```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 20
EXTRAVERSION = .7
NAME = Homicidal Dwarf Hamster

# 以上表明了内核版本。组合起来就是：2.6.20.7 ，yeah，这就是我分析的内核版本

# 注意写 makefile 时不要使用 makefile 的内建的规则和变量

#要想不打印"Entering directory ..."字样，请使用 no-print-directory 选项
MAKEFLAGS += -rR --no-print-directory

# 因为需要递归执行 build,  所以必须注意要保证按照正确顺序执行 make.

# 使用 'make V=1' 可以看到完整命令

ifdef V
  ifeq ("$(origin V)", "command line")
    KBUILD_VERBOSE = $(V)
  endif
endif
ifndef KBUILD_VERBOSE
  KBUILD_VERBOSE = 0
endif

# 使用 'make C=1' 仅检查需要重新使用 c 编译器编译的文件
# 使用 'make C=2' 检查所有 c 编译器编译的文件

ifdef C
  ifeq ("$(origin C)", "command line")
    KBUILD_CHECKSRC = $(C)
  endif
endif
ifndef KBUILD_CHECKSRC
  KBUILD_CHECKSRC = 0
endif

# 使用 make M=dir 表明需要编译的模块目录
# 旧的语法 make ... SUBDIRS=$PWD 依然支持
# 这里通过环境变量 KBUILD_EXTMOD 来表示
ifdef SUBDIRS
  KBUILD_EXTMOD ?= $(SUBDIRS)
endif
```

```
ifdef M
  ifeq ("$(origin M)", "command line")
    KBUILD_EXTMOD := $(M)
  endif
endif


# kbuild 可以把输出文件放到一个分开的目录下
# 定位输出文件目录有两种语法支持：
# 两种方法都要求工作目录是内核源文件目录的根目录
# 1) O=
# 使用 "make O=dir/to/store/output/files/"
#
# 2) 设置 KBUILD_OUTPUT
# 设置环境变量 KBUILD_OUTPUT 来指定输出文件存放的目录
# export KBUILD_OUTPUT=dir/to/store/output/files/
# make
#
# 但是 O= 方式优先于 KBUILD_OUTPUT 环境变量方式

# KBUILD_SRC 会再 obj 目录的 make 调用
# KBUILD_SRC 到目前还不是为了给一般用户使用的
ifeq ($(KBUILD_SRC),)

ifdef O
  ifeq ("$(origin O)", "command line")
    KBUILD_OUTPUT := $(O)
  endif
endif

# 缺剩目标
PHONY := _all
_all:

ifneq ($(KBUILD_OUTPUT),)
# 调用输出目录的第二个 make, 传递相关变量检查输出目录确实存在

saved-output := $(KBUILD_OUTPUT)
KBUILD_OUTPUT := $(shell cd $(KBUILD_OUTPUT) && /bin/pwd)
$(if $(KBUILD_OUTPUT),, \
    $(error output directory "$(saved-output)" does not exist))

PHONY += $(MAKECMDGOALS)
```

```
$(filter-out _all,$(MAKECMDGOALS)) _all:
    $(if $(KBUILD_VERBOSE:1=),@)$(MAKE) -C $(KBUILD_OUTPUT) \
    KBUILD_SRC=$(CURDIR) \
    KBUILD_EXTMOD="$(KBUILD_EXTMOD)" -f $(CURDIR)/Makefile $@

# Leave processing to above invocation of make
skip-makefile := 1
endif # ifneq ($(KBUILD_OUTPUT),)
endif # ifeq ($(KBUILD_SRC),)

ifeq ($(skip-makefile),)

PHONY += all
ifeq ($(KBUILD_EXTMOD),)
_all: all
else
_all: modules
endif

srctree        := $(if $(KBUILD_SRC),$(KBUILD_SRC),$(CURDIR))
TOPDIR         := $(srctree)
# FIXME - TOPDIR is obsolete, use srctree/objtree
objtree        := $(CURDIR)
src        := $(srctree)
obj        := $(objtree)

VPATH        := $(srctree)$(if $(KBUILD_EXTMOD),:$(KBUILD_EXTMOD))

export srctree objtree VPATH TOPDIR


# SUBARCH 告诉 usermode build 当前的机器是什么体系.  它是第一个设置的，并且如果
# 是usermode build ， 命令行中的 "ARCH=um" 优先下面的 ARCH 设置. 如果是 native
build ，
# 设置ARCH ， 获取正常的数值, 将会忽略SUBARCH .

SUBARCH := $(shell uname -m | sed -e s/i.86/i386/ -e s/sun4u/sparc64/ \
                -e s/arm.*/arm/ -e s/sa110/arm/ \
                -e s/s390x/s390/ -e s/parisc64/parisc/ \
                -e s/ppc.*/powerpc/ -e s/mips.*/mips/ )

# 交叉编译和选择不同的gcc/bin-utils
# ---------------------------------------------------------------------------
#
```

```
# 当交叉编译其他体系的内核时，ARCH 应该设置为目标体系.
# ARCH 可以再 make 执行间设置:
# make ARCH=ia64
# 另外一种方法是设置 ARCH 环境变量.
# 默认 ARCH 是当前执行的宿主机.

# CROSS_COMPILE 作为编译时所有执行需要使用的前缀
# 只有 gcc 和 相关的 bin-utils 使用 $(CROSS_COMPILE)设置的前缀.
# CROSS_COMPILE 可以再命令行设置：
# make CROSS_COMPILE=ia64-linux-
# 另外 CROSS_COMPILE 也可以再环境变量中设置.
# CROSS_COMPILE 的默认值是空的
# Note: 一些体系的 CROSS_COMPILE 是再其 arch/*/Makefile 中设置的

ARCH        ?= $(SUBARCH)
CROSS_COMPILE    ?=

# Architecture as present in compile.h
UTS_MACHINE := $(ARCH)

KCONFIG_CONFIG    ?= .config

# SHELL used by kbuild
CONFIG_SHELL := $(shell if [ -x "$$BASH" ]; then echo $$BASH; \
     else if [ -x /bin/bash ]; then echo /bin/bash; \
     else echo sh; fi ; fi)

HOSTCC      = gcc
HOSTCXX      = g++
HOSTCFLAGS  = -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer
HOSTCXXFLAGS = -O2

# Decide whether to build built-in, modular, or both.
# Normally, just do built-in.

KBUILD_MODULES :=
KBUILD_BUILTIN := 1

#    如果仅编译模块 "make modules", 就不会编译内建的 objects.
#    当使用 modversions 编译 modules 时 , 就需要考虑 build-in objects,
#    目的是再记录前确认 checksums 已经更新.

ifeq ($(MAKECMDGOALS),modules)
  KBUILD_BUILTIN := $(if $(CONFIG_MODVERSIONS),1)
```

```
endif

#    If we have "make <whatever> modules", compile modules
#    in addition to whatever we do anyway.
#    Just "make" or "make all" shall build modules as well

ifneq ($(filter all _all modules,$(MAKECMDGOALS)),)
  KBUILD_MODULES := 1
endif

ifeq ($(MAKECMDGOALS),)
  KBUILD_MODULES := 1
endif

export KBUILD_MODULES KBUILD_BUILTIN
export KBUILD_CHECKSRC KBUILD_SRC KBUILD_EXTMOD

# Beautify output
# ---------------------------------------------------------------------------
#
# 一般，make 再执行命令前会打印整个命令信息. 现在通过$($(quiet)$(cmd))方式
# 可以设置 $(quiet) 来选择不同的命令输出方式等.
#
#        quiet_cmd_cc_o_c = Compiling $(RELDIR)/$@
#        cmd_cc_o_c       = $(CC) $(c_flags) -c -o $@ $<
#
# 如果 $(quiet) 为空, 整个命令行将会打印.
# 如果 $(quiet)设置为 "quiet_", 只打印简短的版本信息.
# 如果 $(quiet)设置为 "silent_", 将不会打印任何信息, 因为没有
$(silent_cmd_cc_o_c) 变量存在.
#
# 一个简单的前缀 $(Q)放到命令前面，以便再 non-verbose 模式下可以隐藏命令：
#
#    $(Q)ln $@ :<
#
# 如果 KBUILD_VERBOSE 等于 0 ，那么上面的命令将隐藏.
# 如果 KBUILD_VERBOSE 等于 1 ，那么上面的命令将显示.

ifeq ($(KBUILD_VERBOSE),1)
  quiet =
  Q =
else
  quiet=quiet_
  Q = @
```

```
endif

# 如果make -s (silent mode), 不会显示命令

ifneq ($(findstring s,$(MAKEFLAGS)),)
  quiet=silent_
endif

export quiet Q KBUILD_VERBOSE


# Look for make include files relative to root of kernel src
MAKEFLAGS += --include-dir=$(srctree)

# We need some generic definitions.
include $(srctree)/scripts/Kbuild.include

# Make variables (CC, etc...)

AS        = $(CROSS_COMPILE)as
LD        = $(CROSS_COMPILE)ld
CC        = $(CROSS_COMPILE)gcc
CPP        = $(CC) -E
AR        = $(CROSS_COMPILE)ar
NM        = $(CROSS_COMPILE)nm
STRIP        = $(CROSS_COMPILE)strip
OBJCOPY        = $(CROSS_COMPILE)objcopy
OBJDUMP        = $(CROSS_COMPILE)objdump
AWK        = awk
GENKSYMS    = scripts/genksyms/genksyms
DEPMOD        = /sbin/depmod
KALLSYMS    = scripts/kallsyms
PERL        = perl
CHECK        = sparse

CHECKFLAGS    := -D__linux__ -Dlinux -D__STDC__ -Dunix -D__unix__ -Wbitwise $(CF)

MODFLAGS    = -DMODULE
CFLAGS_MODULE  = $(MODFLAGS)
AFLAGS_MODULE  = $(MODFLAGS)
LDFLAGS_MODULE  = -r
CFLAGS_KERNEL    =
AFLAGS_KERNEL    =
```

```
# Use LINUXINCLUDE when you must reference the include/ directory.
# Needed to be compatible with the O= option
LINUXINCLUDE    := -Iinclude \
                   $(if $(KBUILD_SRC),-Iinclude2 -I$(srctree)/include) \
           -include include/linux/autoconf.h

CPPFLAGS        := -D__KERNEL__ $(LINUXINCLUDE)

CFLAGS          := -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs \
                   -fno-strict-aliasing -fno-common
AFLAGS          := -D__ASSEMBLY__

# Read KERNELRELEASE from include/config/kernel.release (if it exists)
KERNELRELEASE = $(shell cat include/config/kernel.release 2> /dev/null)
KERNELVERSION = $(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)

export VERSION PATCHLEVEL SUBLEVEL KERNELRELEASE KERNELVERSION
export ARCH CONFIG_SHELL HOSTCC HOSTCFLAGS CROSS_COMPILE AS LD CC
export CPP AR NM STRIP OBJCOPY OBJDUMP MAKE AWK GENKSYMS PERL UTS_MACHINE
export HOSTCXX HOSTCXXFLAGS LDFLAGS_MODULE CHECK CHECKFLAGS

export CPPFLAGS NOSTDINC_FLAGS LINUXINCLUDE OBJCOPYFLAGS LDFLAGS
export CFLAGS CFLAGS_KERNEL CFLAGS_MODULE
export AFLAGS AFLAGS_KERNEL AFLAGS_MODULE

# When compiling out-of-tree modules, put MODVERDIR in the module
# tree rather than in the kernel tree. The kernel tree might
# even be read-only.
export MODVERDIR := $(if $(KBUILD_EXTMOD),$(firstword
$(KBUILD_EXTMOD))/).tmp_versions

# Files to ignore in find ... statements

RCS_FIND_IGNORE := \( -name SCCS -o -name BitKeeper -o -name .svn -o -name CVS -o
-name .pc -o -name .hg -o -name .git \) -prune -o
export RCS_TAR_IGNORE := --exclude SCCS --exclude BitKeeper --exclude .svn
--exclude CVS --exclude .pc --exclude .hg --exclude .git
# ===========================================================================
# 下面是设置 config 和 build 内核时共同使用的规则

# scripts 里面是最基本的帮助 builds,在 scripts/basic 目录下的 makefile 是所有 build
时都会用到的工具：fixdep/
PHONY += scripts_basic
```

```
scripts_basic:
    $(Q)$(MAKE) $(build)=scripts/basic

# 把对 scripts/basic/makefile 转化到 scripts_basic 处理
scripts/basic/%: scripts_basic ;

#这里先说说 script 目录里面的 makefile 文件作用
#makefile：配置目标，包括编译岛内核和模块方式的 target
#makefile_clean:当然是删除了
#kbuild_include:一般的配置选项，会在 makefile_build 中调用
#makefile_build:build 配置
#makefile_lib:module，vmlinux 的配置
#makefile_host:配置 binary
#makefile_modinst：安装 module
#makefile_modpost:

PHONY += outputmakefile
# outputmakefile 规则目的是在输出目录产生一个 makefile 文件。这会为 make 提供方便。
这里是调用 scripts/mkmakefile 创建 makfile 文件
outputmakefile:
ifneq ($(KBUILD_SRC),)
    $(Q)$(CONFIG_SHELL) $(srctree)/scripts/mkmakefile \
       $(srctree) $(objtree) $(VERSION) $(PATCHLEVEL)
endif

# 设置是.config，还是 module，还是 build 命令

no-dot-config-targets := clean mrproper distclean \
             cscope TAGS tags help %docs check% \
             include/linux/version.h headers_% \
             kernelrelease kernelversion

config-targets := 0
mixed-targets  := 0
dot-config     := 1

ifneq ($(filter $(no-dot-config-targets), $(MAKECMDGOALS)),)
    ifeq ($(filter-out $(no-dot-config-targets), $(MAKECMDGOALS)),)
        dot-config := 0
    endif
endif

ifeq ($(KBUILD_EXTMOD),)
        ifneq ($(filter config %config,$(MAKECMDGOALS)),)
```

```
                config-targets := 1
                ifneq ($(filter-out config %config,$(MAKECMDGOALS)),)
                        mixed-targets := 1
                endif
        endif
endif

ifeq ($(mixed-targets),1)
# ==========================================================================
# We're called with mixed targets (*config and build targets).
# Handle them one by one.

%:: FORCE
    $(Q)$(MAKE) -C $(srctree) KBUILD_SRC= $@

else
ifeq ($(config-targets),1)
# ==========================================================================
#仅配置 linux

# Read arch specific Makefile to set KBUILD_DEFCONFIG as needed.
# KBUILD_DEFCONFIG may point out an alternative default configuration
# used for 'make defconfig'
include $(srctree)/arch/$(ARCH)/Makefile
export KBUILD_DEFCONFIG

config %config: scripts_basic outputmakefile FORCE
    $(Q)mkdir -p include/linux include/config
    $(Q)$(MAKE) $(build)=scripts/kconfig $@

else
# ==========================================================================
# 仅编译内核

ifeq ($(KBUILD_EXTMOD),)
# Additional helpers built in scripts/
# Carefully list dependencies so we do not try to build scripts twice
# in parallel
PHONY += scripts
scripts: scripts_basic include/config/auto.conf
    $(Q)$(MAKE) $(build)=$(@)

#下面就不用说了吧
# Objects we will link into vmlinux / subdirs we need to visit
```

```
init-y        := init/
drivers-y     := drivers/ sound/
net-y         := net/
libs-y        := lib/
core-y        := usr/
endif # KBUILD_EXTMOD

ifeq ($(dot-config),1)
# Read in config
-include include/config/auto.conf

ifeq ($(KBUILD_EXTMOD),)
# Read in dependencies to all Kconfig* files, make sure to run
# oldconfig if changes are detected.
-include include/config/auto.conf.cmd

# To avoid any implicit rule to kick in, define an empty command
$(KCONFIG_CONFIG) include/config/auto.conf.cmd: ;

# If .config is newer than include/config/auto.conf, someone tinkered
# with it and forgot to run make oldconfig.
# if auto.conf.cmd is missing then we are probably in a cleaned tree so
# we execute the config step to be sure to catch updated Kconfig files
include/config/auto.conf: $(KCONFIG_CONFIG) include/config/auto.conf.cmd
    $(Q)$(MAKE) -f $(srctree)/Makefile silentoldconfig
else
# external modules needs include/linux/autoconf.h and include/config/auto.conf
# but do not care if they are up-to-date. Use auto.conf to trigger the test
PHONY += include/config/auto.conf

include/config/auto.conf:
    $(Q)test -e include/linux/autoconf.h -a -e $@ || (        \
    echo;                                   \
    echo "  ERROR: Kernel configuration is invalid.";        \
    echo "         include/linux/autoconf.h or $@ are missing.";    \
    echo "         Run 'make oldconfig && make prepare' on kernel src to fix
it.";    \
    echo;                                   \
    /bin/false)

endif # KBUILD_EXTMOD

else
# Dummy target needed, because used as prerequisite
```

```
include/config/auto.conf: ;
endif # $(dot-config)


# ===========================================================================
# 仅编译内核

ifeq ($(KBUILD_EXTMOD),)
# Additional helpers built in scripts/
# Carefully list dependencies so we do not try to build scripts twice
# in parallel
PHONY += scripts
scripts: scripts_basic include/config/auto.conf
    $(Q)$(MAKE) $(build)=$(@)

#下面就不用说了吧
# Objects we will link into vmlinux / subdirs we need to visit
init-y        := init/
drivers-y     := drivers/ sound/
net-y         := net/
libs-y        := lib/
core-y        := usr/
endif # KBUILD_EXTMOD

.....

all: vmlinux

#配置 gcc 选项
ifdef CONFIG_CC_OPTIMIZE_FOR_SIZE
CFLAGS        += -Os
else
CFLAGS        += -O2
endif
....

# 缺剩的编译的内核镜像
export KBUILD_IMAGE ?= vmlinux

#
# vmlinux 与 map 安装路径，默认时 boot 目录
export    INSTALL_PATH ?= /boot

#
```

```
#module 安装目录
#

MODLIB    = $(INSTALL_MOD_PATH)/lib/modules/$(KERNELRELEASE)
export MODLIB

#
#  如果定义了 INSTALL_MOD_STRIP,在安装 module 后会 strip 这些安装的 modules

ifdef INSTALL_MOD_STRIP
ifeq ($(INSTALL_MOD_STRIP),1)
mod_strip_cmd = $(STRIP) --strip-debug
else
mod_strip_cmd = $(STRIP) $(INSTALL_MOD_STRIP)
endif # INSTALL_MOD_STRIP=1
else
mod_strip_cmd = true
endif # INSTALL_MOD_STRIP
export mod_strip_cmd

#如果不是安装 module
ifeq ($(KBUILD_EXTMOD),)
core-y        += kernel/ mm/ fs/ ipc/ security/ crypto/ block/

vmlinux-dirs    := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
             $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
             $(net-y) $(net-m) $(libs-y) $(libs-m)))

vmlinux-alldirs    := $(sort $(vmlinux-dirs) $(patsubst %/,%,$(filter %/, \
             $(init-n) $(init-) \
             $(core-n) $(core-) $(drivers-n) $(drivers-) \
             $(net-n)  $(net-)  $(libs-n)    $(libs-))))

init-y        := $(patsubst %/, %/built-in.o, $(init-y))
core-y        := $(patsubst %/, %/built-in.o, $(core-y))
drivers-y     := $(patsubst %/, %/built-in.o, $(drivers-y))
net-y         := $(patsubst %/, %/built-in.o, $(net-y))
libs-y1         := $(patsubst %/, %/lib.a, $(libs-y))
libs-y2         := $(patsubst %/, %/built-in.o, $(libs-y))
libs-y        := $(libs-y1) $(libs-y2)

# Build vmlinux
...
```

#配置 syms
...

#以下是内核 link 时的配置
....

[摘要] 由于Linux的独特优势，使越来越多的企业和科研机构把目光转向Linux的开发和研究上。目前Linux最新的稳定内核版本为2.6.17，但是当今绝大部分对于Linux Makefile的介绍文章都是基于2.4内核的，可以说关于2.6内核Makefile相关的文章凤毛麟角，笔者抽时间完成了这篇分析文章，让读者迅速熟悉Linux最新Makefile体系，从而加深对内核的理解，同时也希望能对Linux在公司的推广起到一定的推动作用，算是抛砖引玉吧！

1 Makefile组织层次

Linux的Make体系由如下几部分组成：

Ø 顶层Makefile

顶层Makefile通过读取配置文件，递归编译内核代码树的相关目录，从而产生两个重要的目标文件：vmlinux和模块。

Ø 内核相关Makefile

位于arch/$(ARCH) 目录下，为顶层Makefile提供与具体硬件体协结构相关的信息。

Ø 公共编译规则定义文件。

包括Makefile.build 、Makefile.clean、Makefile.lib、Makefile.host等文件组成。这些文件位于scripts目录中，定义了编译需要的公共的规则和定义。

Ø 内核配置文件 .config

通过调用make menuconfig或者make xconfig命令，用户可以选择需要的配置来生成期望的目标文件。

Ø 其他Makefile

主要为整个Makefile体系提供各自模块的目标文件定义，上层Makefile根据它所定义的目标来完成各自模块的编译。

2 Makefile的使用

在编译内核之前，用户必须首先完成必要的配置。Linux内核提供了数不胜数的功能，支持众多的硬件体系结构，这就需要用户对将要生成的内核进行裁减。内核提供了多种不同的工具来简化内核的配置，最简单的一种是字符界面下命令行工具：

```
make config
```

这个工具会依次遍历内核所有的配置项，要求用户进行逐项的选择配置。这个工具会耗费用户太多时间，除非万不得以（你的编译主机不支持其他配置工具）一般不建议使用。

用户还可以使用利用ncurse库编制的图形界面工具，这就是大名鼎鼎的：

```
make menuconfig
```

相信以前对2.4内核比较熟悉的用户一定不会陌生。当然在2.6内核中提供了更漂亮和方便的基于X11的图形配置工具：

```
make xconfig
```

当用户使用这个工具对Linux内核进行配置时，界面下方会出现这个配置项相关的帮助信息和简单描述，当你对内核配置选项不太熟悉时，建议你使用这个工具来进行内核配置。

当用户完成配置后，配置工具会自动生成.config文件，它被保存在内核代码树的根目录下。

用户可以很容易找到它，当然用户也可以直接对这个文件进行简单的修改。但是当你修改过配置文件之后，你必须通过下面的命令来验证和更新配置：

make oldconfig

跟 2.4 版本的不同之处在于，用户不需要显示的调用 make dep 命令来生成依赖文件，内核会自动维护代码间的依赖关系。

当一切工作完成以后，用户只需要简单键入 make，剩下所有的工作 makefile 就会自动替你完成了。

3 Makefile 编译流程

当用户使用 Linux 的 Makefile 编译内核版本时，Makefile 的编译流程如下：

Ø 使用命令行或者图形界面配置工具，对内核进行裁减，生成.config 配置文件

Ø 保存内核版本信息到 include/linux/version.h

Ø 产生符号链接 include/asm,指向实际目录 include/asm-$(ARCH)

Ø 为最终目标文件的生成进行必要的准备工作

Ø 递归进入 /init 、/core、 /drivers、 /net、 /lib 等目录和其中的子目录来编译生成所有的目标文件

Ø 链接上述过程产生的目标文件生成 vmlinux，vmlinux 存放在内核代码树的根目录下

Ø 最后根据 arch/$(ARCH)/Makefile 文件定义的后期编译的处理规则建立最终的映象 bootimage,包括创建引导记录、准备 initrd 映象和相关处理

4 Makefile 关键规则和定义描述

1) 目标定义

目标定义是 Makefile 文件的核心部分，目标定义通知 Makefile 需要生成哪些目标文件、如何根据特殊的编译选项链接目标文件，同时控制哪些子目录要递归进入进行编译。

这个例子 Makefile 文件位于/fs/ext2 目录 ：

```
#
# Makefile for the linux ext2-filesystem routines.
#


obj-$(CONFIG_EXT2_FS) += ext2.o


ext2-y := balloc.o bitmap.o dir.o file.o fsync.o ialloc.o inode.o \
                      ioctl.o namei.o super.o symlink.o


ext2-$(CONFIG_EXT2_FS_XATTR)   += xattr.o xattr_user.o xattr_trusted.o
ext2-$(CONFIG_EXT2_FS_POSIX_ACL) += acl.o
ext2-$(CONFIG_EXT2_FS_SECURITY)     += xattr_security.o
ext2-$(CONFIG_EXT2_FS_XIP) += xip.o
```

这表示与 ext2 相关的目标文件由 ext2-y 定义的文件列表组成，其中 ext2-$(*)是由内核配置文件.config 中的配置项决定，最终 Makefile 会在这个目录下统一生成一个目标文件 ext2.o（由 obj-$(CONFIG_EXT2_FS)决定）。其中 obj-y 表示为生成 vmlinux 文件所需要的目标文件集合，具体的文件依赖于内核配置。

Makefile 会编译所有的$(obj-y)中定义的文件，然后调用链接器将这些文件链接到 built-in.o 文件中。最终 built-in.o 文件通过顶层 Makefile 链接到 vmlinux 中。值得注意的是$(obj-y)的文件顺序很重要。列表文件可以重复，文件第一次出现时将会链接到 built-in.o 中，后来出现的同名文件将会被忽略。文件顺序直接决定了他们被调用的顺序，这一点读者需要特别注意。

读者可能会在某些 Makefile 中发现 lib-y 定义，所有包含在 lib-y 定义中的目标文件都将会被编译到该目录下一个统一的库文件中。值得注意的是 lib-y 定义一般被限制在 lib 和 arch/$(ARCH)/lib 目录中。

体系 makefile 文件和顶层 makefile 文件共同定义了如何建立 vmlinux 文件的规则。

$(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。

然后是如下位置顺序：

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

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

例如: #arch/i386/Makefile

```
        libs-y                  += arch/i386/lib/
core-y                  += arch/i386/kernel/ \
                                        arch/i386/mm/ \
                                        arch/i386/$(mcore-y)/ \
                                        arch/i386/crypto/
drivers-$(CONFIG_MATH_EMULATION) += arch/i386/math-emu/
drivers-$(CONFIG_PCI)           += arch/i386/pci/
        ..........................................
```

2) 目录递归

Makefile 文件只负责当前目录下的目标文件，子目录中的文件由子目录中的 makefile 负责编译，编译系统使用 obj-y 和 obj-m 来自动递归编译各个子目录中的文件。

对于 fs/Makefile:

```
obj-$(CONFIG_EXT2_FS) += ext2/
```

如果在内核配置文件.config 中，CONFIG_EXT2_FS 被设置为 y 或者 m，则内核 makefile 会自动进入 ext2 目录来进行编译。内核 Makefile 只使用这些信息来决定是否需要编译这个目录，子目录中的 makefile 规定哪些文件编译为模块，哪些文件编译进内核。

3) 依赖关系

Linux Makefile 通过在编译过程中生成的 .文件名.o.cmd（比如对于 main.c 文件，它对应的依赖文件名为.main.o.cmd）来定义相关的依赖关系。

一般文件的依赖关系由如下部分组成：

Ø 所有的前期依赖文件（包括所有相关的*.c 和 *.h）

Ø 所有与 CONFIG_选项相关的文件

Ø 编译目标文件所使用到的命令行

位于 init 目录下的 main.c 文件的依赖文件.main.o.cmd 内容如下，读者可以结合

起来理解上述文件依赖关系的三个组成部分:

```
cmd_init/main.o  :=  gcc  -m32  -Wp,-MD,init/.main.o.d    -nostdinc  -isystem
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/include    -D__KERNEL__     -Iinclude
-Iinclude2           -I/home/linux/linux-2.6.17.11/include           -include
include/linux/autoconf.h -I/home/linux/linux-2.6.17.11/init -Iinit -Wall -Wundef
-Wstrict-prototypes  -Wno-trigraphs  -fno-strict-aliasing  -fno-common  -Os
-fomit-frame-pointer -pipe -msoft-float -mpreferred-stack-boundary=2 -march=i686
-mcpu=pentium4                    -mregparm=3                    -ffreestanding
-I/home/linux/linux-2.6.17.11/include/asm-i386/mach-default
-Iinclude/asm-i386/mach-default                                -D"KBUILD_STR(s)=\#s"
-D"KBUILD_BASENAME=KBUILD_STR(main)"  -D"KBUILD_MODNAME=KBUILD_STR(main)" -c -o
init/.tmp_main.o /home/linux/linux-2.6.17.11/init/main.c
deps_init/main.o := \
               /home/linux/linux-2.6.17.11/init/main.c \
               $(wildcard include/config/x86/local/apic.h) \
$(wildcard include/config/acpi.h) \
# 由于篇幅的关系，此处略去一些定义
……..
include2/asm/mpspec_def.h \

/home/linux/linux-2.6.17.11/include/asm-i386/mach-default/mach_mpspec.h \
           include2/asm/io_apic.h \
           include2/asm/apic.h \


init/main.o: $(deps_init/main.o)


$(deps_init/main.o):
```

4)　　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 makefile 规则。

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期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)。

5) 引导映象

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。在 Linux 中 Makefile 无法为所有的体系结构提供标准化的方法，因此常需要具体硬件体系结构下 makefile 提供附加处理规则。

附加处理过程常位于 arch/$(ARCH)/下的 boot/目录。

内核编译体系无法在 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。

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。

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) 常用编译命令

if_changed

如果必要，执行传递的命令。

用法:

```
$(builtin-target): $(obj-y) FORCE
$(call if_changed,link_o_target)
```

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$(builtin-target)中，否则命令行检查将失败，目标一直会编译。

if_changed_dep

如果必要，执行传递的命令并更新依赖文件。

用法:

```
%.o: %.S FORCE
    $(call if_changed_dep,as_o_S)
```

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。同时它会重新检测依赖关系的改变并将生成新的依赖文件。这是与 if_changed 命令的区别。

7)　　定制命令

当正常执行带编译命令时命令的简短信息会被显示（要想显示详细的命令，请在命令行中加入 V＝1)。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 命令编译$(obj)/bzImage 目标时将显示：

BUILD　arch/i386/boot/bzImage

8)　　预处理链接脚本

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。

9)　　主机辅助程序的编译

内核编译系统支持在编译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。这一部分的内容相对于其他内核文件的编译要简单的多，感兴趣的读者可以参考 scripts/Makefile.build 中的相关内容。

10)　Clean 机制

clean 命令清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。

4 小结随着 Linux 的飞速发展,越来越多的开发人员将关注的焦点集中到 Linux 的研究和开发上。如果想对 Linux 内核进行研究和开发，就必须首先熟悉 Linux 内核 Makefile 的组织和编译过程。目前 Linux 最新的稳定内核版本为 2.6.17，但是当今绝大部分对于 Linux Makefile 的介绍都是基于 2.4 内核的,可以说关于 2.6 内核 Makefile 相关的文章凤毛麟角,我特意抽时间完成了这篇分析文章，让读者迅速熟悉 Linux 最新 Makefile 体系，从而加深对内核的理解，同时也希望能对 Linux 在公司的推广起到一定的推动作用。