1 概述

Linux的Makefile有５个部分：

1. Makefile 顶层Makefile文件
2. .config 内核配置文件
3. arch/$(ARCH)/Makefile 架构相关的Makefile
4. scripts/Makefile.\* 所有kbuild　Makefiles文件的通用规则等
5. kbuild Makefiles 内核中有500个这样的文件

顶层Makefile读取由配置内核时得到的.config文件。

顶层Makefile负责编译两个主要文件：vmlinux（固有的内核映像）和modules（任何模块文件）。

它通过递归进入内核源码树的子目录下构建目标。

需要进入的子目录列表依赖于内核配置，顶层Makefile里面包含一个arch/$(ARCH)/Makefile架构下的Makefile文件。这个架构下的Makefile提供架构相关的信息给顶层Makefile。

每个子目录下有一个kbuild Makefile它执行从上面传递下来的命令。这个kbuild Makefile使用.config文件的信息来构造kbuild编译built-in（内置）或modular（模块）目标所需要的文件列表。

scripts/Makefile.\*包含所有根据kbuild makefiles文件来编译内核时所需要的定义和规则等等。

2 谁做什么

内核Makefile文件对于不同的人又四种不同的关系。

Users编译内核的人。这些人使用例如make menuconfig或者make命令。他们通常不会阅读或者编辑任何内核Makefile文件（和任何源码文件）。

Normal developers开发某一特性的人如设备驱动，文件系统和网络协议。这些人需要维护他们工作的子系统的kbuild Makefile文件。为了有效的做这些工作他们需要了解内核总体Makefile结构的知识，和kbuild公共接口的细节知识。

Arch developers开发整个架构体系的人，例如is64或者sparc架构。架构开发者需要了解　arch Makefile和kbuild Makefiles。

Kbuild developers开发内核构建系统本身的人，这些人需要了解所有内核Makefiles的细节。

这个文档的目标读者是Normal developers和Arch developers。

3 kbuild文件

内核中的大多数Makefile文件是kbuild Makefiles它是kbuild的基础。这章介绍kbuild makefiles中使用的语法知识。

kbuild文件的首选名字是Makefile但是Kbuild也可以使用，当Makefile和Kbuild同时出现时Kbuild会被使用。

3.1节目标定义是一个快速的简介，下面的章节将会提供细节和例子。

3.1 目标定义

目标定义是kbuild Makefile的主要部分（核心）。定义了哪些文件被编译，哪些特殊编译选项，哪些子目录会递归进入。

最简单的kbuild makefile包含一行：

obj-y += foo.o

它告诉kbuild这个目录有一个名字为foo.o的目标，这个目标依赖foo.c或者foo.S文件被编译。   
如果foo.o将被编译为模块，使用obj-m变量因此下面的例子经常使用：

obj-$(CONFIG\_FOO) += foo.o

$(CONFIG\_FOO)设置为y（内置）或者m（模块）。 如果CONFIG\_FOO既不是y和m,这个文件不会被编译和链接。

3.2 内置目标 - obj-y

kbuild Makefile通过$(obj-y)列表指定目标文件编译到vmlinux。这些列表依赖于内核配置。

Kbuild编译所有$(obj-y)文件，然后调用$(LD) -r将这些文件打包为built-in.o文件。built-in.o稍后会被父Makefile链接到vmlinux中。

$(obj-y)文件的顺序是有意义的，列表中的文件允许重复：第一个实例会被链接到built-in.o中，剩下的会被忽略。

链接顺序是有意义的，因为某些函数module\_init() / \_\_initcall在启动时会按他们出现的顺序被调用。所以要记住改变链接顺序可能　例如：会改变SCSI控制器的检测顺序，导致磁盘被重新编号。

#drivers/isdn/i4l/Makefile

# Makefile for the kernel ISDN subsystem and device drivers.

# 内核　ISDN　子系统和设备驱动的Makefile

# Each configuration option enables a list of files.

# 每一个配置选项使能一个列表中的文件

obj-$(CONFIG\_ISDN\_I4L) += isdn.o

obj-$(CONFIG\_ISDN\_PPP\_BSDCOMP) += isdn\_bsdcomp.o

3.3 可加载的目标 - obj-m

$(obj-m)指定目标文件被编译为可加载到内核的模块。

一个模块可能由一个源文件或者几个源文件编译而成。当只有一个源文件时，kbuild makefile简单的使用$(obj-m)来编译。例如:

#drivers/isdn/i4l/Makefile

obj-$(CONFIG\_ISDN\_PPP\_BSDCOMP) += isdn\_bsdcomp.o

注意：这个例子$(CONFIG\_ISDN\_PPP\_BSDCOMP)的值为m

如果内核模块是由几个源文件编译而成的，你想使用上述相同的方法编译模块，但是kbuild需要知道哪些目标文件是你需要编译进模块中，所以你不得不通过设置$(<module\_name>-y)变量来告诉它。例如:

#drivers/isdn/i4l/Makefile

obj-$(CONFIG\_ISDN\_I4L) += isdn.o

isdn-y := isdn\_net\_lib.o isdn\_v110.o isdn\_common.o

在这个例子中，模块名字是isdn.o。Kbuild会编译$(isdn-y)列出的所有目标文件然后执行$(LD) -r使这些文件生成isdn.o文件。

由于kbuild可以识别$(<module\_name>-y)来合成目标，你可以使用CONFIG\_符号的值来确定某些目标文件是否作为合成目标的一部分。例如:

#fs/ext2/Makefile

obj-$(CONFIG\_EXT2\_FS) += ext2.o

ext2-y := balloc.o dir.o file.o ialloc.o inode.o ioctl.o \

namei.o super.o symlink.o

ext2-$(CONFIG\_EXT2\_FS\_XATTR) += xattr.o xattr\_user.o \

xattr\_trusted.o

在这个例子中，xattr.o, xattr\_user.o 和 xattr\_trusted.o只是合成目标ext2.o的一部分当$(CONFIG\_EXT2\_FS\_XATTR)的值是y时。

注意：当然，当你编译目标到内核中去，上述语法也是可以用的。因此，如果CONFIG\_EXT2\_FS=y，kbuild会编译一个ext2.o文件然后链接到built-in.o文件中，正如你所期望的。

3.5 库文件 - lib-y

obj-\*列出的目标文件用于模块或者合成为特定目录下的built-in.o文件，也有可能被合成为一个库lib.a。  
lib-y列出的所有文件被合成为此目录下的一个库文件。   
obj-y列出的目标文件和另外加入的文件不会被包含在库中，因为他们会在任何时候被访问。   
为了保持一致性lib-m列出的文件会包含在lib.a文件中。

注意相同的kbuild　makefile会编译文件到built-in文件中和一部分到库文件中。 因此相同的目录可能同时包含built-in.o文件和lib.a文件例如:

#arch/x86/lib/Makefile

lib-y := delay.o

这会根据delay.o生成lib.a库文件，对于kbuild会意识到这是一个lib.a文件被编译， 这个目录会被列为libs-y。参见　6.3节。

lib-y一般仅限于使用在lib/和arch/\*/lib目录中。

3.6 Descending down in directories

一个Makefile文件只负责编译本目录下的目标，子目录下的文件归子目录下的Makefiles负责。编译系统会自动递归进入子目录并调用make，你需要知道这一点。

要做到这一点obj-y和obj-m会被使用。ext2分布在一个单独的目录中，fs/目录下的Makefile告诉kbuild使用下面规则进入子目录。例如:

#fs/Makefile

obj-$(CONFIG\_EXT2\_FS) += ext2/

如果CONFIG\_EXT2\_FS被设置为y或者m，相应的obj-变量会被设置，然后kbuild会进入ext2目录。Kbuild只是使用这些信息来决定是否需要进入此目录，然后子目录中的Makefile指定哪些需要编译为模块，哪些需要编译为built-in。

使用CONFIG\_变量指定目录名字是非常好的做法，这可以使kbuild跳过那些CONFIG\_选项不是y 和m的目录。

3.7 编译标志

ccflags-y, asflags-y和ldflags-y

这三个标志只适用于被分配的kbuild makefile文件，他们用于正常递归编译时调用cc（编译器） 和ld（链接器）时。   
注意：这是以前使用的EXTRA\_CFLAGS, EXTRA\_AFLAGS和EXTRA\_LDFLAGS三个标志，他们仍然可以使用但是过时了。

ccflags-y指定$(CC)的编译选项，例如:

# drivers/acpi/Makefile

ccflags-y := -Os

ccflags-$(CONFIG\_ACPI\_DEBUG) += -DACPI\_DEBUG\_OUTPUT

这个变量是必要的因为定层Makefile拥有$(KBUILD\_CFLAGS)变量它在整个文件树中使用这个编译标志。

asflags-y指定$(AS)的编译选项，例如:

#arch/sparc/kernel/Makefile

asflags-y := -ansi

ldflags-y指定$(LD)的编译选项，例如:

#arch/cris/boot/compressed/Makefile

ldflags-y += -T $(srctree)/$(src)/decompress\_$(arch-y).lds

subdir-ccflags-y,subdir-asflags-y

这两个标志和ccflags-y，asflags-y相似。不同的是subdir-前缀的变量对kbuild所在的目录以及子目录都有效。   
使用subdir-\*指定的选项会被加入到那些使用在没有子目录的变量之前。例如:

subdir-ccflags-y := -Werror

CFLAGS\_$@,AFLAGS\_$@

CFLAGS\_$@和AFLAGS\_$@只应用于当前kbuild　makefile文件中   
$(CFLAGS\_$@)是$(CC)针对单个文件的选项。$@代表某个目标文件。例如:

# drivers/scsi/Makefile

CFLAGS\_aha152x.o = -DAHA152X\_STAT -DAUTOCONF

CFLAGS\_gdth.o = # -DDEBUG\_GDTH=2 -D\_\_SERIAL\_\_ -D\_\_COM2\_\_ \

-DGDTH\_STATISTICS

这两行单独指定aha152x.o和gdth.o文件的编译选项。

$(AFLAGS\_$@)应用于汇编文件，例如:

# arch/arm/kernel/Makefile

AFLAGS\_head.o := -DTEXT\_OFFSET=$(TEXT\_OFFSET)

AFLAGS\_crunch-bits.o := -Wa,-mcpu=ep9312

AFLAGS\_iwmmxt.o := -Wa,-mcpu=iwmmxt

3.9 依赖跟踪

Kbuild按照如下方式跟踪依赖文件：

1. 所有必须的文件（包括 \*.c 和 \*.h）
2. CONFIG\_选项应用于所有必需文件
3. 命令行应用于编译目标

因此，如果你改变$(CC)的编译选项所有受影响的文件都会被重新编译。

3.10 特殊规则

特殊规则应用于kbuild基础选项没有提供需求。一个典型的例子是在编译的时候产生头文件。 另一个例子是架构相关的Makefiles需要特殊的规则来生成启动映像等等。

特殊规则像普通Make规则一样写，Kbuild不会执行该目录下的Makefile， 所以特殊规则需要提供必需文件和目标文件的相对路径。

定义特殊规则使用的两个变量：

**$(src)**

$(src)是Makefile在哪个目录下的相对路径，经常使用$(src)来指定文件在源码树的位置。

**$(obj)**   
$(obj)是目标文件存放的相对路径，经常使用$(obj)指定生成文件。例如:

#drivers/scsi/Makefile

$(obj)/53c8xx\_d.h: $(src)/53c7,8xx.scr $(src)/script\_asm.pl

$(CPP) -DCHIP=810 - < $< | ... $(src)/script\_asm.pl

这是一个特殊规则，遵循make要求的语法。   
这个目标文件依赖两个必需文件，使用$(obj)前缀指定目标文件，使用$(src)前缀指定必需文件。（因为他们不生成文件）

**$(kecho)**   
输出编译信息对于使用者是有益的但是当执行make -s时表示不输出任何信息除了警告和错误。为了支持这个功能kbuild定义了$(kecho)使信息输出到标准输出除非使用make -s命令。例如:

#arch/blackfin/boot/Makefile

$(obj)/vmImage: $(obj)/vmlinux.gz

$(call if\_changed,uimage)

@$(kecho) 'Kernel: $@ is ready'

3.11 $(CC)支持的函数

内核可以使用不同版本的$(CC)编译，每一种编译器有自己特性和选项。kbuild提供基本的检查$(CC)有效选项的功能，$(CC)通常是gcc 编译器，但是其他备用编译器也是可以的。

**as-option**   
as-option用于检查$(CC) – 当使用编译器编译汇编文件(\*.S)时 – 支持给定选项。如果第一种选项不支持时可以指定第二种选项。例如:

#arch/sh/Makefile

cflags-y += $(call as-option,-Wa$(comma)-isa=$(isa-y),)

在上面的例子中，cflags-y会使用-Wa$(comma)-isa=$(isa-y)选项当$(CC)支持这种选项时。第二个参数是可选的，当不支持第一个参数是他会被使用。

**cc-ldoption**   
cc-ldoption用于检查当$(CC)链接目标文件是是否支持给定选项。如果不支持第一种选项时第二种选项会被使用。例如:

#arch/i386/kernel/Makefile

vsyscall-flags += $(call cc-ldoption, -Wl$(comma)--hash-style=sysv)

在上面的例子中，vsyscall-flags会使用-Wl$(comma)--hash-style=sysv如果$(CC)支持它，第二个参数是可选的，当不支持第一个参数是他会被使用。

**还有更多函数请参考原文件**

6 架构相关的Makefiles

顶层Makefile在进入单个子目录前设置环境和做一些准备工作它包含通用部分，而arch/$(ARCH)/Makefile包含架构相关的kbuild设置选项。所以arch/$(ARCH)/Makefile需要设置几个变量和定义几个目标。

当kbuild运行时，遵循下面几个步骤（大概）：

1. 配置内核=>生成.config文件
2. 将内核版本号存放在include/linux/version.h文件中。
3. 更新全部其他先决条件为目标做准备,其他先决条件在arch/$(ARCH)/Makefile文件中指定
4. 递归进入所有列为init-\* core\* drivers-\* net-\* libs-\*的目录编译所有目标。上述变量的值在arch/$(ARCH)/Makefile文件中被扩展
5. 所有目标文件被链接，最终生成文件vmlinux放在源码根目录下。被列为head-y的目标第一个被链接，它是由arch/$(ARCH)/Makefile文件指定
6. 最后架构相关的部分做任何必要的后期处理并最终生成bootimage文件。这包括建立引导记录,准备initrd映像文件等等。

6.1 设置变量来配合编译此架构

**LDFLAGS** –通用$(LD)选项，此标志用于所有调用连接器的地方。

#arch/s390/Makefile

LDFLAGS := -m elf\_s390

注意: ldflags-y可以用于进一步的制定，这个标志的使用见3.7章。

**LDFLAGS\_MODULE** –$(LD)链接模块时的选项   
LDFLAGS\_MODULE用于当$(LD)链接.ko模块文件时，默认选项是-r重定位输出。

**LDFLAGS\_vmlinux** –$(LD)链接vmlinux时的选项

LDFLAGS\_vmlinux指定其他选项传递给连接器链接最终的vmlinux映像，是LDFLAGS\_$@的一个特例。

#arch/i386/Makefile

LDFLAGS\_vmlinux := -e stext

**OBJCOPYFLAGS** –objcopy选项

当$(call if\_changed,objcopy)用于反汇编.o文件时OBJCOPYFLAGS指定的选项会被使用。$(call if\_changed,objcopy)经常用来生成vmlinux的原始二进制文件。

#arch/s390/Makefile

OBJCOPYFLAGS := -O binary

#arch/s390/boot/Makefile

$(obj)/image: vmlinux FORCE

$(call if\_changed,objcopy)

在这个例子中$(obj)/image文件是vmlinux的二进制版本。$(call if\_changed,xxx)的用法将在后面说明。

**KBUILD\_AFLAGS** –$(AS)汇编器选项

默认值见顶层Makefile文件可以针对每一种架构进行扩展或者修改例如:

#arch/sparc64/Makefile

KBUILD\_AFLAGS += -m64 -mcpu=ultrasparc

**KBUILD\_CFLAGS** –$(CC)编译器选项

默认值见顶层Makefile文件，可以针对每一种架构进行扩展或者修改，通常KBUILD\_CFLAGS的值依赖于配置。例如:

#arch/i386/Makefile

cflags-$(CONFIG\_M386) += -march=i386

KBUILD\_CFLAGS += $(cflags-y)

很多架构Makefiles文件动态运行目标编译器来测试所支持的选项：

#arch/i386/Makefile

...

cflags-$(CONFIG\_MPENTIUMII) += $(call cc-option,\

-march=pentium2,-march=i686)

...

# Disable unit-at-a-time mode ...

KBUILD\_CFLAGS += $(call cc-option,-fno-unit-at-a-time)

...

第一个例子利用当配置选项被选择时展开为y这个技巧。

**后面还有其他变量请参考原文**

6.3 List directories to visit when descending

一个架构Makefile配合顶层Makefile定义变量来指定怎样编译vmlinux文件。注意模块和架构没有联系，所有模块编译都是架构无关的。

**head-y, init-y, core-y, libs-y, drivers-y, net-y**

$(head-y)列出最先链接到vmlinux的目标  
$(libs-y)列出lib.a库文件放置的目录   
剩下的变量列出built-in.o目标文件放置的目录

$(init-y)目标会放在$(head-y)后面,剩下的以这种顺序排列：   
$(core-y), $(libs-y), $(drivers-y)和$(net-y).

顶层Makefile定义所有普通目录的值，arch/$(ARCH)/Makefile只添加架构相关的目录例如:

#arch/sparc64/Makefile

core-y += arch/sparc64/kernel/

libs-y += arch/sparc64/prom/ arch/sparc64/lib/

drivers-$(CONFIG\_OPROFILE) += arch/sparc64/oprofile/

6.4 架构特定的启动映象

架构下的Makefile的目标是生成vmlinux文件，压缩，打包在引导代码中，并复制生成的文件到指定位置。这包含很多不同的安装命令。实际的目标在不同的体系架构下是不能统一的。

通常在arch/$(ARCH)/boot/目录下面进行额外的处理。

Kbuild不提供任何方法支持在boot/目录中编译特定的目标。因此arch/$(ARCH)/Makefile会调用make在boot/目录下编译目标。

推荐的方法是在arch/$(ARCH)/Makefile文件中包含快捷方式， 在arch/$(ARCH)/boot/Makefile目录中使用全路径调用。例如:

#arch/i386/Makefile

boot := arch/i386/boot

bzImage: vmlinux

$(Q)$(MAKE) $(build)=$(boot) $(boot)/$@

"$(Q)$(MAKE) $(build)=<dir>"是一种被推荐的方式在子目录中调用make

没有规则命名架构特定的目标，但是当执行"make help"时会列出所有相应的目标。 为了支持它必须定义$(archhelp)，例如:

#arch/i386/Makefile

define archhelp

echo '\* bzImage - Image (arch/$(ARCH)/boot/bzImage)'

endif

当不带参数执行make时遇到的第一个目标是built，在顶层Makefile中第一个目标是all:体系架构下会始终默认生成一个可引导的映像文件。   
当执行"make help"时默认目标会使用'\*'标记出来。 在all:增加一个新的目标使他作为默认目标不同于vmlinux，例如:

#arch/i386/Makefile

all: bzImage

当不带参数执行"make"时，bzImage会被生成。

6.6 编译启动映像的几个命令

Kbuild提供了几个很有用的宏来编译启动映像。

**if\_changed**   
下面的命令是if\_changed的基本用法:

target: source(s) FORCE

$(call if\_changed,ld/objcopy/gzip)

当这条规则被执行时，他会检查所有文件是否需要更新或者在上一次调用后命令是否被改变了。如果任何选项改变了都会强制重新编译。

任何使用if\_changed的目标必须列入到$(targets),否则命令会检查失败，这个目标将会始终被编译。

分配到$(targets)中的目标没有$(obj)/前缀。if\_changed会结合自定义命令使用，定义在6.7节”自定义kbuild命令”.

注意： 忘记FORCE先决条件是一个典型的错误。另一个陷阱是空白有时候是有意义的 例如：下面的命令是错误的（注意逗号后面额外的空格）

target: source(s) FORCE

#WRONG!# $(call if\_changed, ld/objcopy/gzip)

6.8 预处理链接脚本

当vmlinux映像被编译时，arch/$(ARCH)/kernel/vmlinux.lds链接脚本会被使用。   
这个脚本是同一目录下的vmlinux.lds.S文件的变种。   
kbuild知道.lds文件，包含一个规则将\*lds.S->\*lds。例如:

#arch/i386/kernel/Makefile

always := vmlinux.lds

#Makefile

export CPPFLAGS\_vmlinux.lds += -P -C -U$(ARCH)

分配到$(always)的变量是用来告诉kbuild编译vmlinux.lds这个目标。   
分配到$(CPPFLAGS\_vmlinux.lds)的变量是用来告诉kbuild　   
使用指定选项来编译vmlinux.lds目标

当编译\*.lds目标时kbuild使用下面几个变量：

KBUILD\_CPPFLAGS : 在顶层Makefile中设置   
cppflags-y : 可能在kbuild makefile文件中设置   
CPPFLAGS\_$(@F) : 目标特定的选项，注意这里使用了完整文件名。

kbuild编译\*lds文件的基本命令用于多个架构特定文件。

后记

此篇文章是翻译Linux内核源码树中/Documentation/kbuild/makefiles.txt文件。   
我只是将我认为重要的部分进行了翻译，还有很多内容没有翻译。   
但是也花了我三天时间 **翻译不易**。   
最后说明：作者水平有限，翻译的很差劲可能有些地方会有错误，只能对着原文参考此文章，其实翻译后我自己都不想看这篇文章。。。