Linux内核编程：Kbuild子系统

主讲：王利涛

《嵌入式工程师自我修养》系列教程

第04步：Linux内核编程

01 Kbuild简介

- 什么是Kbuild？
  - Kernel build，用来编译Linux内核
  - 基于GNU make设计，对Makefile进行扩充
    - 菜单式配置：Kconfig
    - 预定义目标和变量：xx_defconfig、menuconfig、obj-y
    - 跨平台工具、递归式Makefile
  - Linux模块化设计、高度可以裁剪
    - 模块机制
    - Kbuild子系统

- Kbuild的优势
  - 高度灵活可定制：编译参数、模块编译选项
  - 使用方便：内核裁剪、添加模块、删除模块
  - 配置简单：可交互的图形菜单
- Kbuild应用广泛
  - 被越来越多的开源软件使用
  - Xen、seabios
  - Buildroot、 U-boot、 Busybox

- 学习Kbuild，有哪些收获？
  - 深刻理解Makefile是如何编译Linux内核、U-boot等大型工程的
  - 遇到编译错误时，提供更多的视角去分析
  - 提供了一张地图，破解Linux内核的“黑暗森林”
  - 有助于理解内核启动流程、组织架构
  - 掌握Kbuild工作原理，对学习其他开源软件有帮助
- 本期课程规划
  - 如何使用Kbuild Makefile
  - Kbuild工作流程分析
  - 分析案例：内核编译、模块编译、安装、头文件
  - Makefile：Linux三剑客—Makefile工程实践

02 Kbuild工作流程

• 内核编译流程

```
# make ARCH=arm vexpress_defconfig
# make menuconfig
# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-  uImage  LOADADDR=0x60003000
```

```
# make ARCH=arm vexpress_defconfig
        HOSTCC  scripts/basic/fixdep
        HOSTCC  scripts/kconfig/conf.o
        HOSTCC  scripts/kconfig/confdata.o
        LEX     scripts/kconfig/lexer.lex.c
        YACC    scripts/kconfig/parser.tab.[ch]
        HOSTCC  scripts/kconfig/lexer.lex.o
        HOSTCC  scripts/kconfig/parser.tab.o
        HOSTCC  scripts/kconfig/preprocess.o
        HOSTCC  scripts/kconfig/symbol.o
        HOSTCC  scripts/kconfig/util.o
        HOSTLD  scripts/kconfig/conf
        #
        # configuration written to .config
        #
```

• 内核编译流程

```
# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
  SYNC    include/config/auto.conf.cmd
  UPD     include/config/kernel.release
  UPD     include/generated/uapi/linux/version.h
  MKELF   scripts/mod/elfconfig.h
  HOSTCC  scripts/mod/modpost.o
  CC      scripts/mod/devicetable-offsets.s
  UPD     scripts/mod/devicetable-offsets.h
  HOSTLD  scripts/mod/modpost
  CC      init/main.o
  CHK     include/generated/compile.h
  UPD     include/generated/compile.h
  AR      sound/x86/built-in.a
  AR      sound/xen/built-in.a
  CC      sound/sound_core.o
  AR      sound/built-in.a
  CC      lib/strncpy_from_user.o
  CC      lib/strnlen_user.o
  CC      lib/net_utils.o
  CC      lib/sg_pool.o
  AR      lib/built-in.a
  LD      vmlinux.o
  LD      vmlinux
SORTTAB vmlinux
SYSMAP  System.map
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
LDS     arch/arm/boot/compressed/vmlinux.lds
AS      arch/arm/boot/compressed/head.o
GZIP    arch/arm/boot/compressed/piggy_data
AS      arch/arm/boot/compressed/piggy.o
CC      arch/arm/boot/compressed/misc.o
CC      arch/arm/boot/compressed/decompress.o
CC      arch/arm/boot/compressed/string.o
AS      arch/arm/boot/compressed/hyp-stub.o
AS      arch/arm/boot/compressed/lib1funcs.o
AS      arch/arm/boot/compressed/ashldi3.o
AS      arch/arm/boot/compressed/bswapsdi2.o
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
```

- 编译三步骤
  - 配置阶段：编译平台、目标、配置文件
  - 编译阶段：解析Makefile、建立目标依赖关系、按照依赖关系依次生成各个目标及目标依赖
  - 安装阶段：
    - 桌面PC：内核镜像安装、模块安装、头文件安装
    - 嵌入式：根文件系统、Flash镜像制作等

- Makefile中的预定义
  - 预定义目标：
    - xxx_defconfig 、 menuconfig 、 gconfig
    - vmlinux、bzImage、zImage
    - modules、install、modules_install
    - clean、mrproper、distclean
  - 预定义变量：
    - ARCH、CROSS_COMPILE
    - obj-m、obj-y、xxx-objs
- Config symbols

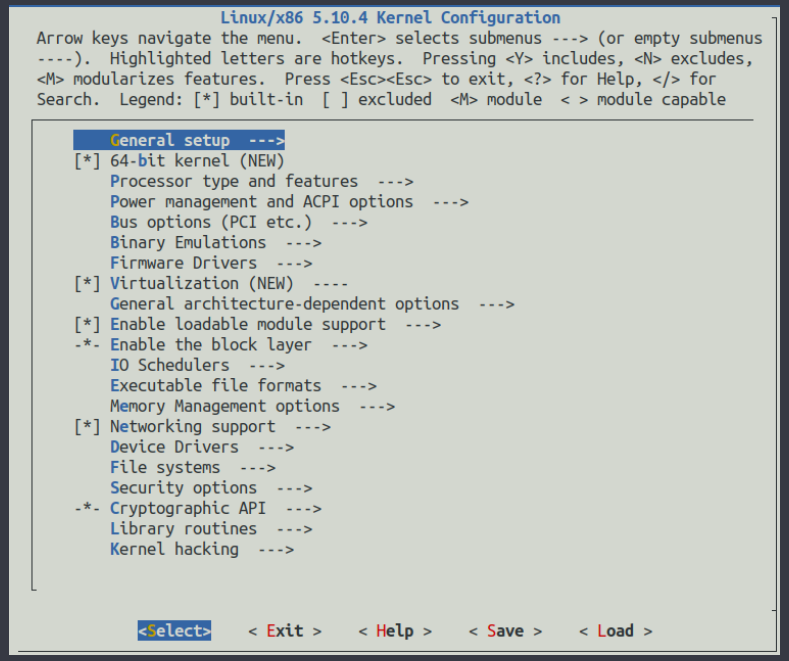
```
obj-$(CONFIG_HELLO)
    config symbol   ←  .config
     配置变量              config entry ← Kconfig文件←menuconfig
    CONFIG_XXX          config entry ← xxx_defconfig
```

03 Kbuild 编译系统构成

- Kbuild本质
  - 一个可扩展、可配置的Makefile框架
  - 递归式Makefile、菜单式配置
- 构成：
  - Makefile：顶层目录下的Makefile
  - .config：内核的配置文件
  - arch/$(ARCH)/Makefile：跟平台架构相关的Makefile
  - scripts/Makefile.*：通用编译规则
  - Kbuild Makefile：分布在各个子目录下
  - Kconfig：配置菜单，定义每个config symbol的属性（类型、描述、依赖等）

04 Kconfig 简介

- Kconfig作用
  - 用来生成配置菜单，配置各种config symbol
  - 生成对应的配置变量：CONFIG_XXX
  - 每个目录下都有一个Kconfig文件
  - 各个Kconfig文件通过source命令构建多级菜单
  - 解析工具：scripts/kconfig/*conf




```
# make menuconfig
  HOSTCC  scripts/kconfig/mconf.o
  HOSTCC  scripts/kconfig/lxdialog/checklist.o
  HOSTCC  scripts/kconfig/lxdialog/inputbox.o
  HOSTCC  scripts/kconfig/lxdialog/menubox.o
  HOSTCC  scripts/kconfig/lxdialog/textbox.o
  HOSTCC  scripts/kconfig/lxdialog/util.o
  HOSTCC  scripts/kconfig/lxdialog/yesno.o
  HOSTLD  scripts/kconfig/mconf
```

- 实验：
  - make menuconfig: apt-get install libncurses5-dev
  - 内核模块添加配置菜单
- Kconfig语法
  - config：用来定义菜单选项
  - menuconfig
  - choice/endchooice
  - comment
  - if/endif
  - source：生成一个树型菜单

05 Kconfig 菜单条目

• 菜单示例

"config" <symbol>
<config options>

drivers/char/Kconfig：

```
config VIRTIO_CONSOLE
        tristate
        prompt  "Virtio console"
        depends on TTY
        select HVC_DRIVER
        select VIRTIO
        help
          Virtio console for use with hypervisors.
```

CONFIG_VIRTIO_CONSOLE

- config symbol
  - bool：y n
  - tristate： y m n
  - int：数值
  - hex：数值
  - string：字符串

```
bool "Networking support“

bool
prompt "Networking support"
```

06 依赖关系：depends on

• 依赖关系示例

```
config LCD
    bool "lcd driver"
    depends on TEST
    default n
    help
        This is a lcd driver config symbol for test
        this config entry depends on CONFIG_TEST
```

• 内核中的依赖关系示例

```
config STACKPROTECTOR
        bool "Stack Protector buffer overflow detection"
        depends on $(cc-option,-fstack-protector)
```

```
config CC_HAS_ASM_GOTO
        def_bool $(success,$(srctree)/scripts/gcc-goto.sh $(CC))
```

```
config USB_DISK
        depends on TEST && m
```

07 反向依赖：select / imply

• 反向依赖：select

```
config TEST
    bool "config menu test"
    default y
    select RTC
    help
        This is a configuration menu test
        choose Y to display menu entries
```

• 弱反向依赖：imply

```
config TEST
    bool "config menu test"
    default y
    select RTC
    imply KEY
    help
        This is a configuration menu test
        choose Y to display menu entries
```

08 内核配置中的反向依赖

- 示例1：
  - 一个子系统绑定（依赖）几个驱动，当用户选择这个子系统中，这几个关联的驱动都会自动选中。

```
A_init()
{
        if (IS_REACHABLE(CONFIG_C))
                C_register(&a);
        ...
}
```

- 示例2：

```
# Generic IOMAP is used to ...
 config HAVE_GENERIC_IOMAP

 config GENERIC_IOMAP
        depends on HAVE_GENERIC_IOMAP && FOO

lib/Makefile :

        obj-$(CONFIG_GENERIC_IOMAP) += iomap.o

For each architecture using the generic IOMAP functionality we would see:

 config X86
        select ...
        select HAVE_GENERIC_IOMAP
        select ...
```

09 Kconfig menuconfig菜单

- menuconfig菜单
  - 所有的子选项在子菜单中单独显示

```
(1):
 menuconfig A
 if A
    config A1
    config A2
 endif

(2):
 menuconfig A
 config A1
    depends on A
 config A2
    depends on A
```

• menuconfig菜单

```
(3):
 menuconfig A
    config A0
 if A
    config A1
    config A2
 endif

 (4):
 menuconfig A
 config A0
 config A1
    depends on A
 config A2
    depends on A
```

10 Kconfig choice/endchoice

- 互斥选择

```
"choice" [symbol]
<choice options>
<choice block>
"endchoice"
```

```
choice
        prompt “lcd ratio setting"
        config "320*240"
        config "1920*1080"
        config "16*16"
        ...
endchoice
```

11 Kconfig 子菜单

- 方法一
  - 通过依赖关系生成菜单
  - 若菜单条目依赖前项，则其为该选项的子菜单

```
config A
    bool "A configuration"
config B
    bool "B configuration"
    depends on A
```

- 方法二
  - 子菜单：menu/endmenu
  - 所有的菜单条目都在menu和和endmenu之间的块中
  - 子菜单会继承父菜单的依赖关系

```
menu “test menu”
config xxx_1
…
config xxx_2
endmenu
```

12 更多编译目标

- 目标
  - make config：
  - make nconfig：基于文本的菜单配置
  - make menuconfig：依赖ncurses图形库
    - # apt-get install libncurses5-dev
  - make xconfig：基于窗口的配置菜单，依赖Qt库
    - # add-apt-repository ppa:rock-core/qt4
    - # apt-get install libqt4-dev
  - make gconfig：基于GTK的菜单配置
    - # apt-get install gtk+-2.0 glib-2.0 libglade2-dev
  - .config

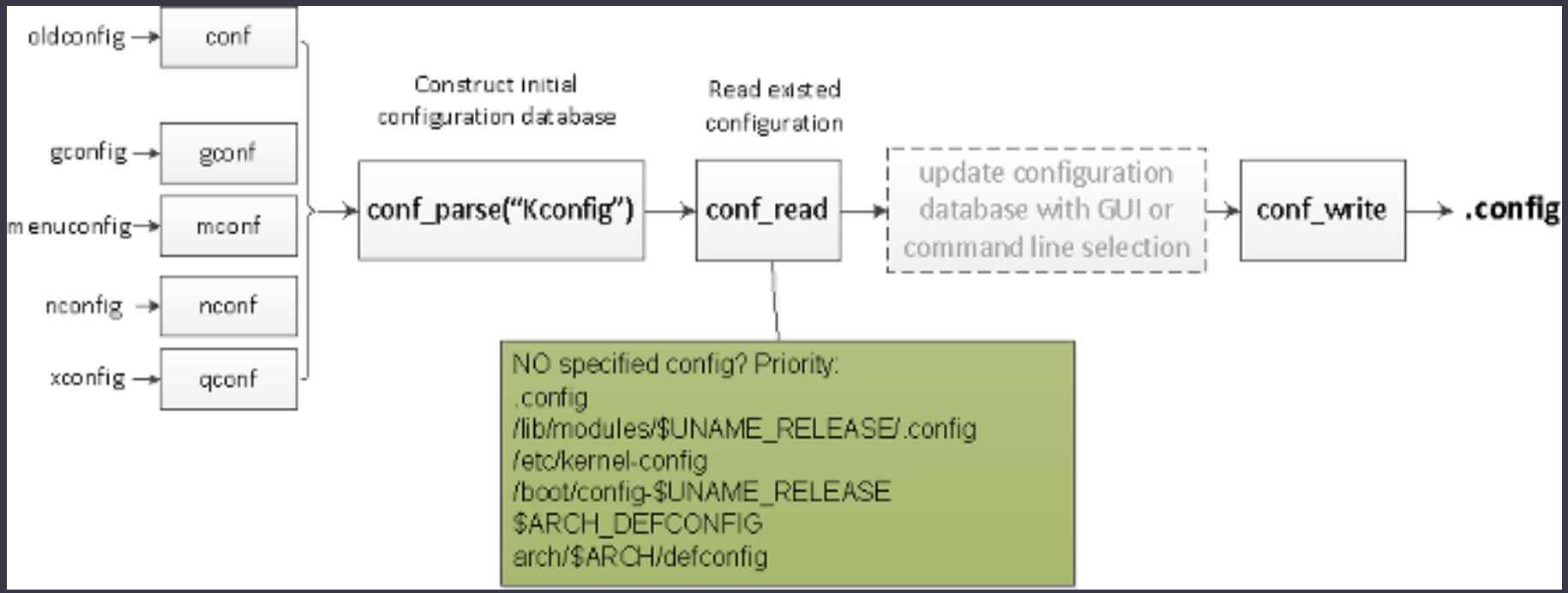
```
# make xconfig
  HOSTCC  scripts/kconfig/images.o
*
* Could not find Qt via pkg-config.
* Please install either Qt 4.8 or 5.x. and make sure it's in PKG_CONFIG_PATH
*
make[1]: *** [scripts/kconfig/Makefile:214: scripts/kconfig/qconf-cfg] Error 1
make: *** [Makefile:602: xconfig] Error 2
```

```
# make gconfig
*
* Unable to find the GTK+ installation. Please make sure that
* the GTK+ 2.0 development package is correctly installed.
* You need gtk+-2.0 gmodule-2.0 libglade-2.0
*
make[1]: *** [scripts/kconfig/Makefile:214: scripts/kconfig/gconf-cfg] Error 1
make: *** [Makefile:602: gconfig] Error 2
```

- 目标
  - .config
  - make clean
  - make mrproper
  - make distclean

13 .config文件（上）

- .config简介
  - .config文件是如何生成的？
  - .config文件里都是什么？
  - .config文件有什么用？如何参与编译工作？
  - 参考：scripts/kconfig/mconf.c、conf.c



- .config的生成
  - make vexpress_defconfig → .config
  - make menuconfig

Makefile:

```
%config: outputmakefile scripts_basic FORCE
	$(Q)$(MAKE) $(build)=scripts/kconfig $@
build := -f $(srctree)/scripts/Makefile.build obj
```

展开后：

```
make -f scripts/Makefile.build  obj= scripts/kconfig  menuconfig/vexpress_defconfig
```

scripts/kconfig/Makefile:

```
%_defconfig: $(obj)/conf
	$(Q)$< $(silent) --defconfig=arch/$(SRCARCH)/configs/$@ $(config)
menuconfig: $(obj)/mconf
	$(Q)$< $(silent) $(Kconfig)
```

- .config的第二阶段
  - .config –> syncconfig –> Makefile
    - include/config/auto.conf：用来配置Makefile
    - include/generated/autoconf.h：供C程序引用
    - include/config/*.h：空头文件，用于构建依赖关系

```
Makefile:
KCONFIG_CONFIG   ?= .config
cmd_syncconfig = $(MAKE) -f  $(srctree)/Makefile syncconfig
PHONY += include/config/auto.conf
%/config/auto.conf  %/generated/autoconf.h: $(KCONFIG_CONFIG)
            +$(call cmd,syncconfig)
include/config/auto.conf:
deps_config := kernel/trace/Kconfig  certs/Kconfig fs/udf/Kconfig …
include/config/auto.conf: \
            $(deps_config)
scripts/Kbuild.include:
cmd = @set -e; $(echo-cmd) $(cmd_$(1))
```

14 .config文件（下）

- .config如何参与编译
  - .config –> syncconfig –> include/config/auto.conf
  - .config –> syncconfig –> include/config/tristate.conf
  - 在Makefile中引用auto.conf定义的配置变量(config symbols)：

顶层Makefile：

```
need-config          := 1
ifdef need-config
include include/config/auto.conf
endif
```

include include/config/auto.conf：

```
CONFIG_USB=y
```

drivers/usb/Makefile：

```
obj-$(CONFIG_USB)            += core/
```

- .config如何被C语言引用
  - .config –> syncconfig –> include/generated/autoconf.h
  - 配置变量(config symbols) –> C语言的宏定义
  - 在C程序中引用autoconf.h定义的宏：

include/generated/autoconf.h：

```
#define CONFIG_USB_MON 1
```

include/linux/usb.h:

```
#if defined(CONFIG_USB_MON) || defined(CONFIG_USB_MON_MODULE)
        struct mon_bus *mon_bus;    /* non-null when associated */
        int monitored;              /* non-zero when monitored */
#endif
```

```
gcc –linclude  include/generated/autoconfig.h –c hello.c
```

15 Kbuild Makefile 工作流程

- Linux内核镜像编译流程

```
# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- vexpress_defconfig
# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- menuconfig
# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- -j4
# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- uImage
LOADADDR=ox60003000
```

- Kbuild Makefile构成
  - 顶层Makefile：主要用来调用相应规则的Makefile
  - .config：用户配置的各种选项
  - arch/$(ARCH)/Makefile：跟平台相关的Makefile
  - 各个目录下的Makefile：负责编译各个模块
  - scripts/Makefile.*：定义各种通用规则

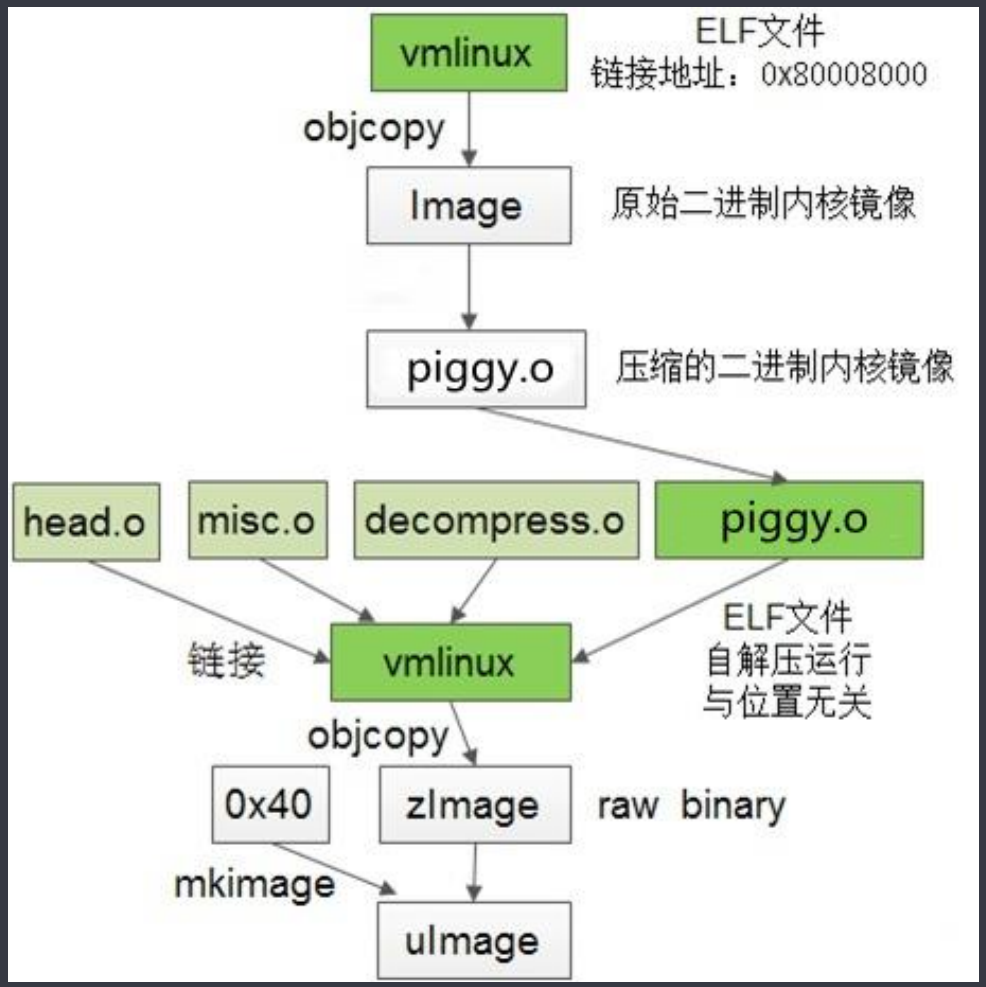
- scripts/Makefile.*：各类规则文件
  - scripts/Makefile.build：通用规则，用来编译built-in.a、lib.a
  - scripts/Makefile.lib：负责分析obj-y、obj-y和子目录中的subdir-y等
  - scripts/Makefile.include：一些通用定义，被Makefile.*包含使用
  - scripts/Makefile.host：编译各种主机工具
  - scripts/Makefile.headerinst：头文件安装规则
  - scripts/Makefile.modinst：模块install规则
  - scripts/Makefile.modpost：模块编译，由.o和.mod生成module.ko
  - scripts/Makefile.modsign：模块签名
  - scripts/Makefile.clean：clean 规则，make clean时调用

- Kbuild Makefile预定义目标和变量
  - obj-m：将当前文件编译为独立的模块
  - obj-y：将当前文件编译进内核
  - xxx-objs：一个模块依赖的多个源文件
  - bzImage：
  - menuconfig：
  - CONFIG_xxx：
    - -include include/config/auto.conf
    - include/config/auto.conf.cmd

- Kbuild Makefile工作流程
  - 根据ARCH变量，首先include arch/$(ARCH)/Makefile
  - 读取 .config文件：读取用户的各种配置变量
  - 解析预定义目标、目标，构建依赖关系
  - 编译各个模块或组件（使用scripts/Makefile.*）
    - 将每个目录下的源文件编译为对应的.o目标文件
    - 将.o目标文件归档为built-in.a
  - 将所有对象链接成 vmlinux
  - 编译模块…

16 vmlinux编译过程分析

• 内核镜像编译流程


vmlinux
ELF文件
链接地址：0x80008000
objcopy
Image
原始二进制内核镜像
piggy.o
压缩的二进制内核镜像
head.o
misc.o
decompress.o
piggy.o
链接
vmlinux
ELF文件
自解压运行
与位置无关
objcopy
0x40
zImage
raw binary
mkimage
uImage

• 默认目标的依赖：vmlinux

Top Makefile：

```
# That's our default target when none is given on the command line
PHONY := __all
__all: all
all: vmlinux
vmlinux: scripts/link-vmlinux.sh  autoksyms_recursive  $(vmlinux-deps)
        +$(call if_changed,link-vmlinux)
```

```
vmlinux-deps := $(KBUILD_LDS) $(KBUILD_VMLINUX_OBJS)
$(KBUILD_VMLINUX_LIBS)

export KBUILD_LDS          := arch/$(SRCARCH)/kernel/vmlinux.lds
KBUILD_VMLINUX_OBJS := $(head-y) $(patsubst %/,%/built-in.a, $(core-y))
KBUILD_VMLINUX_ OBJS += $(addsuffix built-in.a, $(filter %/, $(libs-y)))
KBUILD_VMLINUX_OBJS += $(patsubst %/,%/built-in.a, $(drivers-y))
```

• KBUILD_VMLINUX_OBJ变量

arch/arm/kernel/Makefile：

```
head-y                  := arch/arm/kernel/head$(MMUEXT).o
core-y                  += arch/arm/
core-y                  += $(machdirs) $(platdirs)
libs-y                  := arch/arm/lib/ $(libs-y)
```

Makefile：

```
core-y                  := init/ usr/
core-y                  += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/
libs-y                  := lib/
drivers-y               := drivers/  sound/
drivers-y     += net/  virt/
```

展开后：

```
KBUILD_VMLINUX_OBJS := arch/arm/kernel/head.o  arch/arm/built-in.a
init/built-in.a  usr/built-in.a
 kernel/built-in.a  certs/built-in.a mm/ built-in.a  fs/ built-in.a  ipc/ built-in.a
 security/built-in.a  crypto/built-in.a  block/built-in.a lib/built-in.a
arch/arm/lib/built-in.a drivers/built-in.a  sound/built-in.a  net/ built-in.a
virt/ built-in.a
```

- KBUILD_VMLINUX_LIBS变量

```
KBUILD_VMLINUX_LIBS  := $(patsubst %/, %/lib.a, $(libs-y))
libs-y                          := lib/
libs-y                          := arch/arm/lib/ $(libs-y)
 展开后：
KBUILD_VMLINUX_LIBS  := lib/lib.a  arch/arm/lib/lib.a
```

- autoksyms_recursive

```
autoksyms_recursive:  descend  modules.order
        $(Q)$(CONFIG_SHELL)  $(srctree)/scripts/adjust_autoksyms.sh \
          "$(MAKE)  -f  $(srctree)/Makefile  vmlinux"
```

• 生成vmlinux的规则

Makefile：

```
vmlinux: scripts/link-vmlinux.sh  autoksyms_recursive  $(vmlinux-deps)
        +$(call if_changed,link-vmlinux)
scripts/Kbuild.include:
if_changed = $(if $(newer-prereqs)$(cmd-check),                    \
        $(cmd);                                                    \
        printf '%s\n' 'cmd_$@ := $(make-cmd)' > $(dot-target).cmd, @:)
cmd = @set -e; $(echo-cmd) $(cmd_$(1))
```

Makefile：

```
cmd_link-vmlinux =                                                 \
        $(CONFIG_SHELL) $< "$(LD)" "$(KBUILD_LDFLAGS)" "$(LDFLAGS_vmlinux)";
        $(if $(ARCH_POSTLINK), $(MAKE) -f $(ARCH_POSTLINK) $@, true)
# SHELL used by kbuild
CONFIG_SHELL := sh
LDFLAGS_vmlinux = --build-id=sha1  --orphan-handling=warn
```

展开后：

```
cmd_link-vmlinux = sh scripts/link-vmlinux.sh “ld.lld”“ --build-id=sha1  --
orphan-handling=warn”；  true
```

- scripts/link-vmlinux.sh脚本
  - 链接$(KBUILD_VMLINUX_OBJS)中的所有built-in.a
  - 链接$(KBUILD_VMLINUX_LIBS)
  - 符号处理：生成system.map、include/generated/autoksyms.h等文件

```
modpost_link()
{
        local objects
        objects="--whole-archive                        \
                ${KBUILD_VMLINUX_OBJS}                  \
                --no-whole-archive                      \
                --start-group                           \
                ${KBUILD_VMLINUX_LIBS}                  \
                --end-group"
        ${LD} ${KBUILD_LDFLAGS} -r -o ${1} ${objects}
}
```

17 built-in.a 生成分析

• 默认目标的依赖分析

Makefile：

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

展开后：build-dirs := init lib drivers net sound certs crypto ipc kernel mm …

```
PHONY += descend $(build-dirs)
$(build-dirs): prepare
        $(Q)$(MAKE) $(build)=$@ \
        single-build=$(if $(filter-out $@/, $(filter $@/%,
        $(KBUILD_SINGLE_TARGETS))),1) \
        need-builtin=1 need-modorder=1

KBUILD_SINGLE_TARGETS := $(addprefix $(extmod-prefix), $(single-no-ko))
extmod-prefix = $(if $(KBUILD_EXTMOD),$(KBUILD_EXTMOD)/)
single-no-ko := $(sort $(patsubst %.ko, %.mod, $(MAKECMDGOALS)))
```

Kbuild.include：

```
build := -f $(srctree)/scripts/Makefile.build obj
```

• 示例：编译sound目录

```
sound：
        make -f scripts/Makefile.build  obj=sound
         need-builtin=1  need-modorder=1 single-build=0
```

scripts/Makefile.build

```
PHONY := __build
__build: $(if $(KBUILD_BUILTIN), $(targets-for-builtin)) $(if
$(KBUILD_MODULES), $(targets-for-modules)) $(subdir-ym) $(always-y)

targets-for-builtin := $(extra-y)
targets-for-builtin += $(obj)/built-in.a
```

__build展开后：

```
__build: sound/built-in.a
```

- sound/built-in.a
  - 单文件模块：obj-y=hello.o
  - 复合模块：
    - obj-y=hello.o hello-y= a.o b.o c.o
    - obj-y=hello.o hello-objs= a.o b.o c.o
  - 子目录：obj-y=subdir

scripts/Makefile.build：

```
$(obj)/built-in.a: $(real-obj-y) FORCE
        $(call if_changed,ar_builtin)

cmd_ar_builtin = rm -f $@; $(AR) cDPrST $@ $(real-prereqs)
real-prereqs = $(filter-out $(PHONY), $^)

real-obj-y := $(foreach m, $(obj-y), $(if $(strip $($(m:.o=-objs))
$($(m:.o=-y)) $($(m:.o=-))),$($(m:.o=-objs)) $($(m:.o=-y)),$(m)))
real-obj-y := $(addprefix $(obj)/,$(real-obj-y))
```

18 单个目标文件生成分析

• 单个目标文件编译

scripts/Makefile.build：

```
$(obj)/%.o: $(src)/%.c $(recordmcount_source) $(objtool_dep) FORCE
	$(call if_changed_rule,cc_o_c)
	$(call cmd,force_checksrc)

define rule_cc_o_c
	$(call cmd_and_fixdep,cc_o_c)
	$(call cmd,gen_ksymdeps)
	$(call cmd,checksrc)
	$(call cmd,checkdoc)
	$(call cmd,objtool)
	$(call cmd,modversions_c)
	$(call cmd,record_mcount)
endef

quiet_cmd_cc_o_c = CC $(quiet_modtag)  $@
cmd_cc_o_c = $(CC) $(c_flags) -c -o $@ $<

cmd_force_checksrc = $(CHECK) $(CHECKFLAGS) $(c_flags) $<
```

• 单个目标文件编译

Makefile：

```
CHECK                   = sparse
CHECKFLAGS  := -D__linux__ -Dlinux -D__STDC__ -Dunix -D__unix__ \
-Wbitwise -Wno-return-void -Wno-unknown-attribute $(CF)
-----------------------------------------------------------
scripts/Kbuild.include:
if_changed_rule = $(if $(newer-prereqs)$(cmd-check),$(rule_$(1)),@:)
cmd = @set -e; $(echo-cmd) $(cmd_$(1))
cmd_and_fixdep =
            $(cmd);                                                 \
            scripts/basic/fixdep $(depfile) $@ '$(make-cmd)' > $(dot-target).cmd;\
            rm -f $(depfile)
```

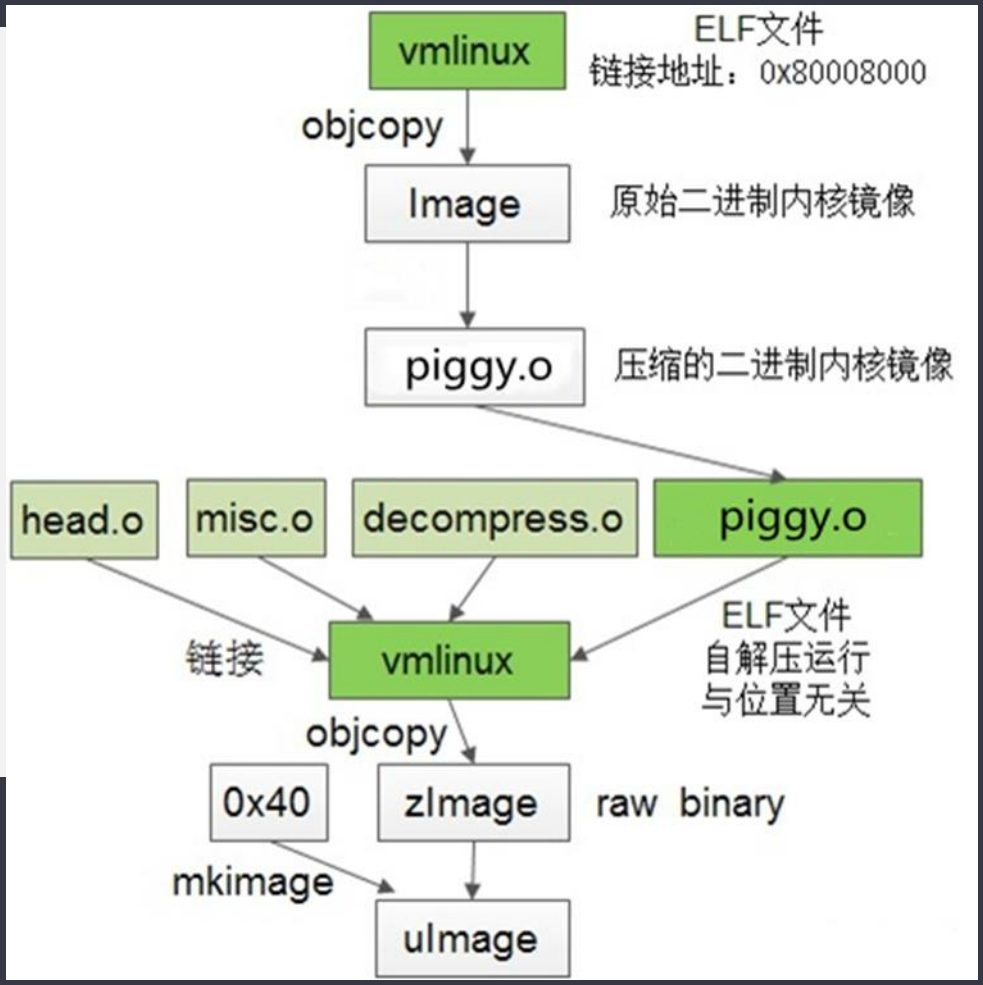
命令展开后：

```
%.o : %.c
   gcc –c –o $@ %.c
   sparse -D__linux__ -Dlinux -D__STDC__ …
```

19 zImage生成分析

• 内核镜像生成过程

```
…
CC  sound/xx.o
CC   xxxx/xxx.o
…
LD      vmlinux
SORTTAB  vmlinux
SYSMAP  System.map
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
GZIP    arch/arm/boot/compressed/piggy_data
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
UIMAGE  arch/arm/boot/uImage
```



• Image镜像生成分析

arch/arm/Makefile：

```
KBUILD_IMAGE := $(boot)/zImage
all: $(notdir $(KBUILD_IMAGE))
zImage: Image
BOOT_TARGETS      = zImage  Image  uImage
$(BOOT_TARGETS): vmlinux
            $(Q)$(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $(boot)/$@
            @$(kecho) '  Kernel: $(boot)/$@ is ready'
```

其中：

build := -f $(srctree)/scripts/Makefile.build obj

boot := arch/arm/boot

machine-y为空， MACHINE := arch/arm/mach-$(word 1,$(machine-y))/

所以　　MACHINE :=

　　boot := arch/arm/boot

```
zImage：  vmlinux
          make -f scripts/Makefile.build  obj=arch/arm/boot  arch/arm/boot/zImage
Image：  vmlinux
          make -f scripts/Makefile.build  obj=arch/arm/boot  arch/arm/boot/Image
```

• Image镜像生成分析

arch/arm/Makefile：

```
$(obj)/Image: vmlinux FORCE
        $(call if_changed,objcopy)
$(obj)/compressed/vmlinux: $(obj)/Image FORCE
        $(Q)$(MAKE) $(build)=$(obj)/compressed $@
$(obj)/zImage: $(obj)/compressed/vmlinux FORCE
        $(call if_changed,objcopy)
```

展开后：

```
arch/arm/boot/Image：vmlinux
        arm-linux-gnueabi-objcopy -O binary -R .comment -S  vmlinux  Image
arch/arm/boot/compressed/vmlinux：arch/arm/boot/Image
        make -f scripts/Makefile.build  obj=arch/arm/boot/compressed  vmlinux
arch/arm/boot/zImage：arch/arm/boot/compressed/vmlinux
        arm-linux-gnueabi-objcopy -O binary -R .comment -S  vmlinux  zImage
```

• Image生成分析

```
arch/arm/boot/Image： vmlinux
        arm-linux-gnueabi-objcopy -O binary -R .comment -S  vmlinux  Image
```

参数说明：

-O：生成一个二进制文件
-R：从一个目标文件中删除指定的section
-S：--strip-all，全方位压缩vmlinux文件

```
$(obj)/piggy.o: $(obj)/piggy_data
$(obj)/piggy_data: $(obj)/../Image FORCE
        $(call if_changed,$(compress-y))

.config:
compress-$(CONFIG_KERNEL_GZIP) = gzip

Makfile.lib:
cmd_gzip = cat $(real-prereqs) | $(KGZIP) -n -f -9 > $@
KGZIP                   = gzip
```

• arch/arm/boot/compressed/vmlinux

```
arch/arm/boot/compressed/vmlinux：arch/arm/boot/Image
        make -f scripts/Makefile.build obj=arch/arm/boot/compressed  vmlinux
arch/arm/boot/compressed/Makefile：
$(obj)/vmlinux:  $(obj)/vmlinux.lds  $(obj)/$(HEAD)  $(obj)/piggy.o
        $(addprefix $(obj)/, $(OBJS)) $(lib1funcs) $(ashldi3) \
        $(bswapsdi2) $(efi-obj-y) FORCE
        $(call if_changed,ld)
        @$(check_for_bad_syms)
```

其中：

```
HEAD       = head.o
OBJS       += misc.o decompress.o
cmd_ld = $(LD) $(ld_flags) $(real-prereqs) -o $@
LD         = $(CROSS_COMPILE)ld
ld_flags  = $(KBUILD_LDFLAGS) $(ldflags-y) $(LDFLAGS_$(@F))
ldflags-y  += $(EXTRA_LDFLAGS)，其中EXTRA_LDFLAGS为空
real-prereqs = $(filter-out $(PHONY), $^)
```

```
arch/arm/boot/compressed/vmlinux: vmlinux.lds head.o misc.o decompress.o piggy.o
        arm-linux-gnueabi-ld -EL vmlinux.lds compressed/head.o  compressed/piggy.o
```

• zImage镜像生成分析

```
arch/arm/boot/zImage： arch/arm/boot/compressed/vmlinux
        arm-linux-gnueabi-objcopy -O binary -R .comment -S  vmlinux  zImage
```

参数说明：

-O：生成一个二进制文件
-R：从一个目标文件中删除指定的section
-S：--strip-all，全方位压缩vmlinux文件

20 uImage镜像生成分析

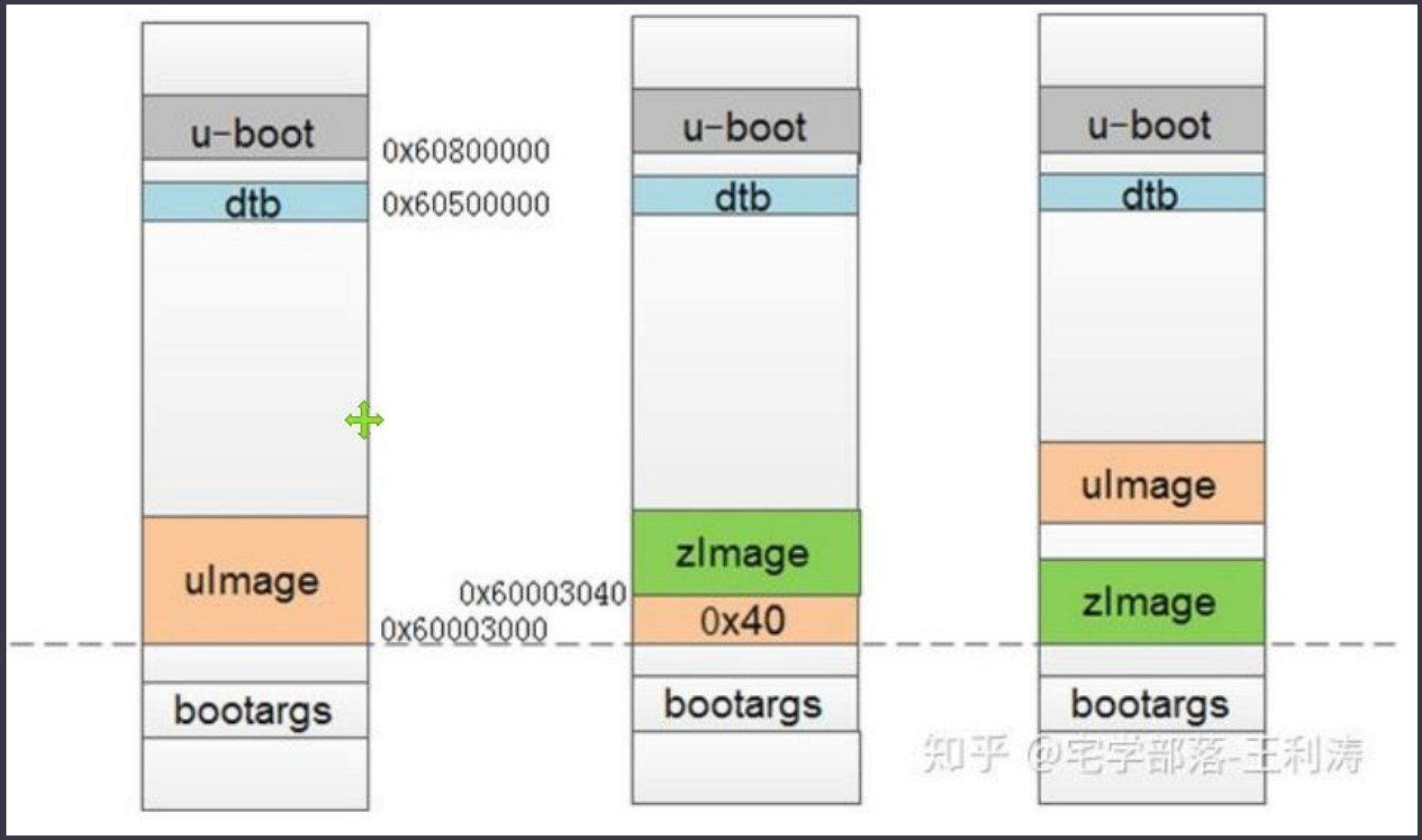
• uImage镜像生成分析

```
$ mkimage –A arm -O linux –T kernel –C none –a 0x60003000  –e 0x60003000  -d
zImage   uImage
```

mkimage参数说明：

-A：指定CPU架构类型
-O：指定操作系统类型
-T：指定image类型
-C：采用的压缩方式：none、gzip、bzip2等
-a：内核加载地址
-e：内核镜像入口地址

- uImage启动过程


u-boot
dtb
uImage
bootargs
0x60800000
0x60500000
0x60003040
0x60003000
u-boot
dtb
zImage
0x40
bootargs
u-boot
dtb
uImage
zImage
bootargs
知乎 @宅学部落-王利涛

21 内核模块编译分析

• 内核模块编译信息

```
# make modules
  CC [M]  drivers/char/hello.o
  MODPOST Module.symvers
  CC [M]  drivers/char/hello.mod.o
  LD [M]  drivers/char/hello.ko
```

modules目标对应的规则：

```
Makefile:
modules: $(if $(KBUILD_BUILTIN),vmlinux)  modules_check  modules_prepare
            $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modpost
```

- 内核模块编译步骤
  - 步骤01
    - 将每个源文件编译为对应的.o目标文件
    - 将单个.o目标文件链接成模块文件module.o
    - 生成对应的module.mod文件
    - 生成module.order文件，里面保存所有的ko文件信息
  - 步骤02
    - 从modules.order文件中查找所有的KO文件
    - 使用modpost，为每个KO模块创建module.mod.c文件
    - 创建modules.symvers文件，保存导出的符号及CRC值
    - 将module.o和module.mod.o链接生成module.ko
  - 步骤03
    - 生成和内核模块相关的信息：版本魔幻数
    - License、version、alias

• 模块编译对应的Makefile

scripts/Makefile.modpost：

```
PHONY := __modpost
__modpost:
__modpost: $(output-symdump)
	$(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modfinal
```

scripts/Makefile.modfinal：

```
PHONY := __modfinal
__modfinal:
__modfinal: $(modules)

# find all .ko modules listed in modules.order
modules := $(sort $(shell cat $(MODORDER)))
```

```
# cat modules.order
drivers/char/hello.ko
```

• 模块编译对应的Makefile

scripts/Makefile.modfinal：

```
$(modules): %.ko :  %.o %.mod.o scripts/module.lds FORCE
        +$(call if_changed,ld_ko_o)

cmd_ld_ko_o =                                                   \
$(LD) -r $(KBUILD_LDFLAGS) $(KBUILD_LDFLAGS_MODULE) $(LDFLAGS_MODULE) \
        -T scripts/module.lds -o $@ $(filter %.o, $^);                  \
        $(if $(ARCH_POSTLINK), $(MAKE) -f $(ARCH_POSTLINK) $@, true)
```

22 modules_install过程分析

• 模块安装信息

```
root@pc:/home/linux-5.10.4# make modules_install
  INSTALL drivers/char/hello.ko
  INSTALL fs/nfs/flexfilelayout/nfs_layout_flexfiles.ko
  DEPMOD  5.10.4
```

```
root@pc:/usr/lib/modules/5.10.4# tree
.
├── build -> /home/linux-5.10.4
├── kernel
│   ├── drivers
│   │   └── char
│   │       └── hello.ko
│   └── fs
│       └── nfs
│           └── flexfilelayout
│               └── nfs_layout_flexfiles.ko
```

• 模块安装对应的Makefile

Makefile：

```
modules_install:  _emodinst_  _emodinst_post
_emodinst_:
        $(Q)mkdir -p $(MODLIB)/$(install-dir)
        $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modinst

_emodinst_post: _emodinst_
        $(call cmd,depmod)
```

• 模块安装对应的Makefile

scripts/Makefile.modinst：

```
PHONY := __modinst
__modinst:
__modinst: $(modules)
modules := $(sort $(shell cat $(if $(KBUILD_EXTMOD),$(KBUILD_EXTMOD)/)modules.order))

$(modules):
	$(call cmd,modules_install,$(MODLIB)/$(modinst_dir))

modinst_dir = $(if $(KBUILD_EXTMOD),$(ext-mod-dir),kernel/$(@D))

cmd_modules_install = \
   mkdir -p $(2) ; \
   cp $@ $(2) ; \
   $(mod_strip_cmd) $(2)/$(notdir $@) ; \
   $(mod_sign_cmd) $(2)/$(notdir $@) $(patsubst %,|| true,$(KBUILD_EXTMOD)) ;
\
   $(mod_compress_cmd) $(2)/$(notdir $@)
```

23 headers_install过程分析

• 目标header对应的规则

Makefile：

```
headers: $(version_h) scripts_unifdef uapi-asm-generic archheaders archscripts
        $(if $(wildcard $(srctree)/arch/$(SRCARCH)/include/uapi/asm/Kbuild),, \
        $(error Headers not exportable for the $(SRCARCH) architecture))
        $(Q)$(MAKE) $(hdr-inst)=include/uapi
        $(Q)$(MAKE) $(hdr-inst)=arch/$(SRCARCH)/include/uapi
```

其中：

```
hdr-inst := -f $(srctree)/scripts/Makefile.headersinst  obj
```

简化一下

```
headers: $(version_h)  scripts_unifdef  uapi-asm-generic  archheaders  archscripts
        make -f  scripts/Makefile.headersinst obj=include/uapi
        make -f  scripts/Makefile.headersinst obj=arch/arm/include/uapi
```

• 目标header对应的Makefile

scripts/Makefile.headersinst：

```
PHONY := __headers
__headers:
__headers: $(all-headers)
        $(call cmd,remove)
all-headers := $(src-headers) $(gen-headers)
src-headers := $(if $(src-subdirs), $(shell cd $(src) && find $(src-subdirs) -name '*.h'))
```

其中：

```
src = include/uapi
src-headers = include/uapi/$(src-subdirs)/*.h
src-headers := $(filter-out $(no-export-headers), $(src-headers))
```

```
src-headers := include/uapi/asm-generic/*.h include/uapi/linux/*.h
include/uapi/sound/*.h …
```

- src-headers对应的规则

```
$(src-headers): $(dst)/%.h: $(src)/%.h $(srctree)/scripts/headers_install.sh FORCE
        $(call if_changed,install)
```

其中：

src = include/uapi

dst := usr/include

cmd_install = $(CONFIG_SHELL) $(srctree)/scripts/headers_install.sh $< $@

• gen-headers对应的规则

scripts/Makefile.headersinst:

```
gen := $(objtree)/$(subst include/,include/generated/,$(obj))
gen-headers := $(if $(gen-subdirs), $(shell cd $(gen) && find $(gen-subdirs) -name '*.h'))

$(gen-headers): $(dst)/%.h: $(gen)/%.h $(srctree)/scripts/headers_install.sh FORCE
        $(call if_changed,install)
```

其中：

gen = include/generated/uapi

dst := usr/include

cmd_install = $(CONFIG_SHELL) $(srctree)/scripts/headers_install.sh $< $@

24 内核中的空头文件探秘

- 再回首
  - .config生成的三个主要文件
  - include/config/auto.conf：用来配置Makefile
  - include/generated/autoconf.h：在C程序中引用
  - include/config/*.h：

- Kbuild Makefile
  - 跟踪三种依赖关系：
  - 编译所需要的所有源文件：*.c
  - 源文件.c中包含的各种头文件：*.h
  - 所有程序中使用的配置选项：CONFIG_XXX

```
#include <xx.h>
#ifdef CONFIG_SMP
__boot_cpu_id = cpu;
#endif
```

• 头文件依赖

```
hello.o : hello.c---hello.d
        gcc –c hello.c –o hello.o
```

scripts/Makefile.lib：

```
c_flags      = -Wp,-MD,$(depfile) $(NOSTDINC_FLAGS) $(LINUXINCLUDE)     \
            -include $(srctree)/include/linux/compiler_types.h       \
            $(__c_flags) $(modkern_cflags)                          \
            $(basename_flags) $(modname_flags)
```

.hello.o.cmd：

```
deps_drivers/char/hello.o := \
   $(wildcard include/config/smp.h) \
   $(wildcard include/config/wanglitao.h) \
  include/linux/kconfig.h \
   $(wildcard include/config/cc/version/text.h)
            …
 drivers/char/pi.h \

drivers/char/hello.o: $(deps_drivers/char/hello.o)
```

• 配置变量依赖

Makefile.build：

```
$(obj)/%.o: $(src)/%.c $(recordmcount_source) $(objtool_dep) FORCE
	$(call if_changed_rule,cc_o_c)
-include $(foreach f,$(existing-targets),$(dir $(f)).$(notdir $(f)).cmd)

existing-targets := $(wildcard $(sort $(targets)))
targets += $(targets-for-builtin) $(targets-for-modules)
```

Kbuild.include：

```
# Execute the command and also postprocess generated .d dependencies file.
if_changed_dep = $(if $(newer-prereqs)$(cmd-check),$(cmd_and_fixdep),@:)
cmd_and_fixdep =                                                \
	$(cmd);                                                 \
	scripts/basic/fixdep $(depfile) $@ '$(make-cmd)' > $(dot-target).cmd;\
	rm -f $(depfile)

depfile = $(subst $(comma),_,$(dot-target).d) # depfile保存gcc -MD生成的依赖文件
# Name of target with a '.' as filename prefix. foo/bar.o => foo/.bar.o
dot-target = $(dir $@).$(notdir $@)
```

宅学部落

专注嵌入式、Linux精品教程

更多信息

王利涛老师个人店：https://wanglitao.taobao.com/
嵌入式在线教程网：www.zhaixue.cc
嵌入式技术交流群：
宅学部落02群：398294860
宅学部落03群：559671596
宅学部落04群：528718820

欢迎关注公众号：