Outils de développement - 3A STI

Jean-Francois Lalande - May 2016 - Version 1

Ce cours présente le b.a.-ba des outils d'aide au développement. Il s'agit d'automatiser les processus de compilation, de gestion de versions.

Plan du module

1 Introduction

1.1 De l'automatisation de la compilation

Dans ce qui suit, on présente des benchmarks de performance du processus de compilation datant de 2010. Ces benchmarks ont été réalisés chez Amadeus qui compte plus de 10 000 collaborateurs et qui offre des service notamment pour la gestion des vols pour les compagnies aériennes (entre autre) [GH].

Machine de test: 4 processeurs à 6 coeurs à 2.4 GHz (opteron), 32Go de RAM

Quelques chiffres représentatifs pour le code:

Nb Lines	366 452
Nb functions	15 227
Blank lines	55 878
Nb structs	3483

Temps de compilation + link: 6780s soit presque 2h (avec 4 processus en permanence)

Temps de compilation hacké (concaténation par groupe de 25 des sources): 2854s = 47 min

Temps de re-compilation pour 1 fichier touché: 1 min (calcul du makefile) + quelques secondes (compilation) + 7 min (link)

1.2 De la qualité du code

Sans faire un cours sur la qualité logicielle, des règles de bon sens s'appliquent pour que le code soit compréhensible et maintenables. Toujours en prenant l'exemple du code Amadeus, des indicateurs simples permettent d'apprécier la qualité du code:

Type	Count	Percent	Quality Notice
1	61031	39.22	Physical line length > 80 characters
2	905	0.58	Function name length > 32 characters
4	21	0.01	Assignment "=" within "if" statement
5	9	0.01	Assignment "=" within "while" statement
13	100	0.06	"switch" statement does not have a "default"
14	88	0.06	"case" conditions do not equal "break"
17	5971	3.84	Function comment content less than 10.0%
23	552	0.35	"?" ternary operator identified
29	184	0.12	Number of function parameters > 6
43	497	0.32	Keyword "continue" has been identified
55	1180	0.76	Scope level exceeds the defined limit of 7
107	32	0.02	The closing brace is not on a standalone line

2 Cmake

Références importantes: Cmake

2.1 Les vénérables automake et autoconf

La suite d'outils la plus utilisée pour la génération de la compilation et du fameux **./configure**; **make**; **make install** est la suite des autotools:

- automake: permet d'écrire les squelettes de makefiles
- autoconf: gère les dépendances et génère les makefiles depuis le squelette
- libtool: gestion de création de bibliothèques

La difficulté d'utilisation de ces outils réside en partie dans les langages utilisés, *m4* pour configure.ac utilisé par autoconf, une syntaxe proche du makefile pour makefile.ac utilisé par automatke, tous ces outils étant écrits en *perl*.

La compréhension des messages d'erreur, la faible évolutivité de la suite d'outils la pousse petit à petit vers des chaines de compilation plus évoluées.

Les objectifs de cmake sont donc:

- unifier la syntaxe dans un unique fichier
- unifier les commandes dans un seul outil **cmake**
- augmenter la vitesse d'exécution des opérations de gestion de compilation
- augmenter l'intéropérabilité, par exemple avec kdevelop ou Visual Studio

2.2 CMakeLists.txt

On définit tout d'abord un nom de projet à l'aide de la directive **PROJECT**. Des dépendances peuvent être définies. Il s'agit de bibliothèques du système qui ne sont pas incluses dans la glibc. Dans l'exemple suivant, il s'agit de la librairie zlib:

```
PROJECT(jfl)
FIND_PACKAGE(ZLIB REQUIRED)
```

Le fait d'utiliser zlib impose de passer le répertoire contenant les entêtes au compilateur (gcc). Cela se fait en utilisant la directive **INCLUDE_DIRECTORIES**:

```
INCLUDE_DIRECTORIES(${ZLIB_INCLUDE_DIR})
```

On déclare ensuite l'exécutable à créer et les sources afférentes:

```
ADD_EXECUTABLE(jfl test.c)
```

Nous obtenons, au final:

```
PROJECT(jfl)
FIND_PACKAGE(ZLIB REQUIRED)
INCLUDE_DIRECTORIES(${ZLIB_INCLUDE_DIR})
ADD_EXECUTABLE(jfl test.c)
```

2.2.1 Construction de la chaine de compilation

L'appel à **cmake** permet de générer le Makefile. Il faut préciser le répertoire dans lequel **cmake** trouvera les sources et le fichier CMakeFiles.txt. La suite de la compilation se déroule classiquement en appelant **make**.

```
jf@lalande:cours_outils/codes> cmake .
-- Configuring done Generating done
-- Build files have been written to: .../codes
jf@lalande:cours_outils/codes> make
[100%] Building C object CMakeFiles/jfl.dir/test.o
Linking C executable jfl
[100%] Built target jfl
jf@lalande:cours_outils/codes> ./jfl
HelloWorld !
```

La phase de compilation est silencieuse par défaut. Pour activer les traces il suffit de modifier la variable d'environnement **VERBOSE** à 1.

```
jf@lalande:cours_outils/codes> export VERBOSE=1
[100%] Building C object CMakeFiles/jfl.dir/test.o
/usr/bin/gcc    -o CMakeFiles/jfl.dir/test.o   -c test.c
Linking C executable jfl
/usr/bin/cmake -E cmake_link_script CMakeFiles/jfl.dir/link.txt --verbose=1
/usr/bin/gcc     CMakeFiles/jfl.dir/test.o  -o jfl -rdynamic
```

2.2.2 Les directives SET, IF

La directive **SET** permet de créer des variables réutilisables dans d'autres directives:

```
SET(jfl_src main.c truc.c machin.c)
ADD_EXECUTABLE(jfl ${jfl_src})
```

La variable peut être liée à l'environnement auquel cas, la variable d'environnement est modifiée:

```
set(ENV{PATH} /home/jf)
```

La directive **IF** permet d'introduire des choix conditionnels:

```
IF(ZLIB_FOUND)
  MESSAGE(STATUS "ZLIB présent")
  INCLUDE_DIRECTORIES(${ZLIB_INCLUDE_DIR})
ELSE(ZLIB_FOUND)
  MESSAGE(STATUS "ZLIB absent")
ENDIF(ZLIB_FOUND)
```

La directive **IF** permet de tester de nombreuses choses: le positionnement d'une constante ou d'expressions logiques de constantes, l'existence de fichiers ou répertoires, la comparaison d'entiers ou de chaines.

2.2.3 La construction de librairies

La construction d'une librairie partagée se fait de la même manière qu'un exécutable, en utilisant la directive ADD_LIBRARY. Si l'on place les sources de la librairie dans un sous répertoire, il faut alors ajouter un nouveau fichier CMakeLists.txt dans ce sous-répertoire qui contiendra les directives de construction de cette librairie.

```
cmake_minimum_required(VERSION 2.6)
PROJECT(malib)
SET(lib_src malib.c)
ADD_LIBRARY(malib SHARED ${lib_src})
```

Dans le fichier principal du projet, il faut alors préciser que **cmake** doit évaluer le fichier CMakeLists.txt dans le sous répertoire et l'informer que l'exécutable à produire dépend de la librairie construite dans le sous répertoire:

```
ADD_SUBDIRECTORY(malib)
TARGET_LINK_LIBRARIES(jfl malib ${ZLIB_LIBRARY})
```

Si la librairie est partagée, elle est placée dans le répertoire SHARED:

```
jf@lalande:cours_outils/codes> rm SHARED/libmalib.so
jf@lalande:cours_outils/codes> ./jfl
./jfl: error while loading shared libraries: libmalib.so: cannot open
shared object file: No such file or directory
```

2.2.4 Paramétrer la compilation

Commend dans l'outil d'autotools au travers du script **./configure**, il est possible de passer des options au générateur afin d'activer une option lors de la compilation. On utilise pour se faire la directive **OPTION**:

```
OPTION(WITH_GUI "Compiler l'interface graphique" OFF)
IF(WITH_GUI)
  MESSAGE(STATUS "Compilation de l'interface graphique activée")
ENDIF(WITH_GUI)
```

Lors de l'appel à **cmake**, l'option **-D** permet de passer la valeur de l'option:

```
cmake -DWITH_GUI=ON
```

2.2.5 Fichier CMakeLists.txt final

```
cmake_minimum_required(VERSION 2.6)
PROJECT(jfl)
# Dépendances
FIND_PACKAGE(ZLIB REQUIRED)
INCLUDE_DIRECTORIES(${ZLIB_INCLUDE_DIR})
# Variable des sources et cible
SET(jfl_src test.c)
ADD_EXECUTABLE(jfl ${jfl_src})
# Information sur ZLIB
IF(ZLIB_FOUND)
  MESSAGE(STATUS "ZLIB présent")
  INCLUDE_DIRECTORIES(${ZLIB_INCLUDE_DIR})
ELSE(ZLIB_FOUND)
  MESSAGE(STATUS "ZLIB absent")
ENDIF(ZLIB_FOUND)
# Librairies
ADD_SUBDIRECTORY(malib SHARED)
TARGET_LINK_LIBRARIES(jfl malib ${ZLIB_LIBRARY})
# Option GUI
OPTION(WITH_GUI "Compiler l'interface graphique" OFF)
IF(WITH_GUI)
```

```
  MESSAGE(STATUS "Compilation de l'interface graphique activée")
ENDIF(WITH_GUI)
```

2.3 Installation

La génération des scripts d'installation du Makefile se fait au travers de la directive **INSTALL**. Elle permet de placer automatiquement les binaires, librairies, headers dans l'arborescence du système. Classiquement, un exécutable est placé dans /usr/bin. Lorsque l'on compile et installe un paquet "à la main", celui-ci est classiquement placé dans /usr/local.

```
INSTALL(TARGETS jfl DESTINATION "bin")
INSTALL(FILES mon_header.h DESTINATION "include")
INSTALL(TARGETS malib DESTINATION "lib")
```

Le chemin de destination est relatif par rapport à la variable **CMAKE_INSTALL_PREFIX**. Celle-ci contient une valeur par défaut mais peut-être modifiée à loisir.

Pour installer le binaire, il faut disposer des droits root et exécuter:

```
make install
```

Des permissions spécifiques peuvent êtres spécifiées, par exemple:

```
INSTALL(TARGETS jfl DESTINATION "bin" PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE)
```

2.4 Ant

Apache ant est un autre outil de *build* qui n'est pas "orienté shell" c'est-à-dire qu'il ne repose pas sur bash ou sh. C'est un projet Apache, fait pour compiler du Java. Le fichier de configuration par défaut, build.xml, est assez explicite. Il peut ressembler à:

```
<project>

  <target name="clean">
      <delete dir="build"/>
  </target>

  <target name="compile">
      <mkdir dir="build/classes"/>
      <javac srcdir="src" destdir="build/classes"/>
  </target>

  <target name="jar">
      <mkdir dir="build/jar"/>
      <jar destfile="build/jar/HelloWorld.jar" basedir="build/classes">
          <manifest>
              <attribute name="Main-Class" value="oata.HelloWorld"/>
          </manifest>
      </jar>
  </target>

  <target name="run">
      <java jar="build/jar/HelloWorld.jar" fork="true"/>
  </target>

</project>
```

Qui s'utilise ensuite pour par exemple construire le .jar de l'aplication:

```
ant compile
ant jar
ant run
```

Ceci est un exemple, cliquez sur le lien de téléchargement pour obtenir le cours complet.