# Cmake原生构建工具学习（NDK第二十八节课）

**20.05 准时开始**

**21.20 准时开始**


课程大纲：

```
Cmake原生构建工具学习

1.本节课总体安排概况。
2.什么是CMake。
3.Makefile导入动态库静态库。
4.CmakeList详解。
5.Cmake流程控制与函数等。
6.动态库与静态库画图分析。
7.Cmake预编译库与依赖源码方式。

预习资料：本节课知识点还不需要预习资料，Derry讲的课会让所有学生都听得明明白白。
```

**01-本节课总体安排概况。**

Cmake原生构建工具学习

1.本节课总体安排概况。　　【先说本节课一共会讲哪些内容】
2.什么是CMake。　　【认识CMake是什么，Cmake之前的方式Makefile为何被替换(引出下面环节)】
3.Makefile导入动态库静态库。【Makefile构建代码介绍并分析各个弊端，所以目前都是Cmake工具构建(引出下面环节)】
4.CmakeList详解。　　【目前原生构建工具基本上都是Cmake了,Cmake其实就是对Makefile进行封装】
5.Cmake流程控制与函数等。　【根据(上一个环节)的基础，学习此环节的内容】
6.动态库与静态库画图分析。　【因为(下面环节)会各种操作动态库与静态库，所以先分析】
7.Cmake预编译库与依赖源码方式。【此环节与(上一个环节)有着关系】

预习资料：本节课知识点还不需要预习资料，Derry讲的课会让所有学生都听得明明白白。

### 02.什么是CMake。

在Android Studio 2.2及以上，构建原生库的默认工具是CMake。

CMake是一个跨平台的构建工具，可以用简单的语句来描述所有平台的安装（编译过程）。
能够输出各种各样的makefile或者project文件。CMake并不直接构建出最终的软件，
而是产生其他工具的脚本（如makefile），然后再依据这个工具的构建方式使用。

CMake是一个比make更高级的编译配置工具，它可以根据不同的平台、不同的编译器，
生成相应的makefile或vcproj项目，从而达到跨平台的目的。

Android Studio利用CMake生成的是ninja。ninja是一个小型的关注速度的构建系统。
我们不需要关心ninja的脚本，知道怎么配置CMake就可以了。

CMake其实是一个跨平台的支持产出各种不同的构建脚本的一个工具。

### 03.Makefile导入动态库静态库。

Android.mk

```
# 这里面能够决定编译 Login.c Test.c

# 1.源文件在的位置。宏函数 my-dir 返回当前目录（包含 Android.mk 文件本身的目录）的路径。
# LOCAL_PATH 其实就是Android.mk文件本身的目录的路径
LOCAL_PATH := $(call my-dir)

$(info "LOCAL_PATH:========= ${LOCAL_PATH}")

# 2.清理
include $(CLEAR_VARS)

# TODO 预编译库的引入 == 提前编译好的库
LOCAL_MODULE := getndk

LOCAL_SRC_FILES := libgetndk.so
# LOCAL_SRC_FILES := libgetndk.a

# 预编译动态库的Makeifle脚本
```

```
# include $(PREBUILT_STATIC_LIBRARY)

# 预编译共享库的Makeifle脚本
include $(PREBUILT_SHARED_LIBRARY)

#引入其他makefile文件。CLEAR_VARS 变量指向特殊 GNU Makefile，可为您清除许多 LOCAL_XXX 变量
#不会清理 LOCAL_PATH 变量
include $(CLEAR_VARS)
# TODO end

# 3.指定库名字
#存储您要构建的模块的名称 每个模块名称必须唯一，且不含任何空格
#如果模块名称的开头已是 lib，则构建系统不会附加额外的前缀 lib；而是按原样采用模块名称，并添加 .so 扩展名。
LOCAL_MODULE := MyLoginJar

#包含要构建到模块中的 C 和/或 C++ 源文件列表 以空格分开
LOCAL_SRC_FILES := Login.c \
Test.c

# TODO 开始链接进来
# 静态库的链接
# LOCAL_STATIC_LIBRARIES := getndk
# 动态库链接
LOCAL_SHARED_LIBRARIES := getndk

# 导入 log
#LOCAL_LDLIBS := -llog
LOCAL_LDLIBS    := -lm -llog

# 4.动态库
#构建动态库BUILD_SHARED_LIBRARY 最后生成总动态库  ---> apk/lib/armeabi-v7a/libMyLoginJar.so
include $(BUILD_SHARED_LIBRARY)
```

### 04.CmakeList详解

```
# 最低支持的版本，注意：这里并不能代表最终的版本，最终版本在app.build.gradle中设置的
cmake_minimum_required(VERSION 3.10.2)

# 当前工程名，以前的旧版本，是没有设置的，这个可以设置，也可以不设置
project("ndk28_cmake")

# 批量导入 cpp c源文件
# file 可以定义一个变量 SOURCE， GLOB（使用GLOB从源树中收集源文件列表，就可以开心的 *.cpp *.c *.h）
# https://www.imooc.com/wenda/detail/576408
file(GLOB SOURCE *.cpp *.c)

# 添加一个库（动态库SHARED，静态库STATIC）
add_library(native-lib # 库的名字 ---> libnative-lib.so
            SHARED  # 动态库

            # cpp的源文件：把cpp源文件编译成 libnative-lib.so 库
```

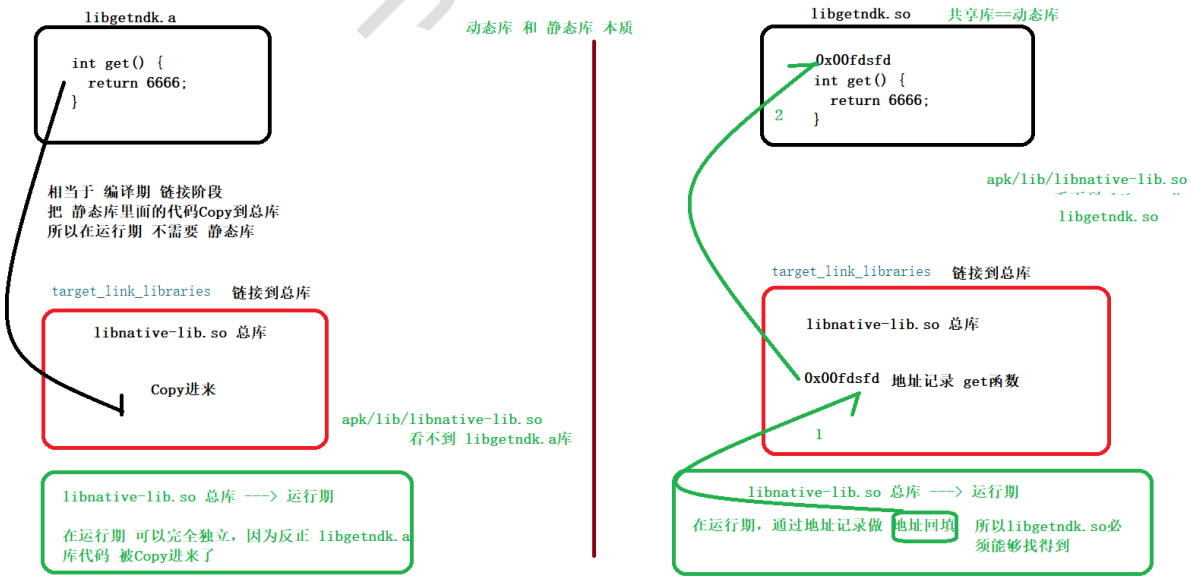
```
            ${SOURCE}
        )

# 查找一个 NDK工具中的 动态库(liblog.so)

# 思考：我如何知道 哪些库是可以写的，你怎么知道些一个log就可以？
# 答：请查看 D:\Android\Sdk\ndk\21.4.7075529\build\cmake\system_libs.cmake

# 思考：D:\Android\Sdk\ndk\21.0.6113669\toolchains\llvm\prebuilt\windows-
x86_64\sysroot\usr\lib\arm-linux-androideabi\16\liblog.so
# 答：你怎么知道是在 21.4.7075529？，arm-linux-androideabi？，16？
# 答：?1(因为local.properties知道了NDK版本，或者是你当前的NDK版本)
# 答：?2(因为我的手机是arm32的 所以 == arm-linux-androideabi 而且还我运行过)
# 答：?3(因为 minSdkVersion 16)
find_library(log-lib
             log )

# native-lib是我们的总库，也就是我们在 apk/lib/libnative-lib.so
# 然后 把log库链接到 总库中去，总库的cpp代码就可以使用 android/log.h的库实现代码了
target_link_libraries(native-lib # 被链接的总库
                      ${log-lib} # 链接的具体库

                    # getndk
        )

# log 信息输出的查看
# 以前的Cmake版本都是在output.txt，现在最新版本Cmake在metadata_generation_stderr.txt或
cmake_server_log，害我寻找了半天
# 想及时更新你的日志，请安装一次即可 or Linked_C++_Projects
# 在Build也可以查看，注意：是点击Sync Now 才会看到

#[[
(无) = 重要消息；
STATUS = 非重要消息；
WARNING = CMake 警告，会继续执行；
AUTHOR_WARNING = CMake 警告 (dev)，会继续执行；
SEND_ERROR = CMake 错误，继续执行，但是会跳过生成的步骤；
FATAL_ERROR = CMake 错误，终止所有处理过程；
]]

message(STATUS "1DerrySuccessD>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
message(STATUS "2DerrySuccessD>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
message(STATUS "3DerrySuccessD>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
message(STATUS "4DerrySuccessD>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
message(STATUS "5DerrySuccessD>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
message(STATUS "6DerrySuccessD>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
message(STATUS "7DerrySuccessD>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
message(STATUS "8DerrySuccessD>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
message(STATUS "9DerrySuccessD>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
message(STATUS "0DerrySuccessD>>>>>>>>>>>>>>>>>>>>>>>>>>>>")
message("10 OldCmakeVersion:output.txt, NewCmakeVersion:cmake_server_log.txt")
```

**05.Cmake流程控制与函数等**

```
# TODO ------------------------------------------------------------------------
-

# TODO CMake变量
# 声明变量：set(变量名 变量值)
set(var 666)
# 引用变量：message 命令用来打印
message("var = ${var}")
# CMake中所有变量都是string类型。可以使用set()和unset()命令来声明或移除一个变量
# 移除变量
unset(var)
message("my_var = ${var}") # 会取不到值，因为被移除了

# TODO CMake列表（lists）
# 声明列表：set(列表名 值1 值2 ... 值N) 或 set(列表名 "值1;值2;...;值N")
set(list_var 1 2 3 4 5) # 字符串列表呢？ CMake中所有变量都是string类型
# 或者
set(list_var2 "1;2;3;4;5") # 字符串列表呢？  CMake中所有变量都是string类型

message("list_var = ${list_var}")
message("list_var2 = ${list_var2}")

# TODO CMake流程控制-条件命令
# true(1, ON, YES, TRUE, Y, 非0的值)
# false(0, OFF, NO, FALSE, N, IGNORE, NOTFOUND)
set(if_tap OFF) # 定义一个变量if_tap，值为false
set(elseif_tap ON) # 定义一个变量elseif_tap，值为ture

if(${if_tap})
    message("if")
elseif(${elseif_tap})
    message("elseif")
else(${if_tap}) # 可以不加入 ${if_tap}
    message("else")
# endif(${if_tap}) # 结束if
endif() # 结束if 可以不加
# 注意：elseif和else部分是可选的，也可以有多个elseif部分，缩进和空格对语句解析没有影响

# TODO CMake流程控制-循环命令
set(a "")
#  a STREQUAL "xxx"（a等不等xxx，不等于）
#  NOT == !
while(NOT a STREQUAL "xxx")
    set(a "${a}x")
    message(">>>>>>a = ${a}")
endwhile()
#[[ 注意：
break()命令可以跳出整个循环
continue()可以继续当前循环
]]

foreach(item 1 2 3)
    message("1item = ${item}")
endforeach(item) # 结束for
```

```
foreach(item RANGE 2) # RANGE 默认从0开始， 所以是：0 1 2
    message("2item = ${item}")
endforeach(item)

foreach(item RANGE 1 6 2) #  1 3 5 每次跳级2
    message("3item = ${item}")
endforeach(item)

set(list_va3 1 2 3) # 列表
# foreach(item IN LISTS ${list_va3}) 没有报错，没有循环
foreach(item IN LISTS list_va3)
    message("4item = ${item}")
endforeach(item)

# TODO CMake自定义函数  Shell的函数很类似
#[[
ARGC：表示传入参数的个数
ARGV0：表示第一个参数，ARGV1、ARGV2以此类推即可
ARGV：表示所有参数
]]
function(num_method n1 n2 n3)
    message("call num_method method")
    message("n1 = ${n1}")
    message("n2 = ${n2}")
    message("n3 = ${n3}")
    message("ARGC = ${ARGC}")
    message("arg1 = ${ARGV0} arg2 = ${ARGV1} arg3 = ${ARGV2}")
    message("all args = ${ARGV}")
endfunction(num_method)
num_method(1 2 3) # 调用num_method函数

# 静态库和动态库本质
```

## 06.动态库与静态库画图分析



**07.Cmake预编译库与依赖源码方式**

```
cmake_minimum_required(VERSION 3.10.2)
project("ndk28_cmake")

# 批量导入 cpp c源文件
file(GLOB SOURCE ${CMAKE_SOURCE_DIR}/cpp/*.cpp ${CMAKE_SOURCE_DIR}/cpp/*.c)

# 添加一个库（动态库SHARED，静态库STATIC）
add_library(native-lib # 库的名字 ---> libnative-lib.so
             SHARED  # 动态库
            # cpp的源文件：把cpp源文件编译成 libnative-lib.so 库
             ${SOURCE}
        )


# TODO >>>>>>>>>>>>>>>>>>>>>>>> 预编译库（已经有了 xxx.so / xxx.a）的导入方式
# TODO 方式一：推荐的方式
#[[
# 第一步：导入fmod头文件
include_directories("${CMAKE_SOURCE_DIR}/cpp/inc")
# 第二步：导入库文件 （方式一）
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -
L${CMAKE_SOURCE_DIR}/jniLibsaaa/${CMAKE_ANDROID_ARCH_ABI}")
# 第三步链接到总库中去
target_link_libraries( # native-lib是我们的总库
        native-lib # 被链接的总库
        log # 自动寻找  # 具体的库 链接到 libnative-lib.so里面去
        fmod # 具体的库 链接到 libnative-lib.so里面去
        fmodL # 具体的库 链接到 libnative-lib.so里面去
        )
]]

# TODO 方式二：以前更多使用的方式，老程序员使用的方式
#[[
# 第一步：导入fmod头文件
include_directories("${CMAKE_SOURCE_DIR}/cpp/inc")
# 第二步：导入库文件 （方式二）
add_library(fmod SHARED IMPORTED)
set_target_properties(fmod PROPERTIES
        IMPORTED_LOCATION
${CMAKE_SOURCE_DIR}/jniLibsaaa/${CMAKE_ANDROID_ARCH_ABI}/libfmod.so)

add_library(fmodL SHARED IMPORTED)
set_target_properties(fmodL PROPERTIES
        IMPORTED_LOCATION
${CMAKE_SOURCE_DIR}/jniLibsaaa/${CMAKE_ANDROID_ARCH_ABI}/libfmodL.so)

# 第三步链接到总库中去
target_link_libraries( # native-lib是我们的总库
        native-lib # 被链接的总库
        log # 自动寻找  # 具体的库 链接到 libnative-lib.so里面去
        fmod # 具体的库 链接到 libnative-lib.so里面去
        fmodL # 具体的库 链接到 libnative-lib.so里面去
        )
]]
```

```
# TODO >>>>>>>>>>>>>>>>>>>>>>>> 判断静态库还是动态库 (静态库会直接Copy到总库，动态库则不会)
#[[
# >>>>>>>>>>> 下面代码不参与 判断 start
# 第一步：导入fmod头文件
include_directories("${CMAKE_SOURCE_DIR}/cpp/inc")
# 第二步：导入库文件 （方式二）
add_library(fmod SHARED IMPORTED)
set_target_properties(fmod PROPERTIES
        IMPORTED_LOCATION
${CMAKE_SOURCE_DIR}/jniLibsaaa/${CMAKE_ANDROID_ARCH_ABI}/libfmod.so)
add_library(fmodL SHARED IMPORTED)
set_target_properties(fmodL PROPERTIES
        IMPORTED_LOCATION
${CMAKE_SOURCE_DIR}/jniLibsaaa/${CMAKE_ANDROID_ARCH_ABI}/libfmodL.so)
# >>>>>>>>>>> 上面代码不参与 判断 end

## OFF=0=false    ON=1=true
# set(isSTATIC OFF)
set(isSTATIC ON)

if(${isSTATIC})
    # 导入静态库
    add_library(getndk STATIC IMPORTED)
    # 开始真正导入 静态库   System.loadLibrary("getndk"); // 如果是动态库，这里需要加载，否则注释
    set_target_properties(getndk PROPERTIES IMPORTED_LOCATION
${CMAKE_SOURCE_DIR}/cpp/libgetndk.a)
    message("isSTATIC == static")
else(${isSTATIC})
    # 导入动态库
    add_library(getndk SHARED IMPORTED)
    # 开始真正导入 动态库  System.loadLibrary("getndk"); // 如果是动态库，这里需要加载，否则注释
    set_target_properties(getndk PROPERTIES
            IMPORTED_LOCATION
${CMAKE_SOURCE_DIR}/jniLibsaaa/${CMAKE_ANDROID_ARCH_ABI}/libgetndk.so)
    message("isSTATIC == shared")

endif(${isSTATIC})

target_link_libraries( # native-lib是我们的总库
        native-lib # 被链接的总库
        log # 自动寻找  # 具体的库 链接到 libnative-lib.so里面去
        getndk # TODO 具体的库 链接到 libnative-lib.so里面去【这个库，有可能是静态库，有可能是动态库】
        fmod # 具体的库 链接到 libnative-lib.so里面去
        fmodL # 具体的库 链接到 libnative-lib.so里面去
        )
]]


# rtmp的时候，就全盘采用 源码构建方式
# TODO >>>>>>>>>>>>>>>>>>>>>>>> 依赖源码的方式（已经有了 xxxc / xxx.c / xxx.c ...）的导入方式

#引入get子目录下的CMakeLists.txt
add_subdirectory(${CMAKE_SOURCE_DIR}/cpp/libget)
```

```
#引入count子目录下的CMakeLists.txt
add_subdirectory(${CMAKE_SOURCE_DIR}/cpp/libcount)

target_link_libraries( # native-lib是我们的总库
        native-lib # 被链接的总库
        log # 自动寻找  # 具体的库 链接到 libnative-lib.so里面去
        get # 具体的库 链接到 libnative-lib.so里面去
        count # 具体的库 链接到 libnative-lib.so里面去
        )
```