windows" if="os.is.windows"/>
<target name="build.no.windows" unless="os.is.windows"/>
```

Tasks

- Chaque target est constitué d'un ensemble de tasks
- Une task représente une action à exécuter
- Une task a un certains nombres de paramètres définis par:
  - Des attributs
  - Des sous éléments

Tasks

Ant fournit un grand nombre de tâches par défaut qui correspondent aux traitements courants de gestion d'un logiciel:

- Créer un répertoire
- Compiler du code source Java
- Exécuter l'outil Javadoc sur des fichiers
- Créer un jar
- Supprimer fichiers/répertoires
- Et bien plus encore:
  - `http://ant.apache.org/manual/tasksoverview.html`

Tasks

Possibilité d'ajouter:

- Des tâches optionnelles
  - `http://ant.apache.org/external.html`
  - `http://ant-contrib.sourceforge.net`
- Des tâches propriétaires
- Vos propres tâches

Target init

```
<project name="Sample Project" default="compile" basedir=".">

  ...

  <!-- set up some directories used by this project -->
  <target name="init" description="setup project directories">
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${doc.dir}"/>
  </target>

  ...

</project>
```

Target compile

```
<project name="Sample Project" default="compile" basedir=".">

  ...

  <!-- Compile the java code in ${src.dir} into ${build.dir} -->
  <target name="compile" depends="init" description="compile java sources">
    <javac srcdir="${source.dir}" destdir="${build.dir}"/>
  </target>

  ...

</project>
```

Target doc

```
<project name="Sample Project" default="compile" basedir=".">

  ...

  <!-- Generate javadocs for current project into ${doc.dir} -->
  <target name="doc" depends="init" description="generate documentation">
    <javadoc sourcepath="${source.dir}" destdir="${doc.dir}"/>
  </target>

  ...

</project>
```

Target clean

```
<project name="Sample Project" default="compile" basedir=".">
  ...
 <!-- Delete the build & doc directories and Emacs backup (*~) files -->
  <target name="clean" description="tidy up the workspace">
    <delete dir="${build.dir}"/>
    <delete dir="${doc.dir}"/>
    <delete>
      <fileset defaultexcludes="no" dir="${source.dir}" includes="**/*~"/>
    </delete>
  </target>
  ...
</project>
```

Les ensembles de fichiers

- Tag `fileset`
- Attributs:
  - dir: Définit le répertoire de départ de l'ensemble de fichiers
  - includes: Liste des fichiers à inclure
  - excludes: Liste des fichiers à exclure
- L'expression "**/" permet de désigner tous les sous-répertoires du répertoire défini dans l'attribut dir

Exemple

```
<fileset dir="src" includes="**/*.java">
```

Les ensembles de motifs

- Tag `patternset`
- Utilisé dans un fileset
- Attributs:
  - id: Définit un identifiant pour le pattern qui pourra ainsi être réutilisé dans plusieurs filesets
  - includes: Liste des fichiers à inclure
  - excludes: Liste des fichiers à exclure
  - refid: Demande la réutilisation d'un ensemble dont l'identifiant est fourni comme valeur

Exemple

```
<fileset dir="src">
  <patternset id="source_code">
    <includes="**/*.java"/>
  </patternset>
</fileset>
```

Les listes de fichiers

- Tag `filelist`
- Attributs:
  - id: Définit un identifiant pour la liste qui pourra ainsi être réutilisé
  - dir: Définit le répertoire de départ de la liste de fichiers
  - files: liste des fichiers séparés par des virgules
  - refid: Demande la réutilisation d'une liste dont l'identifiant est fourni comme valeur

Exemple

```
<filelist dir="texte" files="fichier1.txt,fichier2.txt" />
```

Filelist vs fileset

- Un fileset est un filtre sur un ensemble de fichiers existants dans le système de fichiers
- Une filelist peut inclure des fichiers qui existent ou pas.

Les éléments de chemin[1]

- Tag `pathelement`
- Permet de définir un élément qui sera ajouté à la variable `classpath`
- Attributs:
  - location: Définit un chemin d'une ressource qui sera ajoutée

Exemple

```
<classpath>
  <pathelement location="bin/mabib.jar">
  <pathelement location="lib/">
</classpath>
```

[1] `http://ant.apache.org/manual/using.html#path`

Ant et Eclipse

- Eclipse supporte Ant nativement
  - Pas besoin d'installer/configurer Ant séparément
- Eclipse a une "vue Ant"
  - Window $\rightarrow$ Show View $\rightarrow$ Ant
- Glisser-déposer un fichier `build.xml` dans la vue
- Double-clic sur une target pour exécuter

Références

- Notes de D. Donsez
- Notes de K. Anderson sur Ant[1]
- `http://www.jmdoudoux.fr/java/dej/chap-ant.htm`

[1] `www.cs.colorado.edu/~kena/classes/3308/f06/lectures/10/`