# 一、Cmake简介

cmake是一个跨平台、开源的构建系统。它是一个集软件构建、测试、打包于一身的软件。它使用与平台和编译器独立的配置文件来对软件编译过程进行控制。

# 二、常用命令

## 1. 指定cmake的最小版本

cmake\_minimum\_required(VERSION 3.4.1)

这行命令是可选的，我们可以不写这句话，但在有些情况下，如果 CMakeLists.txt 文件中使用了一些高版本cmake特有的一些命令的时候，就需要加上这样一行，提醒用户升级到该版本之后再执行cmake。

## 2. 设置项目名称

project(demo)

这个命令不是强制性的，但最好都加上。它会引入两个变量demo\_BINARY\_DIR 和 demo\_SOURCE\_DIR，同时，cmake自动定义了两个等价的变量 PROJECT\_BINARY\_DIR 和 PROJECT\_SOURCE\_DIR。

## 3. 设置编译类型

add\_executable(demo demo.cpp) # 生成可执行文件

add\_library(common STATIC util.cpp) # 生成静态库

add\_library(common SHARED util.cpp) # 生成动态库或共享库

add\_library 默认生成是静态库，通过以上命令生成文件名字，

在Linux下是：

demo

libcommon.a

libcommon.so

在Windows下是：

demo.exe

common.lib

common.dll

## 4. 指定编译包含的源文件

4.1 明确指定包含哪些源文件

add\_library(demo demo.cpp test.cpp util.cpp)

4.2 搜索所有的cpp文件

aux\_source\_directory(dir VAR) 发现一个目录下所有的源代码文件并将列表存储在一个变量中。

aux\_source\_directory(. SRC\_LIST) # 搜索当前目录下的所有.cpp文件

add\_library(demo ${SRC\_LIST})

4.3 自定义搜索规则

file(GLOB SRC\_LIST "\*.cpp" "protocol/\*.cpp")

add\_library(demo ${SRC\_LIST})

# 或者

file(GLOB SRC\_LIST "\*.cpp")

file(GLOB SRC\_PROTOCOL\_LIST "protocol/\*.cpp")

add\_library(demo ${SRC\_LIST} ${SRC\_PROTOCOL\_LIST})

# 或者

aux\_source\_directory(. SRC\_LIST)

aux\_source\_directory(protocol SRC\_PROTOCOL\_LIST)

add\_library(demo ${SRC\_LIST} ${SRC\_PROTOCOL\_LIST})

## 5. 查找指定的库文件

find\_library(VAR name path)查找到指定的预编译库，并将它的路径存储在变量中。

默认的搜索路径为cmake包含的系统库，因此如果是 NDK 的公共库只需要指定库的 name 即可。

find\_library( # Sets the name of the path variable.

log-lib

# Specifies the name of the NDK library that

# you want CMake to locate.

log )

类似的命令还有find\_file()、find\_path()、find\_program()、find\_package()。

## 6. 设置包含的目录

include\_directories(

${CMAKE\_CURRENT\_SOURCE\_DIR}

${CMAKE\_CURRENT\_BINARY\_DIR}

${CMAKE\_CURRENT\_SOURCE\_DIR}/include

)

Linux下还可以通过如下方式设置包含的目录

set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -I${CMAKE\_CURRENT\_SOURCE\_DIR}")

## 7. 设置链接库搜索目录

link\_directories(

${CMAKE\_CURRENT\_SOURCE\_DIR}/libs

)

Linux 下还可以通过如下方式设置包含的目录

set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -L${CMAKE\_CURRENT\_SOURCE\_DIR}/libs")

## 8. 设置 target 需要链接的库

target\_link\_libraries( # 目标库

demo

# 目标库需要链接的库

# log-lib 是上面 find\_library 指定的变量名

${log-lib} )

在 Windows 下，系统会根据链接库目录，搜索xxx.lib 文件，Linux 下会搜索 xxx.so 或者 xxx.a 文件，如果都存在会优先链接动态库（so 后缀）。

8.1 指定链接动态库或静态库

target\_link\_libraries(demo libface.a) # 链接libface.a

target\_link\_libraries(demo libface.so) # 链接libface.so

8.2 指定全路径

target\_link\_libraries(demo ${CMAKE\_CURRENT\_SOURCE\_DIR}/libs/libface.a)

target\_link\_libraries(demo ${CMAKE\_CURRENT\_SOURCE\_DIR}/libs/libface.so)

8.3 指定链接多个库

target\_link\_libraries(demo

${CMAKE\_CURRENT\_SOURCE\_DIR}/libs/libface.a

boost\_system.a

boost\_thread

pthread)

## 9. 设置变量

9.1 set 直接设置变量的值

set(SRC\_LIST main.cpp test.cpp)

add\_executable(demo ${SRC\_LIST})

## 9.2 set追加设置变量的值

set(SRC\_LIST main.cpp)

set(SRC\_LIST ${SRC\_LIST} test.cpp)

add\_executable(demo ${SRC\_LIST})

## 9.3 list追加或者删除变量的值

set(SRC\_LIST main.cpp)

list(APPEND SRC\_LIST test.cpp)

list(REMOVE\_ITEM SRC\_LIST main.cpp)

add\_executable(demo ${SRC\_LIST})

## 10. 条件控制

10.1 if…elseif…else…endif

逻辑判断和比较：

if (expression)：expression 不为空（0,N,NO,OFF,FALSE,NOTFOUND）时为真

if (not exp)：与上面相反

if (var1 AND var2)

if (var1 OR var2)

if (COMMAND cmd)：如果 cmd 确实是命令并可调用为真

if (EXISTS dir) if (EXISTS file)：如果目录或文件存在为真

if (file1 IS\_NEWER\_THAN file2)：当 file1 比 file2 新，或 file1/file2 中有一个不存在时为真，文件名需使用全路径

if (IS\_DIRECTORY dir)：当 dir 是目录时为真

if (DEFINED var)：如果变量被定义为真

if (var MATCHES regex)：给定的变量或者字符串能够匹配正则表达式 regex 时为真，此处 var 可以用 var 名，也可以用 ${var}

if (string MATCHES regex)

数字比较：

if (variable LESS number)：LESS 小于

if (string LESS number)

if (variable GREATER number)：GREATER 大于

if (string GREATER number)

if (variable EQUAL number)：EQUAL 等于

if (string EQUAL number)

字母表顺序比较：

if (variable STRLESS string)

if (string STRLESS string)

if (variable STRGREATER string)

if (string STRGREATER string)

if (variable STREQUAL string)

if (string STREQUAL string)

示例：

if(MSVC)

set(LINK\_LIBS common)

else()

set(boost\_thread boost\_log.a boost\_system.a)

endif()

target\_link\_libraries(demo ${LINK\_LIBS})

# 或者

if(UNIX)

set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -std=c++11 -fpermissive -g")

else()

add\_definitions(-D\_SCL\_SECURE\_NO\_WARNINGS

D\_CRT\_SECURE\_NO\_WARNINGS

-D\_WIN32\_WINNT=0x601

-D\_WINSOCK\_DEPRECATED\_NO\_WARNINGS)

endif()

if(${CMAKE\_BUILD\_TYPE} MATCHES "debug")

...

else()

...

endif()

10.2 while…endwhile

while(condition)

...

endwhile()

10.3 foreach…endforeach

foreach(loop\_var RANGE start stop [step])

...

endforeach(loop\_var)

start 表示起始数，stop 表示终止数，step 表示步长，示例：

foreach(i RANGE 1 9 2)

message(${i})

endforeach(i)

# 输出：13579

## 11. 打印信息

message(${PROJECT\_SOURCE\_DIR})

message("build with debug mode")

message(WARNING "this is warnning message")

message(FATAL\_ERROR "this build has many error") # FATAL\_ERROR 会导致编译失败

## 12. 包含其它cmake文件

include(./common.cmake) # 指定包含文件的全路径

include(def) # 在搜索路径中搜索def.cmake文件

set(CMAKE\_MODULE\_PATH ${CMAKE\_CURRENT\_SOURCE\_DIR}/cmake) # 设置include的搜索路径

# 三、常用变量

## 1. 预定义变量

PROJECT\_SOURCE\_DIR：工程的根目录

PROJECT\_BINARY\_DIR：运行 cmake 命令的目录，通常是 ${PROJECT\_SOURCE\_DIR}/build

PROJECT\_NAME：返回通过 project 命令定义的项目名称

CMAKE\_CURRENT\_SOURCE\_DIR：当前处理的 CMakeLists.txt 所在的路径

CMAKE\_CURRENT\_BINARY\_DIR：target 编译目录

CMAKE\_CURRENT\_LIST\_DIR：CMakeLists.txt 的完整路径

CMAKE\_CURRENT\_LIST\_LINE：当前所在的行

CMAKE\_MODULE\_PATH：定义自己的 cmake 模块所在的路径，SET(CMAKE\_MODULE\_PATH ${PROJECT\_SOURCE\_DIR}/cmake)，然后可以用INCLUDE命令来调用自己的模块

EXECUTABLE\_OUTPUT\_PATH：重新定义目标二进制可执行文件的存放位置

LIBRARY\_OUTPUT\_PATH：重新定义目标链接库文件的存放位置

## 2. 环境变量

使用环境变量

$ENV{Name}

写入环境变量

set(ENV{Name} value) # 这里没有“$”符号

## 3. 系统信息

CMAKE\_MAJOR\_VERSION：cmake 主版本号，比如 3.4.1 中的 3

CMAKE\_MINOR\_VERSION：cmake 次版本号，比如 3.4.1 中的 4

CMAKE\_PATCH\_VERSION：cmake 补丁等级，比如 3.4.1 中的 1

CMAKE\_SYSTEM：系统名称，比如 Linux-­2.6.22

CMAKE\_SYSTEM\_NAME：不包含版本的系统名，比如 Linux

CMAKE\_SYSTEM\_VERSION：系统版本，比如 2.6.22

CMAKE\_SYSTEM\_PROCESSOR：处理器名称，比如 i686

UNIX：在所有的类 UNIX 平台下该值为 TRUE，包括 OS X 和 cygwin

WIN32：在所有的 win32 平台下该值为 TRUE，包括 cygwin

## 4. 主要开关选项

BUILD\_SHARED\_LIBS：这个开关用来控制默认的库编译方式，如果不进行设置，使用 add\_library 又没有指定库类型的情况下，默认编译生成的库都是静态库。如果 set(BUILD\_SHARED\_LIBS ON) 后，默认生成的为动态库

CMAKE\_C\_FLAGS：设置 C 编译选项，也可以通过指令 add\_definitions() 添加

CMAKE\_CXX\_FLAGS：设置 C++ 编译选项，也可以通过指令 add\_definitions() 添加

add\_definitions(-DENABLE\_DEBUG -DABC) # 参数之间用空格分隔

# 四、项目示例

## 1. 简单项目（单个源文件）

一个经典的C程序，如何通过cmake来编译呢？

1.1 新建文件 main.c

#include <stdio.h>

int main() {

printf("Hello World!\n");

return 0;

}

1.2 新建文件 CMakeLists.txt（命名必须是 CMakeLists.txt，注意大小写和不要漏字母）

project(HELLO)

add\_executable(hello main.c)

1.3 编译和运行

新建一个build文件夹，目录情况如下：

为什么要新建一个build文件夹？

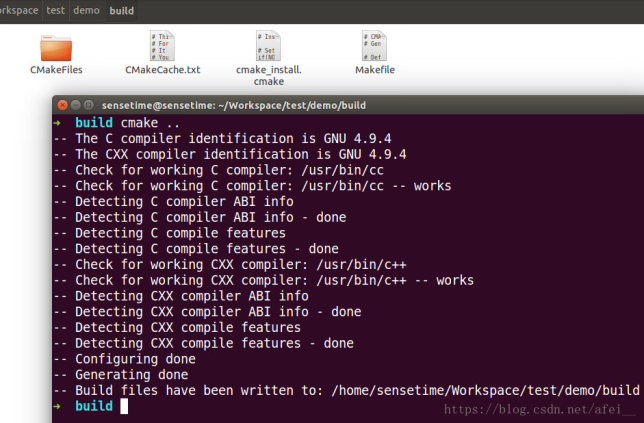
一般我们采用cmake的out-of-source方式来构建（即生成的中间产物和源代码分离），这样做可以让生成的文件和源文件不会弄混，且目录结构看起来也会清晰明了。所以推荐使用这种方式，至于这个文件夹的命名并无限制，我们习惯命名为build。

依次执行以下命令：

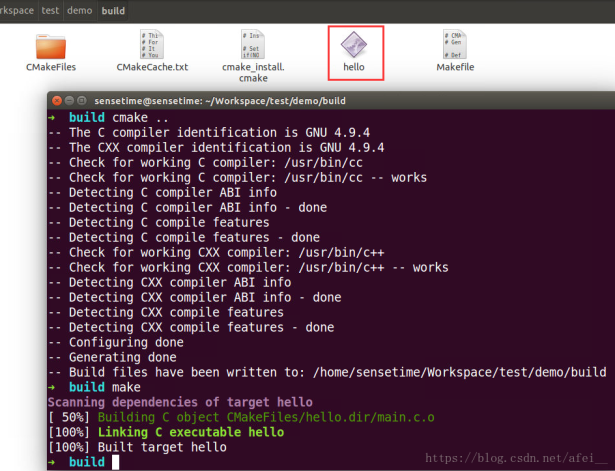
cd build

cmake ..

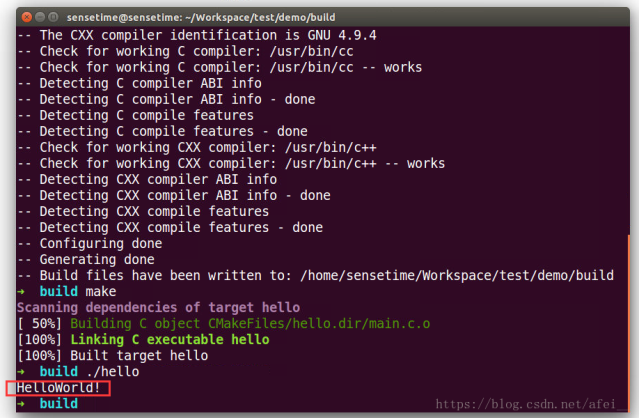
首先进入到 build 目录，然后就是 cmake .. 命令，另外要注意将cmake配置到环境变量中去才可直接使用，否则需要使用绝对路径，..表示上一级目录，cmake会在上一级目录下找到CMakeLists.txt文件并编译，并生成如下图所示的一些中间文件：



直接执行make命令，生成可执行程序，如下图：

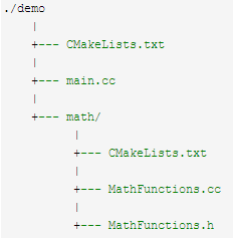


运行程序，如下所示：运行程序，如下所示：



## 2. 复杂项目（多个目录，多个源文件，多个项目）

目录结构如下：



demo根目录下的CMakeLists.txt文件如下：

cmake\_minimum\_required (VERSION 2.8)

project(demo)

aux\_source\_directory(. DIR\_SRCS)

# 添加math子目录

add\_subdirectory(math)

# 指定生成目标

add\_executable(demo ${DIR\_SRCS})

# 添加链接库

target\_link\_libraries(demo MathFunctions)

math 目录下的CMakeLists.txt文件如下：

aux\_source\_directory(. DIR\_LIB\_SRCS)

# 生成链接库

add\_library(MathFunctions ${DIR\_LIB\_SRCS})

## 3. 自定义编译选项

cmake 允许为项目增加编译选项，从而可以根据用户的环境和需求选择最合适的编译方案。

例如，可以将 MathFunctions 库设为一个可选的库，如果该选项为 ON ，就使用该库定义的数学函数来进行运算，否则就调用标准库中的数学函数库。

修改根目录下的 CMakeLists.txt 文件如下：

# CMake 最低版本号要求

cmake\_minimum\_required (VERSION 2.8)

# 项目信息

project (Demo)

# 加入一个配置头文件，用于处理 CMake 对源码的设置

configure\_file (

"${PROJECT\_SOURCE\_DIR}/config.h.in"

"${PROJECT\_BINARY\_DIR}/config.h"

)

# 是否使用自己的 MathFunctions 库

option (USE\_MYMATH

"Use provided math implementation" ON)

# 是否加入 MathFunctions 库

if (USE\_MYMATH)

include\_directories ("${PROJECT\_SOURCE\_DIR}/math")

add\_subdirectory (math)

set (EXTRA\_LIBS ${EXTRA\_LIBS} MathFunctions)

endif (USE\_MYMATH)

# 查找当前目录下的所有源文件

# 并将名称保存到 DIR\_SRCS 变量

aux\_source\_directory(. DIR\_SRCS)

# 指定生成目标

add\_executable(Demo ${DIR\_SRCS})

target\_link\_libraries (Demo ${EXTRA\_LIBS})

configure\_file 命令用于加入一个配置头文件config.h，这个文件由cmake从config.h.in 生成，通过这样的机制，将可以通过预定义一些参数和变量来控制代码的生成。

option命令添加了一个USE\_MYMATH选项，并且默认值为ON。根据USE\_MYMATH变量的值来决定是否使用我们自己编写的MathFunctions库。

修改main.cc文件，让其根据USE\_MYMATH的预定义值来决定是否调用标准库还是MathFunctions 库：

#include "config.h"

#ifdef USE\_MYMATH

#include "math/MathFunctions.h"

#else

#include <math.h>

#endif

int main(int argc, char \*argv[])

{

if (argc < 3){

printf("Usage: %s base exponent \n", argv[0]);

return 1;

}

double base = atof(argv[1]);

int exponent = atoi(argv[2]);

#ifdef USE\_MYMATH

printf("Now we use our own Math library. \n");

double result = power(base, exponent);

#else

printf("Now we use the standard library. \n");

double result = pow(base, exponent);

#endif

printf("%g ^ %d is %g\n", base, exponent, result);

return 0;

}

编写config.h.in文件.

注意main.cc的第一行，这里引用了一个config.h文件，这个文件预定义了 USE\_MYMATH的值。但我们并不直接编写这个文件，为了方便从 CMakeLists.txt 中导入配置，我们编写一个config.h.in文件，内容如下：

#cmakedefine USE\_MYMATH

这样cmake会自动根据CMakeLists.txt配置文件中的设置自动生成config.h文件。

# CMake如何编译CUDA(.cu)源文件

现在的项目，如果需要用到计算加速，**Nvidia**的**CUDA**往往是首选。那么如何在**CMake**中编译写好的**CUDA**源代码，可以参考如下。

首先使用**FIND\_PACKAGE**找到已经安装的**CUDA**，此时需要配置的环境变量等，应该已经自动配置完成了

SET(CUDA\_VERSION 8.0)

FIND\_PACKAGE(CUDA ${CUDA\_VERSION} REQUIRED)

STRING (TOLOWER ${CMAKE\_SYSTEM\_NAME} LOWER\_SYSTEM\_NAME)

SET(CUDA\_TARGET\_INCLUDE${CUDA\_TOOLKIT\_ROOT\_DIR}-${CUDA\_VERSION}/targets/${CMAKE\_HOST\_SYSTEM\_PROCESSOR}-${LOWER\_SYSTEM\_NAME}/include)

接下来，使用**CUDA\_ADD\_LIBRARY**取代原来的**ADD\_LIBRARY**，如下：

CUDA\_ADD\_LIBRARY(mylib SHARED

file1.cpp

file2.cu

file3.cpp

OPTIONS -arch sm\_20

)

TARGET\_LINK\_LIBRARIES(mylib ${CUDA\_LIBRARIES})

如果是可执行程序，请使用**CUDA\_ADD\_EXECUTABLE**取代**ADD\_EXECUTABLE**。