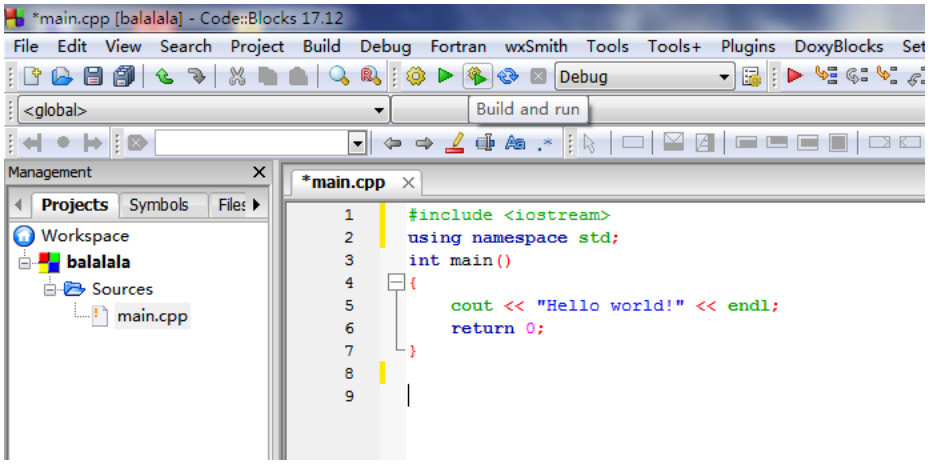
目录

一、Windows 和 linux 下执行单文件

在 windows 环境下，大家都熟悉怎么编写并执行一份代码：

1. 先打开编译器例如 codeblocks 编写源代码，例如一个 c++文件；
2. 点击编译按钮，编译代码，生成.o 的目标文件。
3. 点击执行按钮，生成.exe 的可执行文件，运行完毕。

例如 codeblocks 下的 build & run：



在 linux 环境下呢？

用 vim 编辑器编写代码，得到一个文本文件 main.cpp

```
htfhxx@ubuntu:~/code/hello$ ls
main.cpp
htfhxx@ubuntu:~/code/hello$ cat main.cpp
#include<iostream>
int main(int argc,char *argv[])
{
    std::cout << "hello world" << std::endl;
    return(0);
}
```

1. 使用 g++编译 main.cpp 得到 a.out 文件

```
htfhxx@ubuntu:~/code/hello$ g++ main.cpp
htfhxx@ubuntu:~/code/hello$ ls
a.out  main.cpp
```

2. 执行 a.out 文件，执行完毕得到结果

```
htfhxx@ubuntu:~/code/hello$ ./a.out
hello world
```

当然，更普遍的是使用-o 来编译和执行的

```
htfhxx@ubuntu:~/code/hello$ g++ main.cpp -o hello
htfhxx@ubuntu:~/code/hello$ ls
a.out  hello  main.cpp
htfhxx@ubuntu:~/code/hello$ ./hello
hello world
htfhxx@ubuntu:~/code/hello$
```

二、Windows 和 linux 下执行多文件 or 项目

Windows 自不必多说，在编译器下编译运行 main 函数即可

至于 linux，有如下三个文件 speak.h speak.cpp hellospeak.cpp

```
[huotengfei@node24 test_makefile]$ cat speak.h

/* speak.h */
#include <iostream>
class Speak
{
    public:
        void sayHello(const char *);
};
```

```
[huotengfei@node24 test_makefile]$ cat speak.cpp


/* speak.cpp */
#include "speak.h"
void Speak::sayHello(const char *str)
{
    std::cout << "Hello " << str << "\n";
}
```

```
[huotengfei@node24 test_makefile]$ cat hellospeak.cpp


/* hellospeak.cpp */
#include "speak.h"
int main(int argc,char *argv[])
{
    Speak speak;
    speak.sayHello("world");
    return(0);
}
```

编译执行多个文件：

g++ hellospeak.cpp speak.cpp -o hellospeak

```
[huotengfei@node24 test_makefile]$  g++ hellospeak.cpp speak.cpp -o hellospeak
[huotengfei@node24 test_makefile]$ ./hellospeak
Hello world
```

这个时候会发现，如果源文件太多，一个一个编译时就会特别麻烦。于是人们想到了制作一种类似批处理的程序，来批处理编译源文件，于是就有了 make 工具。它是一个自动化的编译工具，你可以使用一条命令实现完全编译。但是你需要编写一个规则文件，make 依据它来批处理编译，这个文件就是 makefile。

三、解决多文件编译的困难：makefile

文件下包含一个头文件和两个 cpp 文件，以及一个写好的 makefile：

```
[huotengfei@node24 test_makefile]$ ls
hellospeak.cpp  makefile  speak.cpp  speak.h
```

Makefile 大致内容就是要编译两个文件得到 hellospeak.o 和 speak.o，再生成可执行文件 hellospeak，最后删掉.o 文件。

```
[huotengfei@node24 test_makefile]$ cat makefile

all: hellospeak clean
hellospeak: hellospeak.o speak.o
        g++ -o hellospeak hellospeak.o speak.o
hellospeak.o: hellospeak.cpp speak.h
        g++ -c hellospeak.cpp
speak.o: speak.cpp speak.h
        g++ -c speak.cpp
clean:
        rm -f  *o
```

```
[huotengfei@node24 test_makefile]$ make
g++ -c hellospeak.cpp
g++ -c speak.cpp
g++ -o hellospeak hellospeak.o speak.o
rm -f  *o
[huotengfei@node24 test_makefile]$ ls
hellospeak  hellospeak.cpp  makefile  speak.cpp  speak.h
[huotengfei@node24 test_makefile]$ ./hellospeak
Hello world
```

对于一个大工程，编写 makefile 实在是件复杂的事，于是就出现了 cmake 工具，它能够输出各种各样的 makefile 或者 project 文件,从而帮助程序员减轻负担。但是随之而来也就是编写 cmakelist 文件，它是 cmake 所依据的规则。

四、Cmake 工具:编译运行文件

如果没有安装的话就通过 sudo apt-get install cmake 命令安装 cmake 工具：

```
htfhxx@ubuntu:~/code$ cmake .
The program 'cmake' is currently not installed.  You can install it by
sudo apt-get install cmake
htfhxx@ubuntu:~/code$ sudo apt-get install cmake
[sudo] password for htfhxx:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  cmake-data emacsen-common libxmlrpc-core-c3
The following NEW packages will be installed:
```

准备好 cmakelist.txt 文件和要执行的 main.cpp，以及一个 build 文件，用于放入 cmake 编译的繁多的中间文件：

```
[huotengfei@node24 test_cmake]$ ls
build  CMakeLists.txt  main.cpp
[huotengfei@node24 test_cmake]$ pwd
/home/huotengfei/code/cmake/test_cmake
```

要执行的 main.cpp：

```
[huotengfei@node24 test_cmake]$ cat main.cpp
#include<iostream>
int main(int argc,char *argv[])
{
    std::cout << "hello world" << std::endl;
    return(0);
}
```

cmakelist.txt 文件（内容撰写待会再说）：

```
[huotengfei@node24 test_cmake]$ cat CMakeLists.txt
cmake_minimum_required(VERSION 2.8)

project(HelloWorld)

add_executable(HelloWorld main.cpp)
```

第一步，cmake + (cmakelists.txt 所在文件夹)。
此处“..”指的是上一级文件夹，会从文件夹中找到 cmakelists.txt：

```
[huotengfei@node24 build]$ cmake ..
-- The C compiler identification is GNU 4.4.7
-- The CXX compiler identification is GNU 4.4.7
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- 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/huotengfei/code/cmake/test_cmake/build
```

系统自动生成了：CMakeFiles, CMakeCache.txt, cmake_install.cmake 等文件，并且生成了 Makefile

```
[huotengfei@node24 build]$ pwd
/home/huotengfei/code/cmake/test_cmake/build
[huotengfei@node24 build]$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  Makefile
```

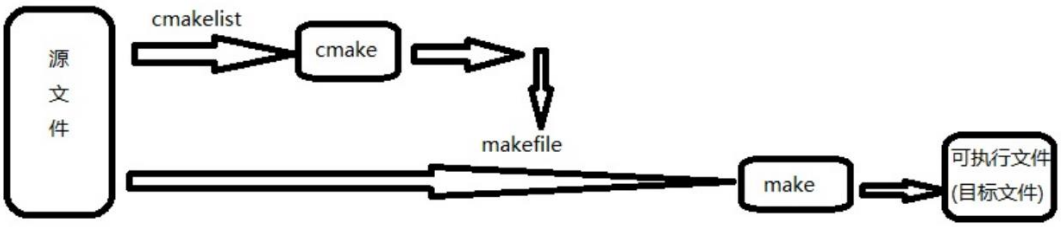
进行工程的实际构建，在这个目录输入 `make` 命令，大概会得到如下的彩色输出：

```
[huotengfei@node24 build]$ make
Scanning dependencies of target HelloWorld
[100%] Building CXX object CMakeFiles/HelloWorld.dir/main.cpp.o
Linking CXX executable HelloWorld
[100%] Built target HelloWorld
```

到这里就已经编译完成了，接下来执行这个项目得到 hello world 的输出：

```
[huotengfei@node24 build]$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  HelloWorld  Makefile
[huotengfei@node24 build]$ ./HelloWorld
hello world
```

即，整个流程为（网图，侵删）：



五、使用 cmake 方便的编译执行单文件 Demo

先分析上一例子中的 cmakelists.txt：

```
[huotengfei@node24 test_cmake]$ cat CMakeLists.txt
cmake_minimum_required(VERSION 2.8)

project(HelloWorld)

add_executable(HelloWorld main.cpp)
```

cmake_minimum_required(VERSION 2.8)
//指的是支持的 cmake 版本，可省略，但是为了方便后人，尽量加上自己所用的版本。
project(HelloWorld)
//指定项目名称，编译完成后生成的名字就是 HelloWorld

add_executable(HelloWorld main.cpp)
//加入执行文件，此处是单文件，待会展开来讲

六、使用 cmake 方便的编译执行多文件项目

两个 cpp 文件一个.h 文件和一个 build，这次我们试着用 cmake 编译执行多文件。

```
[huotengfei@node24 test_cmake_muti]$ pwd
/home/huotengfei/code/cmake/test_cmake_muti
[huotengfei@node24 test_cmake_muti]$ ls
build  CMakeLists.txt  hellospeak.cpp  speak.cpp  speak.h
```

这个 example 的 cmakelists.txt 里包含了之前讲的版本和项目名，加了一个 include_directories，这个参数是把.h 文件所在目录包含进去，可以是一堆的.h 文件。其中 cmake_source_dir 是系统变量，可以通过 set 关键字来设置，默认来说是 cmakelists.txt 所在的文件。

```
[huotengfei@node24 test_cmake_muti]$ cat CMakeLists.txt
cmake_minimum_required(VERSION 2.8)



project(HelloWorld)

#directories of .h
INCLUDE_DIRECTORIES(
 ${CMAKE_SOURCE_DIR}/
)

SET(Sources_code  hellospeak.cpp speak.cpp)

add_executable(hellospeak  ${Sources_code})
```

接着就是执行两个 cpp 文件，当然两个 cpp 文件也可以通过 set 来设置成变量 Sources_code.

```
[huotengfei@node24 build]$ cmake ..
-- The C compiler identification is GNU 4.4.7
-- The CXX compiler identification is GNU 4.4.7
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- 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/huotengfei/code/cmake/test_cmake_muti/build
[huotengfei@node24 build]$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  Makefile
```

```
[huotengfei@node24 build]$ make
Scanning dependencies of target hellospeak
[ 50%] Building CXX object CMakeFiles/hellospeak.dir/hellospeak.cpp.o
[100%] Building CXX object CMakeFiles/hellospeak.dir/speak.cpp.o
Linking CXX executable hellospeak
[100%] Built target hellospeak
[huotengfei@node24 build]$ ./hellospeak
Hello world
```

七、一个复杂的例子：关于 CMakelists 子目录 and 生成库

```
[huotengfei@node24 test_cmake_lib]$ pwd
/home/huotengfei/code/cmake/test_cmake_lib
[huotengfei@node24 test_cmake_lib]$ ls
build  CMakeLists.txt  hellospeak.cpp  MathFunctions
```

这个文件夹下包含 build 文件、一个要执行的 cpp 文件和 CMakelists.txt。除此之外，还有一个 MathFunctions 文件夹。

先来看子目录 MathFunctions：

```
[huotengfei@node24 test_cmake_lib]$ cd MathFunctions/
[huotengfei@node24 MathFunctions]$ ls
CMakeLists.txt  speak.cpp  speak.h
```

这里的 CMakeLists.txt 只有几行，将 speak.cpp 中的函数生成一个 MathFunctions 库。

```
[huotengfei@node24 MathFunctions]$ cat CMakeLists.txt
include_directories(
"${PROJECT_SOURCE_DIR}/MathFunctions"
)

add_library(MathFunctions speak.cpp)
```

在顶层目录下的 CMakeLists.txt 中，通过 add_subdirectories 加入子目录的 CMakeLists.txt。

```
1  cmake_minimum_required(VERSION 2.8)
2  project(HelloWorld)
3
4  #directories of .h
5  INCLUDE_DIRECTORIES(
6   ${CMAKE_SOURCE_DIR}/
7  )
8
9  #should we use our own functions?
10 option (OK  "Use our cpp"  ON)
11
12 if(OK)
13         INCLUDE_DIRECTORIES(
14         #add .h files in MathFunctions
15          ${PROJECT_SOURCE_DIR}/MathFunctions
16         )
17
18         #add the subdirectory
19         add_subdirectory(MathFunctions)
20
21         #give the variable EXTRA_LIBS with ${EXTRA_LIBS} ,default is MathFunctions
22         set (EXTRA_LIBS ${EXTRA_LIBS} MathFunctions)
23
24 endif (OK)
25
26 #add executable files and link the libraries
27 add_executable(hellospeak hellospeak.cpp)
28 target_link_libraries(hellospeak  ${EXTRA_LIBS})
29
```

前两行是版本和项目名，五六行加入.h 文件的目录，10 行设置一个变量默认为 ON,12 行判断是否 OK，如果 OK 为 ON 的话就可以执行 13-23 行。

13-23 行，是确定使用自己本地（即 MathFunctions 文件夹中的库），分别是加入.h 文件、加入子目录（划重点，下面讲）、设置 EXTRA_LIBS 变量，如果未设置 28 行就不再链接这个库。

八、总结 cmakelists.txt 的整理内容

cmakelists.txt 内容不需区分大小写。
一般的 cmakelists.txt 的编写，包括以下几部分：

1. 指定 cmake 版本，就像上面所说的：为了方便后人，尽量加上自己所用的版本 cmake_minimum_required(VERSION 2.8)

2. 指定项目的名称，一般和项目的文件夹名称对应
   project(HelloWorld)

3. 设置环境变量 SET(变量名 变量值)

一般包括（但不仅仅包括）：
CMAKE_C_COMPILER：指定 C 编译器
CMAKE_CXX_COMPILER：指定 C++编译器
CMAKE_C_FLAGS：编译 C 文件时的选项，如-g；也可以通过 add_definitions 添加编译选项
EXECUTABLE_OUTPUT_PATH：可执行文件的存放路径

LIBRARY_OUTPUT_PATH：库文件路径
CMAKE_BUILD_TYPE:：build 的类型(Debug, Release, ...)

变量很多很复杂，根据需要使用即可，可以从官方文档中查找：
https://cmake.org/cmake/help/v3.0/manual/cmake-variables.7.html

当然还有一些自己定义的变量名，也用 set 设置

4. LINK_DIRECTORIES 添加需要链接的库文件目录,即链接库搜索路径
   link_directories(directory1 directory2 ...)

5. 添加可执行文件要链接的库文件的名称
   TARGET_LINK_LIBRARIES(PROJECT_NAME libname.so)

6. 头文件目录
   INCLUDE_DIRECTORIES(
    Include
)

   如果文件夹较多，则可以这样写：
   INCLUDE_DIRECTORIES(
    ${CMAKE_SOURCE_DIR}/include/
    ${CMAKE_SOURCE_DIR}/include/a/
    ${CMAKE_SOURCE_DIR}/include/b/
)

7. 源文件目录
   AUX_SOURCE_DIRECTORY(src DIR_SRCS)

8. 添加要编译的可执行文件
   ADD_EXECUTABLE(PROJECT_NAME TEST_CPP)

9. 生成动态库 or 静态库
   这里多说两句，用 cmake 生成静态动态库，是将在 cmakelists.txt 文件中加入的源文件头文件等等，生成一个类似于.h/.a 的文件
   这与<8>中加入想要编译的可执行文件是二选一的关系。
   add_library(person SHARED ${srcs})
   add_library(person_static STATIC ${srcs})

   install (TARGETS)
   创建规则以将列出的目标安装到给定目录中。