在 linux 下使用 CMake 构建应用程序

级别： 初级
王 程明 (wangchengming.jlu@gmail.com)，硕士研究生，吉林大学计算机科学与技术学院
2009 年 2 月 05 日
本文介绍了一个跨平台的自动化构建系统 CMake 在 linux 上的使用方法。 CMake 是一个比 automake 更加容易使用的工具，能够使程序员从复杂的编译连接过程中解脱出来。文中通过一些例子介绍使用 CMake 处理多源文件目录的方法、查找并使用其他开发包的方法以及生成 debug 版和 release 版程序的方法。

CMake 简介

CMake 是一个跨平台的自动化建构系统,它使用一个名为 CMakeLists.txt 的文件来描述构建过程,可以产生标准的构建文件,如 Unix 的 Makefile 或 Windows Visual C++ 的 projects/workspaces 。文件 CMakeLists.txt 需要手工编写,也可以通过编写脚本进行半自动的生成。CMake 提供了比 autoconfig 更简洁的语法。在 linux 平台下使用 CMake 生成 Makefile 并编译的流程如下:

1. 编写 CmakeLists.txt。
2. 执行命令 “cmake PATH” 或者 “ccmake PATH” 生成 Makefile （ PATH 是 CMakeLists.txt 所在的目录 ）。
3. 使用 make 命令进行编译。

第一个工程

现假设我们的项目中只有一个源文件 main.cpp

清单 1 源文件 main.cpp

```
1 #include<iostream>
2
3 int main()
4 {
5    std::cout<<"Hello word!"<<std::endl;
6    return 0;
7 }
```

为了构建该项目,我们需要编写文件 CMakeLists.txt 并将其与 main.cpp 放在 同一个目录下:

清单 2 CMakeLists.txt

```
1 PROJECT(main)
2 CMAKE_MINIMUM_REQUIRED(VERSION 2.6)
3 AUX_SOURCE_DIRECTORY(. DIR_SRCS)
4 ADD_EXECUTABLE(main ${DIR_SRCS})
```

CMakeLists.txt 的语法比较简单,由命令、注释和空格组成,其中命令是不区分大小写的,符号"#"后面的内容被认为是注释。命令由命令名称、小括号和参数组成,参数之间使用空格进行间隔。例如对于清单 2 的 CMakeLists.txt 文件:第一行是一条命令,名称是 PROJECT ,参数是 main ,该命令表示项目的名称是 main 。第二行的命令限定了 CMake 的版本。第三行使用命令 AUX_SOURCE_DIRECTORY 将当前目录中的源文件名称赋值给变量 DIR_SRCS 。 CMake 手册中对命令 AUX_SOURCE_DIRECTORY 的描述如下:

```
aux_source_directory(<dir> <variable>)
```

该命令会把参数 <dir> 中所有的源文件名称赋值给参数 <variable> 。 第四行使用命令 ADD_EXECUTABLE 指示变量 DIR_SRCS 中的源文件需要编译 成一个名称为 main 的可执行文件。完成了文件 CMakeLists.txt 的编写后需要使用 cmake 或 ccmake 命令生成 Makefile 。 ccmake 与命令 cmake 的不同之处在于 ccmake 提供了一个图形化的操作界面。cmake 命令的执行方式如下:

```
cmake [options] <path-to-source>
```

这里我们进入了 main.cpp 所在的目录后执行 “cmake .” 后就可以得到 Makefile 并使用 make 进行编译,如下图所示。

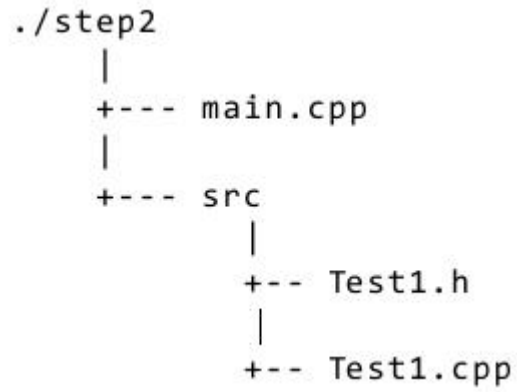
图 1. camke 的运行结果

```
tiger@tiger-laptop:~/test/cmake/step1$ cmake .
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/tiger/test/cmake/step1
tiger@tiger-laptop:~/test/cmake/step1$ make
Scanning dependencies of target main
[100%] Building CXX object CMakeFiles/main.dir/main.cpp.o
Linking CXX executable main
[100%] Built target main
tiger@tiger-laptop:~/test/cmake/step1$ ./main
Hello word!
```

处理多源文件目录的方法

CMake 处理源代码分布在不同目录中的情况也十分简单。现假设我们的源代码分布情况如下：

图 2. 源代码分布情况



其中 src 目录下的文件要编译成一个链接库。

第一步，项目主目录中的 CMakeLists.txt

在目录 step2 中创建文件 CMakeLists.txt 。文件内容如下:

清单 3 目录 step2 中的 CMakeLists.txt

```
1 PROJECT(main)
2 CMAKE_MINIMUM_REQUIRED(VERSION 2.6)
3 ADD_SUBDIRECTORY( src )
4 AUX_SOURCE_DIRECTORY(. DIR_SRCS)
5 ADD_EXECUTABLE(main ${DIR_SRCS}  )
6 TARGET_LINK_LIBRARIES( main Test )
```

相对于清单 2，该文件添加了下面的内容: 第三行，使用命令 ADD_SUBDIRECTORY 指明本项目包含一个子目录 src 。第六行，使用命令 TARGET_LINK_LIBRARIES 指明可执行文件 main 需要连接一个名为 Test 的链接库 。

第二步，子目录中的 CmakeLists.txt

在子目录 src 中创建 CmakeLists.txt。文件内容如下:

清单 4. 目录 src 中的 CmakeLists.txt

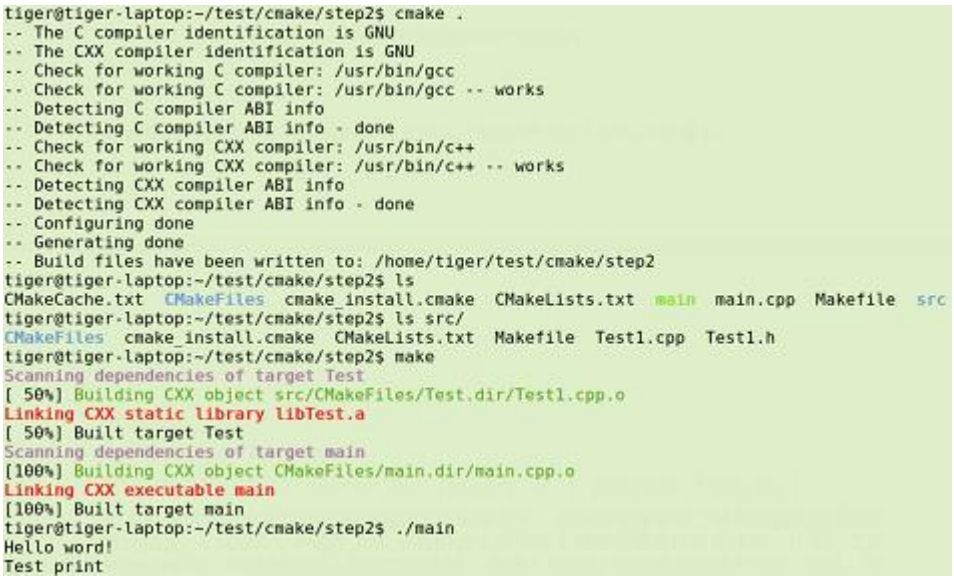
```
1 AUX_SOURCE_DIRECTORY(. DIR_TEST1_SRCS)
2 ADD_LIBRARY ( Test ${DIR_TEST1_SRCS})
```

在该文件中使用命令 ADD_LIBRARY 将 src 目录中的源文件编译为共享库。

第三步，执行 cmake

至此我们完成了项目中所有 CMakeLists.txt 文件的编写,进入目录 step2 中依次执行命令 “cmake .” 和 “make” 得到结果如下:

图 3. 处理多源文件目录时 cmake 的执行结果



在执行 cmake 的过程中,首先解析目录 step2 中的 CMakeLists.txt ,当程序执行命令 ADD_SUBDIRECTORY(src) 时进入目录 src 对其中的 CMakeLists.txt 进行解析。

在工程中查找并使用其他程序库的方法

在开发软件的时候我们会用到一些函数库,这些函数库在不同的系统中安装的位置可能不同,编译的时候需要首先找到这些软件包的头文件以及链接库所在的目录以便生成编译选项。例如一个需要使用博克利数据库项目,需要头文件 db_cxx.h 和链接库 libdb_cxx.so ,现在该项目中有一个源代码文件 main.cpp ，放在项目的根目录中。

第一步，程序库说明文件

在项目的根目录中创建目录 cmake/modules/ ，在 cmake/modules/ 下创建文件 Findlibdb_cxx.cmake ，内容如下：

清单 5. 文件 Findlibdb_cxx.cmake

```
01 MESSAGE(STATUS "Using bundled Findlibdb.cmake...")
02
03 FIND_PATH(
04   LIBDB_CXX_INCLUDE_DIR
05   db_cxx.h
06   /usr/include/
07   /usr/local/include/
08   )
09
10 FIND_LIBRARY(
11   LIBDB_CXX_LIBRARIES NAMES  db_cxx
12   PATHS /usr/lib/ /usr/local/lib/
13   )
```

文件 Findlibdb_cxx.cmake 的命名要符合规范: FindlibNAME.cmake ,其中 NAME 是函数库的名称。Findlibdb_cxx.cmake 的语法与 CMakeLists.txt 相同。这里使用了三个命令： MESSAGE ， FIND_PATH 和 FIND_LIBRARY 。

- 命令 MESSAGE 会将参数的内容输出到终端。
- 命令 FIND_PATH 指明头文件查找的路径，原型如下：
  find_path(<VAR> name1 [path1 path2 ...]) 该命令在参数 path* 指示的目录中查找文件 name1 并将查找到的路径保存在变量 VAR 中。清单 5 第 3－8 行的意思是在 /usr/include/ 和 /usr/local/include/ 中查找文件 db_cxx.h ,并将 db_cxx.h 所在的路径保存在 LIBDB_CXX_INCLUDE_DIR 中。
- 命令 FIND_LIBRARY 同 FIND_PATH 类似,用于查找链接库并将结果保存在变量中。清单 5 第 10－13 行的意思是在目录 /usr/lib/ 和 /usr/local/lib/ 中寻找名称为 db_cxx 的链接库,并将结果保存在 LIBDB_CXX_LIBRARIES 。

第二步, 项目的根目录中的 CmakeList.txt

在项目的根目录中创建 CmakeList.txt ：

清单 6. 可以查找链接库的 CMakeList.txt

```
01 PROJECT(main)
02 CMAKE_MINIMUM_REQUIRED(VERSION 2.6)
03 SET(CMAKE_SOURCE_DIR .)
04 SET(CMAKE_MODULE_PATH ${CMAKE_ROOT}/Modules ${CMAKE_SOURCE_DIR}/cmake/modules)
05 AUX_SOURCE_DIRECTORY(. DIR_SRCS)
06 ADD_EXECUTABLE(main ${DIR_SRCS})
07
08 FIND_PACKAGE( libdb_cxx REQUIRED)
09 MARK_AS_ADVANCED(
10 LIBDB_CXX_INCLUDE_DIR
11 LIBDB_CXX_LIBRARIES
12 )
13 IF (LIBDB_CXX_INCLUDE_DIR AND LIBDB_CXX_LIBRARIES)
14 MESSAGE(STATUS "Found libdb libraries")
15   INCLUDE_DIRECTORIES(${LIBDB_CXX_INCLUDE_DIR})
16    MESSAGE( ${LIBDB_CXX_LIBRARIES} )
17    TARGET_LINK_LIBRARIES(main ${LIBDB_CXX_LIBRARIES}
18 )
19 ENDIF (LIBDB_CXX_INCLUDE_DIR AND LIBDB_CXX_LIBRARIES)
```

在该文件中第 4 行表示到目录 ./cmake/modules 中查找 Findlibdb_cxx.cmake ,8-19 行表示查找链接库和头文件的过程。第 8 行使用命令 FIND_PACKAGE 进行查找,这条命令执行后 CMake 会到变量 CMAKE_MODULE_PATH 指示的目录中查找文件 Findlibdb_cxx.cmake 并执行。第 13-19 行是条件判断语句,表示如果 LIBDB_CXX_INCLUDE_DIR 和 LIBDB_CXX_LIBRARIES 都已经被赋值,

则设置编译时到 LIBDB_CXX_INCLUDE_DIR 寻找头文件并且设置可执行文件 main 需要与链接库 LIBDB_CXX_LIBRARIES 进行连接。

第三步，执行 cmake

完成 Findlibdb_cxx.cmake 和 CMakeList.txt 的编写后在项目的根目录依次执行“cmake . ”和“make ”可以进行编译,结果如下图所示：

图 4. 使用其他程序库时 cmake 的执行结果

```
tiger@tiger-laptop:~/learn/cmake/step3$ cmake .
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Using bundled Findlibdb.cmake...
-- Found libdb libraries
/usr/lib/libdb_cxx.so
-- Configuring done
-- Generating done
-- Build files have been written to: /home/tiger/learn/cmake/step3
tiger@tiger-laptop:~/learn/cmake/step3$ make
Scanning dependencies of target main
[ 50%] Building CXX object CMakeFiles/main.dir/main.cpp.o
[100%] Building CXX object CMakeFiles/main.dir/Student.cpp.o
Linking CXX executable main
[100%] Built target main
```

使用 cmake 生成 debug 版和 release 版的程序

在 Visual Studio 中我们可以生成 debug 版和 release 版的程序,使用 CMake 我们也可以达到上述效果。debug 版的项目生成的可执行文件需要有调试信息并且不需要进行优化,而 release 版的不需要调试信息但需要优化。这些特性在 gcc/g++ 中是通过编译时的参数来决定的,如果将优化程度调到最高需要设置参数-O3,最低是 -O0 即不做优化;添加调试信息的参数是 -g -ggdb ,如果不添加这个参数,调试信息就不会被包含在生成的二进制文件中。

CMake 中有一个变量 CMAKE_BUILD_TYPE ,可以的取值是 Debug Release RelWithDebInfo 和 MinSizeRel。当这个变量值为 Debug 的时候,CMake 会使用变量 CMAKE_CXX_FLAGS_DEBUG 和 CMAKE_C_FLAGS_DEBUG 中的字符串作为编译选项生成 Makefile ,当这个变量值为 Release 的时候,工程会使用变量 CMAKE_CXX_FLAGS_RELEASE 和 CMAKE_C_FLAGS_RELEASE 选项生成 Makefile。

现假设项目中只有一个文件 main.cpp ,下面是一个可以选择生成 debug 版和 release 版的程序的 CMakeList.txt ：

清单 7

```
1 PROJECT(main)
2 CMAKE_MINIMUM_REQUIRED(VERSION 2.6)
3 SET(CMAKE_SOURCE_DIR .)
4
5 SET(CMAKE_CXX_FLAGS_DEBUG "$ENV{CXXFLAGS} -O0 -Wall -g -ggdb")
```

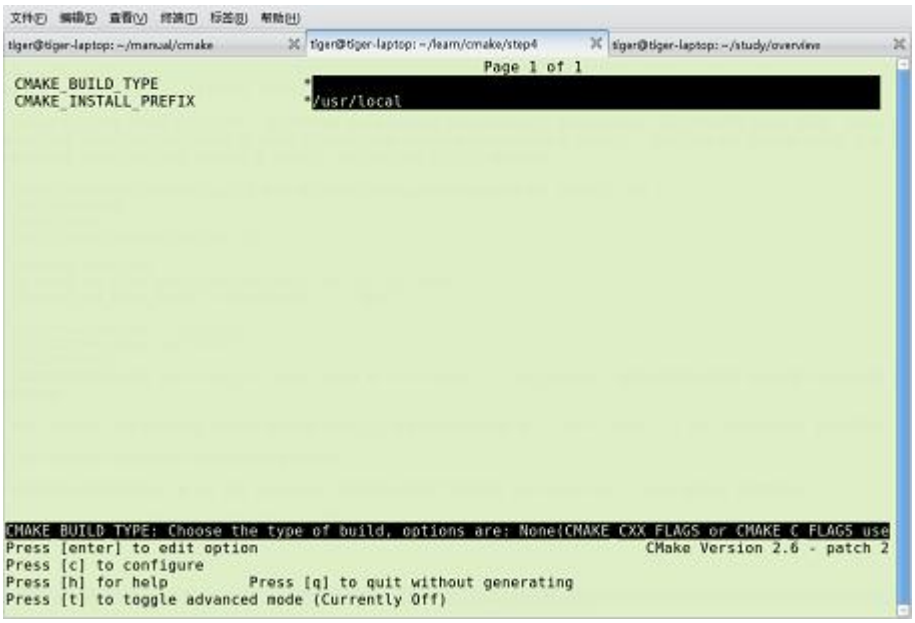
```
6 SET(CMAKE_CXX_FLAGS_RELEASE "$ENV{CXXFLAGS} -O3 -Wall")
7
8 AUX_SOURCE_DIRECTORY(. DIR_SRCS)
9 ADD_EXECUTABLE(main ${DIR_SRCS})
```

第 5 和 6 行设置了两个变量 CMAKE_CXX_FLAGS_DEBUG 和 CMAKE_CXX_FLAGS_RELEASE, 这两个变量是分别用于 debug 和 release 的编译选项。编辑 CMakeList.txt 后需要执行 ccmake 命令生成 Makefile 。在进入项目的根目录,输入 "ccmake ." 进入一个图形化界面,如下图所示：

图 5. ccmake 的界面

按照界面中的提示进行操作,按 "c" 进行 configure ,这时界面中显示出了配置变量 CMAKE_BUILD_TYPE 的条目。如下图所示：

图 6. 执行了 configure 以后 ccmake 的界面



下面我们首先生成 Debug 版的 Makefile ：将变量 CMAKE_BUILD_TYPE 设置为 Debug ,按 "c" 进行 configure ，按 "g" 生成 Makefile 并退出。这时执行命令 find * | xargs grep "O0" 后结果如下:

清单 8 find * | xargs grep "O0"的执行结果

```
CMakeFiles/main.dir/flags.make:CXX_FLAGS = -O0 -Wall -g -ggdb
CMakeFiles/main.dir/link.txt:/usr/bin/c++ -O0 -Wall -g -ggdb
CMakeFiles/main.dir/main.cpp.o -o main -rdynamic
CMakeLists.txt:SET(CMAKE_CXX_FLAGS_DEBUG "$ENV{CXXFLAGS} -O0 -Wall -g -ggdb")
```

这个结果说明生成的 Makefile 中使用了变量 CMAKE_CXX_FLAGS_DEBUG 作为编译时的参数。
下面我们将生成 Release 版的 Makefile ：再次执行命令 "ccmake ." 将变量 CMAKE_BUILD_TYPE 设置为 Release ,生成 Makefile 并退出。执行命令 find * | xargs grep "O0" 后结果如下：

清单 9 find * | xargs grep "O0"的执行结果

```

CMakeLists.txt:SET(CMAKE_CXX_FLAGS_DEBUG "$ENV{CXXFLAGS} -O0 -Wall -g -ggdb")
```

而执行命令 find * | xargs grep "O3" 后结果如下:

清单 10. find * | xargs grep "O3"的执行结果

```
CMakeCache.txt:CMAKE_CXX_FLAGS_RELEASE:STRING=-O3 -DNDEBUG
CMakeCache.txt:CMAKE_C_FLAGS_RELEASE:STRING=-O3 -DNDEBUG
CMakeFiles/main.dir/flags.make:CXX_FLAGS = -O3 -Wall
CMakeFiles/main.dir/link.txt:/usr/bin/c++ -O3 -Wall
CMakeFiles/main.dir/main.cpp.o -o main -rdynamic
CMakeLists.txt:SET(CMAKE_CXX_FLAGS_RELEASE "$ENV{CXXFLAGS} -O3 -Wall")
```

这两个结果说明生成的 Makefile 中使用了变量 CMAKE_CXX_FLAGS_RELEASE 作为编译时的参数。

参考资料

- 要了解 CMake，请参考其官方网站：http://www.cmake.org/。
- 维基百科中对 CMake 也有非常详细的说明，具体请参考：http://zh.wikipedia.org/wiki/CMake。
- 在 developerWorks Linux 专区 寻找为 Linux 开发人员（包括 Linux 新手入门）准备的更多参考资料，查阅我们 最受欢迎的文章和教程。
- 在 developerWorks 上查阅所有 Linux 技巧 和 Linux 教程。

关于作者

王程明，就读于吉林大学计算机科学与技术学网格计算与网络安全实验室，主要研究领域为网络安全。