## Android.mk:常见问题

## 1.undefined reference 问题

缺少链接库或者头文件.

## Manifest:

Manifest结构：

\_\_ .repo

|

|\_\_ manifest.xml (entry,link file)

|

|\_\_ manifests (chechout directory)

| |

| |\_\_ \*.xml

|

|\_\_ manifest.git

<manifest/>

根元素，包含所有其他元素

<manifest>

<remote />

. . .

<default />

<project />

. . .

<manifest/>

<remote/>

用于添加保存仓库URL地址的前缀，与其他元素组合使用，主要属性：

fetch —— URL地址前缀，和<project/>元素的name属性组合成完整的URL、

fetch = ‘..’ 它应该相对于'manifests /'的父目录

$ repo init -u https://android.googlesource.com/platform/manifest

fetch=".." in the manifest.xml is equal to "https://android.googlesource.com/platform/../"

name —— 为remote分配的名字，用于被其他元素指定

<remote fetch="ssh://..." name="origin" />

...

<default/>

用于指定工程元素的默认属性，主要属性：

remote —— 工程默认使用的remote，可以赋值<remote/>元素的name属性

revision —— 工程默认使用的版本，可以是分支、tag

sync-j —— repo sync时默认使用的线程数

<default remote="origin" revision="master" sync-j="4"/>

<project/>

用于指定需要下载的仓库，主要属性：

name —— **仓库名，和<remote/>元素的fetch属性组合起来，就是git仓库完整的URL**地址

path —— 本地路径，如果不设置，则和name属性相同

remote —— 使用的remote，可以赋值<remote/>元素的name属性，如果不设置，则使用<default/>的remote属性

revision —— 使用的版本，可以是分支、tag，如果不设置，则使用<default/>的revision属性

<project name="build" path="build" remote="origin" revision="master"/>

...

<include/>

用于包含其他xml文件的配置，主要属性：

name —— 包含的文件名

<include name="default.xml"/>

**local manifest**

### **local manifest结构**

\_\_ .repo | |\_\_ local\_manifests | | | |\_\_ \*.xml | |

local manifest是新版本repo工具增加的功能，让开发者可以一定程度地自定义工程环境。 repo会先读取init命令指定manifest文件，然后再读取本地.repo/local\_manifest目录下xml文件，从而可以修改原来manifest的配置，起到自定环境的功能。

**<remove-project>**

用于删除工程配置，我们可以在local manifest中加上这个元素，从而取消下载某个仓库。

<?xml version='1.0' encoding='utf-8'?>

<manifest>

<remove-project name="<project-name>" />

</manifest>

还可以和<project/>元素组合使用修改某个仓库的配置（相当于先删除再添加）

例如不取二进制仓库取源代码仓库：

<?xml version='1.0' encoding='utf-8'?>

<manifest>

<remove-project name="<project-name>" /> 《————删除原来二进制仓库的配置

<project name="<project-name>" path="<local-path>" remote="<code-remote>"/> 《————加上源代码仓库配置

</manifest>

## Shell：

**特殊变量名：**

$0 当前脚本的文件名

$n 传递给脚本或函数的参数。n 是一个数字，表示第几个参数。例如，第一个参数是1，第二个参数是2。

$# 传递给脚本或函数的参数个数。

$\* 传递给脚本或函数的所有参数。

$@ 传递给脚本或函数的所有参数。被双引号(" ")包含时，与 $\* 稍有不同，下面将会讲到。

$? 上个命令的退出状态，或函数的返回值。

$$ 当前 Shell 进程 ID。对于 Shell 脚本，就是这些脚本所在的进程 ID。

∗和@ 都表示传递给函数或脚本的所有参数，不被双引号(" ")包含时，都以"1""2" … "$n" 的形式输出所有参数。

但是当它们被双引号(" ")包含时，"∗"会将所有的参数作为一个整体，以"1 2…n"的形式输出所有参数；"@"会将各个参数分开，以"1" "2"…"n" 的形式输出所有参数。

**sed**

sed ‘s/pattern/replace/g’　　分隔符默认是/,但是也可以再s后指定分隔符。如果不加g，只匹配第一个的

## Makefile：

include $(CLEAR\_VARS)

1、解释 ****CLEAR\_VARS：****

****英文****：如下图

****中文****：编译系统提供CLEAR\_VARS变量，它指向了一个用来清除 LOCAL\_ 开头的变量（LOCAL\_PATH除外)的makefile文件，需要它的原因是整个的编译上下文中，所有的变量都是全局的，这样就可以保证这些变量只在局部范围内起作用；

LOCAL\_STRIP\_MODULE : strip 把so中的调试信息去除，相当与debug版到release版的差别。

LOCAL\_MODULE\_STEM

表示编译链接后的目标文件的文件名，不带后缀

LOCAL\_MODULE\_STEM := $(strip $(LOCAL\_MODULE\_STEM))

ifeq ($(LOCAL\_MODULE\_STEM),)

LOCAL\_MODULE\_STEM := $(LOCAL\_MODULE)

endif

例：

recovery模块编译recovery可执行文件：

LOCAL\_MODULE\_STEM:=recovery

LOCAL\_MODULE\_TAGS

模块的tag，为debug eng tests optional samples shell\_ash shell\_mksh等tag的组合，一个模块可有多个Tag,

注意现在模块现在不能使用user作为模块的Tag,

以前如果使用user做为tag,那么这个模块将被自动安装，

如果想定义自动安装的模块，需要在PRODUCT\_PACKAGES变量里添加该模块，

该变量在build/target/product/base.mk和build/target/product/core.mk里有赋值，这是所有产品都将继承的基础配置

另外每个设备可在自己的产品配置文件device\_\*.mk里设置该变量，添加更多的模块

如果当前目录或者父目录有\*\_GPL\*的文件，那么将自动添加gnu的tag

LOCAL\_BUILT\_MODULE\_STEM

表示编译链接后的目标文件的文件名，带后缀

LOCAL\_BUILT\_MODULE\_STEM := $(strip $(LOCAL\_BUILT\_MODULE\_STEM))

ifeq ($(LOCAL\_BUILT\_MODULE\_STEM),)

LOCAL\_BUILT\_MODULE\_STEM := $(LOCAL\_INSTALLED\_MODULE\_STEM)

endif

例：

recovery模块编译recovery可执行文件：

LOCAL\_INSTALLED\_MODULE\_STEM:=recovery

LOCAL\_UNINSTALLABLE\_MODULE

表示该模块是否安装至手机，像sdk文档模块不会被安装至手机，因此LOCAL\_UNINSTALLABLE\_MODULE为true

**Makefile选项CFLAGS,LDFLAGS,LIBS**

**包含其他的文件：include $(LOCAL\_PATH)/dlna.mk**

CFLAGS 表示用于 C 编译器的选项，

CXXFLAGS 表示用于 C++ 编译器的选项。

这两个变量实际上涵盖了编译和汇编两个步骤。

CFLAGS： 指定头文件（.h文件）的路径，如：CFLAGS=-I/usr/include -I/path/include。同样地，安装一个包时会在安装路径下建立一个include目录，当安装过程中出现问题时，试着把以前安装的包的include目录加入到该变量中来。

LDFLAGS：gcc 等编译器会用到的一些优化参数，也可以在里面指定库文件的位置。用法：LDFLAGS=-L/usr/lib -L/path/to/your/lib。每安装一个包都几乎一定的会在安装目录里建立一个lib目录。如果明明安装了某个包，而安装另一个包时，它愣是说找不到，可以抒那个包的lib路径加入的LDFALGS中试一下。

LIBS：告诉链接器要链接哪些库文件，如LIBS = -lpthread -liconv

简单地说，LDFLAGS是告诉链接器从哪里寻找库文件，而LIBS是告诉链接器要链接哪些库文件。不过使用时链接阶段这两个参数都会加上，所以你即使将这两个的值互换，也没有问题。

有时候LDFLAGS指定-L虽然能让链接器找到库进行链接，但是运行时链接器却找不到这个库，如果要让软件运行时库文件的路径也得到扩展，那么我们需要增加这两个库给"-Wl,R"：

LDFLAGS = -L/var/xxx/lib -L/opt/mysql/lib -Wl,R/var/xxx/lib -Wl,R/opt/mysql/lib

如果在执行./configure以前设置环境变量export LDFLAGS="-L/var/xxx/lib -L/opt/mysql/lib -Wl,R/var/xxx/lib -Wl,R/opt/mysql/lib" ，注意设置环境变量等号两边不可以有空格，而且要加上引号（shell的用法）。那么执行configure以后，Makefile将会设置这个选项，链接时会有这个参数，编译出来的可执行程序的库文件搜索路径就得到扩展了。

**= 是最基本的赋值  
:= 是覆盖之前的值  
?= 是如果没有被赋值过就赋予等号后面的值  
+= 是添加等号后面的值**

$(foreach TEMP\_FILE\_NAME, $(bgm\_reslist), $(eval include $(LOCAL\_PATH)/copyres.mk))

这个函数的意思是用TEMP\_FILE\_NAME遍历bgm\_relist，并调用include后的copyres.mk文件（该文件会用到TEMP\_FILE\_NAME的值）

$(filter <pattern...>,<text>)

名称：过滤函数——filter。

功能：以<pattern>模式过滤<text>字符串中的单词，保留符合模式<pattern>的单词。可以有多个模式。

返回：返回符合模式<pattern>;的字串。

示例：

sources := foo.c bar.c baz.s ugh.h

foo: $(sources)

cc $(filter %.c %.s,$(sources)) -o foo

$(filter %.c %.s,$(sources))返回的值是“foo.c bar.c baz.s”。

**Makefiel 学习：**

格式：

target ... : prerequisites ...

command

如果 make

找到一个 whatever.o，那么 whatever.c，就会是 whatever.o 的依赖文件。并且 cc -c　whatever.c 也会被推导出来

exp:

写　whatever.o : \*\*\*.h whaterver.h 　就够了

相当于：whatever.o : \*\*\*.h whaterver.h　whaterver.c

cc -c　whatever.c

= #用＝进行变量的赋值操作时，可以使用该变量之前的或者之后的定义的其他变量。（递归复制时会有问题）

A = $(B)

B = $(A)　　　＃一直复制，make报错

:=　　＃这种方法，前面的变量不能使用后面的变量，只能使用前面已定义好了的变量。

$\* 不包含扩展名的目标文件名称。  
$+ 所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件。  
$< 第一个依赖文件的名称。如果依赖目标是以模式（即"%"）定义的，那么"$<"将是符合模式的一系列的文件集。注意，其是一个一个取出来的。  
$? 所有的依赖文件，以空格分开，这些依赖文件的修改日期比目标的创建日期晚。（也就是需要创建）  
$@ 表示规则中的目标文件集。在模式规则中，如果有多个目标，那么，"$@"就是匹配于目标中模式定义的集合。

$^ 所有的依赖文件，以空格分开，不包含重复的依赖文件。  
$% 如果目标是归档成员，则该变量表示目标的归档成员名称。例如，如果目标名称为 mytarget.so(image.o)，则 $@ 为 mytarget.so，而 $% 为 image.o。

cc -M main.c

得到：main.o : main.c defs.h

C/C++编译的一个功能。大多数的C/C++编译器都支持一个“-M”的选项，即自动找寻源文件中包含的头文件，并生成一个依赖关系。

gcc -MM main.c 　　（使用gcc -M　会把标准库的头文件放进来 ）

得到：main.o : main.c defs.h

**Subst函数**

$(subst FROM, TO, TEXT)，即将字符串TEXT中的子串FROM变为TO。

1. wildcard : 扩展通配符  
   2、notdir ： 去除路径  
   3、patsubst ：替换通配符

如：

“$(var:<pattern>=<replacement>)”

相当于

“$(patsubst <pattern>,<replacement>,$(var))”，

一般我们可以使用“*$(wildcard \*.c)*”来获取工作目录下的所有的*.c*文件列表

***objects := $(patsubst %.c,%.o,$(wildcard \*.c))***

**Makefile中的判断语句：**

make 是在读取 Makefile 时就计算条件表达式的值，并根据条件表达式的值来选择语句，所以，你最好不要把自动化变量（如“$@”等）放入条件表达式中，因为自动化变量是在运行时才有的。

也可以使用函数

$(if <condition>,<then-part>)

或者是$(if <condition>,<then-part>,<else-part>)