Android 编译相关知识讨论

部　门：Honestar CE-技术
菜　鸟：Kevin Xu
日　期：2016年10月8日

基本编译原理

- GCC 编译详解
- C /C++源代码编译过程
- Java 源代码编译与运行过程
- Makefile 与Android.mk 简单介绍

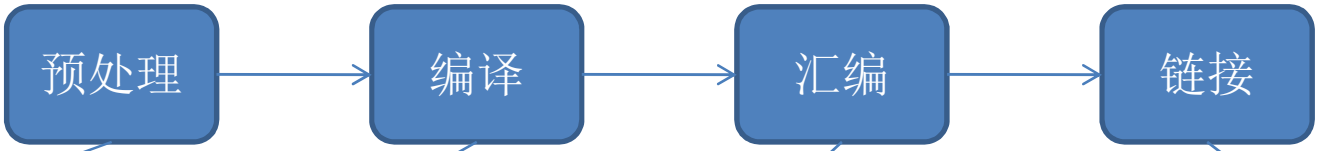
GCC 编译详解

GNU CC（简称为Gcc）是GNU项目中符合ANSI C标准的编译系统，能够编译用C、C++和Object C等语言编写的程序。GCC不仅功能强大，而且可以编译如C、C++、Object C、Java、Fortran、Pascal、Modula-3和Ada等多种语言，而且Gcc又是一个交叉平台编译器，它能够在当前CPU平台上为多种不同体系结构的硬件平台开发软件，因此尤其适合在嵌入式领域的开发编译。我们主要关注与我们相关的C， C++，JAVA 这里不用GCC，而是使用jdk。

GCC所支持后缀名解释

后 缀 名	所对应的语言	后 缀 名	所对应的语言
.c	C原始程序	.s/.S	汇编语言原始程序
.C/.cc/.cxx/.cpp	C++原始程序	.h	预处理文件（头文件）
.m	Objective-C原始程序	.o	目标文件
.i	已经过预处理的C原始程序	.a/.so	编译后的库文件
.ii	已经过预处理的C++原始程序		

GCC的编译流程



编译器将代码中的.h编译进来,将宏展开与替换，并且用户可以使用Gcc的选项”-E”进行查看，该选项的作用是让Gcc在预处理结束后停止编译过程。

GCC首先要检查代码的规范性、是否有语法错误等，以确定代码的实际要做的工作，在检查无误后，Gcc把代码翻译成汇编语言。用户可以使用”-S”选项来进行查看，该选项只进行编译而不进行汇编，生成汇编代码。

汇编阶段是把编译阶段生成的”.s”文件转成目标文件，在此可使用选项”-c”就可看到汇编代码已转化为”.o”的二进制目标代码了

进行二进制与函数库的链接，函数库一般分为静态库和动态库两种。静态库是指编译链接时，把库文件的代码全部加入到可执行文件中，因此生成的文件比较大，但在运行时也就不再需要库文件了。其后缀名一般为”.a”。动态库与之相反，在编译链接时并没有把库文件的代码加入到可执行文件中，而是在程序执行时由运行时链接文件加载库，这样可以节省系统的开销。动态库一般后缀名为”.so”，如前面所述的libc.so.6就是动态库。GCC在编译时默认使用动态库。

C /C++源代码编译过程

test.c

1.预处理(Preproceessing)

预处理的过程主要处理包括以下过程：

将所有的#define删除，并且**展开所有的宏定义**

处理所有的**条件预编译指令**，比如#if #ifdef #elif #else #endif等

处理#include 预编译指令，将被包含的文件插入到该预编译指令的位置。

删除所有注释 “//”和”/* */”.

添加行号和文件标识，以便编译时产生调试用的行号及编译错误警告行号。

保留所有的**#pragma编译器指令**，因为编译器需要使用它们

test.i

gcc -E test.c -o test.i

2. 编译(Compilation)

编译过程就是把预处理完的文件进行一系列的词法分析，语法分析，语义分析及优化后生成相应的汇编代码。

test.s

gcc -S test.i -o test.s

3. 汇编(Assembly)

汇编器是将汇编代码转变成机器可以执行的命令，每一个汇编语句几乎都对应一条机器指令。汇编相对于编译过程比较简单，根据汇编指令和机器指令的对照表一一翻译即可。

test.o

gcc –c hello.c –o hello.o

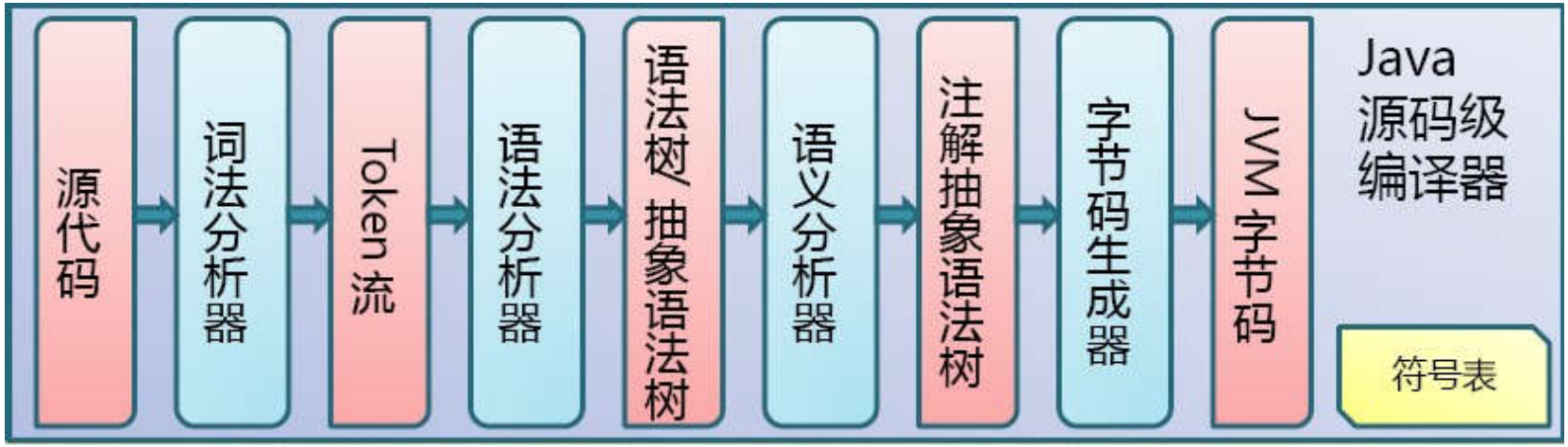
4. 链接(Linking)

通过调用链接器ld来链接程序运行需要的一大堆目标文件，以及所依赖的其它库文件，最后生成可执行文件.

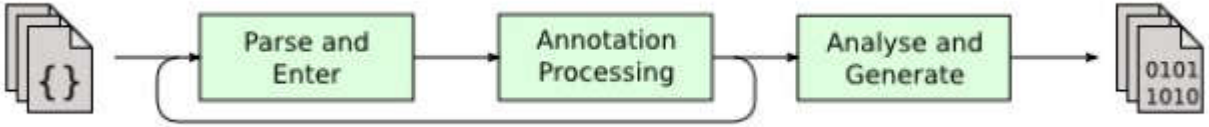
gcc test.c -Wl,--verbose (用来查看链接的库文件)

Java 编译简解

Java代码编译



Java 源码编译：
分析和输入到符号表
注解处理
语义分析和生成class文件

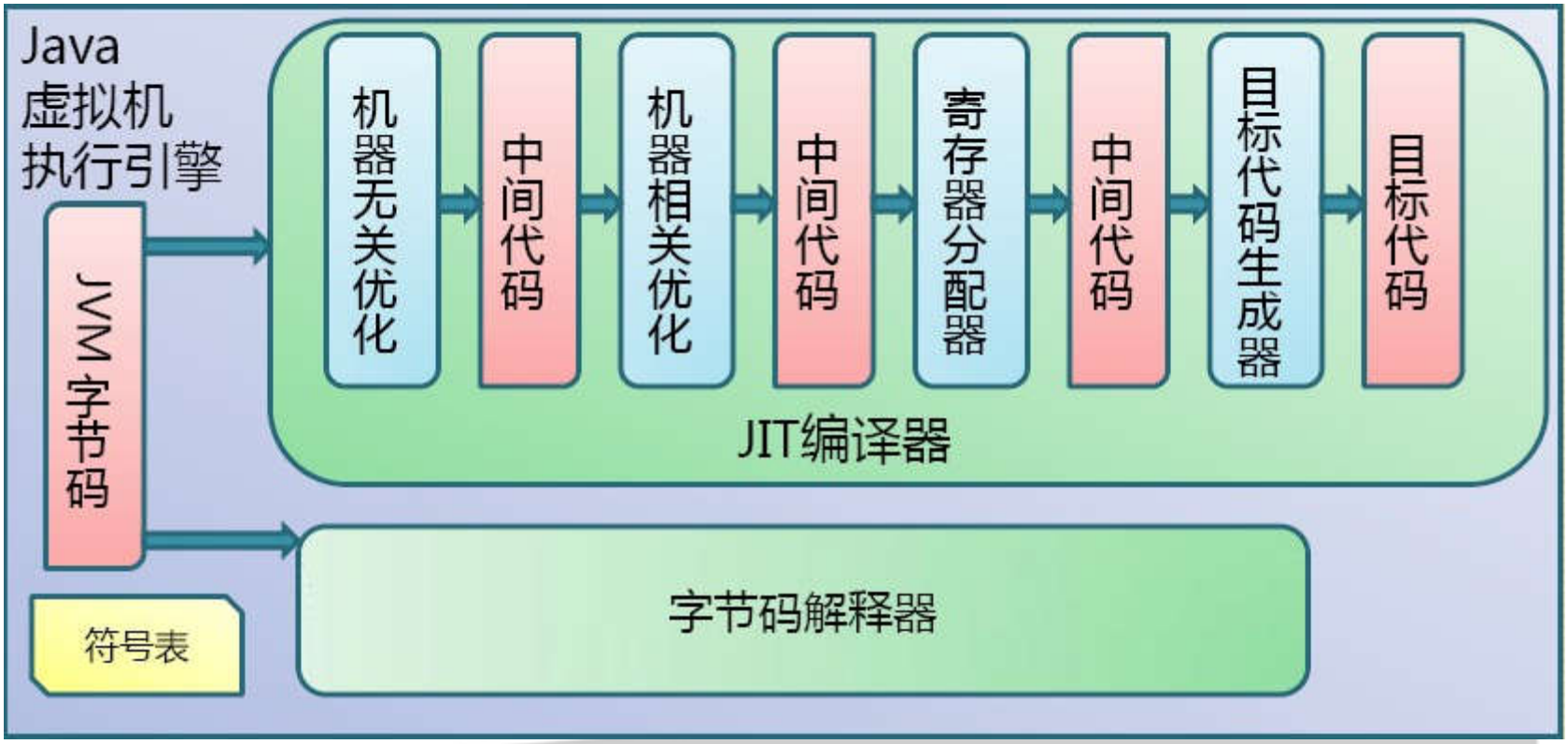


Java 运行简介

Java字节码的执行是由JVM(Java虚拟机)执行引擎来完成

xxx.Class 字节码结构信息。包括class文件格式版本号及各部分的数量与大小的信息元数据。对应于Java源码中声明与常量的信息。包含类/继承的超类/实现的接口的声明信息、域与方法声明信息和常量池

方法信息。对应Java源码中语句和表达式对应的信息。包含字节码、异常处理器表、求值栈与局部变量区大小、求值栈的类型记录、调试符号信息



Java 运行过程

Java类的运行的过程分为两个部分：

1. 类的加载
2. 类的执行

NOTE：JVM只会在程序第一次主动使用类的时候才会去加载要使用的类，
JVM并不是在程序运行的一开始就把所有的类加载到内存中去。

类的加载过程：

JVM会先去方法区中找有没有相应类的.class存在。如果有，就直接使用；如果没有，则把相关类的.class加载到方法区

在.class加载到方法区时，会分为两部分加载：先加载非静态内容，再加载静态内容

加载非静态内容：把.class中的所有非静态内容加载到方法区下的非静态区域内

加载静态内容：

把.class中的所有静态内容加载到方法区下的静态区域内

静态内容加载完成之后，对所有的静态变量进行默认初始化

所有的静态变量默认初始化完成之后，再进行显式初始化

当静态区域下的所有静态变量显式初始化完后，执行静态代码块

当静态区域下的静态代码块，执行完之后，整个类的加载就完成了

类执行机制

JVM是基于栈的体系结构来执行class字节码的。线程创建后，都会产生程序计数器（PC）和栈（Stack），程序计数器存放下一条要执行的指令在方法内的偏移量，栈中存放一个个栈帧，每个栈帧对应着每个方法的每次调用，而栈帧又是有局部变量区和操作数栈两部分组成，局部变量区用于存放方法中的局部变量和参数，操作数栈中用于存放方法执行过程中产生的中间结果。

Makefile 简单介绍

什么是Makefile:

通俗的理解， makefile就像一个Shell脚本一样，其中也可以执行操作系统的命令。makefile带来的好处就是——“自动化编译”

make命令执行：

需要一个 Makefile 文件，让make命令根据条件去编译和链接程序。

1.1 Makefile的规则

目标（标签）：依赖文件

命令

示例：

```
kevin@xu:~/test$ cat Makefile
test:
	gcc -o test test.c
kevin@xu:~/test$   make
gcc -o test test.c
kevin@xu:~/test$   make  test
gcc -o test test.c
```

1.2 Makefile主要包含：显式规则、隐晦规则、变量定义、文件指示和注释

显式规则:显式规则指明，如何生成一个或多的的目标文件。由Makefile的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。
隐晦规则:根据make自有自动推导的功能，所以隐晦的规则可以让我们比较粗糙地简略地书写Makefile，这是由make所支持的。
变量定义：在Makefile中我们要定义一系列的变量，变量一般都是字符串，这个有点你C语言中的宏，当Makefile被执行时，其中的变量都会被扩展到相应的引用位置上。
文件指示：包括了三个部分，一个是在一个Makefile中引用另一个Makefile，就像C语言中的include一样；另一个是指根据某些情况指定Makefile中的有效部分，就像C语言中的预编译#if一样；还有就是定义一个多行的命令。
注释：Makefile中只有行注释，和UNIX的Shell脚本一样，其注释是用“#”字符，这个就像C/C++中的“//”一样。如果你要在你的Makefile中使用“#”字符，可以用反斜框进行转义，如：“\#”。
NOTE:在Makefile中的命令，必须要以[Tab]键开始。

1.3 引用其它的Makefile

在Makefile使用include关键字可以把别的Makefile包含进来，这很像C语言的#include，被包含的文件会原模原样的放在当前文件的包含位置。include的语法是：

include<filename>

filename可以是当前操作系统Shell的文件模式（可以保含路径和通配符）

在include前面可以有一些空字符，但是绝不能是[Tab]键开始。include和可以用一个或多个空格隔开。

make命令开始时，会根据include所指出的其它Makefile，并把其内容置换到当前的位置。像C/C++的#include指令一样。如果文件都没有指定绝对路径或是相对路径的话，make会在当前目录下首先寻找，如果当前目录下没有找到，make会在如下的几个目录下找：

make执行时，有“-I”或“--include-dir”参数，make就会在这个参数所指定的目录下去寻找。

make 会在目录/include（一般是：/usr/local/bin或/usr/include）

如果有文件没有找到的话，make会生成一条警告信息，但不会马上出现致命错误。它会继续载入其它的文件，一旦完成makefile的读取，make会再重试这些没有找到，或是不能读取的文件，如果还是不行，make才会出现一条致命信息。如果你想让make不理那些无法读取的文件，而继续执行，你可以在include前加一个减号“-”。

如：-include<filename>

其表示，无论include过程中出现什么错误，都不要报错继续执行。和其它版本make兼容的相关命令是sinclude，其作用和这一个是一样的。

NOTE:**环境变量 MAKEFILES**

当前环境中定义环境变量MAKEFILES，make会把这个变量中的值做类似于include的动作。这个变量中的值是其它的Makefile，用空格分隔。只是，它和include不同的是，从这个环境变中引入的Makefile的“目标”不会起作用，如果环境变量中定义的文件发现错误，make也会不理。

1.4 make的工作流程

1. 找到所有的Makefile。
2. 载入被include的其它Makefile。
3. 初始化文件中的变量。
4. 推导隐晦规则，并分析所有规则。
5. 为所有的目标文件创建依赖关系链。
6. 根据依赖关系，决定哪些目标要重新生成。
7. 执行生成命令。

2 Makefile书写规则

规则包含两个部分，**依赖关系，生成目标的方法**。

2.1通配符

make支持三各通配符：“*”，“?”和“[...]”

2.2伪目标

为了避免和文件重名的这种情况，我们可以使用一个特殊的标记“.PHONY”来显示地指明一个目标是“伪目标”，向make说明，不管是否有这个文件，这个目标就是“伪目标”。

Eg:

```
.PHONY : test
test:
        gcc  -o test.c test
```

2.3多目标

多个目标依赖同一个（些）文件

2.4 Makefile Shell命令

用“@”字符在命令行前，那么，这个命令将不被make显示出来

命令运行完后，make会检测每个命令的返回码，如果命令返回成功，那么make会执行下一条命令，当规则中所有的命令成功返回后，这个规则就算是成功完成了。如果一个规则中的某个命令出错了（命令退出码非零），那么make就会终止执行当前规则，这将有可能终止所有规则的执行。

在Makefile的命令行前加一个减号“-”（在Tab键之后），标记为不管命令出不出错都认为是成功的。

2.5嵌套执行make

使用make –C 来进入子目录进行掉用子目录的makefile

2.6 命令打包

Makefile中出现一些相同命令序列，那么我们可以为这些相同的命令序列定义一个变量。定义这种命令序列的语法以“define”开始，以“endef”结束

变量在声明时需要给予初使值，而在使用时，需要给在变量名前**加上“$”符号，但最好用小括号“（）”或是大括号“{}”把变量给包括起来**。如果你要使用真实的“$”字符，那么你需要用“$$”来表示。变量可以使用在许多地方，如规则中的“目标”、“依赖”、“命令”以及新的变量中。

变量的赋值　　=　:= ?= +=

“=”号，在“=”左侧是变量，右侧是变量的值;

“?=” 等价于“=”只是在变量的引用上，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;

“?=” 如果之前没定义过此变量，才定义此变量;

“+=” 如果变量之前没有定义过，那么，“+=”会自动变成“=”，如果前面有变量定义，那么“+=”会继承于前次操作的赋值符。如果前一次的是“:=”，那么“+=”会以“:=”作为其赋值符.

4.条件判断

ifeq、else和endif，使用条件判断，可以让make根据运行时的不同情况选择不同的执行分支。条件表达式可以是比较变量的值，或是比较变量和常量的值。

Ifdef/ifndef　ifneq/ ifeq

5.使用函数

函数调用，很像变量的使用，也是以“$”来标识的，其语法如下：

$(<function> <arguments>)

<function>是函数名，<arguments>是函数的参数，参数间以逗号“,”分隔，而函数名和参数之间以“空格”分隔。函数调用以“$”开头，以圆括号或花括号把函数名和参数括起。

5.1字符串处理函数

$(subst <from>,<to>,<text>)

名称：字符串替换函数——**subst**。

功能：把字串<text>中的<from>字符串替换成<to>。

返回：函数返回被替换过后的字符串。

Eg：$(subst ee,EE,feet on the street)，

“feet on the street”中的“ee”替换成“EE”，返回结果是“fEEt on the strEEt”

$(patsubst <pattern>,<replacement>,<text>)

名称：模式字符串替换函数——patsubst。

功能：查找<text>中的单词（单词以“空格”、“Tab”或“回车”“换行”分隔）是否符合模式<pattern>，如果匹配的话，则以<replacement>替换。这里，<pattern>可以包括通配符“%”，表示任意长度的字串。如果<replacement>中也包含“%”，那么，<replacement>中的这个“%”将是<pattern>中的那个“%”所代表的字串。（可以用“\”来转义，以“\%”来表示真实含义的“%”字符）返回：函数返回被替换过后的字符串。

Eg： $(patsubst %.c,%.o,x.c.c bar.c)

把字串“x.c.c bar.c”符合模式[%.c]的单词替换成[%.o]，返回结果是“x.c.o bar.o”

$(strip <string>)

名称：去空格函数——strip。

功能：去掉<string>字串中开头和结尾的空字符。

返回：返回被去掉空格的字符串值。

Eg： $(strip a b c)

把字串“a b c ”去到开头和结尾的空格，结果是“a b c”。

$(findstring <find>,<in>)

名称：查找字符串函数——findstring。

功能：在字串<in>中查找<find>字串。

返回：如果找到，那么返回<find>，否则返回空字符串。

Eg： $(findstring a,a b c)

$(findstring a,b c)

第一个函数返回“a”字符串，第二个返回“ ”字符串（空字符串）

$(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

Eg：
```
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”。

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

名称：反过滤函数——filter-out。

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

返回：返回不符合模式<pattern>的字串。

Eg：objects=main1.o foo.o main2.o bar.o

mains=main1.o main2.o

$(filter-out $(mains),$(objects)) 返回值是“foo.o bar.o”。

$(sort <list>)

名称：排序函数——sort。

功能：给字符串<list>中的单词排序（升序）。

返回：返回排序后的字符串。

Eg：$(sort foo bar lose)返回“bar foo lose”。

备注：sort函数会去掉<list>中相同的单词。

$(word <n>,<text>)

名称：取单词函数——word。

功能：取字符串<text>中第<n>个单词。（从一开始）

返回：返回字符串<text>中第<n>个单词。如果<n>比<text>中的单词数要大，那么返回空字符串。

Eg：$(word 2, foo bar baz)返回值是“bar”。

$(words <text>)

名称：单词个数统计函数——words。
功能：统计<text>中字符串中的单词个数。
返回：返回<text>中的单词数。
Eg：$(words, foo bar baz)返回值是“3”。
备注：如果我们要取<text>中最后的一个单词，我们可以这样：$(word $(words <text>),<text>)。

$(firstword <text>)

名称：首单词函数——firstword。
功能：取字符串<text>中的第一个单词。
返回：返回字符串<text>的第一个单词。
Eg：$(firstword foo bar)返回值是“foo”。
备注：这个函数可以用word函数来实现：$(word 1,<text>)。

以上，是所有的字符串操作函数，如果搭配混合使用，可以完成比较复杂的功能

5.2文件名操作函数

$(dir <names...>)

名称：取目录函数——dir。

功能：从文件名序列<names>中取出目录部分。目录部分是指最后一个反斜杠（“/”）之前的部分。如果没有反斜杠，那么返回“./”。

返回：返回文件名序列<names>的目录部分。

Eg： $(dir src/foo.c hacks)返回值是“src/ ./”。

$(notdir <names...>)

名称：取文件函数——notdir。

功能：从文件名序列<names>中取出非目录部分。非目录部分是指最后一个反斜杠（“/”）之后的部分。

返回：返回文件名序列<names>的非目录部分。

Eg： $(notdir src/foo.c hacks)返回值是“foo.c hacks”。

$(suffix <names...>)

名称：取后缀函数——suffix。

功能：从文件名序列<names>中取出各个文件名的后缀。

返回：返回文件名序列<names>的后缀序列，如果文件没有后缀，则返回空字串。

示例：$(suffix src/foo.c src-1.0/bar.c hacks)返回值是“.c .c”。

$(basename <names...>)

名称：取前缀函数——basename。

功能：从文件名序列<names>中取出各个文件名的前缀部分。

返回：返回文件名序列<names>的前缀序列，如果文件没有前缀，则返回空字串。

Eg：$(basename src/foo.c src-1.0/bar.c hacks)返回值是“src/foo src-1.0/bar hacks”。

$(addsuffix <suffix>,<names...>)

名称：加后缀函数——addsuffix。

功能：把后缀<suffix>加到<names>中的每个单词后面。

返回：返回加过后缀的文件名序列。

Eg：$(addsuffix .c,foo bar)返回值是“foo.c bar.c”。

$(addprefix <prefix>,<names...>)

名称：加前缀函数——addprefix。

功能：把前缀<prefix>加到<names>中的每个单词后面。

返回：返回加过前缀的文件名序列。

Eg：$(addprefix src/,foo bar)返回值是“src/foo src/bar”。

$(join <list1>,<list2>)

名称：连接函数——join。

功能：把<list2>中的单词对应地加到<list1>的单词后面。如果<list1>的单词个数要比<list2>的多，那么，<list1>中的多出来的单词将保持原样。如果<list2>的单词个数要比<list1>多，那么，<list2>多出来的单词将被复制到<list2>中。

返回：返回连接过后的字符串。

Eg：$(join aaa bbb , 111 222 333)返回值是“aaa111 bbb222 333”。

$(foreach <var>,<list>,<text>)

函数的意思，把参数<list>中的单词逐一取出放到参数<var>所指定的变量中，然后再执行<text>所包含的表达式。每一次<text>会返回一个字符串，循环过程中，<text>的所返回的每个字符串会以空格分隔，最后当整个循环结束时，<text>所返回的每个字符串所组成的整个字符串（以空格分隔）将会是foreach函数的返回值。

<var>最好是一个变量名，<list>可以是一个表达式，而<text>中一般会使用<var>

这个参数来依次枚举<list>中的单词。

Eg：

```
names := a b c d
files := $(foreach n,$(names),$(n).o)
```

$(name)中的单词会被挨个取出，并存到变量“n”中，“$(n).o”每次根据“$(n)”计算出一个值，这些值以空格分隔，最后作为foreach函数的返回，所以，$(files)的值是“a.o b.o c.o d.o”。

foreach中的<var>参数是一个临时的局部变量，foreach函数执行完后，参数<var>的变量将不在作用，其作用域只在foreach函数当中。

Q

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

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

if函数可以包含“else”部分，或是不含。即if函数的参数可以是两个，也可以是三个。<condition>参数是if的表达式，如果其返回的为非空字符串，那么这个表达式就相当于返回真，于是，<then-part>会被计算，否则<else-part> 会被计算。if函数的返回值是，如果<condition>为真（非空字符串），那个<then- part>会是整个函数的返回值，如果<condition>为假（空字符串），那么<else-part>会是整个函数的返回值，此时如果<else-part>没有被定义，那么，整个函数返回空字串。所以，<then-part>和<else-part>只会有一个被计算。

$(call <expression>,<parm1>,<parm2>,<parm3>...

call函数是唯一一个可以用来创建新的参数化的函数。写一个非常复杂的表达式，这个表达式中，可以定义许多参数，然后可以用call函数来向这个表达式传递参数。

当 make执行这个函数时，<expression>参数中的变量，如$(1)，$(2)，$(3)等，会被参数<parm1>，<parm2>，<parm3>依次取代。而<expression>的返回值就是 call函数的返回值。

例如：

```
reverse = $(1) $(2)
foo = $(call reverse,a,b)
```

foo的值就是“a b”。参数的次序是可以自定义的，不一定是顺序的，

如：

```
reverse = $(2) $(1)
foo = $(call reverse,a,b)
```

此时的foo的值就是“b a”。

$(origin <variable>)

origin函数不像其它的函数，并不操作变量的值，只是找出这个变量的源头。

NOTE：<variable>是变量的名字，不是引用。所以不要在<variable>中使用“$”字符。Origin函数会以其返回值来表明变量的“源头”；

Origin函数的返回值:

“**undefined**"：如果<variable>从来没有定义过，origin函数返回这个值“undefined”;

“**default**”：如果<variable>是一个默认的定义，比如“CC”这个变量;

“**environment**”:如果<variable>是一个环境变量，并且当Makefile被执行时，“-e”参数没有被打开。

“**file**”:如果<variable>这个变量被定义在Makefile中。

“**command line**”:如果<variable>这个变量是被命令行定义的。

“**override**”:如果<variable>是被override指示符重新定义的。

“**automatic**”:如果<variable>是一个命令运行中的自动化变量

shell函数

shell 函数参数就是操作系统Shell的命令。它和反引号“`”是相同的功能。shell函数把执行操作系统命令后的输出作为函数返回。

Eg:contents := $(shell cat foo)

控制make的函数

make提供了一些函数来控制make的运行。需要检测一些运行Makefile时的运行时信息，并且根据这些信息来决定，让make继续执行，还是停止。

$(error <text ...>)

产生一个致命的错误，<text ...>是错误信息。

$(warning <text ...>)

函数很像error函数，只是它并不会让make退出，只是输出一段警告信息，而make继续执行

Note: 可以使用此函数来打印看make 的执行，还可以使用$(shell echo <text...>)

6.make 的运行

命令行输入”make”

6.1make的退出码

0 —— 表示成功执行。

1 —— 如果make运行时出现任何错误，其返回1。

2 —— 如果使用了make的“-q”选项，并且make使得一些目标不需要更新，那么返回2。

6.2指定Makefile

make –f xxx.mk xxxfile

6.3指定目标

指定要编译的目标 使用： make target

6.4 make的参数

“-B”“--always-make”

所有的目标都需要重编译。(在Android 下经常使用的mm –B)

“-C <dir>”“--directory=<dir>”

指定读取makefile的目录。如果有多个“-C”参数，make的解释是后面的路径以前面的作为相对路径，并以最后的目录作为被指定目录。

“—debug[=<options>]”“-d”相当于“--debug=a”。

输出make的调试信息。它有几种不同的级别可供选择，如果没有参数，那就是输出最简单的调试信息。下面是<options>的取值：

a —all，输出所有的调试信息。

b —basic，只输出简单的调试信息。即输出不需要重编译的目标。

v —verbose，在b选项的级别之上。输出的信息包括哪个makefile被解析，不需要被重编译的依赖文件（或是依赖目标）等。

i —implicit，输出所以的隐含规则。

j —jobs，输出执行规则中命令的详细信息，如命令的PID、返回码等。

m —makefile，输出make读取makefile，更新makefile，执行makefile的信息。

“-e” “--environment-overrides”
指明环境变量的值覆盖makefile中定义的变量的值。
“-f=<file>” “--file=<file>” “--makefile=<file>”
指定需要执行的makefile。
“-h” “--help”
显示帮助信息。
“-I” “--ignore-errors”
在执行时忽略所有的错误。
“-I <dir>” “--include-dir=<dir>”
指定一个被包含makefile的搜索目标。可以使用多个“-I”参数来指定多个目录。
“-j [<jobsnum>]” “--jobs[=<jobsnum>]”
指同时运行命令的个数。如果没有这个参数，make运行命令时能运行多少就运行多少。
如果有一个以上的“-j”参数，那么仅最后一个“-j”才是有效的。
编译时使用的make –j8(一般jn n不要大于cpu 的核心线程数量);
“-w” “--print-directory”
输出运行makefile之前和之后的信息。这个参数对于跟踪嵌套式调用make时很有用。
“--no-print-directory”
禁止“-w”选项。

7.隐含规则

Makefile中的“隐含的”，早先约定了的，不需要再写出来的规则。

“隐含规则”会使用一些系统变量，我们可以改变这些系统变量的值来定制隐含规则的运行时的参数。如系统变量“CFLAGS”可以控制编译时的编译器参数。

如果不明确地写下规则，make就会在这些规则中寻找所需要规则和命令。当然，也可以使用make的参数“-r”或“--no-builtin-rules”选项来取消所有的预设置的隐含规则。

7. 1编译C程序的隐含规则

“<n>.o”的目标的依赖目标会自动推导为“<n>.c”，并且其生成命令是“$(CC) –c $(CPPFLAGS) $(CFLAGS)”

7. 2编译C++程序的隐含规则

“<n>.o” 的目标的依赖目标会自动推导为“<n>.cc”或是“<n>.C”，并且其生成命令是“$(CXX) –c $(CPPFLAGS) $(CFLAGS)”。（建议使用“.cc”作为C++源文件的后缀，而不是“.C”）

7.3汇编和汇编预处理的隐含规则

“<n>.o” 的目标的依赖目标会自动推导为“<n>.s”，默认使用编译品“as”，并且其生成命令是：“$(AS) $(ASFLAGS)”。“<n>.s” 的目标的依赖目标会自动推导为“<n>.S”，默认使用C预编译器“cpp”，并且其生成命令是：“$(AS) $(ASFLAGS)”。

7.4链接Object文件的隐含规则

“<n>” 目标依赖于“<n>.o”，通过运行C的编译器来运行链接程序生成（一般是“ld”），其生成命令是：“$(CC) $(LDFLAGS) <n>.o $(LOADLIBES) $(LDLIBS)”。这个规则对于只有一个源文件的工程有效，同时也对多个Object文件（由不同的源文件生成）的也有效

命令的变量

隐含规则中的命令中，基本上都是使用了预先设置的变量。可以在makefile中改变这些变量的值，或是在make的命令行中传入这些值，或是在环境变量中设置这些值.

AR 函数库打包程序。默认命令是“ar”。
AS 汇编语言编译程序。默认命令是“as”。
CC C语言编译程序。默认命令是“cc”。
CXX C++语言编译程序。默认命令是“g++”。
RM 删除文件命令。默认命令是“rm –f”。

命令参数的变量

ARFLAGS 函数库打包程序AR命令的参数。默认值是“rv”。
ASFLAGS 汇编语言编译器参数。（当明显地调用“.s”或“.S”文件时）。
CFLAGS C语言编译器参数。
CXXFLAGS C++语言编译器参数。
LDFLAGS 链接器参数。（如：“ld”）

Android.mk简介

Android.mk文件用来向编译系统描述如何编译你的源代码。更确切地说，该文件其实就是一个小型的Makefile。由于该文件会被NDK的编译工具解析多次，因此应该尽量减少源码中声明变量，因为这些变量可能会被多次定义从而影响到后面的解析。这个文件的语法允许把源代码组织成模块，每个模块属于下列类型之一：

APK程序：一般的Android程序，编译打包生成apk文件。

JAVA库：java类库，编译打包生成jar包文件。

C\C++应用程序：可执行的C/C++应用程序。

C\C++静态库：编译生产C/C++静态库，并打包成.a文件。

C\C++共享库：编译生成共享库，并打包成.so文件，有且只有共享库才能被安装/复制到APK包中。

模块变量

下面的变量用于向系统描述模块，它应该定义在include $(CLEAR_VARS)和include $(BUILD_***)之间。正如前面讲述的那样，$(CLEAR_VARS)是一个脚本，清除所有这些变量，除非在描述中显示注明。

1. **LOCAL_PATH**：这个变量用于给出当前文件的路径，必须在Android.mk的开头定义，可以这样使用：LOCAL_PATH := $(call my-dir)，这样这个变量不会被$(CLEAR_VARS)清除，因为每个Android.mk只需要定义一次（即使一个文件中定义了多个模块的情况下）。
2. **LOCAL_SRC_FILES**：当前模块包含的所有源代码文件。
3. **LOCAL_MODULE**：当前模块的名称，这个名称应当是唯一的，并且不能包含空格。模块间的依赖关系就是通过这个名称来引用的。
4. **LOCAL_MODULE_CLASS**：标识所编译模块最后放置的位置。ETC表示放置在/system/etc.目录下，APPS表示放置在/system/app目录下，SHARED_LIBRARIES表示放置在/system/lib目录下。如果具体指定，则编译的模块不会放到编译系统中，最后会在out对应product的obj目录下的对应目录中。
5. **LOCAL_SRC_FILES**：这是要编译的源代码文件列表。只要列出要传递给编译器的文件即可，编译系统会自动计算依赖关系。源代码文件路径都是相相对于LOCAL_PATH的，因此可以使用相对路径进行描述。
6. **LOCAL_JAVA_LIBRARIES**：当前模块依赖的Java共享库，也叫Java动态库。例如framework.jar包。
7. **LOCAL_STATIC_JAVA_LIBRARIES**：当前模块依赖的Java静态库，在Android里，导入的jar包和引用的第三方工程都属于Java静态库。

8. **LOCAL_STATIC_LIBRARIES**：当前模块在运行时依赖的静态库的名称。
9. **LOCAL_SHARED_LIBRARIES**：当前模块在运行时依赖的动态库的名称。
10. **LOCAL_C_INCLUDES**：c或c++语言需要的头文件的路径。
11. **LOCAL_CFLAGS**：提供给C/C++编译器的额外编译参数。
12. **LOCAL_PACKAGE_NAME**：当前APK应用的名称。
13. **LOCAL_CERTIFICATE**：签署当前应用的证书名称。
14. **LOCAL_MODULE_TAGS**：当前模块所包含的标签，一个模块可以包含多个标签。标签的值可能是eng、user、debug、development、optional。其中，optional是默认标签。
15. **LOCAL_DEX_PREOPT**：apk的odex优化开关，默认是false。
16. **LOCAL_REQUIRED_MODULES**：指定模块运行所依赖的模块（模块安装时将会同步安装它所依赖的模块）
17.**LOCAL_PROGUARD_FLAG_FILES** 在源码中进行混淆编译 也可以为null
18.**LOCAL_OVERRIDES_PACKAGES** 使其他的模块不参加编译，此处即使Browser不加入编译

Build系统中还定义了一些函数方便在Android.mk中使用，包括：

1. **$(call my-dir)**：获取当前文件夹的路径。
2. **$(call all-java-files-under, <src>)**：获取指定目录下的所有java文件。
3. **$(call all-c-files-under, <src>)**：获取指定目录下的所有c文件。
4. **$(call all-laidl-files-under, <src>)**：获取指定目录下的所有AIDL文件。
5. **$(call all-makefiles-under, <folder>)**：获取指定目录下的所有Make文件。
6. **$(call intermediates-dir-for, <class>, <app_name>, <host or target>, <common?>)**：获取Build输入的目标文件夹路径。

Android.mk 实例解析

MTvPlayer/Android.mk

```
LOCAL_PATH:= $(call my-dir)   # 获取当前路径
include $(CLEAR_VARS)   #调用函数清除之前路径

LOCAL_PACKAGE_NAME := MTvPlayer    #目标文件的名称
LOCAL_MODULE_TAGS := optional         #在eng user user-debug 哪种模式下编译

LOCAL_CERTIFICATE := platform              #当前应用的签名

LOCAL_SRC_FILES := $(call all-java-files-under, src)    #编译目标需要的源文件

LOCAL_JAVACFLAGS += -Xlint:all                          #编译Java文件时的参数

LOCAL_JAVA_LIBRARIES := com.mstar.android     #依赖的java库文件

include $(BUILD_PACKAGE)                          #调用函数进行编译
```

```
include $(CLEAR_VARS) #调用函数清除之前路径
LOCAL_MODULE := libcecmanager_jni #目标文件的名称
LOCAL_MODULE_TAGS := optional #当前应用的签名
LOCAL_SRC_FILES := \       #编译目标源文件
   com_mstar_android_tvapi_common_CecManager.cpp
LOCAL_C_INCLUDES := \      #头文件包含
   $(JNI_H_INCLUDE) \
   frameworks/base/core/jni \
   $(TARGET_TVAPI_LIBS_DIR)/include \
   $(TARGET_TVAPI_LIBS_DIR)/../msrv/common/inc \
   $(TARGET_TVAPI_LIBS_DIR)/../../core/muf/tvos/include
LOCAL_SHARED_LIBRARIES := \    #静态库文件包含
   libandroid_runtime \
   libnativehelper \
   libcutils \
   libutils \
   libbinder \
   libcecmanager
LOCAL_REQUIRED_MODULES := libcecmanager   #指定模块运行所依赖的模块
LOCAL_CFLAGS += $(local_tvjni_cflags)    #C++编译使用参数
include $(BUILD_SHARED_LIBRARY)         #编译生成静态库
```

linux中（ Android特有）基本命令使用介绍

在Android目录下使用

kevin@ubuntu:~/android-4.2.1$ source build/envsetup.sh

使用hmm命令，如下

kevin@ubuntu:~/android-4.2.1$ hmm

Invoke ". build/envsetup.sh" from your shell to add the following functions to your environment:

- lunch: lunch <product_name>-<build_variant>//选择使用某个项目

- tapas: tapas [<App1> <App2> ...] [arm|x86|mips] [eng|userdebug|user]//为编译平台设置，不用关注

-croot: Changes directory to the top of the tree.//回到Android根目录（使用gettop 获得）

-gettop 使用此命令获取当前Android的根目录

- m: Makes from the top of the tree.//在Android根目录下编译make等同与m

linux中（ Android特有）基本命令使用介绍

-mm: Builds all of the modules in the current directory.//编译当前的目录

- mmm: Builds all of the modules in the supplied directories. //编译指定的目录

- cgrep: Greps on all local C/C++ files. //grep 的封装

- jgrep: Greps on all local Java files. //grep 的封装

- resgrep: Greps on all local res/*.xml files. //grep 的封装

-godir: Go to the directory containing a file.//定位制定关键字的目录

-Printconfig 输出当前的配置信息

-Printenv 输出当前环境变量，exprot 命令的封装

Android 基本文件架构及快速调试技巧

Android 源码架构

|-- Makefile　　顶层**Makefile**文件，用于**make**
|-- abi　　ABI：applicationbinary interface，应用程序二进制接口
|-- bionic　　bionic C库 标准C， C++库
|-- bootable　　启动引导相关代码（recovery等）
|-- build　　存放系统编译规则及generic等基础开发配置包
|-- cts　　Android兼容性测试套件标准
|-- dalvik　　dalvik JAVA虚拟机
|-- development　　应用程序开发相关
|-- device 设备相关代码
|-- external　　android使用的一些开源的模组
|-- frameworks　　java及C++语言，是Android应用程序的框架。
|-- hardware　　主要是硬件适配层HAL代码
|-- libcore　　核心库相关
|-- ndk　　ndk相关代码。Android NativeDevelopment Kit
|-- out　　编译完成后的代码输出与此目录
|-- packages　　应用程序包
|-- prebuilt　　x86和arm架构下预编译的一些资源
|-- sdk　　sdk及模拟器
|-- system　　文件系统、应用及组件（系统system）

Android 基本文件架构及快速调试技巧

Android out目录

```
|-- boot.img            kernel+ramdisk.img 打包生成的文件
|-- cache
|-- cache.img           高速缓存分段,主要用于OTA升级
|-- clean_steps.mk
|-- data                主要是用于apk安装等系统运行数据存放
|-- installed-files.txt
|-- kernel              linux kernel
|-- obj                 编译时缓存，用于生成各个img
|-- previous_build_config.mk
|-- ramdisk-recovery.img      recovery模式使用的跟文件系统；
|-- ramdisk.img         根文件系统；
|-- recovery
|-- recovery.img        recovery模式运行的镜像，用于系统恢复
|-- root                根目录，用于生成ramdisk.img
|-- symbols
|-- system              android system文件系统
|-- system.img          android system文件系统镜像
|-- system.img.lzo
|-- userdata.img    用户数据分段镜像
`-- userdata.img.lzo
```

Android 根文件系统

Customer -> /tvcustomer/Customer mstar 特有customer 分段

Database -> /tvdatabase/Database mstar 特有Database分段
DatabaseBackup -> /tvdatabase/DatabaseBackup Database备份
acct
applications -> /tvservice/applications
blcr_zygote.sh
cache 用与OTA升级
certificate -> /tvconfig/config/certificate
config -> /tvconfig/config
d -> /sys/kernel/debug
data user_data分区。主要存放用户文件
default.prop 默认属性配置文件
dev 驱动创建的设备结点
etc -> /system/etc 系统配置文件
file_contexts
fstab.napoli 系统分区表
init init 文件
init.napoli.rc 启动配置文件
init.rc
init.recovery.napoli.rc
init.trace.rcinit.usb.rc
lib -> /tvservice/std_lib/lib

Android 根文件系统

mnt	挂载点目录
tvservice	
ueventd.napoli.rc	
ueventd.rc	
var	临时文件
vendor -> /system/vendor	不同厂家的特有目录
mslib -> /tvservice/mslib	
proc	系统进程运行文件
property_contexts	
root	超级用户root 根目录
sbin	命令存放文件
sdcard -> /mnt/sdcard	外部SD卡
seapp_contexts	
selinux	
sepolicy	
storage	所有存储设备
sys	驱动创建的文件目录
system	Android system 系统文件目录
tmp -> /var/tmp	
tvconfig	
tvcustomer	
tvdatabase	

Android system
文件系统

app	所有不可卸载的apk
bin	可执行的二进制文件
build.prop	编译时候的属性
etc	系统配置文件
fonts	字体配置
framework	Android框架中生成的各个jar包
lib	系统的C++ c生成的动态.so的库
lost+found	
media	系统影音目录
tts	文本语音转换文件
usr	用户文件夹，包含共享、键盘布局、时间区域文件等
vendor	
xbin	常用开发工具可执行的二进制文件