

键盘的起始页

为钱做事，容易累；为理想做事，能够耐风寒；为兴趣做事，则永不倦怠。

目录视图



摘要视图

RSS 订阅

0



(0)

收藏

举报



个人资料



ab6326795

原创	粉丝	喜欢	评论
81	100	143	116



等级： 博客 6

访问量：74万+

积分：8840

排名：2640

AndroidStudio之NDK开发CMake CMakeLists.txt编写入门

2017年10月27日 14:07:48

1022人阅读

分类： JNI (7)

目录(?)

[+]

一、CmakeList的编写和参数详解

在linux 下进行开发很多人选择编写makefile 文件进行项目环境搭建，而makefile 文件依赖关系复杂，工作量很大，搞的人头很大。采用自动化的项目构建工具cmake 可以将程序员从复杂的makefile 文件中解脱出来。cmake 根据内置的规则和语法来自动生成相关的makefile 文件进行编译，同时还支持静态库和动态库的构建，我把工作中用到的东东总结在此，方便忘记时随时查看，具体cmake的介绍和详细语法还是参考官方文档（<http://www.cmake.org/>），有一篇中文的[cmake 实践](#)写的不错，可以google下。



四十平小户型装修



40平小户型装修



装修房子顺序



种头发原理

文章搜索

文章分类

- 名人历史 (1)
- JAVA学习 (28)
- C语言学习 (14)
- C#学习 (1)
- C++学习 (21)
- 斯凯冒泡mrp平台学习 (1)
- android开发 (306)
- MySQL/SqlServer数据库 (5)
- 经验分享 (66)
- 网络开发 (6)
- 跟我一起学JAVA WEB (31)
- windows程序设计 (15)

使用cmake 很简单，只需要执行cmake, make 两个命令即可，用我工作中的一个工程举例说明。

假设当前的项目代码在src 目录。 src 下有子目录：server, utility, lib, bin, build

server ----- 存放项目的主功能类文件

utility ----- 存放项目要用到相关库文件，便已成为库文件存放到子目录lib 中

lib ----- 存放utility 生成的库

bin ----- 存放association 生成的二进制文件

build ----- 编译目录，存放编译生成的中间文件

cmake 要求工程主目录和所有存放源代码子目录下都要编写CMakeList 文件，注意大小写（cm 大写，list中l 大写且落下s）。

src/CMakeLists.txt 文件如下：

```
[cpp]
1.  #cmake file for project association                                #表示注释
2.      #author:>---double__song
3.      #created:>---2011/03/01
4.
5.      CMAKE_MINIMUM_REQUIRED(VERSION 2.8)    #cmake 最低版本要求，低于2.6 构建过程会被终止。
6.
7.      PROJECT(server_project)                                #定义工程名称
8.
9.      MESSAGE(STATUS "Project: SERVER")                #打印相关消息消息
10.     MESSAGE(STATUS "Project Directory: ${PROJECT_SOURCE_DIR}")
11.     SET(CMAKE_BUILD_TYPE DEBUG)                    #指定编译类型
12.     SET(CMAKE_C_FLAGS_DEBUG "-g -Wall")            #指定编译器
13.
14.     ADD_SUBDIRECTORY/utility)                        #添加子目录
15.     ADD_SUBDIRECTORY(server)
```

相关解释：



windows phone手机应用开发 (1)

linux学习 (37)

设计模式 (1)

JavaScript (7)

生活 (5)

JNI (8)

cocos2d-js/x (4)

vuforia(VR) (1)

OpenCV (1)

android安全与逆向 (24)

自动化测试 (1)

文章存档

2018年3月 (1)

2018年2月 (8)

2018年1月 (7)

2017年12月 (4)

2017年11月 (19)

展开

阅读排行

usb调试模式已打开 , adb devi... (52551)

电脑上安装windows phone 8模... (26258)

android使用mount挂载/system/... (17578)

android 获取控件大小和设置... (17105)

【30秒】android模拟器获取R... (16672)

android mediaPlayer error (-38,0... (15416)

1. CMakeLists.txt 文件中不区分大小写

2. PROJECT(project_name) 定义工程名称

语法：project(projectname [cxx] [c] [java])

可以指定工程采用的语言，选项分别表示：C++，C，java，如不指定默认支持所有语言

3. MESSAGE (STATUS, "Content") 打印相关消息

输出消息，供调试CMakeLists.txt 文件使用。

4. SET(CMAKE_BUILD_TYPE DEBUG) 设置编译类型debug 或者release debug 版会生成相关调试信息，可以使用GDB 进行

调试；release不会生成调试信息。当无法进行调试时查看此处是否设置为debug.

5. SET(CMAKE_C_FLAGS_DEBUG "-g -Wall") 设置编译器的类型

CMAKE_C_FLAGS_DEBUG ---- C 编译器

CMAKE_CXX_FLAGS_DEBUG ---- C++ 编译器

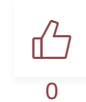
6. ADD_SUBDIRECTORY(utility) 添加要编译的子目录

为工程主目录下的存放源代码的子目录使用该命令，各子目录出现的顺序随意。

如上便是工程server_project 主目录src 下的CMakeLists.txt 文件，下一篇我们解释子目录utility中的CMakeLists.txt 文件。

子目录utility 下的CMakeLists.txt 文件如下：

```
[cpp]
1. #Cmake file for library utility.a
```



0



android使用百度地图、定位S...	(14374)
CMD命令查看局域网内所有...	(11576)
你不可忘记的历史，人民好总...	(10624)
ANSI/UTF-8/UCS2(UTF-16)，...	(7936)

最新评论

usb调试模式已打开，adb de...

大昱：博主果然会玩~，只是不知道这个adb程序稳定性怎么样..

usb调试模式已打开，adb de...

qq_22672291：跟root没关系，别随便root

Android逆向之旅---And...

孙华强：CSDN博友你好，我是孙华强，现将此篇博文收录进“Android知识库”。CSDN现在在做CSDN...

判断两个矩形是否相交的4个方法

骄人：2个矩形相交明明是一个平行四边形，哪里能用矩形替代

【教程】查看某个程序的占用的端口

Tomes_V_White：这个拿来看看Android的adb.exe端口是否被占用时，非常有用。

解决error while loa...

TOP大文哥：sudo apt-get install libgtk2.0-0:i386 libxxf86vm1...

ANR机制以及问题分析

Dawish_大D：虽然是转载的，但是不错的文章

Eclipse中使用正则表达式替换...

qq361301276：精华啊！！

Android(java方法)上实...

小马哥nice：你好，请问有没有工程的项目？

Java实现解压Apk、往apk中...

陈美圆：重新压缩后apk文件变小了，会不会有什么影响？

```

2. #Author:      double__song
3. #Created:      2011/3/3
4.
5. SET(SOURCE_FILES                                #设置变量，表示所有的源文件
6.   ConfigParser.cpp
7.   StrUtility.cpp
8. )
9. INCLUDE_DIRECTORIES(                            #相关头文件的目录
10.  /usr/local/include
11.  ${PROJECT_SOURCE_DIR}/utility
12. )
13.
14. LINK_DIRECTORIES(                               #相关库文件的目录
15.  /usr/local/lib
16. )
17.
18. ADD_LIBRARY(association ${SOURCE_FILES})         #生成静态链接库libassociation.a
19. TARGET_LINK_LIBRARIES(association core)          #依赖的库文件
20.
21. SET_TARGET_PROPERTIES(utility PROPERTIES         #表示生成的执行文件所在路径
22.  RUNTIME_OUTPUT_DIRECTORY> "${PROJECT_SOURCE_DIR}/lib")

```



0



相关解释：

1. SET(SOURCE_FILES)

表示要编译的源文件，所有的源文件都要罗列到此处。set 设置变量，变量名SOURCE_FILES自定义。

2. INCLUDE_DIRECTORY(...)

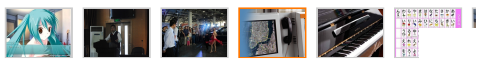
include头文件时搜索的所有目录

变量PROJECT_SOURCE_DIR 表示工程所在的路径，系统默认变量

3. LINK_DIRECTORIES(...)



免费云主机试用一年



联系我们



请扫描二维码联系客服

✉ webmaster@csdn.net

☎ 400-660-0108

💬 QQ客服 🗨 客服论坛

关于 招聘 广告服务 百度

©1999-2018 CSDN版权所有

京ICP证09002463号

经营性网站备案信息

网络110报警服务

中国互联网举报中心

北京互联网违法和不良信息举报中心

库文件存放的目录，在程序连接库文件的时候要再这些目录下寻找对应的库文件

4. ADD_LIBRARY(...)

表示生成静态链接库libassociation.a，由\${PROJECT_SOURCE_DIR}代表的文件生成。

语法：ADD_LIBRARY(libname [SHARED|STATIC]

SHARED 表示生成动态库，STATIC表示生成静态库。

5. TARGET_LINK_LIBRARY(association core)

表示库association 依赖core库文件

6. SET_TARGET_PROPERTIES

设置编译的库文件存放的目录，还可用于其他属性的设置。如不指定，

生成的执行文件在当前编译目录下的各子目录下的build目录下，好拗口！简单一点：

如指定在：./src/lib 下

不指定在：./src/build/utility/build 目录下

生成的中间文件在./src/build/utility/build 目录下，不受该命令影响

子目录server 下的CMakeLists.txt 文件：

```
[cpp]
1. -----
2.     SET(SOURCE_FILES
3.       Gassociation.cpp
4.       ConfigurationHandler.cpp
5.     )
6.     INCLUDE_DIRECTORIES (
7.       /usr/local/include
```



0



```
8.     ${PROJECT_SOURCE_DIR}/utility
9.     ${PROJECT_SOURCE_DIR}/association
10.    )
11.    LINK_LIBRARIES(
12.    /usr/local/lib
13.    ${PROJECT_SOURCE_DIR}/lib
14.    )
15.    ADD_EXECUTABLE(server ${SOURCE_FILES})
16.    TARGET_LINK_LIBRARIES(server
17.    utility
18.    )
19.    SET_TARGET_PROPERTIES(server PROPERTIES #表示生成的执行文件所在路径
20.    RUNTIME_OUTPUT_DIRECTORY "${PROJECT_SOURCE_DIR}/bin")
```



0



相关解释：

1. ADD_EXECUTABLE() #指定要生成的执行文件的名称server

其他用法同utilty/CMakeLists.txt

2. SET_TARGET_PROPERTIES

设置生成的执行文件存放的路径，

注意：

执行文件server 依赖的子目录utility 子目录生成的静态库libutility.a,在指定的时候要写成：

TARGET_LINK_LIBRARIES(server utility)

而不能写成：

TARGET_LINK_LIBRARIES(server libutility.a)

否则编译总会提示找不到libutility库文件。

但使用第三方的库却要指定成具体的库名，如：libACE-6.0.0.so

这一点很诡异，暂时还没找到原因。

完成对应的CMakeLists.txt 文件编写后，便可以进行编译了。

编译：

进入 ./src/build

执行cmake ..

make



cmake 的使用很简单，更高级的应用好比版本信息，打包，安装等相本的应用后面会一一介绍，



复杂的语法使用要参考官方文档。



二、Android Studio NDK CMake 指定so输出路径以及生成多个so的案例与总结

前文

一直想用Android Studio的新方式Cmake来编译JNI 代码，之前也尝试过，奈何有两个难题挡住了我

1. 只能生成一个 so库，不能一次性生成多个 so库，之前的mk是可以有子模块的。
2. 每次生成的so所在的目录不是在 jniLibs下，虽然知道如果打包，会将它打包进去，但就是觉得看不见它，想提供给别人用，还要去某个目录找。

经过尝试，这两个问题都可以解决了。

生成单个so的案例

demo下载地址: <http://download.csdn.net/detail/b2259909/9766081>

直接看CMakeLists.txt文件:

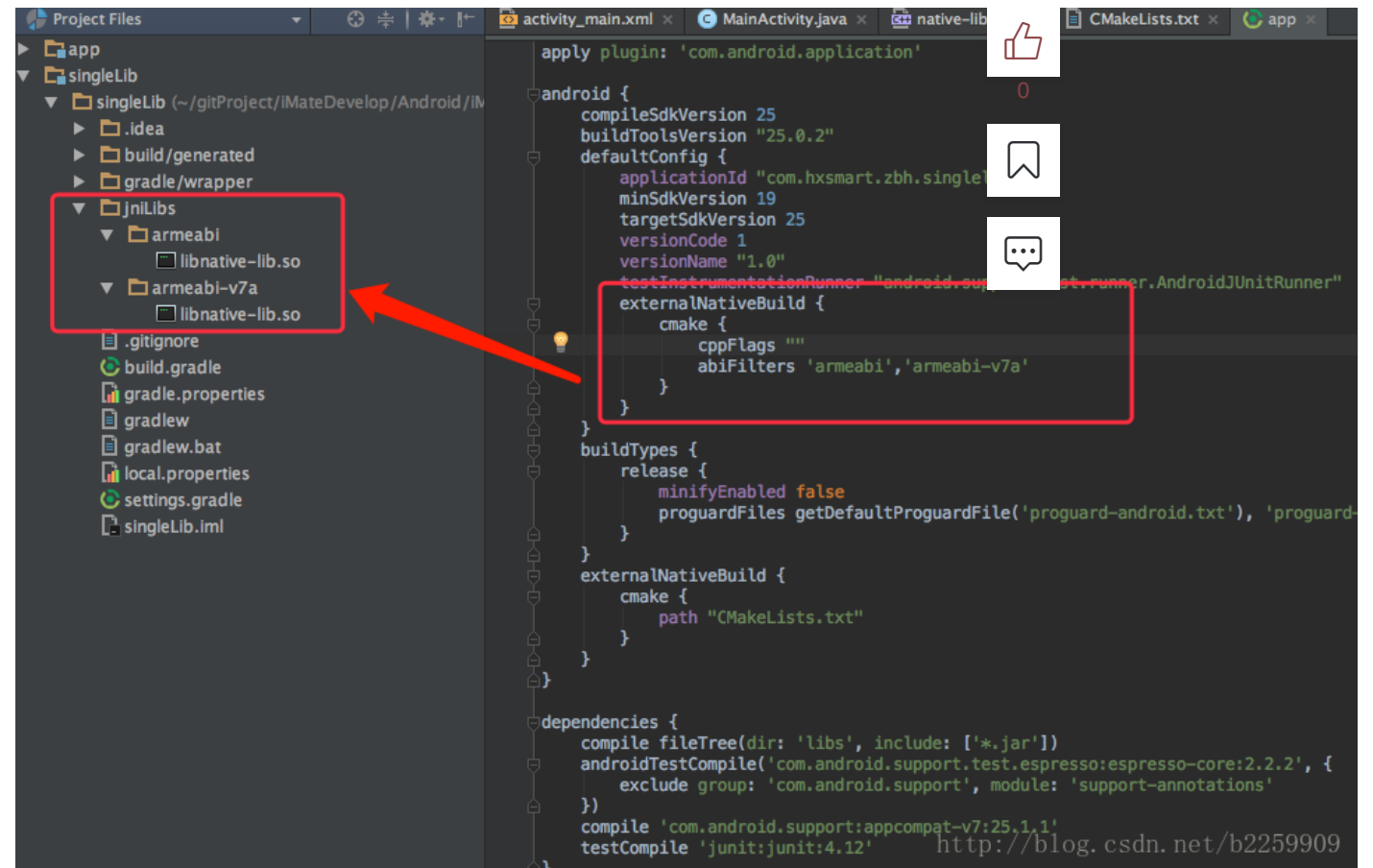
```
1  #指定需要CMAKE的最小版本
2  cmake_minimum_required(VERSION 3.4.1)
3
4
5  #C 的编译选项是 CMAKE_C_FLAGS
6  # 指定编译参数，可选
7  SET(CMAKE_CXX_FLAGS "-Wno-error=format-security -Wno-error=pointer-sign")
8
9  #设置生成的so动态库最后输出的路径
10 set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${PROJECT_SOURCE_DIR}/../jniLibs/${ANDROID_ABI})
11
12 #设置头文件搜索路径（和此txt同个路径的头文件无需设置），可选
13 #INCLUDE_DIRECTORIES(${CMAKE_CURRENT_SOURCE_DIR}/common)
14
15 #指定用到的系统库或者NDK库或者第三方库的搜索路径，可选。
16 #LINK_DIRECTORIES(/usr/local/lib)
17
18 add_library( native-lib
19             SHARED
20             src/main/cpp/native-lib.cpp )
21
22 target_link_libraries( native-lib
23                       log )
```



其中 各个设置都有说明。主要看这个：

```
1 | set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${PROJECT_SOURCE_DIR}/../jniLibs/${ANDROID_ABI})
```

它将会把生成的so库按照你在 build.gradle 指定的 abi分别放置在 jniLibs下

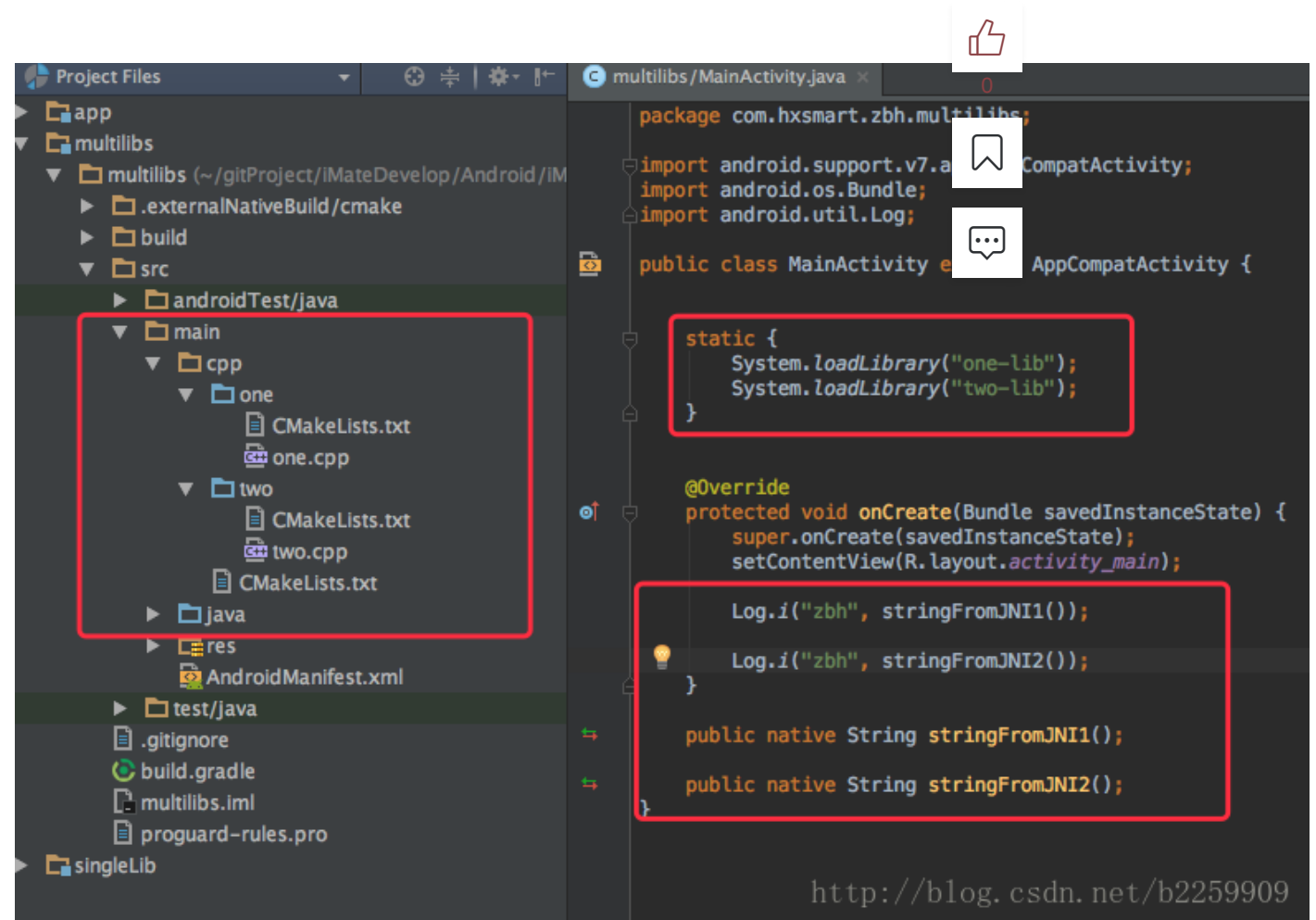


非常好，先解决了第二个问题了。

生成多个so案例


还是上面那个demo，重新建一个module。

cpp的目录结构:



直接看CMakeLists.txt文件:

```
1 #指定需要CMAKE的最小版本
2 cmake_minimum_required(VERSION 3.4.1)
3
4
5 #C 的编译选项是 CMAKE_C_FLAGS
6 # 指定编译参数，可选
7 SET(CMAKE_CXX_FLAGS "-Wno-error=format-security -Wno-error=pointer-sign")
8
9 #设置生成的so动态库最后输出的路径
10 set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${PROJECT_SOURCE_DIR}/../jniLibs/${ARCH}_ABI)
11
12 #设置头文件搜索路径（和此txt同个路径的头文件无需设置），可选
13 #INCLUDE_DIRECTORIES(${CMAKE_CURRENT_SOURCE_DIR}/common)
14
15 #指定用到的系统库或者NDK库或者第三方库的搜索路径，可选。
16 #LINK_DIRECTORIES(/usr/local/lib)
17
18 #添加子目录,将会调用子目录中的CMakeLists.txt
19 ADD_SUBDIRECTORY(one)
20 ADD_SUBDIRECTORY(two)
```



不同的地方是改为添加子目录：

```
1 #添加子目录,将会调用子目录中的CMakeLists.txt
2 ADD_SUBDIRECTORY(one)
3 ADD_SUBDIRECTORY(two)
```

这样就会先去跑到子目录下的 one 和 two 的CmakeLists.txt，执行成功再返回。

此时子目录one下的CmakeLists.txt:

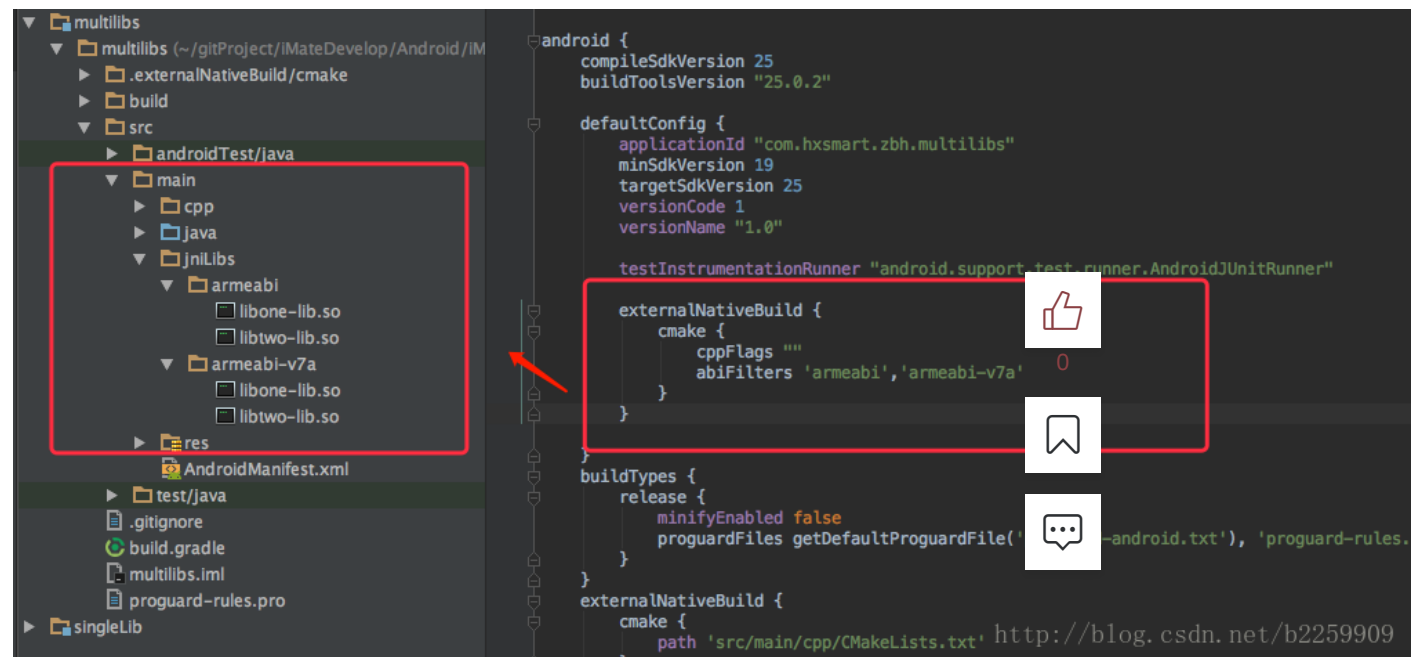
```
1 #继承上一层的CMakeLists.txt的变量，也可以在这里重新赋值
2 #C 的编译选项是 CMAKE_C_FLAGS
3 # 指定编译参数，可选
4 #SET(CMAKE_CXX_FLAGS "-Wno-error=format-security -Wno-error=pointer-sign")
5
6 #生成so动态库
7 ADD_LIBRARY(one-lib SHARED one.cpp)
8
9 target_link_libraries(one-lib log)
```



子目录two下的CmakeLists.txt:

```
1 #继承上一层的CMakeLists.txt的变量，也可以在这里重新赋值
2 #C 的编译选项是 CMAKE_C_FLAGS
3 # 指定编译参数，可选
4 #SET(CMAKE_CXX_FLAGS "-Wno-error=format-security -Wno-error=pointer-sign")
5
6 #生成so动态库
7 ADD_LIBRARY(two-lib SHARED two.cpp)
8
9 target_link_libraries(two-lib log)
```

最后生成了以下两个so文件,并自动按照abi分别放置在了 jniLibs下:



第一个问题也成功了。

总结

最后，除了 设定abiFilters 必须在 build.gradle

主要是发现CmakeLists.txt里 其实可以指定很多东西:

1. so输出路径 CMAKE_LIBRARY_OUTPUT_DIRECTORY
2. .a 静态库输出路径 CMAKE_ARCHIVE_OUTPUT_DIRECTORY
2. 获取当前编译的abi , ANDROID_ABI
3. 编译选项 :

CMAKE_C_FLAGS

CMAKE_CXX_FLAGS

CMAKE_CXX_FLAGS_DEBUG/CMAKE_CXX_FLAGS_RELEASE

4. 子目录编译: ADD_SUBDIRECTORY

5. #设置.c文件集合的变量

```
1  #当前cmakelists.txt所在目录的所有.c .cpp源文件
2  AUX_SOURCE_DIRECTORY(. SRC_LIST)
3
4  #增加其他目录的源文件到集合变量中
5  list(APPEND SRC_LIST
6      ../common/1.c
7      ../common/2.c
8      ../common/3.c
9      ../common/4.c
10     ../common/5.c
11     ../common/WriteLog.c
12 )
13
14 #生成so库，直接使用变量代表那些.c文件集合
15 add_library(mylib SHARED ${SRC_LIST})
```



0



6. 执行自定义命令:

```
1  # copy头文件到 静态库相同文件夹下
2  add_custom_command(TARGET myjni
3      PRE_BUILD
4      COMMAND echo "executing a copy command"
5      COMMAND cp ${CMAKE_CURRENT_SOURCE_DIR}/myjni.h ${PROJECT_SOURCE_DIR}/../build/outputs/staticLib/n
6      COMMENT "PRE_BUILD, so This command will be executed before building target myjni"
7  )
```

最后，因为很多时候，JNI的参数还要转为C的方式，当我们在JAVA层写了native方法，android IDE自动提示红色，这时按下 ALT + ENTER 可以自动生成JNI下的方法声明，并且入参也帮我们转换好了。不过有时候这个插件不生效。

所以我写了一个JNI 的入参转为 C/C++的代码插件: JNI-Convert-Var,直接在 plugin 的仓库搜就有了。最近尝试实现android studio的ALT + ENTER 可以自动生成JNI下的方法声明，结果发现好多IntelliJ IDEA的接口不熟悉。只能先放弃了，以下是我的逻辑：



当鼠标点击在 Native声明方法上时：

1. 检查文件类型，如果为java就继续
2. 获取当前行的上下共三行字符串数据，使用正则表达式获取native声明的方法。
3. 检查当前模块目录下的jni或者cpp目录下的.c或者.cpp文件。
4. 如果没有文件，弹窗让用户创建一个C/C++文件，并追加转换后(如何转换会有一个专门的类)的Java2C方法在文件末尾. 在IDE打开此文件。
5. 如果JNI或者cpp目录有一个以上的C/C++文件, 弹窗让用户选择一个C/C++文件或者创建，之后打开文件追加转换后(如何转换会有一个专门的类)的Java2C方法在文件末尾.。 在IDE打开此文件。

上面逻辑中：

1. 文件类型 ，IntelliJ IDEA 的plugin开发API中可以获取到
2. 获取当前行的上下共三行字符串数据 ，IntelliJ IDEA 的plugin开发API中可以获取到
3. 模块目录的API暂时没找到
4. 在IDE打开C/C++文件，不知道用什么接口

三、CMake之CMakeLists.txt编写入门

自定义变量

主要有隐式定义和显式定义两种。

隐式定义的一个例子是 `PROJECT` 指令，它会隐式的定义 `< projectname >_BINARY_DIR` 和 `< projectname >_SOURCE_DIR` 两个变量；显式定义使用 `SET` 指令构建自定义变量，比如: `SET(HELLO_SRCmain.c)` 就可以通过 `${HELLO_SRC}` 来引用这个自定义变量。



0



变量引用方式

使用 `${}` 进行变量的引用；在 `IF` 等语句中,是直接使用变量名而不通过 `${}` 取...

常用变量

`CMAKE_BINARY_DIR`

`PROJECT_BINARY_DIR`

`< projectname >_BINARY_DIR`

这三个变量指代的内容是一致的，如果是in-source编译，指得就是工程顶层目录；如果是out-of-source编译，指的是工程编译发生的目录。`PROJECT_BINARY_DIR` 跟其它指令稍有区别，目前可以认为它们是一致的。

`CMAKE_SOURCE_DIR`

`PROJECT_SOURCE_DIR`

`< projectname >_SOURCE_DIR`

这三个变量指代的内容是一致的，不论采用何种编译方式，都是工程顶层目录。也就是在in-source编译时,他跟 `CMAKE_BINARY_DIR` 等变量一致。`PROJECT_SOURCE_DIR` 跟其它指令稍有区别,目前可以认为它们

是一致的。

(out-of-source build与in-source build相对，指是否在CMakeLists.txt所在目录进行编译。)

CMAKE_CURRENT_SOURCE_DIR

当前处理的CMakeLists.txt所在的路径，比如上面我们提到的src子目录。

CMAKE_CURRENT_BINARY_DIR

如果是in-source编译，它跟 `CMAKE_CURRENT_SOURCE_DIR` 一致；如果是out-of-source编译，指的是target编译目录。使用 `ADD_SUBDIRECTORY(src bin)` 可以更改这个变量的值。使用 `SET(EXECUTABLE_OUTPUT_PATH <新路径>)` 并不会对这个变量造成影响,它仅仅修改了最终目标文件存放的路径。



0



CMAKE_CURRENT_LIST_FILE

输出调用这个变量的CMakeLists.txt的完整路径

CMAKE_CURRENT_LIST_LINE

输出这个变量所在的行

CMAKE_MODULE_PATH

这个变量用来定义自己的cmake模块所在的路径。如果工程比较复杂，有可能会自己编写一些cmake模块，这些cmake模块是随工程发布的，为了让cmake在处理CMakeLists.txt时找到这些模块，你需要通过SET指令将cmake模块路径设置一下。比如 `SET(CMAKE_MODULE_PATH,${PROJECT_SOURCE_DIR}/cmake)` 这时候就可以通过INCLUDE指令来调用自己的模块了。

EXECUTABLE_OUTPUT_PATH

新定义最终结果的存放目录

LIBRARY_OUTPUT_PATH

新定义最终结果的存放目录

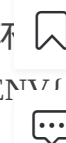
PROJECT_NAME

返回通过 **PROJECT** 指令定义的项目名称。

cmake调用环境变量的方式



使用 `$ENV{NAME}` 指令就可以调用系统的环境变量了。比如 `MESSAGE(STATUS "HOME dir: $ENV{HOME}")` 设置环境变量的方式是 `SET(ENV{变量名} 值)`。



1. CMAKE_INCLUDE_CURRENT_DIR

自动添加 **CMAKE_CURRENT_BINARY_DIR** 和 **CMAKE_CURRENT_SOURCE_DIR** 到当前处理的 CMakeLists.txt , 相当于在每个 CMakeLists.txt 加入 : **INCLUDE_DIRECTORIES(\${CMAKE_CURRENT_BINARY_DIR} \${CMAKE_CURRENT_SOURCE_DIR})**

2. CMAKE_INCLUDE_DIRECTORIES_PROJECT_BEFORE

将工程提供的头文件目录始终置于系统头文件目录的前面,当定义的头文件确实跟系统发生冲突时可以提供一些帮助。

3. CMAKE_INCLUDE_PATH和CMAKE_LIBRARY_PATH

系统信息

- **CMAKE_MAJOR_VERSION** , CMAKE主版本号, 比如2.4.6中的2
- **CMAKE_MINOR_VERSION** , CMAKE次版本号, 比如2.4.6中的4

- `CMAKE_PATCH_VERSION` , CMAKE补丁等级, 比如2.4.6中的6
- `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平台为 `TRUE` , 包括OSX和cygwin
- `WIN32` , 在所有的Win32平台为 `TRUE` , 包括cygwin



0



主要的开关选项

1. CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS

用来控制 `IF ELSE` 语句的书写方式。

2. 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` 并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_LIBSON)` 后, 默认生成的为动态库。

3. CMAKE_C_FLAGS

设置C编译选项, 也可以通过指令 `ADD_DEFINITIONS()` 添加。

4. MAKE_CXX_FLAGS

设置C++编译选项, 也可以通过指令 `ADD_DEFINITIONS()` 添加。

cMake常用指令

这里引入更多的cmake指令,为了编写的方便,将按照cmakeman page 的顺序介绍各种指令,不再推荐使用的指令将不再介绍。

基本指令

PROJECT(HELLO)

指定项目名称,生成的VC项目的名称,使用 `${HELLO_SOURCE_DIR}` 表示项目目录。



0

INCLUDE_DIRECTORIES

指定头文件的搜索路径,相当于指定gcc的-I参数

`INCLUDE_DIRECTORIES(${HELLO_SOURCE_DIR}/Hello)` #增加Hello为include目录



TARGET_LINK_LIBRARIES

添加链接库,相同于指定-l参数

`TARGET_LINK_LIBRARIES(demoHello)` #将可执行文件与Hello连接成最终文件demo

LINK_DIRECTORIES

动态链接库或静态链接库的搜索路径,相当于gcc的-L参数

`LINK_DIRECTORIES(${HELLO_BINARY_DIR}/Hello)` #增加Hello为link目录

ADD_DEFINITIONS

向C/C++编译器添加-D定义,比如:

`ADD_DEFINITIONS(-DENABLE_DEBUG-DABC)`

参数之间用空格分割。如果代码中定义了:

```
1  #ifdef ENABLE_DEBUG
2
3  #endif
```

这个代码块就会生效。如果要添加其他的编译器开关,可以通过 `CMAKE_C_FLAGS` 变量和 `CMAKE_CXX_FLAGS` 变量设置。

ADD_DEPENDENCIES*

定义 target 依赖的其它 target , 确保在编译本 target 之前 , 其它的 target 已经被构建。 `ADD_DEPENDENCIES(target-name depend-target1 depend-target2 ...)`

ADD_EXECUTABLE

`ADD_EXECUTABLE(helloDemo demo.cxx demo_b.cxx)`

指定编译, 好像也可以添加.o文件, 将.cxx编译成可执行文件

ADD_LIBRARY

`ADD_LIBRARY(Hellohello.cxx) #将hello.cxx编译成静态库如libHello.a`

ADD_SUBDIRECTORY

`ADD_SUBDIRECTORY(Hello) #包含子目录`

ADD_TEST

ENABLE_TESTING

`ENABLE_TESTING` 指令用来控制Makefile是否构建test目标, 涉及工程所有目录。语法很简单, 没有任何参数, `ENABLE_TESTING()` 一般放在工程的主CMakeLists.txt中。

`ADD_TEST` 指令的语法是: `ADD_TEST(testnameExename arg1 arg2 ...)`

testname是自定义的test名称, Exename可以是构建的目标文件也可以是外部脚本等等, 后面连接传递给可执行文件的参数。

如果没有在同一个CMakeLists.txt中打开 `ENABLE_TESTING()` 指令, 任何 `ADD_TEST` 都是无效的。比如前面的HelloWorld例子, 可以在工程主CMakeLists.txt中添加




0



```
1 ADD_TEST(mytest ${PROJECT_BINARY_DIR}/bin/main)
2 ENABLE_TESTING
```

生成Makefile后，就可以运行 `make test` 来执行测试了。

AUX_SOURCE_DIRECTORY

基本语法是: `AUX_SOURCE_DIRECTORY(dir VARIABLE)`，作用是发现一个目录下  的源代码文件并将列表存储在一个变量中，这个指令临时被用来自动构建源文件列表，因为目前cmake还不能自动发现新添加的源文件。比如：

```
1 AUX_SOURCE_DIRECTORY(. SRC_LIST)
2 ADD_EXECUTABLE(main ${SRC_LIST})
```

可以通过后面提到的 `FOR EACH` 指令来处理这个LIST。

CMAKE_MINIMUM_REQUIRED

语法为 `CMAKE_MINIMUM_REQUIRED(VERSION versionNumber [FATAL_ERROR])`，

比如: `CMAKE_MINIMUM_REQUIRED(VERSION 2.5 FATAL_ERROR)`

如果cmake版本小与2.5，则出现严重错误，整个过程中止。

EXEC_PROGRAM

在CMakeLists.txt处理过程中执行命令，并不会在生成的Makefile中执行。具体语法为:

```
1 EXEC_PROGRAM(Executable [directory in which to run] [ARGS <arguments to executable>] [OUTPUT_VARIABLE <var>]
```

用于在指定的目录运行某个程序，通过 ARGs 添加参数，如果要获取输出和返回值，可通过 **OUTPUT_VARIABLE** 和 **RETURN_VALUE** 分别定义两个变量。

这个指令可以帮助在CMakeLists.txt处理过程中支持任何命令，比如根据系统情况去修改代码文件等等。举个简单的例子，我们要在src目录执行ls命令，并把结果和返回值存下来，可以直接在src/CMakeLists.txt中添加：

```
1 EXEC_PROGRAM(ls ARGS "*.c" OUTPUT_VARIABLE LS_OUTPUT RETURN_VALUE LS_RESULT)
2 IF(not LS_RESULT)
3     MESSAGE(STATUS "ls result: " ${LS_OUTPUT})
4 ENDIF(not LS_RESULT)
```

在cmake生成Makefile过程中，就会执行ls命令，如果返回0，则说明成功执行了，那么就输出 **ls *.c** 的结果。关于 **IF** 语句，后面的控制指令会提到。

FILE指令


文件操作指令，基本语法为:

```
1 FILE(WRITEfilename "message to write"... )
2 FILE(APPENDfilename "message to write"... )
3 FILE(READfilename variable)
4 FILE(GLOBvariable [RELATIVE path] [globbing expression_r_rs]...)
5 FILE(GLOB_RECURSEvariable [RELATIVE path] [globbing expression_r_rs]...)
6 FILE(REMOVE[directory]...)
7 FILE(REMOVE_RECURSE[directory]...)
8 FILE(MAKE_DIRECTORY[directory]...)
9 FILE(RELATIVE_PATHvariable directory file)
10 FILE(TO_CMAKE_PATHpath result)
11 FILE(TO_NATIVE_PATHpath result)
```

INCLUDE指令

用来载入CMakeLists.txt文件，也用于载入预定义的cmake模块。

```
1 INCLUDE(file1[OPTIONAL])
2 INCLUDE(module[OPTIONAL])
```

OPTIONAL参数的作用是文件不存在也不会产生错误，可以指定载入一个 ，如果定义的是一个模块，那么将在CMAKE_MODULE_PATH中搜索这个模块并载入，载入的内容将在处理到INCLUDE语句是直接执行。



INSTALL指令



FIND_指令

FIND_系列指令主要包含一下指令:

```
1 FIND_FILE(<VAR>name1 path1 path2 ...) VAR变量代表找到的文件全路径,包含文件名
2 FIND_LIBRARY(<VAR>name1 path1 path2 ...) VAR变量表示找到的库全路径,包含库文件名
3 FIND_PATH(<VAR>name1 path1 path2 ...) VAR变量代表包含这个文件的路径
4 FIND_PROGRAM(<VAR>name1 path1 path2 ...) VAR变量代表包含这个程序的全路径
5 FIND_PACKAGE(<name>[major.minor] [QUIET] [NO_MODULE] [[REQUIRED | COMPONENTS][componets...]]) 用来调用
```

FIND_LIBRARY示例:

```
1 FIND_LIBRARY(libXX11 /usr/lib)
2 IF(NOT libX)
3     MESSAGE(FATAL_ERROR "libX not found")
4 ENDIF(NOT libX)
```


控制指令

IF指令，基本语法为：

```
1 IF(expression_r_r)
2   #THEN section.
3   COMMAND1(ARGS...)
4   COMMAND2(ARGS...)
5   ...
6 ELSE(expression_r_r)
7   #ELSE section.
8   COMMAND1(ARGS...)
9   COMMAND2(ARGS...)
10  ...
11 ENDIF(expression_r_r)
```



0



另外一个指令是 **ELSEIF**，总体把握一个原则，凡是出现IF的地方一定要有对应的 **ENDIF**，出现 **ELSEIF** 的地方，**ENDIF** 是可选的。表达式的使用方法如下：

```
1 IF(var) 如果变量不是：空, 0, N, NO, OFF, FALSE, NOTFOUND 或 <var>_NOTFOUND时，表达式为真。
2 IF(NOT var)，与上述条件相反。
3 IF(var1AND var2)，当两个变量都为真是为真。
4 IF(var1OR var2)，当两个变量其中一个为真时为真。
5 IF(COMMANDcmd)，当给定的cmd确实是命令并可以调用是为真。
6 IF(EXISTS dir) or IF(EXISTS file)，当目录名或者文件名存在时为真。
7 IF(file1IS_NEWER_THAN file2)，当file1比file2新，或者file1/file2其中有一个不存在时为真文件名请使用完整路径。
8 IF(IS_DIRECTORY dirname)，当dirname是目录时为真。
9 IF(variableMATCHES regex)
```

IF(string MATCHES regex) 当给定的变量或者字符串能够匹配正则表达式 **regex** 时为真。比如：

```
1 IF("hello" MATCHES "hello")
2     MESSAGE("true")
3 ENDEF("hello" MATCHES "hello")
```

```
1 IF(variable LESS number)
2 IF(string LESS number)
3 IF(variable GREATER number)
4 IF(string GREATER number)
5 IF(variable EQUAL number)
6 IF(string EQUAL number)
```

数字比较表达式

```
1 IF(variable STRLESS string)
2 IF(string STRLESS string)
3 IF(variable STRGREATER string)
4 IF(string STRGREATER string)
5 IF(variable STREQUAL string)
6 IF(string STREQUAL string)
```

按照字母序的排列进行比较。

IF(DEFINED variable) , 如果变量被定义, 为真。

一个小例子,用来判断平台差异:

```
1 IF(WIN32)
2     MESSAGE(STATUS"This is windows.") #作一些Windows相关的操作
3 ELSE(WIN32)
4     MESSAGE(STATUS"This is not windows") #作一些非Windows相关的操作
5 ENDEF(WIN32)
```



上述代码用来控制在不同的平台进行不同的控制,但是阅读起来却并不是那么舒服, `ELSE(WIN32)` 之类的语句很容易引起歧义。

这就用到了我们在 常用变量 一节提到的 `CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS` 开关。可以 `SET(CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTSON)` , 这时候就可以写成:

```
1 IF(WIN32)
2
3 ELSE()
4
5 ENDIF()
```



0



如果配合ELSEIF使用,可能的写法是这样:

```
1 IF(WIN32)
2     #dosomething related to WIN32
3 ELSEIF(UNIX)
4     #dosomething related to UNIX
5 ELSEIF(APPLE)
6     #dosomething related to APPLE
7 ENDIF(WIN32)
```

WHILE指令

WHILE指令的语法是:

```
1 WHILE(condition)
2     COMMAND1(ARGS...)
3     COMMAND2(ARGS...)
```

```
4    ...  
5    ENDWHILE(condition)
```

其真假判断条件可以参考IF指令。

FOREACH指令

FOREACH指令的使用方法有三种形式:

(1)列表

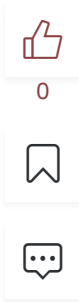
```
1    FOREACH(loop_vararg1 arg2 ...)  
2        COMMAND1(ARGS...)  
3        COMMAND2(ARGS...)  
4        ...  
5    ENDFOREACH(loop_var)
```

像我们前面使用的 **AUX_SOURCE_DIRECTORY** 的例子

```
1    AUX_SOURCE_DIRECTORY(.SRC_LIST)  
2    FOREACH(F ${SRC_LIST})  
3        MESSAGE(${F})  
4    ENDFOREACH(F)
```

(2)范围

```
1    FOREACH(loop_var RANGE total)  
2  
3    ENDFOREACH(loop_var)
```



0

从0到total以1为步进，举例如下：

```
1  FOREACH(VARRANGE 10)
2    MESSAGE(${VAR})
3  ENDFOREACH(VAR)
```

最终得到的输出是：

```
1  0
2  1
3  2
4  3
5  4
6  5
7  6
8  7
9  8
10 9
11 10
```

(3)范围和步进

```
1  FOREACH(loop_var RANGE start stop [step])
2
3  ENDFOREACH(loop_var)
```

从start开始到stop结束，以step为步进。举例如下：



0



```
1 FOREACH(A RANGE 5 15 3)
2   MESSAGE(${A})
3 ENDFOREACH(A)
```

最终得到的结果是:

```
1 5
2 8
3 11
4 14
```



0



这个指令需要注意的是，直到遇到 **ENDFOREACH** 指令，整个语句块才会得到执行。

复杂的例子:模块的使用和自定义模块

这里将着重介绍系统预定义的Find模块的使用以及自己编写Find模块，系统中提供了其他各种模块，一般情况需要使用 **INCLUDE** 指令显式的调用，**FIND_PACKAGE** 指令是一个特例，可以直接调用预定义的模块。

其实纯粹依靠cmake本身提供的基本指令来管理工程是一件非常复杂的事情，所以cmake设计成了可扩展的架构，可以通过编写一些通用的模块来扩展cmake。

首先介绍一下cmake提供的 **FindCURL** 模块的使用，然后基于前面的 **libhello** 共享库，编写一个 **FindHello.cmake** 模块。

（一）使用FindCURL模块

在/backup/cmake目录建立t5目录,用于存放CURL的例子。

建立src目录,并建立src/main.c,内容如下:

```
1 #include<curl/curl.h>
2 #include<stdio.h>
3 #include<stdlib.h>
4 #include<unistd.h>
5
6 FILE*fp;
7
8 intwrite_data(void *ptr, size_t size, size_t nmemb, void *stream)
9 {
10     int written = fwrite(ptr, size, nmemb, (FILE *)fp);
11     return written;
12 }
13
14 int main()
15 {
16     const char *path = "/tmp/curl-test";
17     const char *mode = "w";
18     fp = fopen(path,mode);
19     curl_global_init(CURL_GLOBAL_ALL);
20     CURL coderes;
21     CURL *curl = curl_easy_init();
22
23     curl_easy_setopt(curl, CURLOPT_URL, "http://www.linux-ren.org");
24     curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_data);
25     curl_easy_setopt(curl, CURLOPT_VERBOSE, 1);
26     res = curl_easy_perform(curl);
27     curl_easy_cleanup(curl);
28 }
```



0



这段代码的作用是通过curl取回 www.linux-ren.org 的首页并写入 `/tmp/curl-test` 文件中

建立主工程文件 `CmakeLists.txt` , 如下 :

```
1 PROJECT(CURLTEST)
2 ADD_SUBDIRECTORY(src)
```

建立src/CmakeLists.txt

```
1 ADD_EXECUTABLE(curltest main.c)
```

现在自然是没办法编译的 , 我们需要添加curl的头文件路径和库文件。

方法1:

直接通过 `INCLUDE_DIRECTORIES` 和 `TARGET_LINK_LIBRARIES` 指令添加:

我们可以直接在 `src/CMakeLists.txt` 中添加:

```
1 INCLUDE_DIRECTORIES(/usr/include)
2 TARGET_LINK_LIBRARIES(curltestcurl)
```

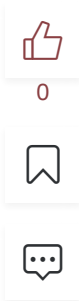
然后建立build目录进行外部构建即可。

现在要探讨的是使用cmake提供的FindCURL模块。

方法2:

使用 `FindCURL` 模块。向 `src/CMakeLists.txt` 中添加:

```
1 FIND_PACKAGE(CURL)
2 IF(CURL_FOUND)
```




```

3  INCLUDE_DIRECTORIES(${CURL_INCLUDE_DIR})
4  TARGET_LINK_LIBRARIES(curltest${CURL_LIBRARY})
5  ELSE(CURL_FOUND)
6  MESSAGE(FATAL_ERROR"CURL library not found")
7  ENDIF(CURL_FOUND)

```

对于系统预定义的 `Find<name>.cmake` 模块,使用方法一般如上例所示,每一块都会定义以下几个变量:

```

1  <name>_FOUND
2  <name>_INCLUDE_DIR or <name>_INCLUDES
3  <name>_LIBRARY or <name>_LIBRARIES

```

可以通过 `<name>_FOUND` 来判断模块是否被找到,如果没有找到,按照工程的需要关闭某些特性、给出提醒或者中止编译,上面的例子就是报出致命错误并终止构建。

如果 `<name>_FOUND` 为真,则将 `<name>_INCLUDE_DIR` 加入 `INCLUDE_DIRECTORIES`,将 `<name>_LIBRARY` 加入 `TARGET_LINK_LIBRARIES` 中。

我们再来看一个复杂的例子,通过 `<name>_FOUND` 来控制工程特性:

```

1  SET(mySourcesviewer.c)
2  SET(optionalSources)
3  SET(optionalLibs)
4
5  FIND_PACKAGE(JPEG)
6
7  IF(JPEG_FOUND)
8      SET(optionalSources${optionalSources} jpegview.c)
9      INCLUDE_DIRECTORIES(${JPEG_INCLUDE_DIR} )
10     SET(optionalLibs${optionalLibs} ${JPEG_LIBRARIES} )

```

```
10     ADD_DEFINITIONS(-DENABLE_JPEG_SUPPORT)
11 ENDIF(JPEG_FOUND)
12
13 IF(PNG_FOUND)
14     SET(optionalSources${optionalSources} pngview.c)
15     INCLUDE_DIRECTORIES(${PNG_INCLUDE_DIR} )
16     SET(optionalLibs${optionalLibs} ${PNG_LIBRARIES} )
17     ADD_DEFINITIONS(-DENABLE_PNG_SUPPORT)
18 ENDIF(PNG_FOUND)
19
20 ADD_EXECUTABLE(viewer${mySources} ${optionalSources}
21 TARGET_LINK_LIBRARIES(viewer${optionalLibs}
```



0



通过判断系统是否提供了JPEG库来决定程序是否支持JPEG功能。

（二）编写属于自己的FindHello模块

接下来在t6示例中演示如何自定义 **FindHELLO** 模块并使用这个模块构建工程。在 **/backup/cmake/** 中建立t6目录，并在其中建立cmake目录用于存放我们自己定义的 **FindHELLO.cmake** 模块，同时建立src目录，用于存放我们的源文件。

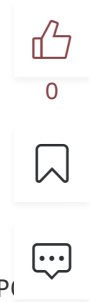
1.定义 **cmake/FindHELLO.cmake** 模块

```
1 FIND_PATH(HELLO_INCLUDE_DIR hello.h /usr/include/hello /usr/local/include/hello)
2
3 FIND_LIBRARY(HELLO_LIBRARY NAMES hello PATH /usr/lib /usr/local/lib)
4
5 IF(HELLO_INCLUDE_DIR AND HELLO_LIBRARY)
6     SET(HELLO_FOUNDTRUE)
7 ENDIF(HELLO_INCLUDE_DIR AND HELLO_LIBRARY)
8
```

```

9  IF(HELLO_FOUND)
10     IF(NOT HELLO_FIND_QUIETLY)
11         MESSAGE(STATUS"Found Hello: ${HELLO_LIBRARY}")
12     ENDIF(NOT HELLO_FIND_QUIETLY)
13 ELSE(HELLO_FOUND)
14     IF(HELLO_FIND_REQUIRED)
15         MESSAGE(FATAL_ERROR"Could not find hello library")
16     ENDIF(HELLO_FIND_REQUIRED)
17 ENDIF(HELLO_FOUND)

```



针对上面的模块让我们再来回顾一下 **FIND_PACKAGE** 指令:

```

1  FIND_PACKAGE(<name>[major.minor] [QUIET] [NO_MODULE] [[REQUIRED | COMPONETS] S][componets...]))

```

前面的CURL例子中我们使用了最简单的 **FIND_PACKAGE** 指令，其实它可以使用多种参数：

QUIET 参数，对应与我们编写的 **FindHELLO** 中的 **HELLO_FIND_QUIETLY**，如果不指定这个参数,就会执行:

```

1  MESSAGE(STATUS"Found Hello: ${HELLO_LIBRARY}")

```

REQUIRED 参数，其含义是指这个共享库是否是工程必须的，如果使用了这个参数，说明这个链接库是必备库，如果找不到这个链接库，则工程不能编译。对应于 **FindHELLO.cmake** 模块中的 **HELLO_FIND_REQUIRED** 变量。

同样,我们在上面的模块中定义了 **HELLO_FOUND**，**HELLO_INCLUDE_DIR**，**HELLO_LIBRARY** 变量供开发者在 **FIND_PACKAGE** 指令中使用。

下面建立 **src/main.c** ,内容为:

```
1 #include<hello.h>
2 int main()
3 {
4     HelloFunc();
5     return 0;
6 }
```

建立 `src/CMakeLists.txt` 文件,内容如下:

```
1 FIND_PACKAGE(HELLO)
2 IF(HELLO_FOUND)
3     ADD_EXECUTABLE(hellomain.c)
4     INCLUDE_DIRECTORIES(${HELLO_INCLUDE_DIR})
5     TARGET_LINK_LIBRARIES(hello${HELLO_LIBRARY})
6 ENDIF(HELLO_FOUND)
```

为了能够让工程找到 FindHELLO.cmake 模块(存放在工程中的cmake目录)我们在主工程文件CMakeLists.txt 中加入:

```
1 SET(CMAKE_MODULE_PATH${PROJECT_SOURCE_DIR}/cmake)
```

(三) 使用自定义的FindHELLO模块构建工程

仍然采用外部编译的方式,建立build目录,进入目录运行:

```
1 cmake ..
```

我们可以从输出中看到:



0



```
1 FoundHello: /usr/lib/libhello.so
```

如果我们把上面的 `FIND_PACKAGE(HELLO)` 修改为 `FIND_PACKAGE(HELLO QUIET)`,

不会看到上面的输出。接下来就可以使用make命令构建工程,运行:

```
1 ./src/hello
```

可以得到输出

```
1 HelloWorld
```

说明工程成功构建。

(四) 如果没有找到hellolibrary呢?

我们可以尝试将 `/usr/lib/libhello.x` 移动到/tmp目录,这样按照 `FindHELLO` 模块的定义,找不到 `hellolibrary` 了,我们再来看一下构建结果:

```
1 cmake ..
```

仍然可以成功进行构建,但是这时候是没有办法编译的。

修改 `FIND_PACKAGE(HELLO)` 为 `FIND_PACKAGE(HELLO REQUIRED)`, 将 `hellolibrary` 定义为工程必须的共享库。

这时候再次运行



```
1 | cmake ..
```

我们得到如下输出:

```
1 | CMakeError: Could not find hello library.
```

因为找不到libhello.x , 所以,整个Makefile生成过程被出错中止。



一些问题

1.怎样区分debug、release版本

建立debug/release两目录，分别在其中执行 `cmake -D CMAKE_BUILD_TYPE=Debug (或Release)`，需要编译不同版本时进入不同目录执行 `make` 即可：

```
1 | Debug版会使用参数-g；  
2 | Release版使用-O3-DNDEBUG
```

另一种设置方法——例如 `DEBUG` 版设置编译参数 `DDEBUG`

```
1 | IF(DEBUG_mode)  
2 |     add_definitions(-DDEBUG)  
3 | ENDIF()
```

在执行 `cmake` 时增加参数即可，例如 `cmake -D DEBUG_mode=ON`

2.怎样设置条件编译

例如 `debug` 版设置编译选项 `DEBUG`，并且更改不应改变 `CMakeList.txt`

使用 `option command` , eg :

```
1 option(DEBUG_mode"ON for debug or OFF for release" ON)
2 IF(DEBUG_mode)
3     add_definitions(-DDEBUG)
4 ENDIF()
```

使其生效的方法：首先 `cmake` 生成 `makefile` , 然后 `make edit_cache` 编辑编译选项；Linux下会打开一个文本框，可以更改，改完后再`make`生成目标文件——`emacs`不支持 `make edit_cache` ；

局限：这种方法不能直接设置生成的`makefile`，而是必须使用命令在`make`中设置参数；对于`debug`、`release`版本，相当于需要两个目录，分别先`cmake`一次，然后分别`makeedit_cache`一次；

期望的效果：在执行`cmake`时直接通过参数指定一个开关项，生成相应的 `makefile` 。

四、cmake 基本命令 & 交叉编译配置 & 模块的编写

cmake 基本命令：

```
1 cmake_minimum_required(VERSION 2.8.2 FATAL_ERROR)
2
3 project("ProjName")
4
5 // 不推荐使用add_definitions来设置编译选项，因为其作用如同cmake -D
6 add_definitions(
7     -std=c++11 # Or -std=c++0x
8     -Wall
9     -Wfatal-errors
```

```
10  -DXXX    #等于gcc -DXXX
11  # Other flags
12  )
13
14  // 一般通过下条语句设置编译选项
15  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -Wall -Wfatal-errors -fPIC")
16
17  //变量 : CMAKE_PREFIX_PATH: Path used for searching by FIND_XXX(), with appropriate suffixes added.
18  set(CMAKE_PREFIX_PATH /path/path/path)
19
20  //引用环境变量
21  export FOO=/use/lib # 在bash中
22  $ENV{FOO} # 在CMakeLists.txt中
23
24  //头文件路径
25  include_directories(
26      include
27      relative/path
28      /absolute/path/
29  )
30
31  //链接库目录
32  link_directories(directory1 directory2 ...)
33
34  //链接函数库
35  target_link_libraries(a.out mylib ompl) //可以是cmake中的target，也可以是某个目录中的库文件，如 libompl.so，等
36
37  //源文件路径，在子目录中
38  add_subdirectory (
39      someDirectory
40      src/otherDirecotry
41  )
```



0




```
42 //寻找某个目录中的所有源文件，格式：aux_source_directory(<dir> <variable>)
43 aux_source_directory(src_srcFiles)
44
45 //生成库文件
46 add_library(mylib [SHARED | STATIC]
47     mylib.cpp
48     other.cpp
49 )
50
51 //生成可执行程序
52 add_executable(a.out
53     main.cpp
54     src/func.cpp
55 )
56
57 //设置变量
58 set(someVariable "some string")
59
60 //打印消息
61 message(STATUS "some status ${someVariable}")
62
63 //list操作
64 list(REMOVE_ITEM _srcFiles "src/f4.cpp") //从变量中去掉某一项
65 list(APPEND <list> <element> [<element> ...]) //添加某一项到变量中
66
67 //检查编译器是否支持某一个编译选项
68 CHECK_CXX_COMPILER_FLAG("-std=c++11" COMPILER_SUPPORTS_CXX11)
69 CHECK_CXX_COMPILER_FLAG("-std=c++0x" COMPILER_SUPPORTS_CXX0X)
70 if(COMPILER_SUPPORTS_CXX11)
71     set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")
72 elseif(COMPILER_SUPPORTS_CXX0X)
73     set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x")
74 else()
```



0



```

74     message(STATUS "The compiler ${CMAKE_CXX_COMPILER} has no C++11 support. Please use a different C++ compiler.")
75 endif()
76

```

常用 find_package 找 boost 库和头文件：

```

1  # Put this in your CMakeLists.txt file (change any options from OFF to ON if you want)
2
3  set(Boost_USE_STATIC_LIBS OFF) # 不写这几个就是默认设置
4  set(Boost_USE_MULTITHREADED ON)
5  set(Boost_USE_STATIC_RUNTIME OFF)
6
7  find_package(Boost 1.59.0 COMPONENTS *boost libraries here* REQUIRED) // 如头文件库，则不需要写 COMPONENTS
8
9  if(Boost_FOUND)
10     include_directories(${Boost_INCLUDE_DIRS})
11     add_executable(progname file1.cxx file2.cxx)
12     target_link_libraries(progname ${Boost_LIBRARIES})
13 endif()
14
15 # Obviously you need to put the libraries you want where I put *boost libraries here*.
16 # For example, if you're using the filesystem and regex library you'd write:
17
18 find_package(Boost 1.59.0 COMPONENTS filesystem regex REQUIRED) # 一般包含头文件是 <boost/xxx/*>, 则 COMPONENTS 是必须的

```

cmake marco & function

```

1  set(var "ABC")
2
3  macro(Moo arg) # 定义macro
4      message("arg = ${arg}")
5      set(arg "abc")
6  endmacro()

```

```
5 message("# After change the value of arg.")
6 message("arg = ${arg}")
7 endmacro()
8 message("=== Call macro ===")
9 Moo(${var}) # 调用macro
10
11 function(Foo arg) # 定义函数
12     message("arg = ${arg}")
13     set(arg "abc")
14     message("# After change the value of arg.")
15     message("arg = ${arg}")
16 endfunction()
17 message("=== Call function ===")
18 Foo(${var}) # 调用函数
19 and the output is
20
21 === Call macro ===
22 arg = ABC
23 # After change the value of arg.
24 arg = ABC
25 === Call function ===
26 arg = ABC
27 # After change the value of arg.
28 arg = abc
29
30 #注意，marco的调用相当于c/c++的预编译器，只会进行字符串替换（这就是为啥 arg 没有被改变）；而函数则可以进
```



0



//常用变量

CMAKE_SOURCE_DIR (相当于工程根目录)

this is the directory, from which cmake was started, i.e. the top level source directory

CMAKE_CURRENT_SOURCE_DIR

this is the directory where the currently processed CMakeLists.txt is located in

PROJECT_SOURCE_DIR (=CMAKE_SOURCE_DIR 相当于工程根目录)

contains the full path to the root of your project source directory, i.e. to the nearest directory where CMakeLists.txt contains the PROJECT() command

CMAKE_PREFIX_PATH (用于找 Findxxx.cmake文件 , 找 库 和 头文件 ,
Path used for searching by FIND_XXX(), with appropriate suffixes added.

CMAKE_INSTALL_PREFIX (安装目录)

Install directory used by install.

If “make install” is invoked or INSTALL is built, this directory is prepended onto all install directories.

This variable defaults to /usr/local on UNIX and c:/Program Files on Windows.

例如 : cmake .. -DCMAKE_INSTALL_PREFIX=/my/paht/to/install



cmake 配置交叉编译环境 :

```
1 // cmake的交叉编译环境设置,创建文件toolchain.cmake , 添加以下内容 :
2
3 # this one is important
4 SET(CMAKE_SYSTEM_NAME Linux)
5 #this one not so much
6 SET(CMAKE_SYSTEM_VERSION 1)
7
8 # specify the cross compiler
9 SET(CMAKE_C_COMPILER /opt/arm/arm-linux-gnueabi-hf-gcc)
10 SET(CMAKE_CXX_COMPILER /opt/arm/arm-linux-gnueabi-hf-g++)
```

```
11
12 # where is the target environment
13 SET(CMAKE_FIND_ROOT_PATH /opt/arm/install)
14
15 # search for programs in the build host directories
16 SET(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
17 # for libraries and headers in the target directories
18 SET(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
19 SET(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
20
21 // If this file is named toolchain.cmake and is located in your home directory
22 // and you are building in the subdirectory build then you can do:
23
24 ~/src$ cd build
25 ~/src/build$ cmake -DCMAKE_TOOLCHAIN_FILE=~/.toolchain.cmake ..
```



0



注意：在交叉编译的时候，如果某些 FindXXX.cmake 模块中有类似 `pkg_search_module` 或者 `pkg_check_modules` 等命令，则会有点问题：

FindXXX.cmake modules, which rely on executing a binary tool like pkg-config may have problems, since the pkg-config of the target platform cannot be executed on the host. Tools like pkg-config should be used only optional in FindXXX.cmake files.

可以找到相应的模块的 FindXXX.cmake 替换其 pkg-config

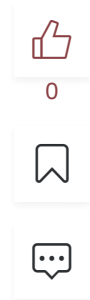
如果不想让 pkg-config 被执行，可以试着：

```
1 SET(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM ONLY) // 即不让cmake的 find_program 去host环境中找可执行文件
```

如果 cmake cache 了一些变量，需要重新运行cmake，只需要删除 CMakeCache.txt 文件即可

关于如何编写自己的 Findxxx.cmake 文件：

尊重原作，以下部分复制了该作者的部分文件内容，[see link](#)



```
1  set(MYSIMPLEPACKAGE_ROOT_DIR
2      "${MYSIMPLEPACKAGE_ROOT_DIR}"
3      CACHE
4      PATH
5      "Directory to search")
6
7  if(CMAKE_SIZEOF_VOID_P MATCHES "8")
8      set(_LIBSUFFIXES /lib64 /lib)
9  else()
10     set(_LIBSUFFIXES /lib)
11 endif()
12
13 find_library(MYSIMPLEPACKAGE_LIBRARY
14             NAMES
15             mysimplepackage
16             PATHS
17             "${MYSIMPLEPACKAGE_ROOT_DIR}"
18             PATH_SUFFIXES
19             "${_LIBSUFFIXES}")
20
21 # Might want to look close to the library first for the includes.
22 get_filename_component(_libdir "${MYSIMPLEPACKAGE_LIBRARY}" PATH)
```

```
23 find_path(MYSIMPLEPACKAGE_INCLUDE_DIR
24     NAMES
25     mysimplepackage.h
26     HINTS
27     "${_libdir}" # the library I based this on was sometimes bundled right next to its include
28     "${_libdir}/.."
29     PATHS
30     "${MYSIMPLEPACKAGE_ROOT_DIR}"
31     PATH_SUFFIXES
32     include/)
33
34 # There's a DLL to distribute on Windows - find where it is.
35 set(_deps_check)
36 if(WIN32)
37     find_file(MYSIMPLEPACKAGE_RUNTIME_LIBRARY
38         NAMES
39         mysimplepackage.dll
40         HINTS
41         "${_libdir}")
42     set(MYSIMPLEPACKAGE_RUNTIME_LIBRARIES
43         "${MYSIMPLEPACKAGE_RUNTIME_LIBRARY}")
44     get_filename_component(MYSIMPLEPACKAGE_RUNTIME_LIBRARY_DIRS
45         "${MYSIMPLEPACKAGE_RUNTIME_LIBRARY}"
46         PATH)
47     list(APPEND _deps_check MYSIMPLEPACKAGE_RUNTIME_LIBRARY)
48 else()
49     get_filename_component(MYSIMPLEPACKAGE_RUNTIME_LIBRARY_DIRS
50         "${MYSIMPLEPACKAGE_LIBRARY}"
51         PATH)
52 endif()
53
54 include(FindPackageHandleStandardArgs)
55 find_package_handle_standard_args(MySimplePackage
```



0



```
55     DEFAULT_MSG
56     MYSIMPLEPACKAGE_LIBRARY
57     MYSIMPLEPACKAGE_INCLUDE_DIR
58     ${_deps_check})
59
60 if(MYSIMPLEPACKAGE_FOUND)
61     set(MYSIMPLEPACKAGE_LIBRARIES "${MYSIMPLEPACKAGE_LIBRARY}")
62     set(MYSIMPLEPACKAGE_INCLUDE_DIRS "${MYSIMPLEPACKAGE_INCLUDE_DIR}")
63     mark_as_advanced(MYSIMPLEPACKAGE_ROOT_DIR)
64 endif()
65
66 mark_as_advanced(MYSIMPLEPACKAGE_INCLUDE_DIR
67     MYSIMPLEPACKAGE_LIBRARY
68     MYSIMPLEPACKAGE_RUNTIME_LIBRARY)
```



0



转载自：

<http://blog.csdn.net/wfei101/article/details/77150234>

<http://blog.csdn.net/b2259909/article/details/58591898>

http://blog.csdn.net/z_h_s/article/details/50699905

<http://blog.csdn.net/gw569453350game/article/details/46683845>

- [上一篇](#) SO(ELF)文件格式详解
- [下一篇](#) android源码在线浏览网站

7



5.00/个

厂家直销 YH-F8挂墙式光纤终端盒8口

8



2690.00/套

厂家直销 1800*900*900MM 冷

9



176.00/套

供应 埋地防水接线盒 GHFS-F4 灌胶式防水

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)


0





AndroidStudio2.2NDK CMakeLists.txt配置新的.cpp

刚接触CMakeLists.txt也不是很了解，记录下自己的学习路程做个笔记，也希望能给后来学习的人一点帮助。创建好ndk项目后android studio2.2会自动生成一个.cpp的文件也会生成一...



cao126197103 2017年01月13日 10:15 3046

Android studio中NDK开发（一）：CMakeLists.txt编写入门

自定义变量 主要有隐式定义和显式定义两种。 隐式定义的一个例子是PROJECT指令，它会隐式的定义< projectname >_BINARY_DIR和_SOURCE_DIR两个变量；显...



liopo 2017年08月29日 20:12 1277

大数据工程师干货来袭！

点我查看，一秒领取！



Android studio 使用Cmake完成C/C++ 的使用以及生成so文件

Android studio 2.2版本以后对C/C++的支持可以说很方便了，当然官方推荐使用Cmake完成对C/C++的支持 2.2版本以上的同学新建一个项目就知道了，步骤如下：File -> N...



u014800493 2017年02月09日 16:04  6248



CMake之CMakeLists.txt编写入门



z_h_s 2016年03月06日 16:18  16454

CMake之CMakeLists.txt编写入门

ROS学习（六）：CMakeLists.txt 文件



xuehuafeiwu123 2016年12月29日 16:34  2591

CMakeLists.txt 文件 为 CMake build 文件。是 CMake 编译系统中软件包的输入。描述如何编译代码、安装到哪里。 ...

AndroidStudio用Cmake方式编译NDK代码



joe544351900 2016年12月14日 12:57  8192

1.cmake是什么？ CMake是一个跨平台的安装（编译）工具，可以用简单的语句来描述所有平台的安装(编译过程)。它能够输出各种各样的makefile或者project文件，能测试编译器所支持的C++...

NDK开发 从入门到放弃(七：Android Studio 2.2 CMAKE 高效NDK开发)

前言之前，每次需要边写C++代码的时候，我的内心都是拒绝的。1. 它没有代码提示 ' ' ' 这意味着我们必须自己手动敲出所有的代码，对于一个新手来说，要一个字母都不错且大小写也要正确，甚至！住所有的...



xiaoyu_93 2016年11月09日 16:36 13641



0



CMakeLists.txt的写法



gxuan 2016年11月28日 23:35 50156

参考：http://blog.csdn.net/cust_hf/article/category/345853 CMakeLists.txt的写法 (1):要求CMake根据指定的源文件生成...

CMakeList的基本写法



u010122972 2017年10月12日 16:03 173

最近需要自己写CMakeList，所以简要写一下一些基本的操作。为图实用，只写了常用的简单操作。1.确定cmake最低版本需求cmake_minimum_required(VERSION 3.0.0)...

Android Studio CMakeLists.txt文件配置



chengkaizone 2016年11月11日 13:07 6435

关于生成可执行文件时依赖的源文件在当前目录及当前目录子目录中的解决办法：因为 aux_source_directory (./ EXE_SRC)中只能将当前目录中的源代码文件添加到变量EX...

Android用CMake进行JNI编程学习



wangjinnan16 2017年09月02日 12:48 186

一、简介 1.1、什么是JNI JNI的全称是Java Native Interface：Java本地开发接口，它提供了若干的API实现了Java和其他语言的通信(主要是C和C++)，目的就...

it培训机构排名

全国it培训学校排名

百度广告



0



Android NDK系列(二)-AS使用CmakeLists生成so文件

接着上个文章，继续看AS能怎么生成so文件。 Android NDK系列（一）-AS使用javah生成so文件：<http://blog.csdn.net/sw5131899/article/details/779>



sw5131899 2017年08月15日 17:24 779

Android中CMake的使用之三调用第三方库



fpcc 2017年04月10日 09:45 3670

Android中CMake的使用之三调用第三方库 在开发过的过程中，难免会调用第三方的库，比如说ffmpeg啊，opencv等等啊，这就会出现这样一个问题，如何使用这种第三方的SO呢（.a）？这里需要...

Cmake安装过程问题，cmake_gui.exe配置问题CMake Error at CMakeLists.txt:1...

开始下载的zip文件不需要安装cmake，出现了问题后以为是下载的文件不对，就又下了遍msi文件，然后安装了cmake，安装后D:\Program Files\CMake\bin中打开cmake-gu...



qq_31806049 2018年02月24日 15:14 92

Android开发学习之路--Android Studio cmake编译ffmpeg

最新的android studio2.2引入了cmake可以很好地实现ndk的编写。这里使用最新的方式，对于以前的android下的ndk编译什么的可以参考之前的文章:Android开发学习之路-ND...



eastmoon502136 2016年10月20日 22:33 13105

嘿！阿里云服务套餐，今日起可免费体验>>

阿里巴巴集团旗下云计算品牌，全球卓越的云计算技术和服务提供商。



0



Android Studio2.2配置MakeList使用cmake编译c文件

初次使用cmake来编译c文件在Android项目中,那个快感能传递到骨髓里,欲罢不能.只能说谁用谁知道.如果还没有使用,请赶快用一次,初次体验一共三部: 第一:studio升级到2.2以后的版本,安...



qq_35599978 2017年02月21日 21:59 917

Android Studio NDK CMake 指定so输出路径以及生成多个so的案例与总结

前文一直想用Android Studio的新方式Cmake来编译JNI 代码，之前也尝试过，奈何有两个难题挡住了我 1. 只能生成一个 so库，不能一次性生成多个 so库，之前的mk是可以有子模块...



b2259909 2017年02月28日 13:06 8441

AndroidStudio使用CMake编译jni的C/C++文件

Android开发主流工具已变成AndroidStudio，新版AS已经支持CMake编译工具，可以用于编译C/C++文件，增强了Android调用jni代码的便捷性。相比于之前繁杂的ndk配置方式，...



binglumeng 2017年01月22日 11:52 3502

cmake 学习笔记(一)



dbzhang800 2011年04月10日 21:07 121776

最大的Qt4程序群(KDE4)采用cmake作为构建系统Qt4的python绑定(pyside)采用了cmake作为构建系统开源的图像处理库 opencv 采用cmake 作为构建系统... 看来不...



0

简单工程CmakeLists.txt的书写



sunbaigui 2011年07月22日 13:24 6191

工程文件夹下的CmakeLists.txt书写cmake_minimum_required(VERSION 2.8)project(earth)
include(\${ENV{HOME}}/usr/share/cmake/Modules/FindPythonInterp.cmake
e...

