Android.mk分析

作者:程姚根 联系方式:chengyaogen123@163.com

在Android的源码中每个目录下几乎都有一个Android.mk的文件，这个文件就是用来管理当前目录或子目录下的文件进行编译的。我们打开Android.mk文件，我们会发现它并没有太多的Makefile语法，更多的是一些大写的宏，例如:LOCAL_PATH,LOCAL_MODULE等。此时我们就会在想，"Android源码下的文件是如何编译的呢?",实际上Android源码已经有比较完善的编译系统，它是以模块化的思想设计编译系统的，Android源码的每个目录几乎都可以当独编译，我们可以给予这个编译系统，编写自己的Android.mk文件，将我们需要的文件进行编译。

Android的编译系统相关的文件存放在Android源码目录下的build/core目录下，这里不介绍Android源码的编译系统，如果想了解Android编译系统的读者可以自己在网上查阅相关的资料。好了下面我们就来看看如何编写自己的Android.mk吧!

一、最简单的Android.mk

从编译一个史上第一个程序开始吧，编写编译hello.c的Android.mk。

```
#获取当前模块的绝对路径
LOCAL_PATH := $(call my-dir)

#清理编译环境中用到的变量
include $(CLEAR_VARS)

#设置当前模块的名称
LOCAL_MODULE = hello

#设置当前模块下需要编译的c文件
LOCAL_SRC_FILES = hello.c

#告诉编译系统，将当前模块下的文件编译成ELF格式的可执行文件
include $(BUILD_EXECUTABLE)
```

hello.c文件中的内容我想你是会写的，这里就不贴出来了，下面我们来编译一下这个模块。

```
a@b:~/workdir/androidL$ source build/envsetup.sh
including device/softwinner/fspad-733/vendorsetup.sh
including device/softwinner/common/vendorsetup.sh
including sdk/bash_completion/adb.bash
a@b:~/workdir/androidL$ lunch

You're building on Linux
```

```
Lunch menu... pick a combo:
     1. aosp_arm-eng
     2. aosp_arm64-eng
     3. aosp_mips-eng
     4. aosp_mips64-eng
     5. aosp_x86-eng
     6. aosp_x86_64-eng
     7. fspad_733-eng
     8. fspad_733-user

Which would you like? [aosp_arm-eng] 7
```

红线标记的部分必须有哦，做这些事情的目的就是设置编译环境啦!

设置好了编译环境，下面我们就来编译我们的模块吧!我的模块放在Android源码目录下的external/test/test1目录下。

```
a@b:~/workdir/androidL$
a@b:~/workdir/androidL$ mmm external/test/test1/
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=5.0
TARGET_PRODUCT=fspad_733
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a-neon
TARGET_CPU_VARIANT=cortex-a7
TARGET_2ND_ARCH=
TARGET_2ND_ARCH_VARIANT=
TARGET_2ND_CPU_VARIANT=
HOST_ARCH=x86_64
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.13.0-32-generic-x86_64-with-Ubuntu-14.04-trusty
HOST_BUILD_TYPE=release
BUILD_ID=LRX21V
OUT_DIR=out
============================================
No private recovery resources for TARGET_DEVICE fspad-733
make: Entering directory `/home/a/workdir/androidL'
Import includes file: out/target/product/fspad-733/obj/EXECUTABLES/hello_intermediates/import_includes
target thumb C: hello <= external/test/test1/hello.c
target Executable: hello (out/target/product/fspad-733/obj/EXECUTABLES/hello_intermediates/LINKED/hello
)
target Symbolic: hello (out/target/product/fspad-733/symbols/system/bin/hello)
Export includes file: external/test/test1/Android.mk -- out/target/product/fspad-733/obj/EXECUTABLES/he
llo_intermediates/export_includes
target Strip: hello (out/target/product/fspad-733/obj/EXECUTABLES/hello_intermediates/hello)
Install: out/target/product/fspad-733/system/bin/hello
make: Leaving directory `/home/a/workdir/androidL'

#### make completed successfully (9 seconds) ####

a@b:~/workdir/androidL$
```

用"mmm"编译我们的模块

编译我们的.c文件

我们生成的可执行文件最终存放路径

嗯，我们很轻松的就完成了在Android源码目录下编译自己的模块，下面我们来详细分析一下每个变量的具体含义。

(1)LOCAL_PATH := $(call my-dir)

$(call my-dir)这种写法是Makefile中调用一个自定义函数的写法,也就是获取my-dir这个自定义函数的结果。下面我们来看看my-dir这个函数具体是如何实现的?

```
############################################################
## Retrieve the directory of the current makefile
## Must be called before including any other makefile!!
############################################################

# Figure out where we are.
define my-dir
$(strip \
  $(eval LOCAL_MODULE_MAKEFILE := $$(lastword $$(MAKEFILE_LIST))) \
  $(if $(filter $(BUILD_SYSTEM)/% $(OUT_DIR)/%,$(LOCAL_MODULE_MAKEFILE)), \
    $(error my-dir must be called before including any other makefile.) \
   , \
    $(patsubst %/,%,$(dir $(LOCAL_MODULE_MAKEFILE))) \
   ) \
 )
endef
```

我们重点关注红颜标注的地方，MAKEFILE_LIST是make解释器内部定义的变量，它的含义就是获取它所寻找到的Makefile文件的绝对路径列表。当我们在源码目录的顶层目录进行"mmm"进行编译的时候，Android的编译系统就会使用make工具找到相关的Makefile文件。我们的模块里面的Makefile就是最后一个Makefile文件了。

LOCAL_MODULE_MAKEFILE := $$(lastword $$(MAKEFILE_LIST)) 获取最后一个Makefile文件绝对路径

$(dir $(LOCAL_MODULE_MAKEFILE)) 获取目录路径，不包含文件 例如:$(dir /home/linux/Makefile) -> /home/linux/

$(patsubst %/, %, /home/linux/) -> /home/linux

好了,我们总结一下 $(call my-dr)的终极含义:获取当前模块的路径

(2)include $(CLEAR_VARS)

CLEAR_VARS这个变量在Android源码树下的build/core/config.mk文件中定义:

```
CLEAR_VARS:= $(BUILD_SYSTEM)/clear_vars.mk
```

include类似于C语言中的头文件包含，嗯，它的含义就是在我们的Android.mk文件中包含编译系统目录下的clear_vars.mk这个文件中的内容。clear_vars.mk中就是将一些编译的时候需要用到的一些变量清空。但是我可以肯定它一定不会把LOCAL_PATH这个变量清空，想想为什么?

(3)LOCAL_MODULE

用来指定当前模块的名称

(4)LOCAL_SRC_FILES

用来指定当前模块需要参与编译的文件

(5)include $(BUILD_EXECUTABLE)

BUILD_EXECUTABLE这个变量在Android源码树下的build/core/config.mk文件中定义:

```
BUILD_EXECUTABLE:= $(BUILD_SYSTEM)/executable.mk
```

executable.mk文件中定义了如何编译设备上的可执行文件。

二、编译模块下的多个文件

上面的Android.mk中我们只编译了一个文件，如果有多个文件需要编译该如何做呢?在Android的编译系统中，我们有两种方法让多个文件可以参与编译。

(1)将需要编译的文件名都指定在LOCAL_SRC_FILES变量

例如:在我们的test1目录下还有两个文件 add.c 和 sub.c ,这两个文件中的内容如下:

```
1 int sub(int a,int b)
2 {
3     return (a - b);
4 }
```

```
1 int add(int a,int b)
2 {
3     return (a + b);
4 }
```

我们的Android.mk的内容如下:

```
1  #获取当前模块的绝对路径
2  LOCAL_PATH := $(call my-dir)
3
4  #清理编译环境中用到的变量
5  include $(CLEAR_VARS)
6
7  #设置当前模块的名称
8  LOCAL_MODULE = hello
9
10 #设置当前模块下需要编译的C文件
11 LOCAL_SRC_FILES = hello.c \
12                   add.c   \
13                   sub.c   \
14
15 #指定模块安装的路径
16 LOCAL_MODULE_PATH = $(LOCAL_PATH)/bin
17
18 #告诉编译系统，将当前模块下的文件编译成ELF格式的可执行文件
19 include $(BUILD_EXECUTABLE)
20
```

LOCAL_MODULE_PATH = $(LOCAL_PATH)/bin

表示将编译好的模块存放在模块所在目录的bin子目录下

```
make: Entering directory `/home/a/workdir/androidL'
target thumb C: hello <= external/test/test1/hello.c
target thumb C: hello <= external/test/test1/add.c
```

```
target thumb C: hello <= external/test/test1/sub.c
target Executable: hello (out/target/product/fspad-733/obj/EXECUTABLES/hello_intermediates/LINKED/hello
)
target Symbolic: hello (out/target/product/fspad-733/symbols/external/test/test1/bin/hello)
Export includes file: external/test/test1/Android.mk -- out/target/product/fspad-733/obj/EXECUTABLES/he
llo_intermediates/export_includes
target Strip: hello (out/target/product/fspad-733/obj/EXECUTABLES/hello_intermediates/hello)
Install: external/test/test1/bin/hello
make: Leaving directory `/home/a/workdir/androidL'

#### make completed successfully (4 seconds) ####
```

（2）调用Android编译系统的函数，获取当前模块下所有需要编译的文件

Android编译系统自定义的函数	功能
all-c-files-under	获取指定目录下所有的c语言文件
all-java-files-under	获取指定目录下所有的java语言文件
all-laidl-files-under	获取指定目录下所有的aidl语言文件
all-makefiles-under	获取指定目录下所有的makefile文件
all-subdir-c-files	获取当前子目录下所有的c语言文件
all-subdir-java-files	获取当前子目录下所有的java语言文件
all-subdir-laidl-files	获取当前子目录下所有的aidl语言文件
all-subdir-makefiles	获取当前子目录下所有的makefile文件

我们以获取C语言为例，来看看函数的具体实现:

(1)all-c-files-under

```
############################################################
## Find all of the c files under the named directories.
## Meant to be used like:
##     SRC_FILES := $(call all-c-files-under,src tests)
############################################################

define all-c-files-under
$(patsubst ./%,%, \
  $(shell cd $(LOCAL_PATH) ; \
          find -L $(1) -name "*.c" -and -not -name ".*") \
 )
endef
```

(2)all-subdir-c-files

```
############################################################
## Find all of the c files from here.  Meant to be used like:
##     SRC_FILES := $(call all-subdir-c-files)
############################################################

define all-subdir-c-files
$(call all-c-files-under,.)
endef
```

嗯，对比一下两者的区别:

(1)all-c-files-under 比较灵活，可以指定模块下一个指定的子目录下搜索所有的c语言文件

(2)all-subdir-c-files 是获取模块下所有的子目录下的C语言文件

注意:在这里函数中，寻找Makefile文件的时候只会递归一级子目录

好了，我们来看看我们修改后的Android.mk文件吧!

```
1 #获取当前模块的绝对路径
2 LOCAL_PATH := $(call my-dir)
3
4 #清理编译环境中用到的变量
5 include $(CLEAR_VARS)
6
7 #设置当前模块的名称
8 LOCAL_MODULE = hello
9
10 #设置当前模块下需要编译的C文件
11 LOCAL_SRC_FILES = $(call all-c-files-under,src)
12
13 #指定模块安装的路径
14 LOCAL_MODULE_PATH = $(LOCAL_PATH)/bin
15
16 #告诉编译系统，将当前模块下的文件编译成ELF格式的可执行文件
17 include $(BUILD_EXECUTABLE)
```

我们把所有的C语言文件存放在了src子目录下，所以这里指定的是在src子目录下搜索。

编译效果如下:

```
make: Entering directory `/home/a/workdir/androidL'
target thumb C: hello <= external/test/test1/src/hello.c
external/test/test1/src/hello.c: In function 'main':
external/test/test1/src/hello.c:6:2: warning: implicit declaration of function 'add' [-Wimplicit-functi
on-declaration]
  add(10,20);
  ^
external/test/test1/src/hello.c:7:2: warning: implicit declaration of function 'sub' [-Wimplicit-functi
on-declaration]
  sub(30,10);
  ^
target Executable: hello (out/target/product/fspad-733/obj/EXECUTABLES/hello_intermediates/LINKED/hello
)
```

我们在hello.c中调用了add和sub函数,没有声明，所以编译的时候报了警告，下面我们自己定义一个hello.h的头文件，在这个头文件中我们声明这两个函数。

问题:如何在Android.mk文件中指定自己的头文件搜索路径?

```
#获取当前模块的绝对路径
LOCAL_PATH := $(call my-dir)

#清理编译环境中用到的变量
include $(CLEAR_VARS)

#设置当前模块的名称
LOCAL_MODULE = hello

#设置当前模块下需要编译的C文件
LOCAL_SRC_FILES = $(call all-subdir-c-files)

#指定模块安装的路径
LOCAL_MODULE_PATH = $(LOCAL_PATH)/bin

#指定头文件搜索路径
LOCAL_C_INCLUDES := $(LOCAL_PATH)/inc

#告诉编译系统，将当前模块下的文件编译成ELF格式的可执行文件
include $(BUILD_EXECUTABLE)
```

三、编译静态库和动态库

前面我们通过Android.mk将我们模块中的代码编译成了ELF格式的可执行文件,下面我们来看看如何将自己的模块编译成库。

```
#获得当前模块所在的路径
LOCAL_PATH := $(call my-dir)

#清除编译过程中用到的变量
include $(CLEAR_VARS)

#模块的名称
LOCAL_MODULE := libadd_sub

#获取模块中需要参与编译的文件
SRC_FILES = $(call all-c-files-under,src)
LOCAL_SRC_FILES := $(SRC_FILES)

#指定模块安装的路径
LOCAL_MODULE_PATH = $(LOCAL_PATH)/lib

#告诉编译系统模块需要编译成动态库
include $(BUILD_SHARED_LIBRARY)
```

(1)BUILD_SHARED_LIBRARY

将模块编译成动态库 ， 例如:libadd_sub.so

(2)BUILD_STATIC_LIBRARY

将模块编译成静态库 ， 例如:libadd_sub.a

四、链接库

1、链接Android系统中自带的库

```
#include <stdio.h>
#define LOG_TAG "TEST"
#include <log/log.h>

int main()
{
    ALOGE("test");
    return 0;
}
```

Android系统中用来输出log信息的函数

ALOGE是Android系统中用来输出log信息的函数，它在liblog.so中，所以我们在编译我们的代码时候，要告诉编译系统，需要去连接liblog.so这个动态库。
我们的Android.mk写成如下形式:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := test
LOCAL_SRC_FILES := src/test.c

#引用系统中的函数库
LOCAL_SHARED_LIBRARIES += liblog
LOCAL_MODULE_PATH := $(LOCAL_PATH)/bin
include $(BUILD_EXECUTABLE)
```

这里不需要加.so哦,Android编译系统会自动去寻找liblog.so进行连接。

(1)LOCAL_SHARED_LIBRARIES

告诉编译系统需要链接的Android系统提供的动态库

(2)LOCAL_STATIC_LIBRARIES

告诉编译系统需要链接的Android系统提供的静态库

2、链接第三方库

我们将add.c和sub.c编译成libadd_sub.so,然后我们在test.c中调用add和sub这两个函数，此时Android.mk应该写成如下形式:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := libadd_sub
SRC_FILES := $(call all-c-files-under,lib)
LOCAL_SRC_FILES := $(SRC_FILES)
LOCAL_MODULE_PATH :=$(LOCAL_PATH)/lib
include $(BUILD_SHARED_LIBRARY)

```

```
 9 ###################################
10 include $(CLEAR_VARS)
11 LOCAL_MODULE := test
12 LOCAL_SRC_FILES := src/test.c
13 LOCAL_MODULE_PATH :=$(LOCAL_PATH)/bin
14 LOCAL_SHARED_LIBRARIES += liblog
15 LOCAL_LDFLAGS += -L$(LOCAL_PATH)/lib/ -ladd sub
16 include $(BUILD_EXECUTABLE)
```

LOCAL_LDFLAGS

指定链接参数， -L 指定需要连接的库所在的路径，-l指定库的名字

五、预置编译

所谓的预置编译指的是将一个**编译好的可执行文件或APK以及他们依赖的库、jar包拷贝到Android系统源码相关的存放路径下**，这样我们在对Android 系统就行打包生成system.img镜像时，这些东西就会被打包进去。哦，还有就是当我们引入第三方库或

问:为了不自己手动把这些东西拷贝到Android源码对应的目录下，然后在打包呀?

答:麻烦，Android版本总是在升级，目录结构也总是在发生变化，使用预置编译，所以的事情都由Android系统自带的编译系统去做，这样就何乐而不为呢?

下面我们以预置libadd_sub.so文件到Android系统中为例，来讲解预置编译的使用方法:

```
 1 LOCAL_PATH := $(call my-dir)
 2
 3 include $(CLEAR_VARS)
 4
 5 LOCAL_MODULE = libaddsub.so
 6
 7 LOCAL_SRC_FILES = libadd_sub.so
 8
 9 LOCAL_MODULE_TAGS = optional
10
11 LOCAL_MODULE_CLASS = SHARED_LIBRARIES
12
13 include $(BUILD_PREBUILT)
14
```

编译效果如下:

```
make: Entering directory `/home/a/workdir/androidL'
Export includes file: external/test/Android.mk -- out/target/product/fspad-733/obj/SHARED_LIBRARIES/libaddsub.so_
intermediates/export_includes
target Prebuilt: libaddsub.so (out/target/product/fspad-733/obj/lib/libaddsub.so)
```

```
Install: out/target/product/fspad-733/system/lib/libaddsub.so
make: Leaving directory `/home/a/workdir/androidL'
```

解释如下:

(1)LOCAL_MODULE

这里的含义和以前的含义不一样，它表示文件(xx.apk/jar/so)预置到系统中之后的名字。例如:libadd_sub.so预置之后的名字为libaddsub.so

(2)LOCAL_SRC_FILES

需要预置到系统中的文件

(3)LOCAL_MODULE_TAGS

这个变量可以赋值为user 、eng、tests、optional

user	**指该模块只在user版本下才编译**
eng	**指该模块只在eng版本下才编译**
tests	**指该模块只在tests版本下才编译**
optional	**指该模块在所有版本下都编译**

(4)LOCAL_MODULE_CLASS

这个变量用来指定文件类型,它可以赋的值有

APPS	**apk文件**
SHARED_LIBRARIES	**动态库文件**
JAVA_LIBRARIES	**dex归档文件**
EXECUTABLES	**ELF格式文件**
ETC	**其他格式文件**

(5)BUILD_PREBUILT 和 BUILD_MULTI_PREBUILT

不同点:

<1>BUILD_PREBUILT 只能针对一个文件, BUILD_MUTI_PREBUILT针对多个文件

<2>BUILD_PREBUILT 在预置的时候可以通过LOCAL_MODULE_PATH将文件拷贝到自己指定的路径下，而BUILD_MULTI_PREBUILT只能将文件拷贝到Android系统指定的路径下。

六、引入第三方jar包

很多时候我们在做APP开发的时候都会用到第三方的jar包，那如何在Android系统中引入第三方jar包，编译java代码生成apk文件呢?

我们来看看Android源码中**packages/apps/Calculator目录下Android.mk的写法。**

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := optional

LOCAL_STATIC_JAVA_LIBRARIES := libarity android-support-v4 guava

LOCAL_SRC_FILES := $(call all-java-files-under, src)

LOCAL_SDK_VERSION := current

LOCAL_PACKAGE_NAME := Calculator

LOCAL_CERTIFICATE := platform

include $(BUILD_PACKAGE)

##################################################
include $(CLEAR_VARS)

LOCAL_PREBUILT_STATIC_JAVA_LIBRARIES := libarity:arity-2.1.2.jar

include $(BUILD_MULTI_PREBUILT)

# Use the following include to make our test apk.
include $(call all-makefiles-under,$(LOCAL_PATH))
```

预置编译

Calculator app应用程序使用到了第三方的arity-2.1.2.jar文件，而arity-2.1.2.jar文件在Calculator当前目录下,为了方便引用arity-2.1.2.jar文件这里先通过预置编译将arity-2.1.2.jar拷贝到Android 系统的相应目录下，这样开始编译Calculator应用程序的时候才可以找到它依赖的jar包。

我们看看它的编译流程:

```
make: Entering directory `/home/a/workdir/androidL'
target R.java/Manifest.java: Calculator (out/target/common/obj/APPS/Calculator_intermediates/src/R.stamp)
target Prebuilt: Calculator (out/target/common/obj/JAVA_LIBRARIES/libarity_intermediates/classes.jar)
target Prebuilt: Calculator (out/target/common/obj/JAVA_LIBRARIES/libarity_intermediates/javalib.jar)
target Java: Calculator (out/target/common/obj/APPS/Calculator_intermediates/classes)
Copying: out/target/common/obj/APPS/Calculator_intermediates/classes-jarjar.jar
Copying: out/target/common/obj/APPS/Calculator_intermediates/emma_out/lib/classes-jarjar.jar
Copying: out/target/common/obj/APPS/Calculator_intermediates/classes.jar
Proguard: out/target/common/obj/APPS/Calculator_intermediates/proguard.classes.jar
ProGuard, version 4.10
Reading program jar [/home/a/workdir/androidL/out/target/common/obj/APPS/Calculator_intermediates/classes.jar]
Reading library jar [/home/a/workdir/androidL/out/target/common/obj/JAVA_LIBRARIES/android_stubs_current_intermediates/classes.jar]
Preparing output jar [/home/a/workdir/androidL/out/target/common/obj/APPS/Calculator_intermediates/proguard.classes.jar]
```

Copying resources from program jar [/home/a/workdir/androidL/out/target/common/obj/APPS/Calculator_intermediates/classes.jar]
target Dex: Calculator
Copying: out/target/common/obj/APPS/Calculator_intermediates/classes.dex
target Package: Calculator (out/target/product/fspad-733/obj/APPS/Calculator_intermediates/package.apk)
Notice file: packages/apps/Calculator/NOTICE -- out/target/product/fspad-733/obj/NOTICE_FILES/src//system/app/Calculator/Calculator.apk.txt
Install: out/target/product/fspad-733/system/app/Calculator/Calculator.apk
target Prebuilt: libarity (out/target/product/fspad-733/obj/JAVA_LIBRARIES/libarity_intermediates/javalib.jar)
make: Leaving directory `/home/a/workdir/androidL'

好了，下面我们来看看这个Android.mk中我们前面没有用过的变量。

(1)LOCAL_STATIC_JAVA_LIBRARIES
指定当前模块依赖的jar包

(2)LOCAL_SDK_VERSION
指定当前SDK的版本为Android源码中的SDK版本

(3)LOCAL_PACKAGE_NAME
指定生成的apk文件的名字

(4)LOCAL_CERTIFICATE
指定apk文件的签名。可以指定的值有:testkey、media、platform、shared这四种，可以在源码build/target/product/security里面看到对应的秘钥，其中shared.pk8代表私钥，shared.x509.perm代表公钥，一定是成对出现的。

其中testkey是作为android编译的时候默认的签名key,如果系统中的apk的Android.mk中没有设置LOCAL_CERTIFICATE的值，就默认使用testkey。而如果设置成LOCAL_CERTIFICATE :=platform就代表使用platform来签名，这样的话这个apk就拥有了和system相同的签名，因为系统级别的签名也是使用platform来签名的。

(5)BUILD_PACKAGE
将模块编译成APK文件

(6)LOCAL_PREBUILT_STATIC_JAVA_LIBRARIES
指定prebuilt jar库的规则，**格式 别名：jar文件路径。注意:别名一定要与LOCAL_STATIC_JAVA_LIBRARIES里所取的别名一致，且不含jar;jar文件路径一定要是真实的存放第三方jar包的路径**。编译用BUILD_MULTI_PREBUILT。