Automatiser des tâches durant le développement

Comparaison de make et de ant

Théophile Mandon Thomas Sauvajon

theophile.mandon@etud.univ-montp2.fr

thomas.sauvajon@gmail.com

Université Montpellier 2

Le 11 décembre 2014

Sommaire

Introduction

- Lors d'un projet c'est compliqué d'allier rapidité et efficacité, surtout pour certaines tâches répétitives comme la compilation.
- Il existe des méthodes pour palier à ce problème mais la plupart du temps elles ont aussi des inconvénients.
- Dans cet exposé nous voulons vous montrer certaines de ses méthodes et comment les exploiter au maximum grâce à des outils spécialisés.

Définition

Définition

Technique qui consiste à découper les gros programmes en différentes unités de compilation pour en maîtriser la complexité. Les différentes unités sont compilées séparément, et (éventuellement) un éditeur de liens est chargé de transformer les modules objets en programme exécutable.

En pratique

- C'est par exemple quand on compile plusieurs fichiers .c en .o puis qu'on fait l'édition de liens pour faire un exécutable.
- On ne peut pas faire ça avec des IDE car on compile tout d'un coup, même si on bénéficie à la fois des fichiers objets et des fichiers exécutables.

Compréhension

- On a plus de fichiers ce qui permets de classer plus facilement son travail.
- Permet de faire un listing plus lisible.
- Permet de donner des noms de fichiers compréhensibles.

Programmation modulaire

Définition

Le concept de décomposer une grosse application en plusieurs modules, groupes de fonctions, de méthodes et de traitement, pour pouvoir les développer indépendamment, et les réutiliser dans d'autres applications.

Comme ça on peut utiliser des travaux déjà faits ou on pourra se servir dans le futur de ce qu'on a fait.

Rapidité

- La rapidité de compilation est augmentée car on a autant ou moins de choses à compiler.
- La rapidité d'accès au code est augmentée car c'est mieux organisé.
- Et la rapidité de maintenance du code est augmentée du fait des deux raisons citées ci dessus.

Inconvénients

- Les commandes pour compiler changent, il y a donc deux solutions possibles :
  - Soit il faut les réécrire à chaque fois.
  - Soit il faut chercher dans son historique de commandes.
- Ce qui finalement peut ralentir le projet au lieu de l'accélérer.

Le logiciel et ce qu'il fait

- Make est un logiciel qui construit automatiquement et intelligemment des fichiers (exécutables ou objets par exemple).
- Il s'utilise avec un fichier Makefile dans lequel on spécifie ce qu'on veut faire.
- Maintenant il existe des logiciels pour générer des Makefile comme autoconf ou cmake.

Histoire

- Il fut développé par Mr Stuart Feldman en 1977.
- En 2003 il a reçu un prix pour l'avoir développé.
- Plusieurs dérivés ont été créé rapidement comme celui de GNU par exemple.

Structure

Structure générale

Cible : dépendance(s)
    commande(s)

- La cible est généralement le nom du fichier qu'on veut créer ou de l'action qu'on veut faire, et de la commande qu'on va utiliser dans le terminal : make cible
- La commande est la ligne de commande dont on veut créer le "raccourci".

Les dépendances

- Les dépendances sont là pour éviter les compilations inutiles : on fait la commande que si la cible est plus ancienne que les dépendances.
- En effet, make va comparer les dates de dernière modifications des deux fichiers et va faire la commande que si on a modifié le code source après avoir crée l'objet.

Généralités sur les commandes

- Il faut mettre une tabulation avant les commandes sinon make ne fonctionne pas.
- Les commandes sont à écrire en shell normalement, mais si on utilise make (ou un dérivé) sous un OS autre que linux il faut s'adapter.
- Si la commande fait plus d'une ligne il faut rajouter un backslash (\ ) à la fin de chaque ligne.

Exemples

Exemple simple

```
main.o :
    gcc -Wall -c main.c -o main.o
```

Exemple un peu plus compliqué

```
main.o : main.c
    gcc -Wall -c main.c -o main.o
```

Lancer le programme

- Il faut d'abord créer le makefile avec les commandes qu'on veut et généralement on le place dans le dossier où se situe le code source.
- Il suffit de faire make cible dans le dossier ou ce situe le makefile.
- Le logiciel fonctionne aussi si on écrit que make, dans ce cas là il lance la première commande qu'il trouve dans le makefile.

Macro

Définition

Une macro est l'association d'un texte de remplacement à un identificateur, tel que l'identificateur est remplacé par le texte dans tout usage ultérieur.

- Traditionnelement les macro sont écrites en majuscules.
- Elles sont utilisées dans les makefile pour pouvoir créer des makefile plus généraux par exemple.
- On peut passer des commandes shell en macro si elles sont précédées de l'accent grave `

Les affectations

- Il existe 4 types d'affectation dans les makefile :
  - Une affectation par référence : le =
  - Une affectation par valeur : le :=
  - Une affectation conditionnelle (on affecte la valeur que si la variable n'est pas initialisée) : le ?=
  - Une affectation par concaténation (qui suppose que la variable existe déjà) : le +=

Se servir des macro

- On définit une macro par NOM affectation valeur
- La valeur peut contenir des macros.
- Si on veut qu'une macro utilise sa propre valeur on utilise l'affectation par concaténation.
- Pour appeller une macro on fait $(Nom de la macro)

Caractères spéciaux

- Les commentaires se font avec le caractère '#'.
- Pour dire n'importe lequel on utilise le caractère '%'
- Pour désigner le nom de la cible on fait $@
- Pour la première dépendance on fait $<, pour toutes $^
- Bien sur, pour les commandes on peut utiliser les caractères jokers du shell.

Exemple

```
clean :
    rm *.o # Ceci est un commentaire
```

Exemple plus complet

Makefile

```
CC = gcc
FLAGS = -Wall
test : test.o
    $(CC) -o $@ $<
test.o : test.c
    $(CC) -o $@ -c $< $(FLAGS)
```

Si on veut changer les options de compilation on peut faire make test.o FLAGS="-Wall -Werror" par exemple.

Les limites

- L'erreur humaine, une courante étant de ne pas mettre les fichiers headers dans les dépendances alors que ça peut poser des problèmes à la compilation si ils n'ont pas été actualisés.
- Rajouter des commentaires modifie le fichier et va donc entraîner une recompilation inutile.
- Problèmes de fichiers et de dates : des fois la date d'un fichier n'est pas sa date de dernière modification. Il peut y avoir des problèmes quand différents groupes modifient le fichier.

Que fait-il

- Outil extensible de construction d'applications
- Permet d'automatiser les tâches de construction d'un projet

Le logiciel

- Acronyme pour « Another Neat Tool » (un autre outil habile)
- Ecrit en Java par la fondation Apache, publié en 2000
- Peut-être utilisé pour des projets écrits dans n'importe quel langage
- Utilise le XML pour décrire le processus de construction (build)

Objectifs d'Ant

- Automatiser les opérations répétitives
- Permettre la portabilité complète du développement en Java

Avantages

- Développé en Java et OpenSource, et donc multiplateforme
- De nombreuses tâches existent déjà
- Extensible : créer une nouvelle tâche est assez facile

Inconvénients

- La structure XML peut être dure à maîtriser
- Pas de persistance d'état, et une gestion des erreurs limitées

La portabilité

- Avec Make :

Unix

```
rm -rf classes/
```

Windows

```
rmdir /S /Q classes
```

- Avec Ant :

Tous les systèmes supportés

```
<delete dir="classes"/>
```

build.xml

- Le fichier build.xml est le fichier principal

build.xml

```
<project name="projetExemple" default="help" basedir=".">
  <property name="base.dir" value="$basedir />
  ...
  <target name="init" >
    <tstamp />
  </target>
  <target name="help" depends "init>...</target>
</project>
```

Descriptions

Projet : <project />

Ensemble de cibles et de propriétés

Propriété : <property />

Une constante, à initialiser. Pour y accéder, ${nomDePropriete}.

Cible : <target />

Ensemble de tâches à exécuter dans un ordre défini.
On peut définir une cible par défaut.

Tâche : <nomDeLaTâche />

Unité de traitement. Exemples de tâches courantes :
delete : Supprimer un fichier
javac : Compiler des sources Java

Projet

build.xml

```
<?xml version="1.0" ?>
<project name="nom du projet" default="main" basedir=".">
  [propriétés]
  [cibles]
</project>
```

- Le projet décrit l'ensemble des étapes de construction de l'application
- l'attribut default est obligatoire

Propriété

build.xml

```
...
  <property name="src.dir" value="src"/>
  <property name="build.dir" value="build"/>
  <property name="classes.dir" value="${build.dir}/classes"/>
  <property name="jar.dir" value="${build.dir}/jar"/>
  <property file="build.properties"/>
...
```

- Chaque propriété doit être initialisée, elle contient un nom et une valeur
- Les propriétés peuvent être contenues dans des fichiers de propriétés

Cible

build.xml

```
...
  <target name="clean">
    [tâches]
  </target>
  <target name="run">
    [tâches]
  </target>
  <target name="main" depends="clean,run"/>
...
```

- Ensemble des tâches qui vont remplir un objectif
- L'attribut name est obligatoire
- Peut utiliser des structures de contrôle (if, sleep, etc.)

Tâche

build.xml

```
...
  <delete dir="${build.dir}"/>
  <mkdir dir="${classes.dir}"/>
  <javac srcdir="${src.dir}" destdir="${classes.dir}"/>
...
```

- Plusieurs dizaines de tâches sont fournies avec Ant
- On peut créer ses propres tâches si nécessaire

Ligne de commande

Syntaxe de base : ant [options] [cible]

Commandes

buildfile
quiet
verbose
version
projecthelp
D[nom]=[valeur]

Conclusion

- La programmation modulaire est quelque chose de très utile, mais il faut utiliser la compilation séparée pour ne pas perdre de temps.
- Pour se simplifier la tâche on peut utiliser des logiciels pour nous aider à faire ça.
- Chaque logiciel a ses avantages et ses inconvénients, make peut être utilisé pour faire plus de choses que Ant, mais ce dernier est plus portable.