Linux 内核2.6 Makefile 文件

=== 目录

=== 1 概述

Makefile 包括五部分：

Makefile	顶层 Makefile 文件
.config	内核配置文件
arch/$(ARCH)/Makefile	机器体系 Makefile 文件
scripts/Makefile.*	所有内核 Makefiles 共用规则
kbuild Makefiles	其它 makefile 文件

通过内核配置操作产生.config 文件，顶层 Makefile 文件读取该文件的配置。顶层 Makefile 文件负责产生两个主要的程序：vmlinux (内核 image)和模块。顶层 Makefile 文件根据内核配置，通过递归编译内核代码树子目录建立这两个文件。顶层 Makefile 文件文本一个名为 arch/$(ARCH)/Makefile 的机器体系 makefile 文件。机器体系 Makefile 文件为顶层 makefile 文件提供与机器相关的信息。每一个子目录有一个 makefile 文件，子目录 makefile 文件根据上级目录 makefile 文件命令启动编译。这些 makefile 使用. config 文件配置数据构建各种文件列表，并使用这些文件列表编译内嵌或模块目标文件。scripts/Makefile.*包含了所有的定义和规则，与 makefile 文件一起编译出内核程序。

=== 2 角色分工

人们与内核 makefile 存在四种不同的关系：
用户 用户使用"make menuconfig"或"make"命令编译内核。他们通常不读或编辑内核 makefile 文件或其他源文件。
普通开发者 普通开发者维护设备驱动程序、文件系统和网络协议代码，他们维护相关子系统的 makefile 文件，因此他们需要内核 makefile 文件整体性的一般知识和关于 kbuild 公共接口的详细知识。
体系开发者 体系开发者关注一个整体的体系架构，比如 sparc 或者 ia64。体系开发者既需要掌握关于体系的 makefile 文件，也要熟悉内核 makefile 文件。
内核开发者 内核开发者关注内核编译系统本身。他们需要清楚内核 makefile 文件的所有方面。
本文档的读者对象是普通开发者和系统开发者。

=== 3 内核编译文件
内核中大多数 makefile 文件是使用 kbuild 基础架构的 makefile 文件。本章介绍 kbuild 的 makefile 中的语法。
3.1节“目标定义”是一个快速导引，后面各章有详细介绍和实例。

--- 3.1 目标定义
目标定义是 makefile 文件的主要部分（核心）。这些目标定义行定义了如何编译文件，特殊的兼容选项和递归子目录。
最简单的 makefile 文件只包含一行：
Example: obj-y += foo.o
这行告诉 kbuild 在该目录下名为 foo.o 的目标文件（object），foo.o 通过编译 foo.c 或者 foo.S 而得到。
如果 foo.o 编译成一个模块，则使用 obj-m 变量，因此常见写法如下：
Example: obj-$(CONFIG_FOO) += foo.o
$(CONFIG_FOO)可以代表 y(built-in 对象)或 m(module 对象)。
如果 CONFIG_FOO 不是 y 或 m，那么这个文件不会被编译和链接。

--- 3.2 内嵌对象 - obj-y
makefile 文件将为编译 vmlinux 的目标文件放在$(obj-y)列表中，这些列表依赖于内核配置。
Kbuild 编译所有的$(obj-y)文件，然后调用"$(LD) -r"合并这些文件到一个 built-in.o 文件中。built-in.o 经过父 makefile 文件链接到 vmlinux。$(obj-y)中的文件顺序很重要。列表中文件允许重复，文件第一次出现将被链接到 built-in.o，后续出现该文件将被忽略。
链接顺序之所以重要是因为一些函数在内核引导时将按照他们出现的顺序被调用，如函数(module_init() / __initcall)。所以要牢记改变链接顺序意味着也要改变 SCSI 控制器的检测顺序和重数磁盘。
例如： #drivers/isdn/i4l/Makefile
```
# 内核 ISDN 子系统和设备驱动程序 Makefile
# 每个配置项是一个文件列表
obj-$(CONFIG_ISDN)              += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
```

--- 3.3 可加载模块 - obj-m

$(obj-m)表示对象文件（object files）编译成可加载的内核模块。

一个模块可以通过一个源文件或几个源文件编译而成。makefile 只需简单地它们加到$(obj-m)。

例如：#drivers/isdn/i4l/Makefile

obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o

注意：在这个例子中$(CONFIG_ISDN_PPP_BSDCOMP)含义是'm'。

如果内核模块通过几个源文件编译而成，使用以上同样的方法。

Kbuild 需要知道通过哪些文件编译模块，因此需要设置一个$(<module_name>-objs)变量。

例如：#drivers/isdn/i4l/Makefile

obj-$(CONFIG_ISDN) += isdn.o

isdn-objs := isdn_net_lib.o isdn_v110.o isdn_common.o

在这个例子中，模块名 isdn.o. Kbuild 首先编译$(isdn-objs)中的 object 文件，然后运行"$(LD) -r"将列表中文件生成 isdn.o.

Kbuild 使用后缀-objs、-y 识别对象文件。这种方法允许 makefile 使用 CONFIG_符号值确定一个 object 文件是否是另外一个 object 的组成部分。

例如： #fs/ext2/Makefile

obj-$(CONFIG_EXT2_FS) += ext2.o

ext2-y := balloc.o bitmap.o

ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o

在这个例子中，如果$(CONFIG_EXT2_FS_XATTR)表示'y'，则 ext2.o 只有 xattr.o 组成部分。

注意： 当然，当你将对象文件编译到内核时，以上语法同样有效。因此，如果 CONFIG_EXT2_FS=y，Kbuild 将先编译 ext2.o 文件，然后链接到 built-in.o。

--- 3.4 导出符号目标

在 makefile 文件中没有特别导出符号的标记。

--- 3.5 库文件 - lib-y

obj-*中的 object 文件用于模块或 built-in.o 编译。object 文件也可能编译到库文件中--lib.a。

所有罗列在 lib-y 中的 object 文件都将编译到该目录下的一个单一的库文件中。

包含在0bj-y 中的 object 文件如果也列举在 lib-y 中将不会包含到库文件中，因为他们不能被访问。但 lib-m 中的 object 文件将被编译进 lib.a 库文件。

注意在相同的 makefile 中可以列举文件到 buit-in 内核中也可以作为库文件的一个组成部分。因此在同一个目录下既可以有 built-in.o 也可以有 lib.a 文件。

例如：#arch/i386/lib/Makefile

lib-y := checksum.o delay.o

这样将基于 checksum.o、delay.o 创建一个 lib.a 文件。

对于内核编译来说，lib.a 文件被包含在 libs-y 中。将“6.3 目录表”。

lib-y 通常被限制使用在 lib/和 arch/*/lib 目录中。

--- 3.6 目录递归

makefile 文件负责编译当前目录下的目标文件，子目录中的文件由子目录中的 makefile 文件负责编译。编译系统将使用 obj-y 和 obj-m 自动递归编译各个子目录中文件。

如果 ext2是一个子目录，fs 目录下的 makefile 将使用以下赋值语句是编译系统编译 ext2 子目录。

例如: #fs/Makefile

obj-$(CONFIG_EXT2_FS) += ext2/

如果 CONFIG_EXT2_FS 设置成'y(built-in)或'm'(modular)，则对应的 obj-变量也要设置，内核编译系统将进入 ext2目录编译文件。

内核编译系统只使用这些信息来决定是否需要编译这个目录，子目录中 makefile 文件规定那些文件编译为模块那些是内核内嵌对象。

当指定目录名时使用 CONFIG_变量是一种良好的做法。如果 CONFIG_选项不为'y'或'm'，内核编译系统就会跳过这个目录。

--- 3.7 编译标记

EXTRA_CFLAGS, EXTRA_AFLAGS, EXTRA_LDFLAGS, EXTRA_ARFLAGS

所有的 EXTRA_变量只能使用在定义该变量后的 makefile 文件中。EXTRA_变量被 makefile 文件所有的执行命令语句所使用。

$(EXTRA_CFLAGS)是使用$(CC)编译 C 文件的选项。

例如: # drivers/sound/emu10k1/Makefile

EXTRA_CFLAGS += -I$(obj)

ifdef

DEBUG EXTRA_CFLAGS += -DEMU10K1_DEBUG

endif

定义这个变量是必须的，因为顶层 makefile 定义了$(CFLAGS)变量并使用该变量编译整个代码树。

$(EXTRA_AFLAGS)是每个目录编译汇编语言源文件的选项。

例如: #arch/x86_64/kernel/Makefile

EXTRA_AFLAGS := -traditional

$(EXTRA_LDFLAGS)和$(EXTRA_ARFLAGS)用于每个目录的$(LD)和$(AR)选项。

例如: #arch/m68k/fpsp040/Makefile

EXTRA_LDFLAGS := -x

CFLAGS_$@, AFLAGS_$@

CFLAGS_$@和 AFLAGS_$@只使用到当前 makefile 文件的命令中。

$(CFLAGS_$@)定义了使用$(CC)的每个文件的选项。$@部分代表该文件。

例如: # drivers/scsi/Makefile

CFLAGS_aha152x.o = -DAHA152X_STAT -DAUTOCONF

CFLAGS_gdth.o = # -DDEBUG_GDTH=2 -D__SERIAL__ -D__COM2__ \

-DGDTH_STATISTICS CFLAGS_seagate.o = -DARBITRATE -DPARITY -DSEAGATE_USE_ASM

这三行定义了 aha152x.o、gdth.o 和 seagate.o 文件的编译选项。

$(AFLAGS_$@)使用在汇编语言代码文件中，具有同上相同的含义。

例如: # arch/arm/kernel/Makefile

AFLAGS_head-armv.o := -DTEXTADDR=$(TEXTADDR) -traditional

AFLAGS_head-armo.o := -DTEXTADDR=$(TEXTADDR) -traditional

--- 3.9 依赖关系

内核编译记录如下依赖关系：

1) 所有的前提文件(both *.c and *.h)

2) CONFIG_ 选项影响到的所有文件

3) 编译目标文件使用的命令行

因此，假如改变$(CC)的一个选项，所有相关的文件都要重新编译。

--- 3.10 特殊规则

特殊规则使用在内核编译需要规则定义而没有相应定义的时候。典型的例子如编译时头文件的产生规则。其他例子有体系 makefile 编译引导映像的特殊规则。特殊规则写法同普通的 Make 规则。

Kbuild(应该是编译程序)在 makefile 所在的目录不能被执行，因此所有的特殊规则需要提供前提文件和目标文件的相对路径。

定义特殊规则时将使用到两个变量：

$(src)： $(src)是对于 makefile 文件目录的相对路径，当使用代码树中的文件时使用该变量$(src)。

$(obj)： $(obj)是目标文件目录的相对路径。生成文件使用$(obj)变量。

例如: #drivers/scsi/Makefile

$(obj)/53c8xx_d.h: $(src)/53c7,8xx.scr $(src)/script_asm.pl

$(CPP) -DCHIP=810 - < $< | ... $(src)/script_asm.pl

这就是使用普通语法的特殊编译规则。

目标文件依赖于两个前提文件。目标文件的前缀是$(obj), 前提文件的前缀是$(src)(因为它们不是生成文件)。

=== 4 辅助程序

内核编译系统支持在编译（compliation）阶段编译主机可执行程序。为了使用主机程序需要两个步骤：第一个步骤使用 hostprogs-y 变量告诉内核编译系统有主机程序可用。第二步给主机程序添加潜在的依赖关系。有两种方法，在规则中增加依赖关系或使用$(always)变量。具体描述如下。

--- 4.1 简单辅助程序

在一些情况下需要在主机上编译和运行主机程序。下面这行告诉 kbuild 在主机上建立 bin2hex 程序。

例如: hostprogs-y := bin2hex

Kbuild 假定使用 makefile 相同目录下的单一 C 代码文件 bin2hex.c 编译 bin2hex。

--- 4.2 组合辅助程序

主机程序也可以由多个 object 文件组成。定义组合辅助程序的语法同内核对象的定义方法。

$(<executeable>-objs)包含了所有的用于链接最终可执行程序的对象。

例如: #scripts/lxdialog/Makefile

hostprogs-y := lxdialog
lxdialog-objs := checklist.o lxdialog.o

扩展名.o 文件都编译自对应的.c 文件。在上面的例子中 checklist.c 编译成 checklist.o，lxdialog.c 编译为 lxdialog.o。最后两个.o 文件链接成可执行文件 lxdialog。

注意：语法<executable>-y 不能用于定义主机程序。

--- 4.3 定义共享库

扩展名为.so 的对象是共享库文件，并且是位置无关的 object 文件。内核编译系统提供共享库使用支持，但使用方法有限制。在下面例子中 libkconfig.so 库文件被链接到可执行文件 conf 中。

例如: #scripts/kconfig/Makefile

```
hostprogs-y     := conf
conf-objs       := conf.o libkconfig.so
libkconfig-objs := expr.o type.o
```

共享库文件需要对应的-objs 定义, 在上面例子中库 libkconfig 由两个对象组成：expr.o 和 type.o。expr.o 和 type.o 将被编译为位置无关代码并被链接如 libkconfig.so。共享库不支持 C++语言。

--- 4.4 C++语言使用方法

内核编译系统提供了对 C++主机程序的支持以用于内核配置，但不主张其它方面使用这种方法。

例如: #scripts/kconfig/Makefile

```
hostprogs-y     := qconf
qconf-cxxobjs := qconf.o
```

在上面例子中可执行文件由 C++文件 qconf.cc 组成 - 通过$(qconf-cxxobjs)标识。

如果 qconf 由.c 和.cc 文件混合组成，附加行表示这种情况。

例如: #scripts/kconfig/Makefile

```
hostprogs-y     := qconf
qconf-cxxobjs := qconf.o
qconf-objs      := check.o
```

--- 4.5 辅助程序编译控制选项

当编译主机程序时仍然可以使用$(HOSTCFLAGS)设置编译选项传递给$(HOSTCC)。这些选项将影响所有使用变量 HOST_EXTRACFLAG 的 makefile 创建的主机程序。

例如: #scripts/lxdialog/Makefile

```
HOST_EXTRACFLAGS += -I/usr/include/ncurses
```

为单个文件设置选项使用下面方式：

例如: #arch/ppc64/boot/Makefile

```
HOSTCFLAGS_piggyback.o := -DKERNELBASE=$(KERNELBASE)
```

也可以使用附加链接选项：

例如: #scripts/kconfig/Makefile

```
HOSTLOADLIBES_qconf := -L$(QTDIR)/lib
```

当链接 qconf 时将使用外部选项"-L$(QTDIR)/lib"。

--- 4.6 何时建立辅助程序

只有当需要时内核编译系统才会编译主机程序。有两种方式：

(1) 在特殊规则中作为隐式的前提需求

例如: #drivers/pci/Makefile

```
hostprogs-y := gen-devlist
$(obj)/devlist.h: $(src)/pci.ids $(obj)/gen-devlist
        ( cd $(obj); ./gen-devlist ) < $<
```

编译目标文件$(obj)/devlist.h 需要先建立$(obj)/gen-devlist。注意在特殊规则中使用主机程序必须加前缀$(obj)。

(2) 使用$(always)

当没有合适的特殊规则可以使用，并且在进入 makefile 文件时就要建立主机程序，可以使用变量$(always)。

例如: #scripts/lxdialog/Makefile

```
hostprogs-y     := lxdialog
always          := $(hostprogs-y)
```

这样就告诉内核编译系统即使没有任何规则使用 lxdialog 也要编译它。

--- 4.7 使用 hostprogs-$(CONFIG_FOO)

在 Kbuild 文件中典型模式如下：

例如: #scripts/Makefile

```
hostprogs-$(CONFIG_KALLSYMS) += kallsyms
```

对 Kbuild 来说'y'用于内嵌对象'm'用于模块。

因此如果 config 符号是'm'，编译系统也将创建该程序。换句话说内核编译系统等同看待 hostprogs-m 和 hostprogs-y。但如果不涉及到 CONFIG 符号仅建议使用 hostprogs-y。

接上文“Linux 内核2.6 Makefile 文件 （一）”

=== 目录

=== 5 编译清除机制

"make clean"命令删除在编译内核生成的大部分文件，例如主机程序，列举在 $(hostprogs-y)、$(hostprogs-m)、$(always)、$(extra-y)和$(targets)中目标文件都将被删除。代码目录数中的"*.[oas]"、"*.ko"文件和一些由编译系统产生的附加文件也将被删除。

附加文件可以使用$(clean-files)进行定义。

例如: #drivers/pci/Makefile

clean-files := devlist.h classlist.h

当执行"make clean"命令时， "devlist.h classlist.h"两个文件将被删除。内核编译系统默认这些文件与 makefile 具有相同的相对路径，否则需要设置以'/'开头的绝对路径。

删除整个目录使用以下方式：

例如: #scripts/package/Makefile

clean-dirs := $(objtree)/debian/

这样就将删除包括子目录在内的整个 debian 目录。如果不使用以'/'开头的绝对路径内核编译系统见默认使用相对路径。

通常内核编译系统根据"obj-* := dir/"进入子目录，但是在体系 makefile 中需要显式使用如下方式：

例如: #arch/i386/boot/Makefile

subdir- := compressed/

上面赋值语句指示编译系统执行"make clean"命令时进入 compressed/目录。

在编译最终的引导映像文件的 makefile 中有一个可选的目标对象名称是 archclean。

例如: #arch/i386/Makefile

archclean:

(Q)(MAKE) $(clean)=arch/i386/boot

当执行"make clean"时编译器进入 arch/i386/boot 并象通常一样工作。arch/i386/boot 中的 makefile 文件可以使用 subdir-标识进入更下层的目录。

注意1: arch/$(ARCH)/Makefile 不能使用"subdir-"，因为它被包含在顶层 makefile 文件中，在这个位置编译机制是不起作用的。

注意2: 所有列举在 core-y、libs-y、drivers-y 和 net-y 中的目录将被"make clean"命令清除。

=== 6 体系 Makefile 文件

在开始进入各个目录编译之前，顶层 makefile 文件设置编译环境和做些准备工作。顶层 makefile 文件包含通用部分，arch/$(ARCH) /Makefile 包含该体系架构所需的设置。因此 arch/$(ARCH)/Makefile 会设置一些变量和少量的目标。

当编译时将按照以下大概步骤执行：

1) 配置内核 => 产生 .config 文件

2) 保存内核版本到 include/linux/version.h 文件中

3) 符号链接 include/asm to include/asm-$(ARCH)

4) 更新所有目标对象的其它前提文件

- 附加前提文件定义在 arch/$(ARCH)/Makefile 文件中

5) 递归进入 init-* core* drivers-* net-* libs-*中的所有子目录和编译所有的目标对象

- 上面变量值都引用到 arch/$(ARCH)/Makefile 文件。

6) 链接所有的 object 文件生成 vmlinux 文件，vmlinux 文件放在代码树根目录下。

最开始链接的几个 object 文件列举在 arch/$(ARCH)/Makefile 文件的 head-y 变量中。
7) 最后体系 makefile 文件定义编译后期处理规则和建立最终的引导映像 bootimage。
- 包括创建引导记录
- 准备 initrd 映像和相关处理

--- 6.1 变量设置
LDFLAGS $(LD)一般选项
选项使用于链接器的所有调用中。通常定义 emulation 就可以了。
例如: #arch/s390/Makefile
LDFLAGS := -m elf_s390
注意: EXTRA_LDFLAGS 和 LDFLAGS_$@可以进一步订制使用选项，将第7章。
LDFLAGS_MODULE $(LD)链接模块的选项
LDFLAGS_MODULE 通常设置$(LD)链接模块的.ko 选项。
默认为"-r"即可重定位输出文件。
LDFLAGS_vmlinux $(LD)链接 vmlinux 选项
LDFLAGS_vmlinux 定义链接最终 vmlinux 时链接器的选项。
LDFLAGS_vmlinux 支持使用 LDFLAGS_$@。
例如: #arch/i386/Makefile
LDFLAGS_vmlinux := -e stext
OBJCOPYFLAGS objcopy 选项
当使用$(call if_changed,objcopy)转化 a .o 文件时，OBJCOPYFLAGS 中的选项将被使用。
$(call if_changed,objcopy)经常被用作为 vmlinux 产生原始的二进制文件。
例如: #arch/s390/Makefile
OBJCOPYFLAGS := -O binary
#arch/s390/boot/Makefile
$(obj)/image: vmlinux FORCE $(call if_changed,objcopy)
在上面例子中$(obj)/image 是 vmlinux 的二进制版本文件。$(call if_changed,xxx)的使用方法见后。
AFLAGS $(AS)汇编选项
默认值见顶层 Makefile 文件
针对每个体系需要另外添加和修改它。
例如: #arch/sparc64/Makefile
AFLAGS += -m64 -mcpu=ultrasparc
CFLAGS $(CC)编译器选项
默认值见顶层 Makefile 文件
针对每个体系需要另外添加和修改它。
通常 CFLAGS 变量值取决于内核配置。
例如: #arch/i386/Makefile
cflags-$(CONFIG_M386) += -march=i386
CFLAGS += $(cflags-y)
许多体系 Makefiles 文件动态启动市场目标机器上的 C 编译器检测支持的选项：
#arch/i386/Makefile
...

```
cflags-$(CONFIG_MPENTIUMII)     += $(call cc-option,\
        -march=pentium2,-march=i686) ...
# Disable unit-at-a-time mode ...
CFLAGS += $(call cc-option,-fno-unit-at-a-time)
...
```

第一个例子当 config 选项是'y'时将被选中。

CFLAGS_KERNEL　　$(CC)编译 built-in 对象的选项

$(CFLAGS_KERNEL)包含外部 C 编译器选项编译本地内核代码。

CFLAGS_MODULE　　$(CC)编译模块选项

$(CFLAGS_MODULE)包含外部 C 编译器选项编译可加载内核代码。

--- 6.2 增加预设置项

prepare: 这个规则用于列举开始进入子目录编译前需要的前提文件。通常是些包含汇编常量的头文件。

例如:

#arch/s390/Makefile

prepare: include/asm-$(ARCH)/offsets.h

在这个例子中 include/asm-$(ARCH)/offsets.h 将在进入子目录前编译。

详见 XXX-TODO 文件描述了 kbuild 如何产生 offset 头文件。

--- 6.3 目录表

体系 makefile 文件和顶层 makefile 文件共同定义了如何建立 vmlinux 文件的变量。注意没有体系相关的模块对象定义部分：所有的模块对象都是体系无关的。

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

$(head-y) 列举首先链接到 vmlinux 的对象文件。

$(libs-y) 列举了能够找到 lib.a 文件的目录。

其余的变量列举了能够找到内嵌对象文件的目录。

$(init-y) 列举的对象位于$(head-y)对象之后。

然后是如下位置秩序：

$(core-y), $(libs-y), $(drivers-y) 和 $(net-y)。

顶层 makefile 定义了所有同用目录，arch/$(ARCH)/Makefile 文件只需增加体系相关的目录。

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 引导映像

体系 makefile 文件定义了编译 vmlinux 文件的目标对象，将它们压缩和封装成引导代码，并复制到合适的位置。这包括各种安装命令。如何定义实际的目标对象无法为所有的体系结构提供标准化的方法。

附加处理过程常位于 arch/$(ARCH)/下的 boot/目录。

内核编译系统无法在 boot/目录下提供一种便捷的方法创建目标系统文件。因此 arch/$(ARCH)/Makefile 要调用 make 命令在 boot/目录下建立目标系统文件。建议使用的方

法是在 arch/$(ARCH)/Makefile 中设置调用，并且使用完整路径引用 arch/$(ARCH)/boot/Makefile。

例如: #arch/i386/Makefile

```
boot := arch/i386/boot
bzImage: vmlinux
    $(Q)$(MAKE) $(build)=$(boot) $(boot)/$@
```

建议使用"(Q)(MAKE) $(build)=<dir>"方式在子目录中调用 make 命令。

没有定义体系目标系统文件的规则，但执行"make help"命令要列出所有目标系统文件，因此必须定义$(archhelp)变量。

例如: #arch/i386/Makefile

```
define
    archhelp echo '* bzImage          - Image (arch/$(ARCH)/boot/bzImage)'
 endef
```

当执行不带参数的 make 命令时，将首先编译第一个目标对象。在顶层 makefile 中第一个目标对象是 all:。

一个体系结构需要定义一个默认的可引导映像。

"make help"命令的默认目标是以*开头的对象。

增加新的前提文件给 all 目标可以设置不同于 vmlinux 的默认目标对象。

例如: #arch/i386/Makefile

```
all: bzImage
```

当执行不带参数的"make"命令时，bzImage 文件将被编译。

--- 6.5 编译非内核目标

extra-y

extra-y 定义了在当前目录下创建没有在 obj-*定义的附加的目标文件。

在 extra-y 中列举目标是处于两个目的：

1) 是内核编译系统在命令行中检查变动情况
   - 当使用$(call if_changed,xxx)时
2) 内核编译系统知道执行"make clean"命令时删除哪些文件

例如: #arch/i386/kernel/Makefile

```
extra-y := head.o init_task.o
```

上面例子 extra-y 中的对象文件将被编译但不会练接到 built-in.o 中。

--- 6.6 编译引导映像命令

Kbuild 提供了一些编译引导映像有用的宏。

if_changed

if_changed 是后面命令使用的基础。

用法:

```
target: source(s)
    FORCE $(call if_changed,ld/objcopy/gzip)
```

当这条规则被使用时它将检查哪些文件需要更新，或命令行被改变。后面这种情况将迫使重新编译编译选项被改变的执行文件。使用 if_changed 的目标对象必须列举在$(targets)中，否则命令行检查将失败，目标一直会编译。

赋值给$(targets)的对象没有$(obj)/前缀。

if_changed 也可以和定制命令配合使用，见6.7"kbuild 定制命令"。

注意: 一个常见错误是忘记了 FORCE 前导词。

ld

链接目标。常使用 LDFLAGS_$@作为 ld 的选项。

objcopy

复制二进制文件。常用于 arch/$(ARCH)/Makefile 中和使用 OBJCOPYFLAGS 作为选项。

也可以用 OBJCOPYFLAGS_$@设置附加选项。

gzip

压缩目标文件。使用最大压缩算法压缩目标文件。

例如: #arch/i386/boot/Makefile

```
LDFLAGS_bootsect := -Ttext 0x0 -s --oformat binary
LDFLAGS_setup    := -Ttext 0x0 -s --oformat binary -e begtext

targets += setup setup.o bootsect bootsect.o
$(obj)/setup $(obj)/bootsect: %: %.o FORCE
        $(call if_changed,ld)
```

在上面例子中有两个可能的目标对象，分别需要不同的链接选项。使用LDFLAGS_$@语法为每个目标对象设置不同的链接选项。

$(targets)包含所有的目标对象，因此内核编译系统知道所有的目标对象并且将：

1) 检查命令行的改变情况

2) 执行 make clean 命令时删除目标对象

": %: %.o"是简写方法，减写 setup.o 和 bootsect.o 文件。

注意: 常犯错误是忘记"target :="语句，导致没有明显的原因目标文件被重新编译。

--- 6.7 定制编译命令

当执行带 KBUILD_VERBOSE=0参数的编译命令时命令的简短信息会被显示。要让定制命令具有这种功能需要设置两个变量：

quiet_cmd_<command> - 将被显示的内容

cmd_<command> - 被执行的命令

例如: #

```
quiet_cmd_image = BUILD     $@
  cmd_image = $(obj)/tools/build $(BUILDFLAGS) \
              $(obj)/vmlinux.bin > $@
targets += bzImage
$(obj)/bzImage: $(obj)/vmlinux.bin $(obj)/tools/build FORCE
        $(call if_changed,image)
        @echo 'Kernel: $@ is ready'
```

执行"make KBUILD_VERBOSE=0"命令编译$(obj)/bzImage 目标时将显示：

```
BUILD     arch/i386/boot/bzImage
```

--- 6.8 预处理连接脚本

当编译 vmlinux 映像时将使用 arch/$(ARCH)/kernel/vmlinux.lds 链接脚本。

相同目录下的 vmlinux.lds.S 文件是这个脚本的预处理的变体。内核编译系统知晓.lds

文件并使用规则*lds.S -> *lds。

例如: #arch/i386/kernel/Makefile

```
always := vmlinux.lds
#Makefile
export CPPFLAGS_vmlinux.lds += -P -C -U$(ARCH)
```

$(always)赋值语句告诉编译系统编译目标是 vmlinux.lds。$(CPPFLAGS_vmlinux.lds)赋值语句告诉编译系统编译 vmlinux.lds 目标的编译选项。

编译*.lds 时将使用到下面这些变量：

CPPFLAGS ：定义在顶层 Makefile

EXTRA_CPPFLAGS ：可以设置在编译的 makefile 文件中

CPPFLAGS_$(@F) ：目标编译选项。注意要使用文件全名。

--- 6.9 $(CC)支持功能

内核可能会用不同版本的$(CC)进行编译，每个版本有不同的性能和选项，内核编译系统提供基本的支持用于验证$(CC)选项。$(CC)通常是 gcc 编译器，但其它编译器也是可以。

cc-option cc-option 用于检测$(CC)是否支持给定的选项，如果不支持就使用第二个可选项。

例如: #arch/i386/Makefile

```
cflags-y += $(call cc-option,-march=pentium-mmx,-march=i586)
```

在上面例子中如果$(CC)支持-march=pentium-mmx 则 cflags-y 等于该值，否则等于-march-i586。如果没有第二个可选项且第一项不支持则 cflags-y 没有被赋值。

cc-option-yn cc-option-yn 用于检测 gcc 是否支持给定的选项，支持返回'y'否则'n'。

例如: #arch/ppc/Makefile

```
biarch := $(call cc-option-yn, -m32)
aflags-$(biarch) += -a32
cflags-$(biarch) += -m32
```

在上面例子中如果$(CC)支持-m32选项则$(biarch)设置为 y。当$(biarch)等于 y 时，变量$(aflags-y)和$(cflags-y)将分别等于-a32和-m32。

cc-option-align gcc 版本>= 3.0用于定义 functions、loops 等边界对齐选项。

```
gcc < 3.00
    cc-option-align = -malign
 gcc >= 3.00
    cc-option-align = -falign
```

例如:

```
CFLAGS += $(cc-option-align)-functions=4
```

在上面例子中对于 gcc >= 3.00来说-falign-functions=4， gcc < 3.00版本使用-malign-functions=4。

cc-version cc-version 返回$(CC)编译器数字版本号。

版本格式是<major><minor>，均为两位数字。例如 gcc 3.41将返回0341。

当一个特定$(CC)版本在某个方面有缺陷时 cc-version 是很有用的。例如-mregparm=3在一些 gcc 版本会失败尽管 gcc 接受这个选项。

例如: #arch/i386/Makefile

```
GCC_VERSION := $(call cc-version)
cflags-y += $(shell \
if [ $(GCC_VERSION) -ge 0300 ] ; then echo "-mregparm=3"; fi ;)
```

在上面例子中-mregparm=3只使用在版本大于等于3.0的 gcc 中。

=== 7 Kbuild 变量

顶层 Makefile 文件导出下面这些变量：

VERSION, PATCHLEVEL, SUBLEVEL, EXTRAVERSION

这几个变量定义了当前内核版本号。很少体系体系 Makefiles 文件直接使用他们，常用$(KERNELRELEASE)代替。

$(VERSION)、$(PATCHLEVEL)和$(SUBLEVEL)定义了三个基本部分版本号，例如"2", "4",和"0"。这三个变量一直使用数值表示。

$(EXTRAVERSION)定义了更细的补钉号，通常是短横跟一些非数值字符串，例如"-pre4"。

KERNELRELEASE

$(KERNELRELEASE)是一个单一字符如"2.4.0-pre4",适合用于构造安装目录和显示版本字符串。一些体系文件使用它用于以上目的。

ARCH

这个变量定义了目标系统体系结构，例如"i386"、“arm"、"sparc". 一些内核编译文件测试$(ARCH)用于确定编译哪个文件。默认情况下顶层 Makefile 文件设置$(ARCH)为主机相同的系统体系。当交叉编译编译时，用户可以使用命令行改变$(ARCH)值：

```
make ARCH=m68k ...
```

INSTALL_PATH

这个变量定义了体系 Makefiles 文件安装内核映项和 System.map 文件的路径。

INSTALL_MOD_PATH, MODLIB

$(INSTALL_MOD_PATH)定义了模块安装变量$(MODLIB)的前缀。这个变量通常不在 Makefile 文件中定义，如果需要可以由用户添加。

$(MODLIB)定义了模块安装目录。

顶层 Makefile 定义 $(MODLIB) 为 $(INSTALL_MOD_PATH)/lib/modules/$(KERNELRELEASE)。用户可以使用命令行修改这个值。

=== 8 Makefile 语言

内核 Makefiles 设计目标用于运行 GNU Make 程序。Makefiles 仅使用 GNU Make 提到的特性，但使用了较多的 GNU 扩展部分。

GNU Make 程序支持基本的列表处理功能。内核 Makefiles 文件结合"if"语句使用了简单的列表建立和维护功能。

GNU Make 程序有两种赋值操作符：":="和"="。 ":="执行时立即计算右值并赋值给左值。"="类似公式定义，当每次使用左值要被使用时计算右值并赋给它。

一些情况中使用"="合适，而一些情况中使用":="才是正确选择。

=== 9 Credits

Original version made by Michael Elizabeth Chastain, <mailto:mec@shout.net> Updates by Kai Germaschewski <kai@tp1.ruhr-uni-bochum.de> Updates by Sam Ravnborg

<sam@ravnborg.org>

=== 10 TODO
- Describe how kbuild support shipped files with _shipped.
- Generating offset header files.
- Add more variables to section 7?

Linux 内核 Makefile 浅析

1． 配置系统的基本结构

Linux 内核的配置系统由三个部分组成，分别是：

1. Makefile：分布在 Linux 内核源代码中的 Makefile，定义 Linux 内核的编译规则；

2. 配置文件（config.in）：给用户提供配置选择的功能；

3.

配置工具：包括配置命令解释器（对配置脚本中使用的配置命令进行解释）和配置用户界面（提供基于字符界面、基于 Ncurses 图形界面以及基于

Xwindows 图形界面的用户配置界面，各自对应于 Make config、Make menuconfig 和 make xconfig）。

这些配置工具都是使用脚本语言，如 Tcl/TK、Perl 编写的（也包含一些用 C

编写的代码）。本文并不是对配置系统本身进行分析，而是介绍如何使用配置系统。所以，除非是配置系统的维护者，一般的内核开发者无须了解它们的原理，只需

要知道如何编写 Makefile 和配置文件就可以。所以，在本文中，我们只对 Makefile 和配置文件进行讨论。另外，凡是涉及到与具体

CPU 体系结构相关的内容，我们都以 ARM 为例，这样不仅可以将讨论的问题明确化，而且对内容本身不产生影响。

2． Makefile

2.1 Makefile 概述

Makefile 的作用是根据配置的情况，构造出需要编译的源文件列表，然后分别编译，并把

目标代码链接到一起，最终形成 Linux 内核二进制文件。
由于 Linux 内核源代码是按照树形结构组织的，所以 Makefile 也被分布在目录树中。Linux 内核中的 Makefile 以及与 Makefile 直接相关的文件有：
1. Makefile：顶层 Makefile，是整个内核配置、编译的总体控制文件。
2. .config：内核配置文件，包含由用户选择的配置选项，用来存放内核配置后的结果（如 make config）。
3. arch/*/Makefile：位于各种 CPU 体系目录下的 Makefile，如 arch/arm/Makefile，是针对特定平台的 Makefile。
4. 各个子目录下的 Makefile：比如 drivers/Makefile，负责所在子目录下源代码的管理。
5. Rules.make：规则文件，被所有的 Makefile 使用。
用
户通过 make config 配置后，产生了 .config。顶层 Makefile 读入 .config 中的配置选择。
顶层
Makefile 有两个主要的任务：产生 vmlinux 文件和内核模块（module）。为了达到此目的，
顶层 Makefile
递归的进入到内核的各个子目录中，分别调用位于这些子目录中的 Makefile。至于到底进入哪些子目录，取决于内核的配置。在顶层
Makefile 中，有一句：include arch/$(ARCH)/Makefile，包含了特定 CPU 体系结构下的 Makefile，这个 Makefile 中包含了平台相关的信息。
位于各个子目录下的 Makefile 同样也根据 .config 给出的配置信息，构造出当前配置下需要的源文件列表，并在文件的最后有 include $(TOPDIR)/Rules.make。
Rules.make 文件起着非常重要的作用，它定义了所有 Makefile 共用的编译规则。比如，如果需要将本目录下所有的 c 程序编译成汇编代码，需要在 Makefile 中有以下的编译规则：
%.s: %.c
$(CC) $(CFLAGS) -S $< -o $@
有
很多子目录下都有同样的要求，就需要在各自的 Makefile 中包含此编译规则，这会比较麻烦。而 Linux
内核中则把此类的编译规则统一放置到 Rules.make 中，并在各自的 Makefile 中包含进了 Rules.make（include
Rules.make），这样就避免了在多个 Makefile 中重复同样的规则。对于上面的例子，在 Rules.make 中对应的规则为：
%.s: %.c
$(CC) $(CFLAGS) $(EXTRA_CFLAGS) $(CFLAGS_$(*F)) $(CFLAGS_$@) -S $< -o $@
2.2 Makefile 中的变量
顶层 Makefile 定义并向环境中输出了许多变量，为各个子目录下的 Makefile 传递一些信息。有些变量，比如 SUBDIRS，不仅在顶层 Makefile 中定义并且赋初值，而且在 arch/*/Makefile 还作了扩充。
常用的变量有以下几类：
1） 版本信息
版
本信息有：VERSION，PATCHLEVEL, SUBLEVEL,
EXTRAVERSION，KERNELRELEASE。版本信息定义了当前内核的版本，比如
VERSION=2，PATCHLEVEL=4，SUBLEVEL=18，EXATAVERSION=-rmk7，它们共同构成

内核的发行版本
KERNELRELEASE：2.4.18-rmk7
2） CPU 体系结构：ARCH
在顶层 Makefile 的开头，用 ARCH 定义目标 CPU 的体系结构，比如 ARCH:=arm 等。许多子目录的 Makefile 中，要根据 ARCH 的定义选择编译源文件的列表。
3） 路径信息：TOPDIR, SUBDIRS
TOPDIR 定义了 Linux 内核源代码所在的根目录。例如，各个子目录下的 Makefile 通过 $(TOPDIR)/Rules.make 就可以找到 Rules.make 的位置。
SUBDIRS
定义了一个目录列表，在编译内核或模块时，顶层 Makefile 就是根据 SUBDIRS 来决定进入哪些子目录。SUBDIRS
的值取决于内核的配置，在顶层 Makefile 中 SUBDIRS 赋值为 kernel drivers mm fs net ipc lib；根据内核的配置情况，在 arch/*/Makefile 中扩充了 SUBDIRS 的值，参见4）中的例子。
4） 内核组成信息：HEAD, CORE_FILES, NETWORKS, DRIVERS, LIBS
Linux 内核文件 vmlinux 是由以下规则产生的：
vmlinux: $(CONFIGURATION) init/main.o init/version.o linuxsubdirs
$(LD) $(LINKFLAGS) $(HEAD) init/main.o init/version.o
--start-group
$(CORE_FILES)
$(DRIVERS)
$(NETWORKS)
$(LIBS)
--end-group
-o vmlinux
可
以看出，vmlinux 是由 HEAD、main.o、version.o、CORE_FILES、DRIVERS、NETWORKS 和 LIBS
组成的。这些变量（如 HEAD）都是用来定义连接生成 vmlinux 的目标文件和库文件列表。其中，HEAD 在 arch/*/Makefile
中定义，用来确定被最先链接进 vmlinux 的文件列表。比如，对于 ARM 系列的 CPU，HEAD 定义为：
HEAD := arch/arm/kernel/head-$(PROCESSOR).o
arch/arm/kernel/init_task.o
表
明 head-$(PROCESSOR).o 和 init_task.o 需要最先被链接到 vmlinux 中。PROCESSOR 为 armv
或 armo，取决于目标 CPU。 CORE_FILES，NETWORK，DRIVERS 和 LIBS 在顶层 Makefile
中定义，并且由 arch/*/Makefile 根据需要进行扩充。 CORE_FILES 对应着内核的核心文件，有
kernel/kernel.o，mm/mm.o，fs/fs.o，ipc/ipc.o，可以看出，这些是组成内核最为重要的文件。同时，
arch/arm/Makefile 对 CORE_FILES 进行了扩充：

```
# arch/arm/Makefile
# If we have a machine-specific directory, then include it in the build.
MACHDIR := arch/arm/mach-$(MACHINE)
ifeq ($(MACHDIR),$(wildcard $(MACHDIR)))
SUBDIRS += $(MACHDIR)
CORE_FILES := $(MACHDIR)/$(MACHINE).o $(CORE_FILES)
endif
HEAD := arch/arm/kernel/head-$(PROCESSOR).o
arch/arm/kernel/init_task.o
SUBDIRS += arch/arm/kernel arch/arm/mm arch/arm/lib arch/arm/nwfpe
CORE_FILES := arch/arm/kernel/kernel.o arch/arm/mm/mm.o $(CORE_FILES)
LIBS := arch/arm/lib/lib.a $(LIBS)
```

5） 编译信息：CPP, CC, AS, LD, AR，CFLAGS，LINKFLAGS

在 Rules.make 中定义的是编译的通用规则，具体到特定的场合，需要明确给出编译环境，编译环境就是在以上的变量中定义的。针对交叉编译的要求，定义了 CROSS_COMPILE。比如：

```
CROSS_COMPILE = arm-linux-
CC = $(CROSS_COMPILE)gcc
LD = $(CROSS_COMPILE)ld
.....
```

CROSS_COMPILE

定义了交叉编译器前缀 arm-linux-，表明所有的交叉编译工具都是以 arm-linux- 开头的，所以在各个交叉编译器工具之前，都加入了

$(CROSS_COMPILE)，以组成一个完整的交叉编译工具文件名，比如 arm-linux-gcc。

CFLAGS 定义了传递给 C 编译器的参数。

LINKFLAGS 是链接生成 vmlinux 时，由链接器使用的参数。LINKFLAGS 在 arm/*/Makefile 中定义，比如：

```
# arch/arm/Makefile
LINKFLAGS :=-p -X -T arch/arm/vmlinux.lds
```

6） 配置变量 CONFIG_*

.config 文件中有许多的配置变量等式，用来说明用户配置的结果。例如 CONFIG_MODULES=y 表明用户选择了 Linux 内核的模块功能。

.config

被顶层 Makefile 包含后，就形成许多的配置变量，每个配置变量具有确定的值：y 表示本编译选项对应的内核代码被静态编译进 Linux

内核；m 表示本编译选项对应的内核代码被编译成模块；n 表示不选择此编译选项；如果根本就没有选择，那么配置变量的值为空。

2.3 Rules.make 变量

前面讲过，Rules.make 是编译规则文件，所有的 Makefile 中都会包括 Rules.make。Rules.make 文件定义了许多变量，最为重要是那些编译、链接列表变量。

O_OBJS，L_OBJS，OX_OBJS，LX_OBJS：本目录下需要编译进 Linux 内核 vmlinux 的目标文件列表，其中 OX_OBJS 和 LX_OBJS 中的 "X" 表明目标文件使用了 EXPORT_SYMBOL 输出符号。

M_OBJS，MX_OBJS：本目录下需要被编译成可装载模块的目标文件列表。同样，MX_OBJS

中的 "X" 表明目标文件使用了 EXPORT_SYMBOL 输出符号。
O_TARGET，
L_TARGET：每个子目录下都有一个 O_TARGET 或 L_TARGET，Rules.make 首先从源代码编译生成 O_OBJS 和
OX_OBJS 中所有的目标文件，然后使用 $(LD) -r 把它们链接成一个 O_TARGET 或 L_TARGET。O_TARGET 以
.o 结尾，而 L_TARGET 以 .a 结尾。
2.4 子目录 Makefile
子目录 Makefile 用来控制本级目录以下源代码的编译规则。我们通过一个例子来讲解子目录 Makefile 的组成：

```
#
# Makefile for the linux kernel.
#
# All of the (potential) objects that export symbols.
# This list comes from 'grep -l EXPORT_SYMBOL *.[hc]'.
export-objs := tc.o
# Object file lists.
obj-y :=
obj-m :=
obj-n :=
obj- :=
obj-$(CONFIG_TC) += tc.o
obj-$(CONFIG_ZS) += zs.o
obj-$(CONFIG_VT) += lk201.o lk201-map.o lk201-remap.o
# Files that are both resident and modular: remove from modular.
obj-m := $(filter-out $(obj-y), $(obj-m))
# Translate to Rules.make lists.
L_TARGET := tc.a
L_OBJS := $(sort $(filter-out $(export-objs), $(obj-y)))
LX_OBJS := $(sort $(filter $(export-objs), $(obj-y)))
M_OBJS := $(sort $(filter-out $(export-objs), $(obj-m)))
MX_OBJS := $(sort $(filter $(export-objs), $(obj-m)))
include $(TOPDIR)/Rules.make
```

a) 注释
对 Makefile 的说明和解释，由#开始。
b) 编译目标定义
类
似于 obj-$(CONFIG_TC) += tc.o 的语句是用来定义编译的目标，是子目录 Makefile
中最重要的部分。编译目标定义那些在本子目录下，需要编译到 Linux
内核中的目标文件列表。为了只在用户选择了此功能后才编译，所有的目标定义都融合了对配置变量的判断。
前面说过，每个配置变量取值范围是：
y，n，m 和空，obj-$(CONFIG_TC) 分别对应着 obj-y，obj-n，obj-m，obj-。如果 CONFIG_TC 配置为

y，那么 tc.o 就进入了 obj-y 列表。obj-y 为包含到 Linux 内核 vmlinux 中的目标文件列表；obj-m
为编译成模块的目标文件列表；obj-n 和 obj- 中的文件列表被忽略。配置系统就根据这些列表的属性进行编译和链接。
export-objs 中的目标文件都使用了 EXPORT_SYMBOL() 定义了公共的符号，以便可装载模块使用。在 tc.c 文件的最后部分，有 "EXPORT_SYMBOL(search_tc_card);"，表明 tc.o 有符号输出。
这
里需要指出的是，对于编译目标的定义，存在着两种格式，分别是老式定义和新式定义。老式定义就是前面 Rules.make
使用的那些变量，新式定义就是 obj-y，obj-m，obj-n 和 obj-。Linux 内核推荐使用新式定义，不过由于
Rules.make 不理解新式定义，需要在 Makefile 中的适配段将其转换成老式定义。
c) 适配段
适配段的作用是将新式定义转换成老式定义。在上面的例子中，适配段就是将 obj-y 和 obj-m 转换成 Rules.make 能够理解的 L_TARGET，L_OBJS，LX_OBJS，M_OBJS，MX_OBJS。
L_OBJS
:= $(sort $(filter-out $(export-objs), $(obj-y))) 定义了 L_OBJS 的生成方式：在
obj-y 的列表中过滤掉 export-objs（tc.o），然后排序并去除重复的文件名。这里使用到了 GNU Make
的一些特殊功能，具体的含义可参考 Make 的文档（info make）。
d) include $(TOPDIR)/Rules.make
3． 配置文件
3.1 配置功能概述
除了 Makefile 的编写，另外一个重要的工作就是把新功能加入到 Linux 的配置选项中，提供此项功能的说明，让用户有机会选择此项功能。所有的这些都需要在 config.in 文件中用配置语言来编写配置脚本，
在 Linux 内核中，配置命令有多种方式：
配置命令 解释脚本
Make config, make oldconfig scripts/Configure
Make menuconfig scripts/Menuconfig
Make xconfig scripts/tkparse
以
字符界面配置（make config）为例，顶层 Makefile 调用 scripts/Configure， 按照
arch/arm/config.in 来进行配置。命令执行完后产生文件 .config，其中保存着配置信息。下一次再做 make config
将产生新的 .config 文件，原 .config 被改名为 .config.old
3.2 配置语言
1） 顶层菜单
mainmenu_name /prompt/ /prompt/ 是用'或"包围的字符串，'与"的区别是'…'中可使用$引用变量的值。mainmenu_name 设置最高层菜单的名字，它只在 make xconfig 时才会显示。
2） 询问语句
bool /prompt/ /symbol/
hex /prompt/ /symbol/ /word/

int /prompt/ /symbol/ /word/
string /prompt/ /symbol/ /word/
tristate /prompt/ /symbol/
询
问语句首先显示一串提示符 /prompt/，等待用户输入，并把输入的结果赋给 /symbol/
所代表的配置变量。不同的询问语句的区别在于它们接受的输入数据类型不同，比如 bool
接受布尔类型（ y 或 n ），hex 接受 16
进制数据。有些询问语句还有第三个参数 /word/，用来给出缺省值。
3） 定义语句
define_bool /symbol/ /word/
define_hex /symbol/ /word/
define_int /symbol/ /word/
define_string /symbol/ /word/
define_tristate /symbol/ /word/
不同于询问语句等待用户输入，定义语句显式的给配置变量 /symbol/ 赋值 /word/。
4） 依赖语句
dep_bool /prompt/ /symbol/ /dep/ ...
dep_mbool /prompt/ /symbol/ /dep/ ...
dep_hex /prompt/ /symbol/ /word/ /dep/ ...
dep_int /prompt/ /symbol/ /word/ /dep/ ...
dep_string /prompt/ /symbol/ /word/ /dep/ ...
dep_tristate /prompt/ /symbol/ /dep/ ...
与
询问语句类似，依赖语句也是定义新的配置变量。不同的是，配置变量/symbol/的取值范围
将依赖于配置变量列表/dep/
…。这就意味着：被定义的配置变量所对应功能的取舍取决于依赖列表所对应功能的选择。
以 dep_bool 为例，如果/dep/
…列表的所有配置变量都取值 y，则显示/prompt/，用户可输入任意的值给配置变量/symbol/，
但是只要有一个配置变量的取值为 n，则
/symbol/被强制成 n。
不同依赖语句的区别在于它们由依赖条件所产生的取值范围不同。
5） 选择语句
choice /prompt/ /word/ /word/
choice 语句首先给出一串选择列表，供用户选择其中一种。比如 Linux for ARM 支持多种
基于 ARM core 的 CPU，Linux 使用 choice 语句提供一个 CPU 列表，供用户选择：
choice 'ARM system type'
"Anakin CONFIG_ARCH_ANAKIN
Archimedes/A5000 CONFIG_ARCH_ARCA5K
Cirrus-CL-PS7500FE CONFIG_ARCH_CLPS7500
……
SA1100-based CONFIG_ARCH_SA1100
Shark CONFIG_ARCH_SHARK" RiscPC
Choice 首先显示 /prompt/，然后将 /word/ 分解成前后两个部分，前部分为对应选择的提示
符，后部分是对应选择的配置变量。用户选择的配置变量为 y，其余的都为 n。

6） if 语句

if [/expr/] ; then

/statement/

...

fi

if [/expr/] ; then

/statement/

...

else

/statement/

...

fi

if 语句对配置变量（或配置变量的组合）进行判断，并作出不同的处理。判断条件 /expr/ 可以是单个配置变量或字符串，也可以是带操作符的表达式。操作符有：=，!=，-o，-a 等。

7） 菜单块（menu block）语句

mainmenu_option next_comment

comment '…..'

…

endmenu

引入新的菜单。在向内核增加新的功能后，需要相应的增加新的菜单，并在新菜单下给出此项功能的配置选项。Comment 后带的注释就是新菜单的名称。所有归属于此菜单的配置选项语句都写在 comment 和 endmenu 之间。

8） Source 语句

source /word/

/word/ 是文件名，source 的作用是调入新的文件。

3.3 缺省配置

Linux

内核支持非常多的硬件平台，对于具体的硬件平台而言，有些配置就是必需的，有些配置就不是必需的。另外，新增加功能的正常运行往往也需要一定的先决条件，

针对新功能，必须作相应的配置。因此，特定硬件平台能够正常运行对应着一个最小的基本配置，这就是缺省配置。

Linux 内核中针对每个 ARCH 都会有一个缺省配置。在向内核代码增加了新的功能后，如果新功能对于这个 ARCH 是必需的，就要修改此 ARCH 的缺省配置。修改方法如下（在 Linux 内核根目录下）：

1. 备份 .config 文件

2. cp arch/arm/deconfig .config

3. 修改 .config

4. cp .config arch/arm/deconfig

5. 恢复 .config

如果新增的功能适用于许多的 ARCH，只要针对具体的 ARCH，重复上面的步骤就可以了。

3.4 help file

大家都有这样的经验，在配置 Linux 内核时，遇到不懂含义的配置选项，可以查看它的帮助，从中可得到选择的建议。下面我们就看看如何给给一个配置选项增加帮助信息。

所有配置选项的帮助信息都在 Documentation/Configure.help 中，它的格式为：

给出本配置选项的名称， 对应配置变量， 对应配置帮助信息。在帮助信息中，首先简单描述此功能，其次说明选择了此功能后会有什么效果，不选择又有什么效果，最后，不要忘了写上"如果不清楚，选择 N（或者）Y"，给不知所措的用户以提示。

4． 实例

对

于一个开发者来说，将自己开发的内核代码加入到 Linux

内核中，需要有三个步骤。首先确定把自己开发代码放入到内核的位置；其次，把自己开发的功能增加到 Linux

内核的配置选项中，使用户能够选择此功能；最后，构建子目录 Makefile，根据用户的选择，将相应的代码编译到最终生成的 Linux

内核中去。下面，我们就通过一个简单的例子--test driver，结合前面学到的知识，来说明如何向 Linux 内核中增加新的功能。

4.1 目录结构

test driver 放置在 drivers/test/ 目录下：$cd drivers/test

$tree

```
.
|-- Config.in
|-- Makefile
|-- cpu
     |
| |-- Makefile
     |
| `-- cpu.c
|-- test.c
|-- test_client.c
|-- test_ioctl.c
|-- test_proc.c
|-- test_queue.c
`-- test
     |
     |-- Makefile
     |
     `-- test.c
```

4.2 配置文件

1） drivers/test/Config.in#

```
# TEST driver configuration
#
mainmenu_option next_comment
comment 'TEST Driver'
bool 'TEST support' CONFIG_TEST
if [ "$CONFIG_TEST" = "y" ]; then
     tristate 'TEST user-space interface' CONFIG_TEST_USER
     bool 'TEST CPU ' CONFIG_TEST_CPU
```

fi
endmenu
由
于 test driver 对于内核来说是新的功能，所以首先创建一个菜单 TEST Driver。然后，显示 "TEST
support"，等待用户选择；接下来判断用户是否选择了 TEST
Driver，如果是（CONFIG_TEST=y），则进一步显示子功能：用户接口与 CPU
功能支持；由于用户接口功能可以被编译成内核模块，所以这里的询问语句使用了 tristate（因为 tristate 的取值范围包括 y、n 和
m，m 就是对应着模块）。
2） arch/arm/config.in
在文件的最后加入：source drivers/test/Config.in，将 TEST Driver 子功能的配置纳入到 Linux 内核的配置中。
4.3 Makefile
1）drivers/test/Makefile

```
    #       drivers/test/Makefile
#
#       Makefile for the TEST.
#
SUB_DIRS        :=
MOD_SUB_DIRS := $(SUB_DIRS)
ALL_SUB_DIRS := $(SUB_DIRS) cpu
L_TARGET := test.a
export-objs := test.o test_client.o
obj-$(CONFIG_TEST)                  += test.o test_queue.o test_client.o
obj-$(CONFIG_TEST_USER)          += test_ioctl.o
obj-$(CONFIG_PROC_FS)              += test_proc.o
subdir-$(CONFIG_TEST_CPU)      += cpu
include $(TOPDIR)/Rules.make
clean:
        for dir in $(ALL_SUB_DIRS); do make -C $$dir clean; done
        rm -f *.[oa] .*.flags
```

drivers/test
目录下最终生成的目标文件是 test.a。在 test.c 和 test-client.c 中使用了 EXPORT_SYMBOL
输出符号，所以 test.o 和 test-client.o 位于 export-objs
列表中。然后，根据用户的选择（具体来说，就是配置变量的取值），构建各自对应的 obj-* 列表。由于 TEST Driver 中包一个子目录
cpu，当 CONFIG_TEST_CPU=y（即用户选择了此功能）时，需要将 cpu 目录加入到 subdir-y 列表中。
2）drivers/test/cpu/Makefile

```
    #       drivers/test/test/Makefile
#
```

```
#       Makefile for the TEST CPU
#
SUB_DIRS        :=
MOD_SUB_DIRS := $(SUB_DIRS)
ALL_SUB_DIRS := $(SUB_DIRS)
L_TARGET := test_cpu.a
obj-$(CONFIG_test_CPU)       += cpu.o
include $(TOPDIR)/Rules.make
clean:
        rm -f *.[oa] .*.flags
```

3）drivers/Makefile……

```
subdir-$(CONFIG_TEST) += test
……
include $(TOPDIR)/Rules.make
```

在 drivers/Makefile 中加入 subdir-$(CONFIG_TEST)+= test，使得在用户选择 TEST Driver 功能后，内核编译时能够进入 test 目录。

4）Makefile……

```
DRIVERS-$(CONFIG_PLD) += drivers/pld/pld.o
DRIVERS-$(CONFIG_TEST) += drivers/test/test.a
DRIVERS-$(CONFIG_TEST_CPU) += drivers/test/cpu/test_cpu.a
DRIVERS := $(DRIVERS-y)
……
```

在
顶层 Makefile 中加入 DRIVERS-$(CONFIG_TEST) += drivers/test/test.a 和
DRIVERS-$(CONFIG_TEST_CPU) += drivers/test/cpu/test_cpu.a。如何用户选择了 TEST Driver，那么 CONFIG_TEST 和 CONFIG_TEST_CPU 都是 y，test.a 和 test_cpu.a 就都位于
DRIVERS-y 列表中，然后又被放置在 DRIVERS 列表中。在前面曾经提到过，Linux 内核文件 vmlinux 的组成中包括
DRIVERS，所以 test.a 和 test_cpu.a 最终可被链接到 vmlinux 中。