kbuild实现分析

x.yin@hotmail.com

Jun 1, 2009

目录

1 前言

人们在很多电影和小说中常常会见到这样的角色:一个受人轻视的落魄小人物，起初别人只是丢给他几个琐碎的任务, 不经意间，他却成了掌控整个局势的显赫人物。最著名的莫过于圣经里的约瑟(joseph), 就连名噪一时的“肖申克的救赎”(The Shawshank Redemption)中的男主角 Andy Dufresne 身上都闪耀着 joseph 的影子（当然还有摩西)。make 在一个复杂项目中扮演的角色，也大体类似，从简单琐碎被人忽视开始，到控制项目中的每个细节结束。 kbuild 之于 kernel 也正是如此。

当 kbuild 本身也成为一个复杂的系统 (big monster) 时，分析和了解 kbuild 也就有了更多的现实意义。不同的用户对 kbuild 的需求不同, 相应地对 kbuild 需要了解的程度也就不同。

- 对于普通用户(编个 kernel image 之类)而言，kbuild只是意味着一些 targets(make menuconfig; make) ， make help 足以帮助他们搞定一切，他们不需要了解除此之外的任何东西，就像吃个鸡蛋并不需要了解蛋是如何长出来的。
- 对于普通的开发者(kernel feature, driver)或者平台 porting 的开发者而言， kbuild 也只是意味着一些实现友好的接口，他们需要对某些实现细节(如何使用 obj-y，obj-m)略知一二，但是并不需要关心为什么这么做(why)，只需要知道怎么去做(how to do it)。Documentaton/kbuild/文档可以满足以上绝大部分人的需求。
- 然而，对于那些好奇心特别重，想知道 Linux 世界为什么如此奇妙的人来说，或者对于那些真正需要自己设计 building system 想从 kbuild 借鉴点什么的人来说，基本上没有比 RTFSC(Read The Fucking Source Code) 更好的选择。

关于 user 和 kbuild 的关系，详见[4, makefile.txt, Section2 ,who does what]

2 概述

本文主要侧重于 kbuild 的实现分析，希望能从一个 building system 设计者的角度来更好地了解 kbuild 的实现和背后的设计思想。本文的主要内容大致可分为5大部分：

- Part1 . chapter1, kbuild 的架构和各个部分的简介。
- Part2 . chapter2, kbuild 常用到的 makefile 基础知识，了解这部分有助于我们对 kbuild 具体实现的分析, 事实上完整通读并理解了 make info page 的人完全可以忽略这一部分。
- Part3 . kbuild 主要功能的分析。我们知道，一份代码运行的时候更多得是以立体的方式表现出来的, 而平面的逐行的注释很难清楚地解释其全部功能，因此，我们采用得是从功能的角度来解释其实现，这一部分分为两章:
  - Chapter3 . 主要分析 kbuild 提供的各类 targets 实现，基本上不涉及 kbuild 规则的实现
  - chapter4 . 将以源代码树下的具体 Makefile 为分析对象，详细分析 kbuild 的各种规则文件.
- Part4 . chapter5, kbuild 专题。单纯讲述 kbuild 而不涉及到一些专题是不可能的，如依赖关系生成，模块版本支持(CONFIG_MODULEVERSION)， kallsyms, bzImage, relocatable kernel 等，这些都和 kbuild 紧密联系在一起. 这些主题同时也会涉及到一些工具程序： fixdep, modpost, kallsyms, relocs 等。
- Part5 . chapter6 , 一些 kbuild 设计思想的总结以及自2.4系列以来的改进。

本文所分析的 kbuild kernel 版本为 2.6.23.1 ， GNU make 版本为 3.81 .

Part1

3 kbuild简介

我们知道，对于一个复杂的大型项目来说，build的过程大致可分为三个阶段:

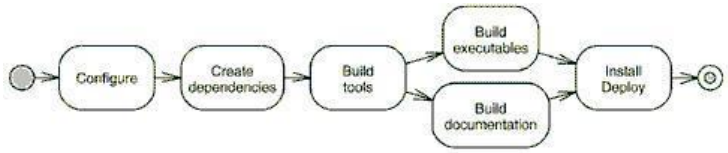

图 1: The steps of a typical build process

- 配置阶段。即首先必须确定要编译的平台，目标(targets)和各种 feature 。
- 编译阶段。系统根据提供的编译工具(build tools)，建立依赖关系图(dependency graphy)，确定正确的目标执行顺序, 然后执行需要更新的命令, 最后构建相应的目标。
- Deploy 阶段。对于kernel而言有可能涉及到模块及头文件的安装。而对于很多嵌入式系统而言，则涉及到 rootfs 或整个 flash image 的构建。

同样， kernel building system 也有相同的三个阶段。

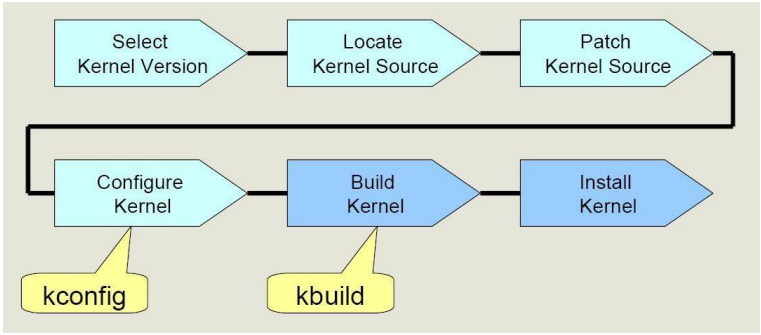

图 2: 编译kernel的典型步骤

kconfig 对应的就是 configuration 阶段，当 .config 生成后， kbuild 会依据 .config 编译指定的目标，最后需要安装模块或头文件，或 kernel image 时，可以通过 make [modinstd | headerinst |install] 实现。

3.1 kconfig

3.1.1 kconfig的结构

每个平台下都有一个 Kconfig, Kconfig 又通过 source 构建出一个 Kconfig 树。当 make %config 时， scripts/kconfig 中的工具程序 conf/mconf/qconf 负责对 Kconfig 的解析。

arch/i386/Kconfig

```
mainmenu "Linux Kernel Configuration"

config X86_32
bool
default y
help
  This is Linux's home port.  Linux was originally native to the Intel
  386, and runs on all the later x86 processors including the Intel
  486, 586, Pentiums, and various instruction-set-compatible chips by
  AMD, Cyrix, and others.
... ...

source "init/Kconfig"

menu "Processor type and features"

source "kernel/time/Kconfig"

... ...

config KTIME_SCALAR
bool
default y
```

Kconfig文件说明:

scripts/kconfig/*	kconfig解析程序
kconfig	各个内核源代码目录中配置文件
arch/$(ARCH)/defconfig	arch缺省配置文件

3.1.2 kconfig language

参见kernel文档：Documentation/kbuild/kconfig-language.txt

3.1.3 kconfig的解析

conf/mconf/gconf/qconf ，引入了 gperf, flex, bison 等对 kconfig 进行词法和语法分析，替代了 2.4 中的 shell, perl 脚本程序。这些分析程序与 kbuild 主题关系不大，也就不必赘述。

3.2 kbuild

3.2.1 kbuild组成

kbuid的Makefile主要有5个部分:

- 顶层Makefile：负责对各类target的分类并调用相应的规则Makefile来生成目标
- 包含用户配置选项的.config
- 具体平台相关的Makefile：arch/$(ARCH)/Makefile
- 各类规则文件:scripts/Makefile.*

- kernel目录树下的具体Makefile：Makefiles

顶层 Makefile 和 Makefile.build, Makefile.modpos, Makefile.clean 等都是相互独立的，每个 Makefile 都包含很多文件，图3是 Makefile 的结构图，而图4则是使用频率最高的也是最典型的规则 Makefile.build 结构图。

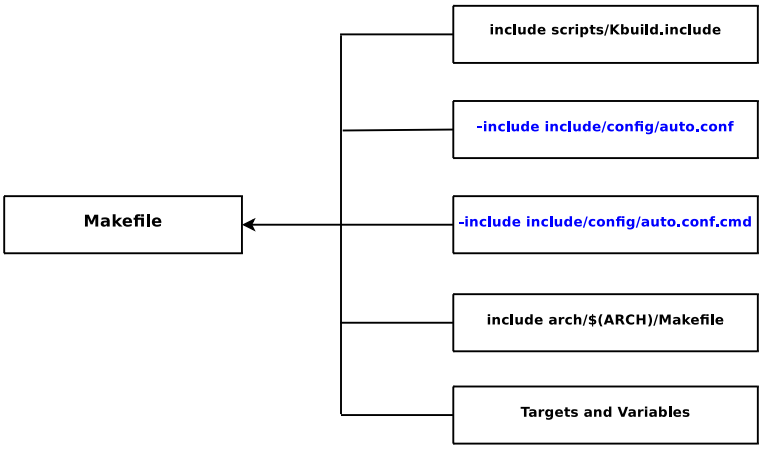

图 3: Makefile结构

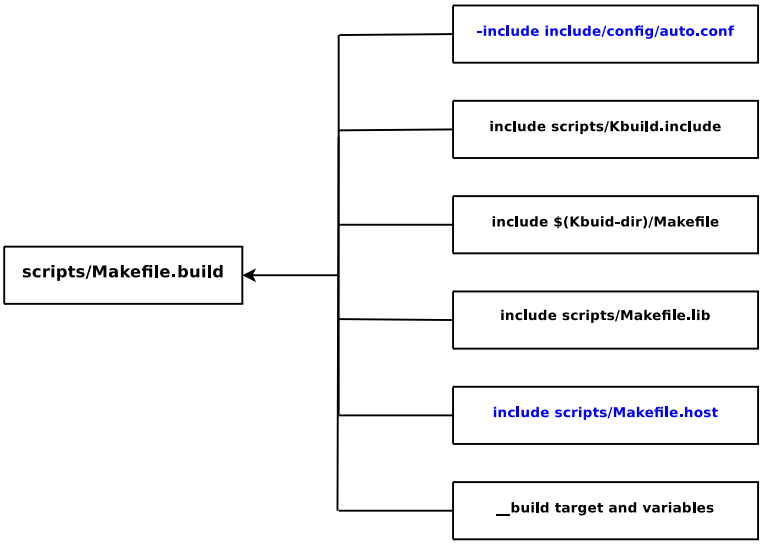

图 4: scripts/Makefile.build结构

3.2.2 kbuild文件功能说明

文件说明

- 顶层Makefile:

Name	Description
Makefile	顶层Makefile,主要提供各类targets的接口,一般并不涉及实现
kbuild	顶层kbuild，生成asm-offset.h并检查缺失的系统调用

- 平台相关Makefile:

Name	Description
arch/$(ARCH)/Makefile	具体架构的Makefile

- scripts目录下的编译规则文件:

Name	Description
kbuild.include	共用的定义。会被所有独立的Makefile包括,如Makefile.build等
Makefile.build	提供编译built-in.o, lib.a等规则的Makefile
Makefile.lib	负责归类分析obj-y obj-m和其中的目录subdir-ym
Makefile.host	hostprog编译规则
Makefile.clean	clean规则
Makefile.headerinst	头文件install规则
Makefile.modinst	模块install规则
Makefile.modpost	模块编译的第二阶段,由.o和.mod生成<module>.ko

Part2

4 kbuid中用到的主要make知识

关于 make 最权威最完整的文档，莫过于[1，GNU make manual]，你能在那里找到所有你想知道的事。本文的主要目的是解读 kbuild 的，而不是一次完整的中文翻译 GNU make manual ([2，GNU make infopage 中文翻译])。因此本章只列出 kbuild 会用到但是现实中的普通 makefile 很少涉及到的主题。本节将以从整体到细节到的方式叙述，首先是 Makefile 概述，其次是 Makefile 的执行过程，最后是 Makefile 中所涉及的细节：规则，变量，命令，函数等。这种叙述方式的好处是能够更好地帮助读者建立一个对 Makefile 的完整印象。

4.1 Makefile概述

make 在执行时，需要一个命名为 Makefile 的文件。在一个完整的 Makefile 中，一般包含了5部分：规则(显示指定和隐含规则)、变量定义(同样包含显示变量和隐含变量)、指示符(include, define 等)和注释。 Makefile 的规则(rules)语法如下:

```
TARGET: PREREQUISITES
[TAB]COMMOAND
```

当规则中的 PREREQUISITES 比 TARGET 新时， make 会执行规则的 COMMOAND 。 make 中的规则分为显示规则和隐含规则。显示规则是由作者显示写出的规则，而隐含规则则是内建在 make 中，为 make 提供了重建某一类目标文件 (.o 等)的通用方法，同时这些隐含规则所用到的变量也就是所谓的隐含变量。隐含规则和隐含变量可以通过 make -p -f /dev/null 查看。隐含规则的好处是在 Makefile 中不需要明确给出重建某一个目标的命令，甚至可以不需要规则。 make 会为你自动搜寻匹配的隐含规则链。例如：

```
1 x:y.o z.o
```

当 x.c, y.c 和 z.c 都存在时，规则执行如下命令：

```
cc -c x.c -o x.o
cc -c y.c -o y.o
cc -c z.c -o z.o
cc x.o y.o z.o -o x
rm -f x.o
rm -f y.o
rm -f z.o
```

但是当 y.c 不存在时，创建文件 y.o 有很多种可能，系统必须为其创建一个可能的规则链:

```
y.y->y.c->y.o(yacc)
y.l->y.c->y.o(lex)
y.p->y.o(Pascal程序)
```

当存在多种可能时，系统必须这个搜索所有匹配的隐含规则，例如本系统中存在 y.y 时就会首先执行 “yacc” 生成文件 “y.c” ，之后由编译器将 “y.c” 编译成为 “y.o” .

这个世界上没有免费的午餐，隐含规则的代价之一就是低效，系统必须搜索可能的隐含规则链。同时隐含规则也有可能应用了不是你想要的规则而引入很难 debug 的错误。这就是为什么 kbuild 在顶层 Makefile 中首先禁用隐含规则和隐含变量的原因：

```
# Do not:
# o  use make's built-in rules and variables
#     (this increases performance and avoid hard-to-debug
    behavour);
# o  print "Entering directory ...";
MAKEFLAGS += -rR --no-print-directory
```

2.6.23.1 版本中顶层 Makefile 禁止了隐含规则，尽管如此， make 的隐含规则还是会出现一些不影响使用的 bug :

```
$ make menuconfig
$ make oldlinux vmlinux V=1 2>&1 |tee build.log
```

你会惊讶的发现 build.log 的第一行:

```
make -C /tmp/linux-2.6.23.1 KBUILD_SRC=
    /tmp/linux-2.6.23.1/scripts/kbuild.include
```

scripts/kbuild.include 居然也会作为规则之一出现，这条规则会出错但是不会影响结果.

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

原因就是

```
# Cancel implicit rules on top Makefile, `-rR' will apply to
    sub-makes.
Makefile: ;
```

在 2.6.23.1 中几乎位于 Makefile 的结尾.

而在 2.6.30 中

```
# Cancel implicit rules on top Makefile
$(CURDIR)/Makefile Makefile: ;
```

基本上位于 Makefile 最前面，上面的由隐含规则引发的错误就消失了。但是你用 make -p 依然能够发现 Makefile 会为 arch/$(ARCH)/Makefile 尝试各种隐含规则，这个并不影响使用。

4.2 Makefile的执行过程

GUN make的执行过程分为两个阶段。

- 第一阶段：读取所有的 makefile 文件，内建所有的变量、明确规则和隐含规则，并建立所有目标和依赖之间的依赖关系结构链表。
  1. 依次读取变量 “MAKEFILES” 定义的 makefile 文件列表。
  2. 读取工作目录下的 makefile 文件，依据文件名依次查找： GNUmakefile, makefile, Makefile ， 首先找到哪个就读取哪个。
  3. 依次读取工作目录 makefile 文件中使用指示符 “include” 包含的文件。
  4. 查找重建所有已读取的 makefile 文件的规则（如果存在一个目标(target)是当前读取的某一个 makefile 文件，则执行此规则重建此 makefile 文件，完成以后从第一步开始重新执行)。

5. 初始化变量值并展开那些需要立即展开的变量和函数并根据预设条件确定执行分支。

- 第二阶段：根据第一阶段已经建立的依赖关系结构链表和最终目标决定哪些目标需要更新，并使用对应的规则来重建这些目标。

1. 根据最终目标(缺省的或指定的）以及依赖关系递归建立依赖关系链表。
2. 执行最终目标的依赖关系链表(和建立依赖关系链表的顺序相反)。
3. 执行最终目标所在的规则。

理解 make 执行过程的两个阶段能帮助我们更深入地了解执行过程中变量以及函数是如何被展开的。关于变量的定义和展开请参考[1, Section 6]，这不是我们关注的重点。我们可以通过一个 kbuild 顶层 Makefile 的例子来加深 Makefile 的两阶段执行过程的理解:

限于篇幅，删除了注释

```
ifeq ($(dot-config),1)
-include include/config/auto.conf

ifeq ($(KBUILD_EXTMOD),)
-include include/config/auto.conf.cmd

$(KCONFIG_CONFIG) include/config/auto.conf.cmd: ;

include/config/auto.conf: $(KCONFIG_CONFIG) include/config/
   auto.conf.cmd
        $(Q)$(MAKE) -f $(srctree)/Makefile silentoldconfig
else
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
include/config/auto.conf: ;
endif # $(dot-config)
```

我们注意到当 make 不带任何 target 缺省编译时: dot-config := 1 ; KBUILD_EXTMOD="" ; 系统会编译 vmlinux，bzImage(i386) 和 modules 。

假设系统中 .config 生成后， make 第一次运行，此时系统中 include/config/auto.conf 和 include/include/config/auto.conf.cmd 根本不存在，由于所用命令形式为：

```
-include include/config/auto.conf
-include include/config/auto.conf.cmd
```

所以系统直接忽略且并不报错。问题是整个 Makefile 中随后又用到了很多 CONFIG_ 选项(CONFIG_MODULES, CONFIG_LOCALVERSION_AUTO 等)，而这些 CONFIG_XXX 正是包含在 include/config/auto.conf 中的，那这些选项到底又是如何起作用的呢？

这就涉及到了 Makefile 第一阶段的 Step4-Makefile 的重建([1, Section 3.7 Remaking Makefiles]). make 在读入所有 makefile 文件之后，首先将所读取的每个 makefile 作为一个目标，寻找更新它们的规则。如果存在一个更新某一个 makefile 文件明确规则或者隐含规则，就去更新对应的 makefile 文件。完成对所有的 makefile 文件的更新之后，如果之前所读取的任何一个 makefile 文件(被 include 的之类)被更新，那么 make 就清除本次执行的状态重新读取一遍所有的 makefile 文件(此过程中，同样在读取完成以后也会去试图更新所有的已经读取的 makefile 文件，但是一般这些文件不会再次被重建，因为它们在时间戳上已经是最新的)。读取完成以后再开始解析已经读取的 makefile 文件并开始执行必要的动作。

尽管 include/config/auto.conf 在第一次执行时并不存在，但是由于存在更新 auto.conf 文件的规则:

```
include/config/auto.conf: $(KCONFIG_CONFIG) include/config/
    auto.conf.cmd
	$(Q)$(MAKE) -f $(srctree)/Makefile silentoldconfig
```

注意make调用的是顶层 Makefile 本身， target 是 silentoldconfig ，奇怪得是 Makefile 并未定义 silentoldconfig ，(%config 规则)，这个将会在随后的章节里5.2详述， silientoldconfig 会产生:

- include/config/auto.conf
- inlcude/config/auto.conf.cmd
- include/config/*.h, 用于fixdep
- include/linux/autoconf.h

所以 Makefile 会重置本次执行的所有状态并重新读取一遍所有的 makefile 。注意，同样在读取完成以后 make 又会去试图更新所有的已经读取的 makefile 文件，如果 includ/config/auto.conf 由于某种原因没有生成，无限循环就不可避免了。关于Makefile的remake 过程，在7.1.2你会看到一个类似的例子。

4.3 规则

4.3.1 伪目标.PHONY

伪目标是这样一个目标：它不代表一个真正的文件名，在执行 make 时可以指定这个目标来执行其所在规则定义的命令。使用伪目标主要有两点原因：

1. 避免目标和文件名重名而使我们的期望失败（此目标的目的为了执行执行一系列命令，而不是创建这个目标）上例中当 dot-config=0, KBUILD_EXTMOD 不为空时，会执行:

```
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;
                          \
```

inlcude/config/auto.conf 作为 .PHONY 命令，会强制执行 include/config/auto.conf 的命令。否则，当 include/config/auto.conf 文件存在时并不会执行命令体。

2. 提高执行make时的效率. 当一个目标被声明为伪目标后， make 在执行此规则时不会试图去查找隐含规则来创建它。对于一个复杂项目而言，这样也提高了编译的整体执行效率.

一般情况下，一个伪目标不作为另外一个目标的依赖。然而当伪目标作为某个目标的依赖时，其命令体都会被强制执行。然而 kbuild 中这样的例子却有很多:

Makefile.build

```
# Built-in and composite module parts
$(obj)/%.o: $(src)/%.c FORCE
        $(call cmd,force_checksrc)
        $(call if_changed_rule,cc_o_c)

# Add FORCE to the prequisites of a target to force it to be
    always rebuilt.
# ---------------------------------------------------------

PHONY += FORCE

FORCE:

.PHONY: $(PHONY)
```

这个例子中， FORCE 是一个 .PHONY 目标，在执行此规则时，目标总会被认为是最新的。而且， FORCE 作为一个规则也没有命令或者依赖，并且它的目标也不是一个存在的文件名，所以也是一个强制目标([1, 4.7 Rules without Commands or Prerequisites]). 也就是说：这个规则一旦被执行， make 就认为它的目标已经被更新过。这样的目标在作为一个规则的依赖时，因为依赖总被认为被更新过，因此作为依赖所在的规则中定义的命令总会被执行。

有人可能会很奇怪，既然 .o 和 .c 的依赖关系已经建立起来了，那为什么不通过依赖关系 (.o .d : .c) 自动更新 .o 而非要强制执行呢？问题的关键是 kbuild 编译命令选项的改变并不能像依赖关系规则一样直接影响 .o 的更新。 if_changed_rule 的作用就是既检查依赖的更新也检查命令行参数的更新。

4.3.2 多规则目标

Makefile 中，一个文件可以作为多个规则的目标。这种情况下，以这个文件为目标的规则的所有依赖文件将会被合并成此目标的一个依赖文件列表，当其中任何一个依赖文件比目标更新（比较目标文件和依赖文件的时间戳）时， make 将会执行特定的命令来重建这个目标。当然，多个规则中最多只能有一个规则定义命令，如果出现了多个规则定义了命令，那么将使用最后一个规则定义的命令并且发出警告信息。 kbuild 中多规则目标最典型的就是 _all ， 当make不带任何目标缺省编译时将使用 _all 作为其缺省目标:

```
Makefile
# That's our default target when none is given on the command line
PHONY := _all
_all:

# If building an external module we do not care about the all: rule
# but instead _all depend on modules
PHONY += all
ifeq ($(KBUILD_EXTMOD),)
_all: all
else
_all: modules
endif

# The all: target is the default when no target is given on the
# command line.
# This allow a user to issue only 'make' to build a kernel including modules
# Defaults vmlinux but it is usually overridden in the arch makefile
all: vmlinux

ifdef CONFIG_MODULES

# By default, build modules as well

all: modules

arch/i386/Makefile
all: bzImage
```

这个例子中 _all 作为缺省目标，而 all 作为 _all 的依赖， all 的依赖目标则为 vmlinux bzImage modules ，所以最终 make 会生成3个 target:vmlinux bzImage modules .

4.3.3 静态模式(Static Pattern rules)

所谓静态模式规则就是：规则存在多个目标，并且不同的目标可以根据目标文件的名字来自动构造出依赖文件。静态模式规则比多目标规则更通用，它不需要多个目标具有相同的依赖。但是静态模式规则中的依赖文件必须是相类似的而不是完全相同的(符合模式)。

静态模式的语法:

```
TARGETS ...: TARGET-PATTERN: PREREQ-PATTERNS ...
```

```
COMMANDS
...
```

TAGETS列出了此规则的一系列目标文件。像普通规则的目标一样可以包含通配符。

TAGET-PATTERN 和 PREREQ-PATTERNS 说明了如何为每一个目标文件生成依赖文件。从目标模式 (TAGET-PATTERN) 的目标名字中抽取一部分字符串（称为“茎”）。使用“茎”替代依赖模式 (PREREQ-PATTERNS) 中的相应部分来产生对应目标的依赖文件。需要明确的一点是：在模式规则的依赖列表中使用不包含模式字符“%”也是合法的。代表这个文件是所有目标的依赖文件.我们来看一个 kbuild 中 static pattern 例子:

Makefile.build

```
$(single-used-m): $(obj)/%.o: $(src)/%.c FORCE
        $(call cmd,force_checksrc)
        $(call if_changed_rule,cc_o_c)
        @{ echo $(@:.o=.ko); echo $@; } > $(MODVERDIR)/$(@F:.o
            =.mod)
```

以 driver/net/Makefile 中的几个编译成 module 的文件为例： sing-used-m := 3c509.o e100.o ，此时静态规则会为 3c508.o e100.o 分别产生对应的规则：

```
$(obj)/3c509.o : $(src)/3c509.c FORCE
        $(call cmd,force_checksrc)
        $(call if_changed_rule,cc_o_c)
        @{ echo $(@:.o=.ko); echo $@; } > $(MODVERDIR)/$(@F:.o
            =.mod)
$(obj)/e100.o : $(src)/e100.c FORCE
        $(call cmd,force_checksrc)
        $(call if_changed_rule,cc_o_c)
        @{ echo $(@:.o=.ko); echo $@; } > $(MODVERDIR)/$(@F:.o
            =.mod)
```

同时应该注意的是:在使用静态模式规则时，指定的目标必须和目标模式相匹配，否则执行 make 时将会得到一个错误提示。如果存在一个文件列表，其中一部分符合某一种模式而另外一部分符合另外一种模式，这种情况下我们可以使用 “filter” 函数来对这个文件列表进行分类，在分类之后对确定的某一类使用模式规则。

4.3.4 ::规则

双冒号规则就是使用 “::” 代替普通规则的 “:” 得到的规则。当同一个文件作为多个规则的目标时，双冒号规则的处理和普通规则的处理过程完全不同（双冒号规则允许在多个规则中为同一个目标指定不同的重建目标的命令）。 Makefile 中，一个目标可以出现在多个规则中。但是这些规则必须是同一类型的规则，要么都是普通规则，要么都是双冒号规则。而不允许一个目标同时出现在两种不同类型的规则中。双冒号规则和普通规则的处理的不同点表现在以下几个方面：

1. 双冒号规则中，当依赖文件比目标更新时。规则将会被执行。对于一个没有依赖而只有命令行的双冒号规则，当引用此目标时，规则的命令将会被无条件执行。而普通规则，当规则的目标文件存在时，此规则的命令永远不会被执行（目标文件永远是最新的）。

2. 当同一个文件作为多个双冒号规则的目标时。这些不同的规则会被独立的处理，而不是像普通规则那样合并所有的依赖到一个目标文件。这就意味着对这些规则

的处理就像多个不同的普通规则一样。就是说多个双冒号规则中的每一个的依赖文件被改变之后，make只执行此规则定义的命令，而其它的以这个文件作为目标的双冒号规则将不会被执行。当同一个目标出现在多个双冒号规则中时，规则的执行顺序和普通规则的执行顺序一样，按照其在 Makefile 中的书写顺序执行。

看一个简单的例子:

```
Newprog :: foo.c
	$(CC) $(CFLAGS) $< -o $@
Newprog :: bar.c
	$(CC) $(CFLAGS) $< -o $@
```

如果 “foo.c” 文件被修改，执行 make 以后将根据 “foo.c” 文件重建目标 “Newprog” 。而如果 “bar.c” 被修改那么 “Newprog” 将根据 “bar.c” 被重建。

kbuild 只在顶层 Makefle 中用到了一次 :: 规则。

Makefile

```
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
```

本例中 %:: 作为 pattern 规则: 当没有显示指定目标时 $(MAKECMDGOALS) 为空，%则等同于缺省目标及其所有依赖目标关系链. 如果 MAKECMDGOALS 不为空，则等同于 $(MAKECMDGOALS):: FORCE

事实上，只有当 make %config 加入其它目标如 vmlinux 时(make oldconfig vmlinux)， mixed-targets 才会为1. 此时 %:: 将被解释为 oldconfig:: FORCE 和 vmlinux:: FORCE ，而且由于其依赖为 .PHONY FORCE 命令，所以 olconfig 和 vmlinux 的命令每次都将被执行.

4.4 命令和变量

4.4.1 命令回显

通常， make 在执行命令行之前会把要执行的命令行输出到标准输出设备。我们称之为“回显”，就好像我们在 shell 环境下输入命令执行时一样。但是，如果规则的命令行以字符 “@” 开始，则make在执行这个命令时就不会回显这个将要被执行的命令。典型的用法是在使用 “echo” 命令输出一些信息时。如 kbuild 顶层 Makefile 的例子：

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

PHONY += $(clean-dirs) clean
$(clean-dirs):
```

```
11          $(Q)$(MAKE) $(clean)=$(patsubst _clean_%,%,$@)
```

当KBUILD_VERBOSE不为1时，上例中的命令展开为:

```
1 $(clean-dirs):
2         @$(MAKE) $(clean)=$(patsubst _clean_%,%,$@)
```

4.4.2 命令的执行

make 通常会在规则中的命令在运行结束后，检测命令执行的返回状态，如果返回成功，那么就启动另外一个子 shell 来执行下一条命令。规则中的所有命令执行完成之后，这个规则就执行完成了。如果一个规则中的某一个命令出错(返回非0状态)， make 就会放弃对当前规则后续命令的执行，也有可能会终止所有规则的执行。在某些情况下，规则中一个命令的执行失败并不代表整个规则执行的错误。为了忽略一些无关命令执行失败的情况，我们可以在命令之前加一个减号“-”(在 [Tab] 字符之后)，来告诉 make 忽略此命令的执行失败。例如对于 “clean” 目标我们就可以这么写：

```
1 clean:
2         -rm  *.o
```

其含义是：即使执行 “rm” 除文件失败， make 也继续执行。

但是我们也常常能够看见命令行前除了 “-” 外，还能看到 “+” 。

Makefile.clean

```
1  __clean: $(subdir-ymn)
2  ifneq ($(strip $(__clean-files)),)
3          +$(call cmd,clean)
4  endif
5  ifneq ($(strip $(__clean-dirs)),)
6          +$(call cmd,cleandir)
7  endif
8  ifneq ($(strip $(clean-rule)),)
9          +$(clean-rule)
10 endif
11         @:
```

“+” 意味着: make 的命令行选项 -n(–just-print), -q(–question), -t(–touch) 并不影响之前带 “+” 号的命令行选项，另外他们也不影响包含 “$(MAKE)” 命令行的执行.也就是说即便 `make -[nqt] clean` ，该删的还是删了

另外，我们在 kbuild 的 Makefile 里也常常能够看到， set -e :

kbuild.include

```
1 try-run = $(shell set -e;                \
2         TMP="$(TMPOUT).$$$$.tmp";        \
3         if ($(1)) >/dev/null 2>&1;       \
4         then echo "$(2)";                \
5         else echo "$(3)";                \
6         fi;                              \
7         rm -f "$$TMP")
```

set -e 命令设置当前Shell进程为这样的状态：如果它执行的任何一条命令的退出状态非零则立刻终止，不再执行后续命令。如果该命令没有加-， make 会立即退出。有

人会很奇怪make缺省行为就是如此，为何还加上 set -e 呢？区别就是即便 make -k 时，“set -e” 也会导致 make 立即退出，而不是接着执行 (keep going) 。

4.4.3 定义命令包(Defining Canned Command Sequences)

Makefile 中，可能有多个规则会使用相同的一组命令。你可以使用指示符 “define” 来定义这样一组命令作为一个变量，在需要的规则中通过变量名直接引用。以 Makefile.build 为例：

```
define rule_cc_o_c
        $(call echo-cmd,checksrc) $(cmd_checksrc)
                                  \
        $(call echo-cmd,cc_o_c) $(cmd_cc_o_c);
                                  \
        $(cmd_modversions)
                                  \
        scripts/basic/fixdep $(depfile) $@ '$(call make-cmd,
            cc_o_c)' >    \
                                      $(dot-target).tmp;
                                                          \
        rm -f $(depfile);
                                  \
        mv -f $(dot-target).tmp $(dot-target).cmd
endef

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

其中 if_changed_rule 定义于 kbuild.include :

```
# Built-in and composite module parts
if_changed_rule = $(if $(strip $(any-prereq) $(arg-check) ),
                       \
        @set -e;

            \
        $(rule_$(1)))
```

上例中的 $(call if_changed_rule, cc_o_c) 最终会展开为 @set -e; $(rule_cc_o_c), 此时的 rule_cc_o_c 的用法和普通的变量没有任何区别。

4.4.4 变量的替换引用

关于变量的替换引用，详见([1, Section 6.3.1 Substitution References])。对于一个已经定义的变量，可以使用“替换引用”将其值中的后缀字符(串)使用指定的字符(字符串)替换。

```
$(modules:.ko=.mod.o): %.mod.o: %.mod.c FORCE
        $(call if_changed_dep,cc_o_c)
```

当 modules := ext2.o ext3.o ，此时 $(modules:.ko=.mod.o) 就等于 ext2.mod.o ext3.mod.o . 变量的替换引用其实是函数 patsubst 的一个简化实现. 例如，本例中的 $(modules:.ko=.mod.o) 等同于 $(patsubst %.ko, %.mod.o，$(modules)) .

4.4.5 目标指定变量和模式指定变量

目标指定变量(Target-specific Variable)允许对于相同变量根据目标指定不同的值，这有点类似于自动化变量。目标指定的变量值只在指定它的目标的上下文中有效，对于其他的目标没有影响。也就是说目标指定变量是此目标的"局部"变量.

设置一个目标指定变量的语法为：

```
TARGET ... : VARIABLE-ASSIGNMENT
```

或者：

```
TARGET ... : override VARIABLE-ASSIGNMENT
```

如顶层 Makefile 编译外部模块时:

```
1 clean:  rm-dirs := $(MODVERDIR)
```

GNU make 除了支持目标指定变量之外，还支持另外一种方式：模式指定变量（ Pattern-specific Variable ）。模式指定变量定义是将一个变量值指定到所有符合此模式的目标上，而不像目标指定变量一样只适用于某一个目标. 设置一个模式指定变量的语法和设置目标变量的语法相似：

```
PATTERN ... : VARIABLE-ASSIGNMENT
```

或者：

```
PATTERN ... : override VARIABLE-ASSIGNMENT
```

模式指定变量和目标指定变量语法的唯一区别就是：这里的目标是一个或者多个“模式”目标（包含模式字符“%”).模式指定变量在 Makefile.build 中比比皆是：

Makefile.build

```
1
2 $(real-objs-m)         : quiet_modtag := [M]
3 $(real-objs-m:.o=.i)   : quiet_modtag := [M]
4 $(real-objs-m:.o=.s)   : quiet_modtag := [M]
5 $(real-objs-m:.o=.lst): quiet_modtag := [M]
```

4.5 call函数

call 函数是唯一一个可以创建定制化参数函数的引用函数。使用这个函数可以实现对用户自己定义函数引用。我们可以将一个变量定义为一个复杂的表达式，用 call 函数根据不同的参数对它进行展开来获得不同的结果。函数语法：

```
1 $(call VARIABLE,PARAM,PARAM,...)
```

执行时， call 将它的参数 “PARAM” 依次赋值给临时变量 “$(1)”,“$(2)” . call 函数对参数的数目没有限制，也可以没有参数值，没有参数值的 “call” 没有任何实际存在的意义。执行时变量 “VARIABLE” 被展开为在函数上下文有效的临时变量，变量定义中的 “$(1)” 作为第一个参数，并将函数参数值中的第一个参数赋值给它；变量中的 “$(2)” 一样被赋值为函数的第二个参数值；依此类推（变量 $(0) 代表变量 “VARIABLE” 本身）。之后对变量 “VARIABLE” 表达式的计算值。返回值为参数值 “PARAM” 依次替换 “$(1)”,“$(2)” ... 之后变量 “VARIABLE” 定义的表达式的计算值。

以 kbuild 的 Makefile.build 为例

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

if_changed_dep 则定义在 kbuild.include 中

```
if_changed_dep = $(if $(strip $(any-prereq) $(arg-check) ),
                      \
        @set -e;
            \
        $(echo-cmd) $(cmd_$(1));
                                                    \
        scripts/basic/fixdep $(depfile) $@ '$(make-cmd)' > $(
            dot-target).tmp;\
        rm -f $(depfile);
            \
        mv -f $(dot-target).tmp $(dot-target).cmd)
```

此例中 if_changed_dep 函数只有一个参数 as_o_S , 而 cmd_as_o_S 又定义为:

```
cmd_as_s_S       = $(CPP) $(a_flags)    -o $@ $<
```

Part3

5 kbuild targets实现分析

5.1 kbuild targets和命令行概述

编译时，我们首先必须确定一个 target ，然后才能确定所有的 Prerequiste ，最后根据更新情况决定是否执行相应的命令。因此要想了解 kbuild 的实现，我们首先了解 kbuild 所支持的 target . kbuild 所支持的的 targets 大体可分为如下几大类(make help 打印出的信息，略作调整）.

```
Cleaning targets:
  clean/mrproper/distclean 删除部分或全部编译生成的文件:
                           config,backup,patch等

Configuration targets:
  %config     生成.config

Other generic targets:
  all   - Build all targets marked with [*]
* vmlinux   - Build the bare kernel
* modules   - Build all modules
  rpm   - Build a kernel as an RPM package
  kernelrelease   - Output the release version string
  kernelversion   - Output the version stored in Makefile

Install targets:
  headers_install - Install sanitised kernel headers to INSTALL_HDR_PATH
                    (default: /mnt/sda2/kernel/linux-2.6.23.1/usr)
  modules_install - Install all modules to INSTALL_MOD_PATH (default: /)

Single targets
  dir/            - Build all files in dir and below
  dir/file.[ois]  - Build specified target only
  dir/file.ko     - Build module including final link

Source code browse:
  tags/TAGS   - Generate tags file for editors
  cscope   - Generate cscope index

Static analysers
  checkstack      - Generate a list of stack hogs
  namespacecheck  - Name space analysis on compiled kernel
  headers_check   - Sanity check on exported headers

Kernel packaging:
  %pkg         - Build the kernel as an % package

Documentation targets:
  %doc             - 把kernel内部文档转成不同格式
```

```
Architecture specific targets (i386):
* bzImage - Compressed kernel image (arch/i386/boot/bzImage)
  install - Install kernel using
   (your) ~/bin/installkernel or
   (distribution) /sbin/installkernel or
   install to $(INSTALL_PATH) and run lilo
  bzdisk       - Create a boot floppy in /dev/fd0
  fdimage      - Create a boot floppy image
  isoimage     - Create a boot CD-ROM image
```

make 或 make all 执行时的所有目标均以 [*] 标记.

kbuild 命令行选项

```
  make V=0|1 [targets] 0 => quiet build (default), 1 => verbose build
  make V=2   [targets] 2 => give reason for rebuild of target

  make O=dir [targets] Locate all output files in "dir", including .config

  make C=1   [targets] Check all c source with $CHECK (sparse by default)
  make C=2   [targets] Force check of all c source with $CHECK

  make -C=KERNELDIR M=dir modules  编译外部模块
  make -C=KERNELDIR SUBDIRS=dir modules 同上，SUBDIRS优先级低于M
```

以上即为 kbuild 所支持的所有 targets 和相关的命令行选项，本章的重点是以下几类：

- configration ，主要是通过 make %config 生成 .config
- 输出和源代码目录的分离.
- 编译 module
- 编译外部模块
- vmlinux
- 编译Single target

而以下几类和 kbuild 规则文件有紧密的联系，这将在下一章详细解释.

- 编译 host program ，这通常是 kbuild 需要的一些工具程序.
- 模块及头文件的安装
- arch Makefile

而那些和编译 module/kernel image 关系不大的 targets 我们将略去不谈.

kbuild 除了支持上述单个 target 外，同时也支持 %config 和其他种类的 target 的混合模式 (mixed-target) :

```
$ make oldconfig vmlinux
```

会产生.config，然后接着执行vmlinux

targets 和命令行选项的不同组合，产生的效果也是完全不一样的。例如，命令行加上了 O=dir 选项和和没有 O=dir 选项的执行过程是有区别的.

```
$ make O=dir oldconfig vmlinux
$ make oldconfig vmlinux
```

在了解这些目标实现之前，有必要了解Makefile的简化版:

```
... ...

ifeq ($(KBUILD_SRC),)

    KBUILD_OUTPUT的赋值
    ... ...

ifneq ($(KBUILD_OUTPUT),)

    KBUILD_OUTPUT不为空时的规则

skip-makefile := 1
endif # ifneq ($(KBUILD_OUTPUT),)
endif # ifeq ($(KBUILD_SRC),)


ifeq ($(skip-makefile),)

    ... ...

    config-targets/mixed-targets/dot-config定义和赋值

    ... ...

ifeq ($(mixed-targets),1)

        mixed-target规则

else # $(mixed-targets)
ifeq ($(config-targets),1)

        config-target规则

else # $(config-targets)

        ... ...

ifeq ($(dot-config),1)

        dot-config变量和规则

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

        vmlinux/module/EXTMOD变量和规则

endif #ifeq ($(config-targets),1)
endif #ifeq ($(mixed-targets),1)

        single target变量和规则

endif   # skip-makefile
```

```
54 ... ...
```

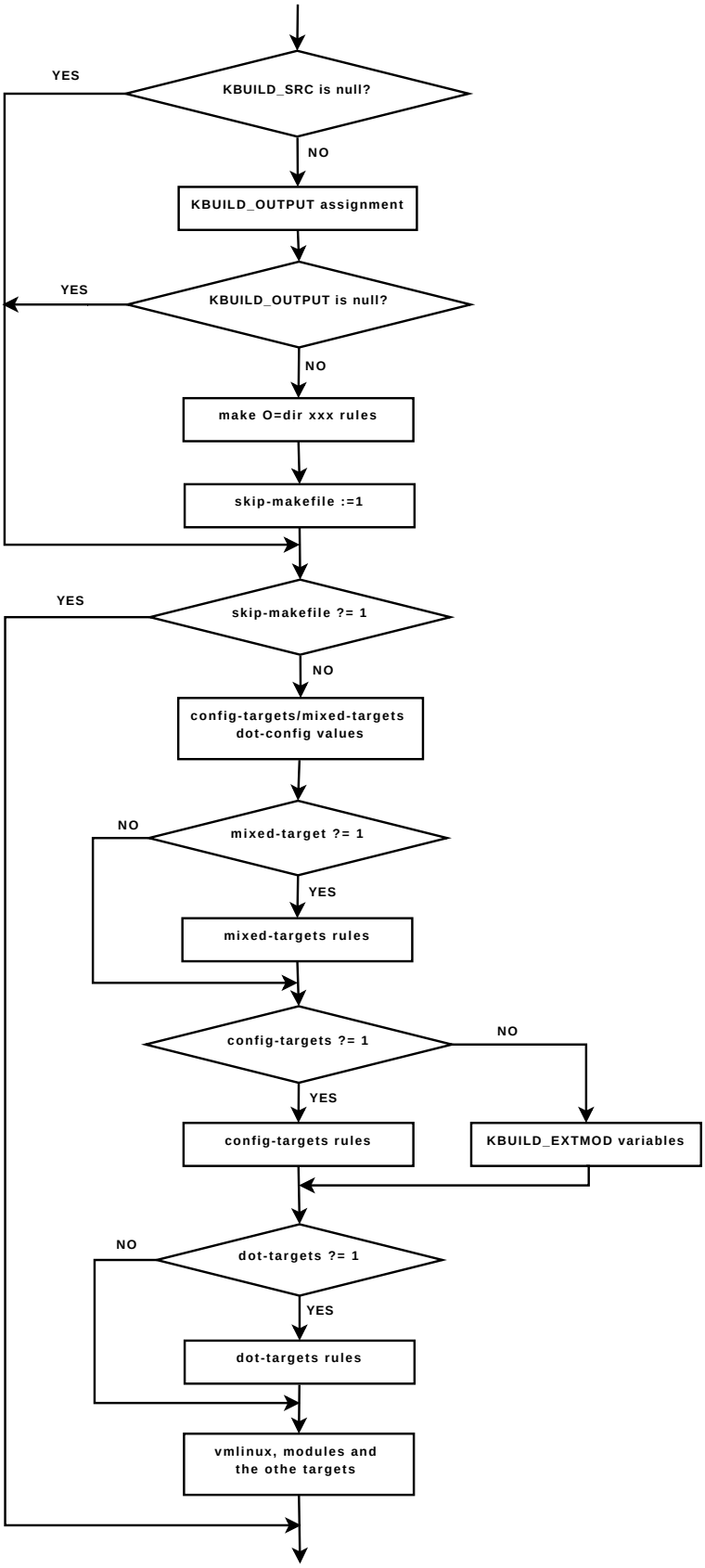

图 5: Makefile简略图

其中的关键变量有: KBUILD_SRC, KBUILD_OUTPUT, skip_makefile dot-config, mixed-target, config-target . 我们在这些 targets 的实现时，也是主要以顶层 Makefile 为参考，先讲述主要变量值，接着是规则链.

5.2 %config target的实现

%config 的目的是产生 .config . make %config 时 Makefile 中的主要变量值为:

- KBUILD_SRC和KBUILD_OUTPUT，skip-makefile都为空;
- dot-config=1
- mixed-targets=0
- config-targets=1

Makefile会最终会调用规则

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

以make oldconfig为例,运行结果为:

```
make -f scripts/Makefile.build obj=scripts/basic
mkdir -p include/linux include/config
make -f scripts/Makefile.build obj=scripts/kconfig oldconfig
scripts/kconfig/conf -o arch/i386/Kconfig
scripts/kconfig/conf会解析Kconfig并最终生成.config
```

5.3 mixed-target的实现

mixed-target 通常指定一个 %config 和其他目标，如: make oldconfig vmlinux ，此时 Makefile 中的主要变量值为: KBUILD_SRC 和 KBUILD_OUTPUT, skip-makefile 都为空;

- dot-config=1
- config-targets=1
- mixed-targets=1

Makefile 匹配的规则为

```
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略去

endif
```

此处用到了模式双冒号规则 %:: ，分别调用递归调用 kbuild 顶层 Makefile 本身重新执行目标:

```
make -C $(srctree) KBUILD_SRC=  oldconfig
make -C $(srctree) KBUILD_SRC=  vmlinux
```

执行oldconfig时，此时的变量为

- CURDIR= $(srctree)
- KBUILD_SRC=""

这和单独执行 make oldconfig 一样

```
make -C $(srctree) KBUILD_SRC=  vmlinux
```

这也和单独执行 make vmlinux 没有任何区别。

5.4 编译输出和源代码目录的分离

```
$ make O=dir oldconfig
```

改变系统输出目录是通过递归调用 Makefile 来实现的，我们来看其实现过程:

第一遍Makefile

开始时的变量值为:

- KBUILD_SRC为空
- KBUILD_OUTPUT = dir

Makefile 最终会运行

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

由于 KBUILD_OUTPUT 并不为空, skip-makefile := 1 ，所以Makefile剩余部分就直接跳过了。

递归运行Makefile-第二遍

```
make -C dir KBUILD_SRC=pwd KBUILD_EXTMOD="" -f pwd/Makefile oldconfig
```

此时: KBUILD_SRC不为空，skip-makefile=0 ; 不过此时 CURDIR=dir (make -C dir) , 从而:

- 源代码目录srctree = $(KBUILD_SRC)
- 输出目录objtree = $(CURDIR)

这就实现了和没有 O=dir 命令行选项时的代码完全一致，这就是为什么当 make O=dir xxx 时， Makefile 采用递归形式的原因.

5.5 make和make all

make 由于没有显示指定 MAKECMDGOALS ， 所以使用 _all 作为缺省的目标.此时 Makefile 中的主要变量:

- dot-config=1
- config-targets=0
- mixed-targets=0
- KBUILD_EXTMOD=""
- KBUILD_OUTPUT=""
- skip-makefile=""

Makefile规则链的确定:

```
PHONY := _all
_all:

... ...
```

_all的依赖目标为:

```
ifeq ($(KBUILD_EXTMOD),)
_all: all
else
_all: modules
endif
... ...

all: vmlinux

... ...

ifdef CONFIG_MODULES

# By default, build modules as well

all: modules
```

以x86台为例，arch/i386/Makefile中:

```
all: bzImage
```

所以缺省目标_all的所有依赖目标为:

```
_all: all
all: vmlinux bzImage modules
```

由此可见， make 和 make all 是等同的，系统会依次生成 vmlinux, bzImage ， 和 modules 。

5.6 vmlinux目标实现

5.6.1 vmlinux所涉及的变量

```
vmlinux-init := $(head-y) $(init-y)
vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
vmlinux-all  := $(vmlinux-init) $(vmlinux-main)
vmlinux-lds  := arch/$(ARCH)/kernel/vmlinux.lds
```

head-y 和部分 core-y， libs-y drivers-y 通常在 arch/$(ARCH)/Makefile 中都有定义，在 i386 中:

```
vmlinux-lds  := arch/i386/kernel/vmlinux.lds
head-y    := arch/i386/kernel/head.o arch/i386/kernel/init_task.o
init-y    := init/built-in.o
core-y    := usr/built-in.o
             arch/i386/kernel/built-in.o  \
             arch/i386/mm/built-in.o \
             arch/i386/mach-default/built-in.o \
             arch/i386/crypto/built-in.o  \
             kernel/built-in.o  mm/built-in.o \
             fs/built-in.o ipc/built-in.o \
             security/built-in.o  crypto/built-in.o \
             block/built-in.o
libs-y    := lib/lib.a  arch/i386/lib/lib.a \
             lib/built-in.o arch/i386/lib/built-in.o
drivers-y := drivers/built-in.o  \
             sound/built-in.o  \
             arch/i386/pci/built-in.o  \
             arch/i386/power/built-in.o  \
             arch/i386/video/built-in.o
net-y     := net/built-in.o
vmlinux-dirs := init usr \
             arch/i386/kernel arch/i386/mm \
             arch/i386/mach-default arch/i386/crypto \
             kernel mm fs ipc security crypto block \
             drivers sound arch/i386/pci arch/i386/power \
             arch/i386/video arch/i386/oprofile \
             net lib arch/i386/lib
vmlinux-alldirs := \
             arch/i386/crypto arch/i386/kernel \
             arch/i386/lib arch/i386/mach-default \
             arch/i386/mach-es7000 arch/i386/math-emu \
             arch/i386/mm arch/i386/oprofile arch/i386/pci \
             arch/i386/power arch/i386/video arch/i386/xen \
             block crypto drivers fs init ipc kernel lib mm \
             net security sound usr
```

kallsyms.o 此处略去不谈，将在 kbuild 专题7.3.2详述.

5.6.2 vmlinux的规则链

```
vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) $(
    kallsyms.o) vmlinux.o FORCE
ifdef CONFIG_HEADERS_CHECK
        $(Q)$(MAKE) -f $(srctree)/Makefile headers_check
endif
        $(call vmlinux-modpost)
        $(call if_changed_rule,vmlinux__)
        $(Q)rm -f .old_version
```

vmlinux 的依赖目标为 $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) $(kallsyms.o) vmlinux.o 。

```
vmlinux.o: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) $(
    kallsyms.o) FORCE
        $(call if_changed_rule,vmlinux-modpost)
```

vmlinux.o 的依赖目标为 $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) $(kallsyms.o) 。

```
$(sort $(vmlinux-init) $(vmlinux-main)) $(vmlinux-lds): $(
    vmlinux-dirs) ;
vmlinux-lds, vmlinux-init, vmlinux-依赖于mainvmlinux-dirs.

PHONY += $(vmlinux-dirs)
$(vmlinux-dirs): prepare scripts
        $(Q)$(MAKE) $(build)=$@
```

vmlinux-dirs 规则会调用: make -f scripts/Makefile.build obj=<dir> 进行实际的编译 .vmlinux-dirs 伪目标是实现 descending down 的关键,当 vmlinux-dirs 遍历完后，上述的 vmlinux-lds, vmlinux-init, vmlinux-main 都已编译完成.

prepare 目标及其依赖规则链为:

```
prepare: prepare0

prepare0: archprepare FORCE
        $(Q)$(MAKE) $(build)=.
        $(Q)$(MAKE) $(build)=. missing-syscalls

archprepare: prepare1 scripts_basic
```

prepare0 依赖目标为 archprepare，prepare0 的命令主要分为两步：

1. make -f scripts/Makefile.build obj=. 此时 srcdir=. ， 也就是说 Makefle.build 会 include 源代码根目录下的 kbuild ， 目的是由 arch/i386/kernel/asm-offset.c 产生 include/asm-i386/asm-offset.h 。

2. make -f scripts/Makefile.build obj=. missing-syscalls 指定了目标 missing-syscalls 。调用 scripts/checksyscall.sh 检查是否有 i386 特定的系统调用未被忽略.

archprepare 目标除了 Makefile 中定义的依赖目标外， arch/$(ARCH)/Makefile 也有可能定义别的依赖。

prepare1规则链为:

```
prepare1: prepare2 include/linux/version.h include/linux/
    utsrelease.h \
```

```
                    include/asm include/config/auto.conf
ifneq ($(KBUILD_MODULES),)
        $(Q)mkdir -p $(MODVERDIR)
        $(Q)rm -f $(MODVERDIR)/*
endif

include/linux/version.h: $(srctree)/Makefile FORCE
        $(call filechk,version.h)

include/linux/utsrelease.h: include/config/kernel.release
   FORCE
        $(call filechk,utsrelease.h)

include/asm:
        @echo '  SYMLINK $@ -> include/asm-$(ARCH)'
        $(Q)if [ ! -d include ]; then mkdir -p include; fi;
        @ln -fsn asm-$(ARCH) $@
```

当需要编译模块时， prepare1 命令会建立目录 .tmp_versions/ ，该目录下保存所有 <modeule>.mod ，用于模块编译的第二阶段. 当 O=dir 编译输出和源代码目录分开时，<dir>/include/asm-$(ARCH)/ 只包含一个 asm-offset.h 文件， <dir>/include/asm 符号链接到 <dir>/include/asm-$(ARCH) .

所以 prepare3 规则要另建 include2 目录链接到 $(srctree)/include/asm-$(ARCH) .

prepare2规则链为:

```
prepare2: prepare3 outputmakefile

outputmakefile:
ifneq ($(KBUILD_SRC),)
        $(Q)$(CONFIG_SHELL) $(srctree)/scripts/mkmakefile \
            $(srctree) $(objtree) $(VERSION) $(PATCHLEVEL)
endif
```

当 KBUILD_SRC 不为空(O=dir，编译输出和源代码目录分开)时，在输出目录建立 Makefile ，其目的是: 当 make -f $(srctree)/scripts/Makefile.build obj=. 时, 通过 dir 刚生成的 Makefile ，重定向到 $(srctree)/Makefile 。

我们来看一个 outputmakefile 生成的 Makefile(<dir>/Makefile) :

```
all:
        $(MAKE) -C $(KERNELSRC) O=$(KERNELOUTPUT)
```

prepare3规则链为:

```
prepare3: include/config/kernel.release
ifneq ($(KBUILD_SRC),)
        @echo '  Using $(srctree) as source for kernel'
        $(Q)if [ -f $(srctree)/.config -o -d $(srctree)/
           include/config ]; then \
                echo "  $(srctree) is not clean, please run '
                   make mrproper'";\
                echo "  in the '$(srctree)' directory.";\
                /bin/false; \
        fi;
        $(Q)if [ ! -d include2 ]; then mkdir -p include2; fi;
        $(Q)ln -fsn $(srctree)/include/asm-$(ARCH) include2/
           asm
```

```
11 endif
12
13 kernelrelease = $(KERNELVERSION)$(localver-full)
14 include/config/kernel.release: include/config/auto.conf FORCE
15         $(Q)rm -f $@
16         $(Q)echo $(kernelrelease) > $@
```

prepare3输出和源代码目录分开(O=dir，KBUILD_SRC 不为空)时:

- 检查源代码目录是否含有未清除文件。
- 建立 include2/asm 到 $(srtree)/include/asm-$(ARCH) 的链接， include2/asm 会加到编译时的 include 搜索路径.

scripts目标及其依赖目标链为：

```
1 scripts: scripts_basic include/config/auto.conf
2         $(Q)$(MAKE) $(build)=$(@)
3
4 PHONY += scripts_basic
5 scripts_basic:
6         $(Q)$(MAKE) $(build)=scripts/basic
7
8 include/config/auto.conf: $(KCONFIG_CONFIG) include/config/
     auto.conf.cmd
9         $(Q)$(MAKE) -f $(srctree)/Makefile silentoldconfig
```

scripts_basic 的目的是生成 fixdep 和 docproc . make -f $(srctree)/Makefile silentoldconfig 会产生

- include/config/auto.conf
- inlcude/config/auto.conf.cmd
- include/config/*.h(用于fixdep)
- include/linux/autoconf.h

而 Makefile 规则的执行则以依赖目标关系链推理相反的顺序执行，所以

```
1 include/config/auto.conf: $(KCONFIG_CONFIG) include/config/
     auto.conf.cmd
2         $(Q)$(MAKE) -f $(srctree)/Makefile silentoldconfig
```

为最先执行的规则。

当 make vmlinux 时 dot-config = 1 ，所以Makefile包含

```
1 # Read in config
2 -include include/config/auto.conf
```

这时 include/config/auto.conf 既是 include 文件又是 target ， 因此， Makefile 在执行过程中会就会引起 Makefile 的 Remaking .

5.6.3 vmlinux的链接

链接 vmlinux 的最终命令为：

```
ld -m elf_i386 --emit-relocs --build-id -o vmlinux
   -T arch/i386/kernel/vmlinux.lds arch/i386/kernel/head.o
   arch/i386/kernel/init_task.o  init/built-in.o
   --start-group  usr/built-in.o  arch/i386/kernel/built-in.o
   arch/i386/mm/built-in.o  arch/i386/mach-default/built-in.o
   arch/i386/crypto/built-in.o  kernel/built-in.o  mm/built-in.o
   fs/built-in.o  ipc/built-in.o  security/built-in.o
   crypto/built-in.o  block/built-in.o  lib/lib.a
   arch/i386/lib/lib.a  lib/built-in.o  arch/i386/lib/built-in.o
   drivers/built-in.o  sound/built-in.o  arch/i386/pci/built-in.o
   arch/i386/power/built-in.o  arch/i386/video/built-in.o
   net/built-in.o --end-group .tmp_kallsyms2.o
```

值得注意的是:

- arch/i386/kernel/head.o 处于 vmlinux 的最前面，其主要作用就是初始化中断描述符表 IDT ，内存页目录表 GDT ，把系统从32位段式寻址的保护模式跳到32位页式寻址的保护模式(Enable paging) .head.S 中首先定义的就是:

  ```
  .section .text.head，"ax"，@progbits
  ENTRY(startup_32)
  ```

  starup_32会在稍后的vmlinux.ld中看到.

- ld 使用了 linker script -T arch/i386/kernel/vmlinux.lds 。一般来说，普通程序是不需要指定 linker script 的，也不需要关心各个 section 的具体位置的。当程序执行时， kernel 中的 ELF Loader 会依据 ELF Program header 解析可执行文件的各个SECTION，并把它们映射到虚拟地址空间。然而，内核启动时，必须首先确定各个 section 的具体位置，这就是 vmlinux.lds 的作用.

arch/i386/kernel/vmlinux.lds

```
/* 定义了输出格式和平台 */
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
/* 定义phy_startup_32为程序入口点 */
ENTRY(phys_startup_32)
/* jiffes和jiffies_64地址相同,也就是说jiffies_64的低32位就是jiffies */
jiffies = jiffies_64;
/* 指定ELF Program header 可以用objdump -p 查看 */
PHDRS {
 text PT_LOAD FLAGS(5); /* R_E */
 data PT_LOAD FLAGS(7); /* RWE */
 note PT_NOTE FLAGS(0); /* ___ */
}
/* 定义输入文件的section如何组织到输出文件的section */
SECTIONS
{
  /* 起始VMA地址为0xC0100000, startup_32也为0xC0100000 */
```

```
  . = 0xC0000000 + ((0x100000 + (0x100000 - 1)) & ~(0x100000 - 1));
  /* phy_startup_32 = 0x100000 */
  phys_startup_32 = startup_32 - 0xC0000000;
  /* .text.head section LMA为0x100000 head.o中的.text.head会放到此
处 */
  .text.head : AT(ADDR(.text.head) - 0xC0000000) {
  /* _text地址为当前VMA地址, 0xC0100000 */
   _text = .; /* Text and read-only data */
 *(.text.head)
  } :text = 0x9090 /*合并section中的空隙用0x9090填充*/
 ... ...
}
```

vmlinux生成后，我们可以看到:

```
$ objdump -p vmlinux

vmlinux:     file format elf32-i386

Program Header:
    LOAD off    0x00001000 vaddr 0xc0100000 paddr 0x00100000 align 2**12
         filesz 0x002c580c memsz 0x002c580c flags r-x
    LOAD off    0x002c7000 vaddr 0xc03c6000 paddr 0x003c6000 align 2**12
         filesz 0x000903b4 memsz 0x00105000 flags rwx
    NOTE off    0x0020c6d0 vaddr 0xc030b6d0 paddr 0x0030b6d0 align 2**2
         filesz 0x00000024 memsz 0x00000024 flags ---
```

和vmlinux.ld中指定的一致.

我们也可以用nm来验证vmlinux.ld中的符号地址.

```
$ nm vmlinux|grep " _text"
c0100000 T _text
$ nm vmlinux|grep startup_32
00100000 A phys_startup_32
c0100000 T startup_32
c01000c0 T startup_32_smp
$ nm vmlinux|grep " jiffies*"
c03ec400 D jiffies
c03ec400 D jiffies_64
```

5.7 modules target实现

5.7.1 modules变量

module目标的编译涉及到两个重要的变量:

- KBUILD_MODULES, 表示是否需要编译模块
- KBUILD_BUILTIN, 表示编译模块时是否先编译vmlinux.

缺省值

- KBUILD_MODULES :=

- KBUILD_BUILTIN := 1

当只 make modules 时，一般并不 KBUILD_BUILTIN=“” ，并不编译 vmlinux ，但是当 CONFIG_MODVERSIONS=y 时，模块编译之前必须确保符号的版本标签值(checksum)是最新的，所以必须设置 KBUILD_BUILTIN=1 。

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

当 make <whatever> modules 时，必须编译模块；而且 make 或 make all 缺省都会编译模块， KBUILD_MODULES=1 .

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

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

5.7.2 modules规则链

编译模块的前提是 CONFIG_MODULES 必须定义(CONFIG_MODULES=y)，否则，即便 make 显示或缺省编译模块， modules 也会什么也不做。

```
ifdef CONFIG_MODULES

# By default, build modules as well

all: modules如果

KBUILD_BUILTIN=1, 则将成为依赖目标vmlinux.
#       Build modules

PHONY += modules
modules: $(vmlinux-dirs) $(if $(KBUILD_BUILTIN),vmlinux)
        @echo '  Building modules, stage 2.';
        $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modpost
... ...

else # CONFIG_MODULES

# Modules not configured
# ---------------------------------------------------------------

modules modules_install: FORCE
        @echo
        @echo "The present kernel configuration has modules
            disabled."
        @echo "Type 'make config' and enable loadable module
            support."
        @echo "Then build a kernel with module support enabled
            ."
        @echo
```

```
27          @exit 1
28
29  endif # CONFIG_MODULES
30
31  PHONY += $(vmlinux-dirs)
32  $(vmlinux-dirs): prepare scripts
33          $(Q)$(MAKE) $(build)=$@
```

我们看到 $(vmlinux-dirs) 也成为 modules 的依赖目标， $(vmlinux-dirs) 和编译 vmlinux 时的 $(vmlinux-dirs) 完全相同，而且 $(vmlinux-dirs) 为 .PHONY 目标，这就意味着缺省编译 make 或 make all 时，编译 modules 时，这些目录又会被再次遍历。 modules 最后会调用 make -f scripts/Makefile.modpos 进行模块编译的第二步，关于 Makefile.modpost 将在下一章6.2.5中详述.

5.8 EXTMOD target实现

5.8.1 EXTMOD命令行

编译外部模块时，系统提供了如下的接口:

```
make M=dir clean     Delete all automatically generated files
make M=dir modules   Make all modules in specified dir
make M=dir       Same as 'make M=dir modules'
make M=dir modules_install
```

dir 下的 Makefile 和 kernel 目录树下的 Makefile 没有任何区别,都会被 include 到 scripts/Makefile.build . 接下来我们将看到 EXTMOD target 是如何在 Makefile 中实现的：

kbuild 为了支持编译外部模块(External module ， 非 kernel 源代码树)，引入了 KBUILD_EXTMOD 变量:

kbuild 目前支持两种指定外部模块所在的目录的方式:

- make ... SUBDIRS=$PWD ，这是过时的方法，这种情况下 KBUILD_EXTMOD 环境变量优先级比 SUBDIRS 高。
- make M=dir ，这是最新的方式， M=dir 会直接赋值给 KBUILD_EXTMOD ，而不考虑环境变量. 当然我们也可以在 make 命令行选项直接赋值 make KBUILD_EXTMOD =dir ，这个指定的值具有最高优先级，会直接忽略 Makefile 中所有变量的赋值。

```
1  ifdef SUBDIRS
2    KBUILD_EXTMOD ?= $(SUBDIRS)
3  endif
4  ifdef M
5    ifeq ("$(origin M)", "command line")
6      KBUILD_EXTMOD := $(M)
7    endif
8  endif
```

当 KBUILD_EXTMOD 不为空时， make M=dir ， kbuild 会自动加上 modules ，所以和 make M=dir modules 等效.

```
1  ifeq ($(KBUILD_EXTMOD),)
2  _all: all
3  else
4  _all: modules
5  endif
```

5.8.2 EXTMOD编译的前提条件

编译外部模块的前提时， kbuild 必须保证 include/config/auto.conf 的存在，也就是说 kbuild 前期的 configuration 阶段必须完成，否则外部模块无法编译。

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

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

ifeq ($(KBUILD_EXTMOD),)略去

else

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  $(dot-config)
# Dummy target needed, because used as prerequisite
include/config/auto.conf: ;
endif # $(dot-config)
```

由上可看出，当 build external modules 时，系统并没有像非 EXTMOD 那样提供重载的机会，所以必须保证 include/config/auto.conf 的存在，否则系统报错,

5.8.3 EXTMOD目标的实现

前面可以看到:

make M=dir时， 缺省目标为 _all: modules
make M=dir modules，显示目标也为moduels

顶层Makefile由于一堆ifdef的定义，难以阅读，其简略版本如下:

```
ifeq ($(KBUILD_EXTMOD),)
        ... ...
```

```
# Build vmlinux
        ... ...
ifdef CONFIG_KALLSYMS
        ... ...
kallsyms.o := .tmp_kallsyms$(last_kallsyms).o
        ... ...
endif # ifdef CONFIG_KALLSYMS

        ... ...
vmlinux.o: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) $(
    kallsyms.o) FORCE
        $(call if_changed_rule,vmlinux-modpost)
        ... ...

ifdef CONFIG_MODULES

# By default, build modules as well

all: modules
        ... ...
else # CONFIG_MODULES

        ... ...

endif # CONFIG_MODULES

        ... ...

else # KBUILD_EXTMOD编译外部模块的变量：
        module-dir, install-dir, clean-dir编译外部模块的目标：
        modules, modules_install, clean, help
endif # KBUILD_EXTMOD
```

外部模块 module 目标自成体系和 make modules 规则明显不同:

```
modules: $(module-dirs)
        @echo '  Building modules, stage 2.';
        $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modpost
```

而 $(module-dirs) 只会编译 M=dir 指定的目录:

```
module-dirs := $(addprefix _module_,$(KBUILD_EXTMOD))
PHONY += $(module-dirs) modules
$(module-dirs): crmodverdir $(objtree)/Module.symvers
        $(Q)$(MAKE) $(build)=$(patsubst _module_%,%,$@)
```

编译外部模块时 $(srctree)/Module.symvers 最好存在，否则会有警告信息.

```
PHONY += $(objtree)/Module.symvers
$(objtree)/Module.symvers:
        @test -e $(objtree)/Module.symvers || ( \
        echo; \
        echo "  WARNING: Symbol version dump $(objtree)/Module
            .symvers"; \
        echo "           is missing; modules will have no
            dependencies and modversions."; \
        echo )
```

外部模块的 EXTMOD clean 命令同样也只是 clean M=dir 指定的目录

```
clean-dirs := $(addprefix _clean_,$(KBUILD_EXTMOD))

PHONY += $(clean-dirs) clean
$(clean-dirs):
        $(Q)$(MAKE) $(clean)=$(patsubst _clean_%,%,$@)

clean:  rm-dirs := $(MODVERDIR)
clean: $(clean-dirs)
        $(call cmd,rmdirs)
        @find $(KBUILD_EXTMOD) $(RCS_FIND_IGNORE) \
                \( -name '*.[oas]' -o -name '*.ko' -o -name
                    '.*.cmd' \
                -o -name '.*.d' -o -name '.*.tmp' -o -name '*.
                    mod.c' \) \
                -type f -print | xargs rm -f
```

外部模块的 EXTMOD modules_install 命令同样也会安装 M=dir 指定的目录下的模块

```
PHONY += modules_install
modules_install: _emodinst_ _emodinst_post

install-dir := $(if $(INSTALL_MOD_DIR),$(INSTALL_MOD_DIR),
    extra)
PHONY += _emodinst_
_emodinst_:
        $(Q)mkdir -p $(MODLIB)/$(install-dir)
        $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modinst

PHONY += _emodinst_post
_emodinst_post: _emodinst_
        $(call cmd,depmod)的作用就是安装完
cmd_depmodM=目录下的模块后，运行更新模块信息dirdepmod.
```

5.9 Single target实现

5.9.1 Single target的命令行

kbuild 提供的 single target 有以下几种形式:

make dir/	编译单个目录,包括遍历子目录。
make dir/file.[ois]	编译单个文件
make dir/file.ko	编译单个模块

5.9.2 Single target的实现

```
# Single targets
#
    ------------------------------------------------------------

# Single targets are compatible with:
# - build whith mixed source and output
# - build with separate output dir 'make O=...'
```

```
# - external modules
#
#  target-dir => where to store outputfile
#  build-dir  => directory in kernel source tree to use

ifeq ($(KBUILD_EXTMOD),)
        build-dir  = $(patsubst %/,%,$(dir $@))
        target-dir = $(dir $@)
else
        zap-slash=$(filter-out .,$(patsubst %/,%,$(dir $@)))
        build-dir  = $(KBUILD_EXTMOD)$(if $(zap-slash),/$(zap-
            slash))
        target-dir = $(if $(KBUILD_EXTMOD),$(dir $<),$(dir $@)
            )
endif

%.s: %.c prepare scripts FORCE
        $(Q)$(MAKE) $(build)=$(build-dir) $(target-dir)$(
            notdir $@)
%.i: %.c prepare scripts FORCE
        $(Q)$(MAKE) $(build)=$(build-dir) $(target-dir)$(
            notdir $@)
%.o: %.c prepare scripts FORCE
        $(Q)$(MAKE) $(build)=$(build-dir) $(target-dir)$(
            notdir $@)
%.lst: %.c prepare scripts FORCE
        $(Q)$(MAKE) $(build)=$(build-dir) $(target-dir)$(
            notdir $@)
%.s: %.S prepare scripts FORCE
        $(Q)$(MAKE) $(build)=$(build-dir) $(target-dir)$(
            notdir $@)
%.o: %.S prepare scripts FORCE
        $(Q)$(MAKE) $(build)=$(build-dir) $(target-dir)$(
            notdir $@)
%.symtypes: %.c prepare scripts FORCE
        $(Q)$(MAKE) $(build)=$(build-dir) $(target-dir)$(
            notdir $@)

# Modules
/ %/: prepare scripts FORCE
        $(Q)$(MAKE) KBUILD_MODULES=$(if $(CONFIG_MODULES),1) \
        $(build)=$(build-dir)
%.ko: prepare scripts FORCE
        $(Q)$(MAKE) KBUILD_MODULES=$(if $(CONFIG_MODULES),1)
              \
        $(build)=$(build-dir) $(@:.ko=.o)
        $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modpost
```

编译单个文件，模块，目录时，所有的目标都是模式目标，而且其依赖目标为 prepare 和 scripts . 例如， make init/main.o 时，make会首先会检查依赖条件 prepare script ，然后调用 make -f scripts/Makefile.build obj=init init/main.o 。

6 kbuild Makefile的实现分析

kbuild源代码目录树下的Makefile的目标通常有以下大类:

- built-in.o 即所有 obj-y 的目标文件，包括: 单个文件的 .o ; 由多个文件复合而成的 .o ; obj-y=dir/ 编译而成的 dir/built-in.o
- lib.a 有一个或多个.o组成lib.a
- 多个模块 (*.ko) .每个模块可能由单个 .o 组成，也可能由多个 .o 复合而成。 obj-m=dir/ 则会递归到 dir 目录编译模块。
- 支持 clean 目标. 如定义一些 clean-files .
- Hostprog ， 如 arch/i386/boot/compressed/Makefile hostprogs-y := relocs
- extra-y ， 如 arch/i386/kernel/Makefile extra-y := head.o init_task.o vmlinux.lds

除了这些与平台无关的Makefile外，我们还将以x86为例来分析arch/$(ARCH)/Makefile的实现.

6.1 Built-in object goals - obj-y

obj-y 文件列表一般都是要 merge 到一起形成一个 built-in.o ， 然后这个 built-in.o 再链接进 kernel 中。 obj-y 列表中的文件可分为三类：

- 单个文件.没有自己的文件列表(name-objs 或 names-y).
- 复合文件.即一个 .o 是由几个 .o (name-objs 或 names-y)链接而成
- 子目录的递归编译 obj-y=dir/

我们以init/Makefile为例:

```
obj-y                           := main.o version.o mounts.o

mounts-y := do_mounts.o
mounts-$(CONFIG_BLK_DEV_RAM) += do_mounts_rd.o
mounts-$(CONFIG_BLK_DEV_INITRD) += do_mounts_initrd.o
```

其中 version.o main.o 为单个文件, 因为不存在 version-y 和 main-y. mount.o 则为复合文件。

分清了 Single object 和 Composite object 之后， kernel 需要编译 init/built-in.o 时会调用 make -f scripts/Makefile.build obj=init , 我们来看其具体实现:

1. Makefile.build执行第一阶段.
a. 读取所有include文件.

```
1 # Read .config if it exist, otherwise ignore
2 -include include/config/auto.conf
3
4 include scripts/kbuild.include
```

待编译目录树下的 Makefile ， 如有 kbuild ， kbuild 优先；唯一使用 kbuild 而不是 Makefile 的就是顶层目录.

```
# The filename kbuild has precedence over Makefile
kbuild-dir := $(if $(filter /%,$(src)),$(src),$(srctree)/$(src))
include $(if $(wildcard $(kbuild-dir)/kbuild), $(kbuild-dir)/kbuild, $(kbuild-dir)/Makefile)

include scripts/Makefile.lib
```

由于 init/Makefile 并没有定义 hostporgs-y hostprogs-m ,所以 scripts/Makefile.host 并没有被包括。

b. 变量的确定.

假设 make 缺省编译， vmlinux，bzImag e和 modules 为编译目标，顶层 Makefile 中的主要变量值为:

```
KBUILD_BUILTIN :=1
KBUILD_MODULES :=1
```

由于顶层 Makefile 执行了 export KBUILD_MODULES KBUILD_BUILTIN ， 所以 Makefile.build 中这些变量也是可见的。

Makefile.build 中的变量值: src := $(obj) 所以 src := init 。 init/Makefile 中只定义了 obj-y obj-m ， 所以 Makefile.build 中的变量初始值如下:

```
lib-target :=
extra-y :=
subdir-ym :=
always :=
obj-m :=
obj-n :=
obj- :=
obj-y := main.o version.o mouts.o initramfs.o calibrate.o
```

```
ifneq ($(strip $(obj-y) $(obj-m) $(obj-n) $(obj-) $(lib-target)),)
builtin-target := $(obj)/built-in.o
endif
```

所以builtin-target := init/built-in.o

Makefile.lib 主要作用就是将包含进来的 Makefile(init/Makefile) 中的各种定义重新分类定义:

```
obj-y: 将obj-y中包含目录的dir/替换为: dir/built-in.o
obj-m: 将obj-m中含有dir/的过滤掉
subdir-ym: obj-y和obj-m中的目录集合
multi-used-y: obj-y中的复合目标文件
multi-used-m: obj-m中的复合目标文件
single-used-m:过滤掉multi-used-m剩下的obj-m
multi-objs-y: 所有obj-y中的复合文件所包含的目标文件总和
multi-objs-m: 所有obj-m中的复合文件所包含的目标文件总和
mult-objs: multi-obj-y和multi-objs-m的集合
subdir-objs-y: obj-y中非本地objects
```

real-objs-y: obj-y中除subdir-objs-y的objs,其中复合文件会展开，还包括extra-y
real-objs-m: obj-m(已无dir/)中所有objs,其中复合文件会展开.

init/Makefile 中 mounts.o 是由 do_mounts.o do_mounts_rd.o do_mounts_initrd.o do_mounts_md.o 复合而成。属于 composite object . init/Makefile 只存在一个 mounts.o , 如果有多个 compiste objects 的话， mult-objs-y/mult-objs-m 将是所有这些的合集

```
multi-objs-y := do_mounts.o do_mounts_rd.o \
                do_mounts_initrd.o do_mounts_md.o
multi-objs-m :=
mult-objs    := do_mounts.o do_mounts_rd.o \
                do_mounts_initrd.o do_mounts_md.o
```

而最后 Makefile.lib 会给所有的目标和依赖文件加上 $(obj)/ 前缀. 变成：

```
obj-y := init/main.o init/version.o init/mouts.o \
                    init/initramfs.o init/calibrate.o
obj-m :=
multi-used-y := init/mounts.o
multi-used-m :=
multi-objs-y :=
multi-objs-m := init/do_mounts.o init/do_mounts_rd.o \
                    init/do_mounts_initrd.o init/do_mounts_md.o
```

c.规则链的确定.
Makefile.build的缺省target即为__build.

```
1 __build: $(if $(KBUILD_BUILTIN),$(builtin-target) $(lib-target
     ) $(extra-y)) \
2           $(if $(KBUILD_MODULES),$(obj-m)) \
3           $(subdir-ym) $(always)
4          @:
```

本例中__build: init/built-in.o

```
1 $(builtin-target): $(obj-y) FORCE
2         $(call if_changed,link_o_target)
3 init/buit-in.依赖于oinit/main.o init/version.o init/mouts.o
     init/noinitramfs.o init/calibrate.o
4
5 $(multi-used-y) : %.o: $(multi-objs-y) FORCE
6         $(call if_changed,link_multi-y)
```

init/mounts.o 依赖于 init/do_mounts.o init/do_mounts_rd.o init/do_mounts_initrd.o init/do_mounts_md.o
init/%.o 依赖于 init/%.c ， 当然编译后形成的依赖关系远非这么简单

```
1 $(obj)/%.o: $(src)/%.c FORCE
2         $(call cmd,force_checksrc)
3         $(call if_changed_rule,cc_o_c)
```

至此，Makefile第一遍运行结束。

2. Makefile.build第二阶段
由于这是第一次执行，所有的命令都会被执行:

```
make -f scripts/Makefile.build obj=init
  CC      init/main.o
  CHK     include/linux/compile.h
  UPD     include/linux/compile.h
  CC      init/version.o
  CC      init/do_mounts.o
  CC      init/do_mounts_rd.o
  CC      init/do_mounts_initrd.o
  LD      init/mounts.o
  CC      init/initramfs.o
  CC      init/calibrate.o
  LD      init/built-in.o

  LD      init/mounts.o展开为:
  ld -m elf_i386 -m elf_i386  -r -o init/mounts.o init/do_mounts.o
          init/do_mounts_rd.o init/do_mounts_initrd.o
  LD      init/built-in.o展开为:
  ld -m elf_i386 -m elf_i386  -r -o init/built-in.o init/main.o
          init/version.o init/mounts.o init/initramfs.o init/calibrate.o
```

6.2 模块目标(Loadable modules goal- obj-m)

类似于 obj-y，obj-m 也分为 Signle object 和 Composite object 。但是，无论 Single object module 还是 composite object module ，模块的编译都是分成两个阶段，而且 kernel 源代码树目录下的 module 和 External module 略有差异:

6.2.1 make modules执行过程

阶段1 生成.o和.mod 根据 Makefile.build 中的 $(signle-used-m) 和 $(multi-used-m) 编译成 .o ,同时调用

```
@{ echo $(@:.o=.ko); echo $@; } > $(MODVERDIR)/$(@F:.o=.mod) 或
@{ echo $(@:.o=.ko); echo $(link_multi_deps); } \
              > $(MODVERDIR)/$(@F:.o=.mod)
```

写入$(MODVERDIR)/.<file>.mod

```
export MODVERDIR := $(if $(KBUILD_EXTMOD),\
                      $(firstword $(KBUILD_EXTMOD))/).tmp_versions
```

make缺省编译时: MODVERDIR := .tmp_versions

阶段2 依据.o和.mod调用Makefile.post生成.ko Makefile kernel目录下的modules

```
1 #       Build modules
2
3 PHONY += modules
4 modules: $(vmlinux-dirs) $(if $(KBUILD_BUILTIN),vmlinux)
5         @echo '  Building modules, stage 2.';
6         $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modpost
```

6.2.2 External module执行过程

当系统调用 make -C KERNELDIR="kernel/dir" M=`pwd` modules 编译外部模块时，目标执行顺序稍有不同，但是也分两个阶段:

顶层Makefile中涉及的代码：

```
export MODVERDIR := $(if $(KBUILD_EXTMOD),$(firstword $(
    KBUILD_EXTMOD))/).tmp_versions

# We are always building modules
KBUILD_MODULES := 1
PHONY += crmodverdir
crmodverdir:
        $(Q)mkdir -p $(MODVERDIR)
        $(Q)rm -f $(MODVERDIR)/*

PHONY += $(objtree)/Module.symvers
$(objtree)/Module.symvers:
        @test -e $(objtree)/Module.symvers || ( \
        echo; \
        echo "  WARNING: Symbol version dump $(objtree)/Module
            .symvers"; \
        echo "           is missing; modules will have no
            dependencies and modversions."; \
        echo )


module-dirs := $(addprefix _module_,$(KBUILD_EXTMOD))
PHONY += $(module-dirs) modules
$(module-dirs): crmodverdir $(objtree)/Module.symvers
        $(Q)$(MAKE) $(build)=$(patsubst _module_%,%,$@)

modules: $(module-dirs)
        @echo '  Building modules, stage 2.';
        $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modpost
```

阶段一: 1.crmodverdir 创建MODVERDIR，并删除目录下所有文件2.检查kernel 目录下Module.symvers是否存在3.调用 make -f scripts/Makefile.build obj= ‘外部模块目录‘，生成 xxx.o 和相应的 $(MODVERDIR)/.xxx.mod 。

阶段二: 调用make -f scripts/Makefile.modpos 生成.ko 调用 scripts/Makefile.modpost ，根据 .tmp_versions/.xxx.mod 重新生成 module .

6.2.3 Single object模块编译

当 CONFIG_8139TOO=m 时， driver/net/Makefile～

```
obj-$(CONFIG_8139TOO) += 8139too.o
```

Makefile.build会应用single-used-m规则

```
$(single-used-m): $(obj)/%.o: $(src)/%.c FORCE
        $(call cmd,force_checksrc)
        $(call if_changed_rule,cc_o_c)
        @{ echo $(@:.o=.ko); echo $@; } > $(MODVERDIR)/$(@F:.o
            =.mod)
```

生成 drivers/net/8139too.o 和 .tmp_versions/8139too.mod 。

6.2.4 Composite object模块编译

当.config中设置如下

```
CONFIG_EXT2_FS=m
CONFIG_EXT2_FS_XATTR=y
CONFIG_EXT2_FS_POSIX_ACL=y
CONFIG_EXT2_FS_SECURITY=y
CONFIG_EXT2_FS_XIP=y
```

fs/ext2/Makefile会编译ext2.ko

```
1 obj-$(CONFIG_EXT2_FS) += ext2.o
2
3 ext2-y := balloc.o dir.o file.o fsync.o ialloc.o inode.o \
4           ioctl.o namei.o super.o symlink.o
5
6 ext2-$(CONFIG_EXT2_FS_XATTR)     += xattr.o xattr_user.o
     xattr_trusted.o
7 ext2-$(CONFIG_EXT2_FS_POSIX_ACL) += acl.o
8 ext2-$(CONFIG_EXT2_FS_SECURITY)  += xattr_security.o
9 ext2-$(CONFIG_EXT2_FS_XIP)       += xip.o
```

Makefile.build会使用multi-used-m规则

```
1 $(multi-used-m) : %.o: $(multi-objs-m) FORCE
2         $(call if_changed,link_multi-m)
3         @{ echo $(@:.o=.ko); echo $(link_multi_deps); } > $(
             MODVERDIR)/$(@F:.o=.mod)
```

将各个.o链接成fs/ext2/ext2.o,同时会产生.tmp_versions/ext2.mod

6.2.5 Makefilemod.post实现分析

Makefile.modpost 负责将相应的 .o(8139too.o 和 ext2.o)编译成.ko的 (8139too.ko 和 ext2.ko)

Makefile.modpost 规则链：
Makefile.modpost的缺省目标为_modpost, 而_modpost的依赖目标为(以ext2为例):

```
1 _modpost: __modpost
```

_mospost依赖__modpost

```
1 __modpost: $(modules:.ko=.o) FORCE
2         $(call cmd,modpost) $(wildcard vmlinux) $(filter-out
             FORCE,$^)
```

__modpost依赖于要编译模块的.o，如ext2.o，调用scripts/mod/modpost生成.mod.c

```
1 _modpost: $(if $(KBUILD_MODPOST_NOFINAL), $(modules:.ko:.o),$(
     modules))
```

当 KBUILD_MODPOST_NOFINAL 为 1 时，不用编 .ko ;但是 KBUILD_MODPOST_NOFINAL 缺省为空，所以依赖目标为 $(modules) ，本例中为 ext2.ko 。

```
$(modules): %.ko :%.o %.mod.o FORCE
        $(call if_changed,ld_ko_o)
```

Static pattern规则，%.ko依赖于%.o和%.mod.o

```
$(modules:.ko=.mod.o): %.mod.o: %.mod.c FORCE
        $(call if_changed_dep,cc_o_c)
```

Static pattern规则，%.mod.o依赖于%.mod.c

```
$(modules:.ko=.mod.c): __modpost ;
```

.mod.c依赖于__modpost

以 ext2.ko 为例，可以看出规则的建立: 变量 modules:= fs/ext2/ext2.ko 缺省规则 _modpost . _modpost 依赖目标为 __modpost 和 fs/ext2/ext2.ko , __modpost 依赖目标为 fs/ext2/ext2.o ，属于强制目标，会产生 fs/ext2/ext2.mod.c

fs/ext2/ext2.ko依赖于fs/ext2/ext2.o和fs/ext2/ext2.mod.o
fs/ext2/ext2.mod.o依赖于fs/ext2/ext2.mod.c
fs/ext2/ext2.mod.c依赖于__modpost

所以最终的执行顺序为：

```
__modpost: fs/ext2/ext2.o FORCE
        scripts/mod/modpost -m -a -o Module.symvers -s fs/ext2
            /ext2.o

fs/ext2/ext2.mod.o: fs/ext2/ext2.mod.c
        CC      fs/ext2/ext2.mod.o

fs/ext2/ext2.ko: fs/ext2/ext2.o fs/ext2/ext2.mod.o
        ld -m elf_i386 -r -m elf_i386  --build-id -o fs/ext2/
            ext2.ko
           fs/ext2/ext2.o fs/ext2/ext2.mod.o
```

6.3 Descending down in directories

以fs/Makefile为例:

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

当 CONFIG_EXT2_FS=y 或 CONFIG_EXT2_FS=m 时 fs/Makefile 将进行递归编译(Descending down)，其实现如下：

Makefile.lib会对obj-m和obj-y进行重新定义和分类: 当CONFIG_EXT2_FS=y时

```
subdir-y        += ext2
```

obj-y 中原先的ext/ 将变为ext2/built-in.o

当CONFIG_EXT2_FS=m时
subdir-m   += ext2

而obj-m 中的ext2/将被过滤掉。

```
subdir-ym       := $(addprefix $(obj)/,$(subdir-ym))
```

对fs/Makefile而言subdir-ym 最终会加上前缀fs/

Makefile.build

```
__build: $(if $(KBUILD_BUILTIN),$(builtin-target) $(lib-target) $(extra-y)) \
         $(if $(KBUILD_MODULES),$(obj-m)) \
         $(subdir-ym) $(always)
        @:

# Descending
# ---------------------------------------------------------------------


PHONY += $(subdir-ym)
$(subdir-ym):
        $(Q)$(MAKE) $(build)=$@
```

由于 __build 为缺省目标， $(sundir-ym) 则为依赖目标，所以对于 fs/Makefile 来说，如果 CONFIG_EXT2_FS=[y|m] ,系统会最终调用:

```
make -f scripts/Makefile.build obj=fs/ext2
```

生成fs/ext2/built-in.o或者ext2.ko

6.4 Library file goals

lib.a目前只应用于lib/和arch/$(ARCH)/lib/，以arch/i386/lib/Makefile 为例:

6.4.1 一个lib.a的例子

arch/i386/lib/Makefile

```
#
# Makefile for i386-specific library files..
#


lib-y = checksum.o delay.o usercopy.o getuser.o putuser.o memcpy.o strstr.o \
        bitops.o semaphore.o string.o

lib-$(CONFIG_X86_USE_3DNOW) += mmx.o

obj-$(CONFIG_SMP)       += msr-on-cpu.o
```

当CONFIG_X86_USE_3DNOW=n，CONFIG_SMP＝y 时，lib目录会生成2个目标:

- lib/built-in.o 由所有的 obj-y 链接而成，本例中只包含 msr-on-cpu.o
- lib/liba.a 由所有的 lib-y 组成，本例中包含 checksum.o delay.o usercopy.o getuser.o putuser.o memcpy.o strstr.o bitops.o semaphore.o string.o

6.4.2 lib.a的实现分析

Makefile.lib中的lib-y变量

```
# Libraries are always collected in one lib file.
# Filter out objects already built-in

lib-y := $(filter-out $(obj-y), $(sort $(lib-y) $(lib-m)))
lib-y := $(addprefix $(obj)/,$(lib-y))
```

本例中由于 obj-y 不存在， lib-m 为空.最后会为所有的目录加上前缀 arch/i386/lib/

Makefile.build 规则实现

```
ifneq ($(strip $(lib-y) $(lib-m) $(lib-n) $(lib-)),)
lib-target := $(obj)/lib.a
endif
```

本例中lib-target := arch/i386/lib/lib.a

```
__build: $(if $(KBUILD_BUILTIN),$(builtin-target) $(lib-target
    ) $(extra-y)) \
         $(if $(KBUILD_MODULES),$(obj-m)) \
         $(subdir-ym) $(always)
        @:
```

本例中， make 缺省执行时 KBUILD_BUILTIN 和 KBUILD_MODULES 都为1，所以展开为:

```
__build: arch/i386/lib/built-in.o arch/i386/lib/lib.a
```

arch/i386/lib/built-in.o 和普通的目录 built-in.o 的生成没有任何区别，此处略去。我们将着重于 lib.a 的实现分析

```
#
# Rule to compile a set of .o files into one .a file
#
ifdef lib-target
quiet_cmd_link_l_target = AR      $@
cmd_link_l_target = rm -f $@; $(AR) $(EXTRA_ARFLAGS) rcs $@ $(
    lib-y)

$(lib-target): $(lib-y) FORCE
        $(call if_changed,link_l_target)

targets += $(lib-target)
endif
```

这就是最终生成lib.a的规则.

6.5 Hostprog

kbuild 通常也需要编译一些本机程序(host program ， 相对于可能不同平台的交叉编译)。 kbuild 处理 hostprog 的规则都集中于 Makefile.host 。

6.5.1 hostprog分类

Makefile.host 最终也是通过包含进 Makefile.build 实现作用的， Makefike.host 被包含的前提是源代码目录树下的 Makefile 中定义了 hostprog-y hostprog-m . Makefile.host 作用类同于 Makefile.lib ， 其主要作用是对 hostprog-y 和hostprog-m 进行分类重定义。

Host 程序按照依赖的 obj 多少， objs 中有无 .so ， obj 是否是由 c++ 编译的，可分为如下几类：

- 单个.c编译的hostprog
- 多个.o链接而成的hostprog
- objs中包含<hostprog>-cxxxobjs
- objs中包含share libarary

6.5.2 单个.c编译的hostprog

例子

```
scripts/Makefile
hostprogs-y := bin2c
```

实现分析: Makefile.host相关变量

```
__hostprogs := $(sort $(hostprogs-y) $(hostprogs-m))

# C code
# Executables compiled from a single .c file
host-csingle    := $(foreach m,$(__hostprogs),$(if $($(m)-objs
    ),,$(m)))
```

展开为

```
__hostprogs      := bin2c
host-csingle     := scripts/bin2c
```

Makefile.host相关规则

```
# Create executable from a single .c file
# host-csingle -> Executable
quiet_cmd_host-csingle  = HOSTCC  $@
      cmd_host-csingle  = $(HOSTCC) $(hostc_flags) -o $@ $< \
                $(HOST_LOADLIBES) $(HOSTLOADLIBES_$(@F))
$(host-csingle): $(obj)/%: $(src)/%.c FORCE
        $(call if_changed_dep,host-csingle)
```

展开为:

```
scripts/bin2c: scripts/bin2c.c
        gcc -Wp,-MD,scripts/.bin2c.d -Wall -Wstrict-prototypes
            -O2 -fomit-frame-pointer     -o scripts/bin2c
           scripts/bin2c.c
```

6.5.3 多个.o链接而成的hostprog

例子

```
scripts/kconfig/Makefile
hostprogs-y := kxgettext
kxgettext-objs  := kxgettext.o zconf.tab.o
```

Makefile.host变量:

```
__hostprogs   := kxgettext

# C executables linked based on several .o files
host-cmulti     := $(foreach m,$(__hostprogs),\
                   $(if $($(m)-cxxobjs),,$(if $($(m)-objs),$(m
                       ))))

# Object (.o) files compiled from .c files
host-cobjs      := $(sort $(foreach m,$(__hostprogs),$($(m)-
    objs)))
```

本例中展开为

```
host-cmulti   := scripts/kconfig/kxgettext
host-cobjs    := scripts/kconfig/kxgettext.o \
                 scripts/kconfig/zconf.tab.o
```

Makefile.host规则：

```
# Link an executable based on list of .o files, all plain c
# host-cmulti -> executable
quiet_cmd_host-cmulti   = HOSTLD  $@
      cmd_host-cmulti   = $(HOSTCC) $(HOSTLDFLAGS) -o $@ \
                          $(addprefix $(obj)/,$($(@F)-objs)) \
                          $(HOST_LOADLIBES) $(HOSTLOADLIBES_$(
                              @F))
$(host-cmulti): $(obj)/%: $(host-cobjs) $(host-cshlib) FORCE
        $(call if_changed,host-cmulti)
```

展开为：

```
scripts/kconfig/kxgettext: scripts/kconfig/kxgettext.o scripts
    /kconfig/zconf.tab.o
          gcc  -o scripts/kconfig/kxgettext scripts/kconfig/
              kxgettext.o scripts/kconfig/zconf.tab.o -lncurses
```

6.5.4 objs中包含<hostprog>-cxxxobjs

例子

```
scripts/kconfig/Makefile

hostprogs-y     := qconf
qconf-objs      := kconfig_load.o zconf.tab.o
qconf-cxxobjs   := qconf.o
```

Makefile.host变量:

```
# C++ code
# C++ executables compiled from at least on .cc file
# and zero or more .c files
host-cxxmulti   := $(foreach m,$(__hostprogs),$(if $($(m)-
    cxxobjs),$(m)))

# C++ Object (.o) files compiled from .cc files
host-cxxobjs    := $(sort $(foreach m,$(host-cxxmulti),$($(m)-
    cxxobjs)))
```

本例中

```
host-cobjs      := scripts/kconfig/qconf.o
host-cxxmulti   := scripts/kconfig/qconf
host-cxxobjs    := scripts/kconfig/kconfig_load.o \
                   scripts/kconfig/zconf.tab.o
```

Makefile.host规则:

```
# Link an executable based on list of .o files, a mixture of .
    c and .cc
# host-cxxmulti -> executable
quiet_cmd_host-cxxmulti = HOSTLD  $@
      cmd_host-cxxmulti = $(HOSTCXX) $(HOSTLDFLAGS) -o $@ \
                          $(foreach o,objs cxxobjs,\
                          $(addprefix $(obj)/,$($(@F)-$(o))))
                              \
                          $(HOST_LOADLIBES) $(HOSTLOADLIBES_$(
                              @F))
$(host-cxxmulti): $(obj)/%: $(host-cobjs) $(host-cxxobjs) $(
    host-cshlib) FORCE
        $(call if_changed,host-cxxmulti)
```

规则展开为:

```
scripts/kconfig/qconf: scripts/kconfig/kconfig_load.o scripts/
    kconfig/zconf.tab.o
        g++  -o scripts/kconfig/qconf  scripts/kconfig/
            kconfig_load.o scripts/kconfig/zconf.tab.o  scripts
            /kconfig/qconf.o -lncurses -L/usr/X11R6/lib -lqt-mt
             -laudio -lXt -ljpeg -lpng -lz -lXi -lXrender -
            lXrandr -lXcursor -lXinerama -lXft -lfreetype -
            lfontconfig -lXext -lX11 -lm -lSM -lICE -ldl -
            lpthread   -ldl
```

上面几个例子中都略去了%.c -> %.o的规则

```
# Create .o file from a single .c file
# host-cobjs -> .o
quiet_cmd_host-cobjs    = HOSTCC  $@
      cmd_host-cobjs    = $(HOSTCC) $(hostc_flags) -c -o $@ $<
$(host-cobjs): $(obj)/%.o: $(src)/%.c FORCE
        $(call if_changed_dep,host-cobjs)
```

6.5.5 objs中包含share libarary

当然如果share library存在的话，必须应用下面的规则:

首先必须把 .so 从 xxx-objs 过滤掉, 同时保证 host-cobjs 不包含 .so :

```
# Shared libaries (only .c supported)
# Shared libraries (.so) - all .so files referenced in "xxx-objs"
host-cshlib     := $(sort $(filter %.so, $(host-cobjs)))
# Remove .so files from "xxx-objs"
host-cobjs      := $(filter-out %.so,$(host-cobjs))
```

host-cshobjs为所有xxx.so所包含的xxx-objs展开后的集合：

```
#Object (.o) files used by the shared libaries
host-cshobjs    := $(sort $(foreach m,$(host-cshlib),$($(m:.so=-objs))))
```

用HOSTCC把单个.c编译成.o：

```
# Compile .c file, create position independent .o file
# host-cshobjs -> .o
quiet_cmd_host-cshobjs  = HOSTCC  -fPIC $@
      cmd_host-cshobjs  = $(HOSTCC) $(hostc_flags) -fPIC -c -o $@ $<
$(host-cshobjs): $(obj)/%.o: $(src)/%.c FORCE
        $(call if_changed_dep,host-cshobjs)
```

用HOSTCC将一组.o链接成.so：

```
# Link a shared library, based on position independent .o files
# *.o -> .so shared library (host-cshlib)
quiet_cmd_host-cshlib = HOSTLLD -shared $@
      cmd_host-cshlib = $(HOSTCC) $(HOSTLDFLAGS) -shared -o $@ \
  $(addprefix $(obj)/,$($(@F:.so=-objs))) \
  $(HOST_LOADLIBES) $(HOSTLOADLIBES_$(@F))
$(host-cshlib): $(obj)/%: $(host-cshobjs) FORCE
$(call if_changed,host-cshlib)
```

6.5.6 hostprog的执行

Makefile.build 中的__build为缺省目标

```
__build: $(if $(KBUILD_BUILTIN),$(builtin-target) $(lib-target) $(extra-y)) \
         $(if $(KBUILD_MODULES),$(obj-m)) \
         $(subdir-ym) $(always)
        @:
```

hostprog-y hostprog-m 并没有作为 __build 的依赖目标出现,要想编译 hostprogs ,有两种方法:

1. 加到always中去如scripts/kconfig/Makefile

```
always := dochecklxdialog
```

2. 作为某个目标的依赖scripts/kconfig/Makefile

```
oldconfig: $(obj)/conf
        $< -o arch/$(ARCH)/Kconfig
```

当执行oldconfig时，$(obj)/conf必须首先执行，而conf并未加到always.

6.5.7 hostprog的Descending down

Makefile.host中只定义了

```
obj-dirs += $(host-objdirs)
```

其中 host-objdirs 是 hostprog-y 和 hostprog-m 中的目录集合，但是 Makefile.host 并未将 $(obj-dirs) 加入 __build 或 subdir-ym ，所以也就无法支持 Descending down 了.

6.5.8 hostprog-y和hostprog-m

以scripts/Makefile为例,

```
hostprogs-$(CONFIG_KALLSYMS)      += kallsyms
hostprogs-$(CONFIG_LOGO)          += pnmtologo
hostprogs-$(CONFIG_VT)            += conmakehash
hostprogs-$(CONFIG_PROM_CONSOLE) += conmakehash
hostprogs-$(CONFIG_IKCONFIG)      += bin2c

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

CONFIG_IKCONFIG为y或是m是并没有区别.

6.6 Makefile.clean

kbuild在3种层次上分别提供了3种clean目标:

- clean: 删除大部分产生的文件，但是，依然能够编译外部模块。
- mrproper: 删除所有产生的文件和配置文件
- distclean : 恢复系统到最原始的状态，出了 mrproper 外，还会删除备份 patch ，和 cscope，tag 等

6.6.1 clean

Makefile中的clean规则，首先定义待删除的目录和文件:

```
# Directories & files removed with 'make clean'
CLEAN_DIRS  += $(MODVERDIR)
CLEAN_FILES +=  vmlinux System.map \
                .tmp_kallsyms* .tmp_version .tmp_vmlinux* .
                   tmp_System.map
```

注意clean规则中用到的rm-dirs和rm-files为模式指定变量

```
# clean - Delete most, but leave enough to build external
   modules
#
clean: rm-dirs  := $(CLEAN_DIRS)
clean: rm-files := $(CLEAN_FILES)
clean-dirs      := $(addprefix _clean_,$(srctree) $(vmlinux-
   alldirs))

PHONY += $(clean-dirs) clean archclean
$(clean-dirs):
        $(Q)$(MAKE) $(clean)=$(patsubst _clean_%,%,$@)

clean: archclean $(clean-dirs)
        $(call cmd,rmdirs)
        $(call cmd,rmfiles)
        @find . $(RCS_FIND_IGNORE) \
                \( -name '*.[oas]' -o -name '*.ko' -o -name
                   '.*.cmd' \
                -o -name '.*.d' -o -name '.*.tmp' -o -name '*.
                   mod.c' \
                -o -name '*.symtypes' \) \
                -type f -print | xargs rm -f
```

clean的执行顺序为：

- archclean 这时 arch/$(ARCH)/Makefile 提供的命令
- $(clean-dirs) 会调用 make -f scripts/Makefile.clean obj=dir 删除每个目录中的 clean-files .

6.6.2 Makefile.clean

Makefile.clean 是和 Makefile.build 平行的，这和前面提到的 Makefile.modpost 功能类似，并不包含到 Makefile.build 中去.

Makefile.clean缺省规则__clean.

Makefile.clean也会包含dir目录下的Makefile

```
# The filename kbuild has precedence over Makefile
kbuild-dir := $(if $(filter /%,$(src)),$(src),$(srctree)/$(src
   ))
include $(if $(wildcard $(kbuild-dir)/kbuild), $(kbuild-dir)/
   kbuild, $(kbuild-dir)/Makefile)

extra-y EXTRA_TARGETS, always, targets clean-files
host-progs hostprogrs-y host-progras-m hostprogs- 都为目录下所定
   义cleanMakefile


Makefile.中定义的变量为：clean
__clean-files   := $(extra-y) $(EXTRA_TARGETS) $(always) \
                   $(targets) $(clean-files)             \
                   $(host-progs)                         \
                   $(hostprogs-y) $(hostprogs-m) $(hostprogs-)

```

```
15 __clean-files   := $(wildcard
                                                \
16                    $(addprefix $(obj)/, $(filter-out /%, $(
                         __clean-files))) \
17                    $(filter /%, $(__clean-files)))
18
19 __clean-dirs    := $(wildcard
                                                \
20                    $(addprefix $(obj)/, $(filter-out /%, $(
                         clean-dirs)))    \
```

subdir-y, subdir-m, subdir-n, subdir-, 分别包含 dir 目录下 Makefile 中 obj-y，objm，obj-n obj- 中所含的目录

```
1  __subdir-y       := $(patsubst %/,%,$(filter %/, $(obj-y)))
2  subdir-y         += $(__subdir-y)
3  __subdir-m       := $(patsubst %/,%,$(filter %/, $(obj-m)))
4  subdir-m         += $(__subdir-m)
5  __subdir-n       := $(patsubst %/,%,$(filter %/, $(obj-n)))
6  subdir-n         += $(__subdir-n)
7  __subdir-        := $(patsubst %/,%,$(filter %/, $(obj-)))
8  subdir-          += $(__subdir-)
9  subdir-ym        := $(sort $(subdir-y) $(subdir-m))
10 subdir-ymn       := $(sort $(subdir-ym) $(subdir-n) $(subdir-))
11 subdir-ymn       := $(addprefix $(obj)/,$(subdir-ymn))
```

Makefile.clean规则链为：

```
1  __clean: $(subdir-ymn)
2  ifneq ($(strip $(__clean-files)),)
3          +$(call cmd,clean)
4  endif
5  ifneq ($(strip $(__clean-dirs)),)
6          +$(call cmd,cleandir)
7  endif
8  ifneq ($(strip $(clean-rule)),)
9          +$(clean-rule)
10 endif
11         @:的依赖目标为
12 __clean$(subdir-ymn)
13
14 $(subdir-ymn):
15         $(Q)$(MAKE) $(clean)=$@
```

最终__clean通过subdir-ymn实现遍历子目录(Descending down).

6.6.3 mrproper

Makefile中mrproper待删除的目录和文件:

```
1  # Directories & files removed with 'make mrproper'
2  MRPROPER_DIRS  += include/config include2 usr/include
3  MRPROPER_FILES += .config .config.old include/asm .version .
       old_version \
4                    include/linux/autoconf.h include/linux/
                        version.h      \
```

```
                  include/linux/utsrelease.h
                                                  \
                  Module.symvers tags TAGS cscope*
# mrproper - Delete all generated files, including .config
#
mrproper: rm-dirs  := $(wildcard $(MRPROPER_DIRS))
mrproper: rm-files := $(wildcard $(MRPROPER_FILES))
mrproper-dirs      := $(addprefix _mrproper_,Documentation/
    DocBook scripts)
```

Makefile中mrproper规则链:

```
PHONY += $(mrproper-dirs) mrproper archmrproper
$(mrproper-dirs):
        $(Q)$(MAKE) $(clean)=$(patsubst _mrproper_%,%,$@)

mrproper: clean archmrproper $(mrproper-dirs)
        $(call cmd,rmdirs)
        $(call cmd,rmfiles)
```

mrproper的执行顺序为：

- clean
- archmrproper，是由 arch/$(ARCH)/Makefile 所定义.
- 对 Documentation/DocBook scripts 目录分别调用: make -f scripts/Makefile.clean obj=dir 删除每个目录中的 clean-files
- 删除$(MRPROPER_DIRS) 目录
- 删除$(MRPROPER_FILES) 文件

6.6.4 distclean

Makefile中distclean的定义相对简单

```
distclean: mrproper
        @find $(srctree) $(RCS_FIND_IGNORE) \
                \( -name '*.orig' -o -name '*.rej' -o -name
                    '*~' \
                -o -name '*.bak' -o -name '#*#' -o -name '.*.
                    orig' \
                -o -name '.*.rej' -o -size 0 \
                -o -name '*%' -o -name '.*.cmd' -o -name 'core
                    ' \) \
                -type f -print | xargs rm -f
```

6.6.5 EXTMOD clean

EXTMOD clean的命令行形式为

```
make M=dir clean
```

EXTMOD clean目标的实现为: Makefile

```
clean-dirs := $(addprefix _clean_,$(KBUILD_EXTMOD))

PHONY += $(clean-dirs) clean
$(clean-dirs):
        $(Q)$(MAKE) $(clean)=$(patsubst _clean_%,%,$@)

clean:  rm-dirs := $(MODVERDIR)
clean: $(clean-dirs)
        $(call cmd,rmdirs)
        @find $(KBUILD_EXTMOD) $(RCS_FIND_IGNORE) \
                \( -name '*.[oas]' -o -name '*.ko' -o -name
                   '.*.cmd' \
                -o -name '.*.d' -o -name '.*.tmp' -o -name '*.
                   mod.c' \) \
                -type f -print | xargs rm -f
```

EXTMOD clean调用顺序为:

- make -f scripts/Makefile.clean obj=dir , dir 为外部模块目录。
- 删除 $(MODVERDIR) 即为 dir/.tmp_version/
- 删除所产生的所有文件.

6.7 Architecure Makefile

相对于 Makefile 包含的都是一些通用的编译参数和目标， arch/$(ARCH)/Makefile 则定义目标和变量则是和具体平台相关的。

6.7.1 平台相关的变量

编译连接选项

```
LDFLAGS              通用$(LD)选项
LDFLAGS_MODULE       联接模块时的联接器的选项
LDFLAGS_vmlinux      联接vmlinux时的选项
OBJCOPYFLAGS     objcopy flags
AFLAGS               $(AS) 汇编编译器选项
CFLAGS               $(CC) 编译器选项
CFLAGS_KERNEL     $(CC) built-in 选项
CFLAGS_MODULE     $(CC) modules选项
```

这些变量的默认值通常在顶层 Makefile ， 在 arch/$(ARCH)/Makefile 扩充或修改

vmlinux 相关的变量: head-y, init-y, core-y, libs-y, drivers-y, net-y , 这些变量通常在顶层 Makefile 中有定义(head-y 除外）， arch/$(ARCH)/Makefile 中增加特定平台相关代码。

```
vmlinux-init := $(head-y) $(init-y)
vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
vmlinux-all  := $(vmlinux-init) $(vmlinux-main)
vmlinux-lds  := arch/$(ARCH)/kernel/vmlinux.lds
```

arch/i386/Makefile中定义了:

```
head-y := arch/i386/kernel/head.o arch/i386/kernel/init_task.o
libs-y += arch/i386/lib/
core-y += arch/i386/kernel/ \
          arch/i386/mm/ \
          arch/i386/$(mcore-y)/ \
          arch/i386/crypto/
```

相应的drivers-y:

```
drivers-$(CONFIG_MATH_EMULATION) += arch/i386/math-emu/
drivers-$(CONFIG_PCI) += arch/i386/pci/
......
```

6.7.2 平台相关的目标

archprepare archprepare在顶层Makefile中有定义Makefile

```
archprepare: prepare1 scripts_basic

prepare0: archprepare FORCE
        $(Q)$(MAKE) $(build)=.
        $(Q)$(MAKE) $(build)=. missing-syscalls
```

不过这并不妨碍 arch/$(ARCH)/Makefile 中 archprepare 的定义,因为顶层 Makefile archprepare 只有目标和依赖，却没有命令体.如: arch/arm/Makefile

```
archprepare: maketools
```

archhelp(可选）当执行 make help 时，顶层 Makefile 会打印出 archhelp 的详细命令.注意： archhelp 并不是目标，只是用 define 定义的 command block，以 arch/i386/Makefile 为例:

```
define archhelp
  echo  '* bzImage      - Compressed kernel image (arch/$(ARCH
     )/boot/bzImage)'
  echo  '  install      - Install kernel using'
  echo  '                   (your) ~/bin/installkernel or'
  echo  '                   (distribution) /sbin/installkernel
     or'
  echo  '                   install to $$(INSTALL_PATH) and run
      lilo'
  echo  '  bzdisk       - Create a boot floppy in /dev/fd0'
  echo  '  fdimage      - Create a boot floppy image'
  echo  '  isoimage     - Create a boot CD-ROM image'
endef
```

archclean(必须) 执行arch相关的clean arch/i386/Makefile

```
archclean:
        $(Q)$(MAKE) $(clean)=arch/i386/boot
```

install(可选) arch/i386/Makefile提供了这个命令，其实现如下:

```
install:
```

```
2         $(Q)$(MAKE) $(build)=$(boot) BOOTIMAGE=$(KBUILD_IMAGE)
              install
```

arch/$(ARCH)/kernel/vmlinux.lds : 顶层 Makefile 中 vmlinux-lds 是 vmlinux.o 的依赖目标之一，并且定义了:

```
1 vmlinux-lds  := arch/$(ARCH)/kernel/vmlinux.lds
```

vmlinux.lds却是由arch/$(ARCH)/Makefile实现的。

```
1 arch/i386/kernel/Makefile
2         extra-y := head.o init_task.o vmlinux.lds
3
4         #Makefile
5         export CPPFLAGS_vmlinux.lds += -P -C -U$(ARCH)
```

Makefile.build

```
1 __build: $(if $(KBUILD_BUILTIN),$(builtin-target) $(lib-target
     ) $(extra-y)) \
2          $(if $(KBUILD_MODULES),$(obj-m)) \
3          $(subdir-ym) $(always)
4         @:
5
6 # Linker scripts preprocessor (.lds.S -> .lds)
7 # ---------------------------------------------------------
8 quiet_cmd_cpp_lds_S = LDS     $@
9       cmd_cpp_lds_S = $(CPP) $(cpp_flags) -D__ASSEMBLY__ -o $@
            $<
10
11 $(obj)/%.lds: $(src)/%.lds.S FORCE
12         $(call if_changed_dep,cpp_lds_S)
```

Makefile.build 中， __build 缺省目标已经包含了 extra-y ,也提供了了从 (.ld.s -> .lds) 的命令

architecure 相关的 boot image . 通常，不同的平台对 vmlinux.o 有不同的处理方式，如将压缩 vmlinux 文件，写入启动代码，并将其拷贝到正确的位置。

arch/i386/Makefile

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

6.7.3 其他辅助变量和目标

extra-y 列出了在当前目录下，所要创建的附加文件，而不包含任何已包含在 obj-* 中的文件。用 extra-y 列目标，主要是两个目的：

1. 可以使kbuild检查命令行是否发生变化- 使用$(call if_changed,xxx) 的时候
2. 让 kbuild 知道哪些文件要在 "make clean" 时删除, __clean_file 中包含 $(extra-y)

例子：arch/i386/kernel/Makefile

```
1         extra-y := head.o init_task.o vmlinux.lds
```

在此例子中， extra-y 用来列出所有只编译，但不联接到 built-in.o 的目标文件。事实上 head.o 和 init_task.o 出现在:

arch/i386/Makefile

```
1 head-y := arch/i386/kernel/head.o arch/i386/kernel/init_task.o
```

6.8 modules_install

kernel 目录树下的模块和 External 模块的 modules_install 还是略有区别的, make modules_install 和 make M=dir modules_install 的实现还是最终会调用 scripts/Makefile.modinst .

6.8.1 make modules_install

变量

```
1 MODLIB  = $(INSTALL_MOD_PATH)/lib/modules/$(KERNELRELEASE)
2 export MODLIB
```

其中， INSTALL_MOD_PATH 缺省为空，但也可以在 make 命令行上指定. export MODLIB 使 scripts/Makefile.modinst 可见.

规则: modules_install 依赖于 _modins_ 和 _modinst_post 目标

```
1 # Target to install modules
2 PHONY += modules_install
3 modules_install: _modinst_ _modinst_post
```

modinst 目标首先会检查 depmod 的存在，然后作一些目录清理和链接，最终调用 Makefile.modinst

```
1  PHONY += _modinst_
2  _modinst_:
3          @if [ -z "`$(DEPMOD) -V 2>/dev/null | grep module-init
               -tools`" ]; then \
4                  echo "Warning: you may need to install module-
                       init-tools"; \
5                  echo "See http://www.codemonkey.org.uk/docs/
                       post-halloween-2.6.txt";\
6                  sleep 1; \
7          fi
8          @rm -rf $(MODLIB)/kernel
9          @rm -f $(MODLIB)/source
10         @mkdir -p $(MODLIB)/kernel
11         @ln -s $(srctree) $(MODLIB)/source
12         @if [ ! $(objtree) -ef  $(MODLIB)/build ]; then \
13                 rm -f $(MODLIB)/build ; \
14                 ln -s $(objtree) $(MODLIB)/build ; \
15         fi
16         $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modinst
```

_modinst_post 运行 depmod 更新模块之间的依赖关系.

```
PHONY += _modinst_post
_modinst_post: _modinst_
        if [ -r System.map -a -x $(DEPMOD) ]; then $(DEPMOD) -
            ae -F System.map $(depmod_opts) $(KERNELRELEASE);
            fi
```

6.8.2 make M=dir modules_install

相关变量:

```
install-dir := $(if $(INSTALL_MOD_DIR),$(INSTALL_MOD_DIR),extra)
```

install-dir 只用于顶层 Makefile ,对 Makefile.modinst 并不可见,其作用就是建立该目录. INSTALL_MOD_DIR 缺省为空,当然也可以在 make 命令行指定.

规则: modules_install 同样依赖于 _emodinst 和 _emodinst_post 目标.

```
PHONY += modules_install
modules_install: _emodinst_ _emodinst_post
```

emodinst 目标首先会检查 install-dir 的存在，然后调用 Makefile.modinst 。

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

_emodinst_post 目标会调用 cmd_depmod 运用 depmod 更新模块依赖关系.

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

6.8.3 Makefile.modinst

Makefile.modinst 同 Makefile.build 一样也是一个独立的 Makefile ， 不同得是 Makefile.modinst 实现非常简单. 变量的确定:

```
__modules := $(sort $(shell grep -h '\.ko' /dev/null $(
    wildcard $(MODVERDIR)/*.mod)))
modules := $(patsubst %.o,%.ko,$(wildcard $(__modules:.ko=.o))
    )
```

modules 变量是 $(MODVERDIR) 下的所有 <模块>.ko 。
对于 kernel 模块来说, MODVERDIR=<kernel root dir>/.tmp_versions 。
对于外部变量而言， MODVERDIR=<dir>/.tmp_versions 。

```

# Modules built outside the kernel source tree go into extra
    by default
INSTALL_MOD_DIR ?= extra
ext-mod-dir = $(INSTALL_MOD_DIR)$(subst $(KBUILD_EXTMOD),,$(@D
    ))
```

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

modinst_dir为 模块安装目录，外部模块缺省为 extra /<模块相对于 KBUILD_EXTMOD 的目录>， kernel 模块为 kernel /<模块相对于 kernel root 目录>。

规则: Makefile.modinst 缺省规则为 __modinst ,其依赖目标为 $(modules)

```
1 PHONY += $(modules)
2 __modinst: $(modules)
3         @:
```

$(modules)展开为*.ko，这是一个多目标规则

```
1 $(modules):
2         $(call cmd,modules_install,$(MODLIB)/$(modinst_dir))
```

cmd_modules_install 的作用就是 cp 模块到相应目录，然后调用 mod_strip_cmd strip 模块 .mod_strip_cmd 是定义在顶层 Makefile ， 然后 export 出来的.

```
1 quiet_cmd_modules_install = INSTALL $@
2       cmd_modules_install = mkdir -p $(2); cp $@ $(2) ; $(
          mod_strip_cmd) $(2)/$(notdir $@)
```

6.9 kbuild.include

kbuild.include 定义了所有的公共变量，被每个独立规则的 Makefile 所包含，例如 Makefile.build, Makefie.modpost 等。

6.9.1 echo_cmd和cmd

```
1 echo-cmd = $(if $($(quiet)cmd_$(1)),\
2         echo '   $(call escsq,$($(quiet)cmd_$(1)))$(echo-why)
              ';)
```

echo_cmd 用于输出一些编译时的辅助信息: 当 V=1, quiet 为空，这时就会输出 cmd_$(1) 的信息，这是编译所用的详细命令. 当V不等于1时， quiet=quiet_ ，一般的每个 cmd 都定义有 quiet_$(1) . 而当 make 命令行中 (MAKEFILES) 含有-s时， quiet=silient_ ，而一般的 cmd 都没有定义 silent_$(1) ，所以什么都不输出.

看个例子:

```
1 quiet_cmd_as_o_S = AS       $@
2 cmd_as_o_S       = $(CC) $(a_flags) -c -o $@ $<
```

当V＝1，输出AS arch/i386/kernel/entry.o V不等于1时

```
gcc -m32 -Wp,-MD,arch/i386/kernel/.entry.o.d -nostdinc \
   -isystem /usr/lib/gcc/i486-linux-gnu/4.2.4/include \
   -D__KERNEL__ -Iinclude  -include include/linux/autoconf.h \
   -D__ASSEMBLY__ -DCONFIG_AS_CFI=1 \
   -DCONFIG_AS_CFI_SIGNAL_FRAME=1 -Iinclude/asm-i386/mach-default\
   -c -o arch/i386/kernel/entry.o arch/i386/kernel/entry.S
```

```
1 # printing commands
2 cmd = @$(echo-cmd) $(cmd_$(1))
```

cmd 有两部分组成: echo-cmd 和 cmd 具体命令的执行,由于 echo-cmd 展开后以;结尾，所以一行可以有两条命令.

6.9.2 if_changed*

kbuild.include 提供了3个 if_changed* 函数: if_changed, if_changed_dep, if_changed_rule 。

```
1 if_changed = $(if $(strip $(any-prereq) $(arg-check)),
                                \
2         @set -e;
                                    \
3         $(echo-cmd) $(cmd_$(1));
                                                        \
4         echo 'cmd_$@ := $(make-cmd)' > $(dot-target).cmd)
```

当目标的依赖有更新或者编译时的命令行有改动时， if_changed 就执行命令体: $(echo-cmd) 输出编译信息， $(cmd_$(1)) 执行该命令.同时把 cmd_xxx 写入 .xxx.cmd

if_changed_dep 和 if_changed 稍有不同，除了输出信息和执行命令外，还会执行 fixdep 规则.

```
1 if_changed_dep = $(if $(strip $(any-prereq) $(arg-check) ),
                          \
2         @set -e;
                                    \
3         $(echo-cmd) $(cmd_$(1));
                                                        \
4         scripts/basic/fixdep $(depfile) $@ '$(make-cmd)' > $(
              dot-target).tmp;\
5         rm -f $(depfile);
                                    \
6         mv -f $(dot-target).tmp $(dot-target).cmd)
```

if_changed_rule 当依赖和命令行参数变化时，会直接执行 rule_xxx .

```
1 if_changed_rule = $(if $(strip $(any-prereq) $(arg-check) ),
                           \
2         @set -e;
                                    \
3         $(rule_$(1)))
```

6.9.3 if_changed*

用于编译 kernel 的编译器通常随着版本，平台的不同，所支持的命令选项也不同，因此 kbuild 引入了一系列的 option :

```
as-option, ld-option, cc-等option我们来看一个例子.:
# cc-option
# Usage: cflags-y += $(call cc-option,-march=winchip-c6,-march=i586)

cc-option = $(call try-run,\
        $(CC) $(CFLAGS) $(1) -S -xc /dev/null -o "$$TMP",$(1),$(2))

# try-run
# Usage: option = $(call try-run, $(CC)...-o "$$TMP",option-ok,otherwise)
# Exit code chooses option. "$$TMP" is can be used as temporary file and
# is automatically cleaned up.
try-run = $(shell set -e;               \
        TMP="$(TMPOUT).$$$$.tmp";       \
        if ($(1)) >/dev/null 2>&1;      \
        then echo "$(2)";               \
        else echo "$(3)";               \
        fi;                             \
        rm -f "$$TMP")
```

其中 -xc 指定输入文件格式为c格式，而不是缺省的按后缀判断。 try-run 的作用就是如果该编译器支持该选项，则编译成功，则返回 try-run 的第二个参数，否则返回第三个参数。 cc-option 通常的用法是:

arch/i386/Makefile

```
# prevent gcc from keeping the stack 16 byte aligned
CFLAGS += $(call cc-option,-mpreferred-stack-boundary=2)
```

cc-option 函数只有一个参数，第二个参数为空，也就是说如果不支持， cc-option 没有任何改变。当然也有上面注释中的例子:

```
cflags-y += $(call cc-option,-march=winchip-c6,-march=i586)
```

cc_option有两个参数，也就是说参数1,如果失败，则选择参数2.这种用法不常见.

7 kbuild相关专题

如果抛开所有涉及到的 kernel 特性，而只讲述 kbuild ，就完全体现不出来 kbuild 的重要性了，而且 kbuild 并非全部只是 Makefile ，它也包含很多工具，例如用来做提取符号的 kallsym ，调整依赖关系的 fixdep 解析 kconfig 并生成 .config 的 conf/mconf/qconf .

7.1 Dependency tracing

7.1.1 kbuild dependency tracing

细心的读者一定会发现， kbuild 并没有定义 %.d:%c 之类的模式规则或单个规则，那么当从 .c 编成 .o 时, .d 又是如何生成的呢？

Makefile.build

```
define rule_cc_o_c
        $(call echo-cmd,checksrc) $(cmd_checksrc)
                                  \
        $(call echo-cmd,cc_o_c) $(cmd_cc_o_c);
                                  \
        $(cmd_modversions);
                                                                  \
        scripts/basic/fixdep $(depfile) $@ '$(call make-cmd,
            cc_o_c)' >    \
                                      $(dot-target).tmp;
                                                                    \
        rm -f $(depfile);
                                  \
        mv -f $(dot-target).tmp $(dot-target).cmd
endef

# Built-in and composite module parts
$(obj)/%.o: $(src)/%.c FORCE
        $(call cmd,force_checksrc)
        $(call if_changed_rule,cc_o_c)
```

scripts/Makefile.build 这条规则除了把 .c 编译成 .o 外，也调用了一些貌似生成 .d 的命令，这个和我们一般项目生成 .d 的方式有很大的不同

注意一个细节:模式规则 %.o:%.c 有一个 .PHONY 依赖目标，只要 .o 存在于最终目标的规则链中，就会直接导致这个规则会每次执行其命令体，这难道不是一种效率很低的方式吗？而且而 Makefile.build 到到处都充斥着这样的规则。事实上编译命令并不会每次都执行，靠的就是 if_changed_rule ，它的主要作用就是检查是否有：依赖目标的更新和编译时命令行选项的改变，如有变动，再执行编译命令也不晚。

```
if_changed_rule = $(if $(strip $(any-prereq) $(arg-check) ),
                    \
        @set -e;

            \
        $(rule_$(1)))
```

我们也可以看到 rule_cc_o_c 中涉及到了 $(cmd_modversions) ，这是一个和模块版本有关的命令，这将在下一节模块版本中详细解释，现在可以暂时忽略。

我们以 init/main.c 为例来解释 Dependency graph 的建立。当编译生成 init/main.o 时，

```
init/main.o: init/main.c FORCE
        $(call cmd,force_checksrc)
        $(call if_changed_rule,cc_o_c)
```

由于 KBUILD_CHECKSRC=0, cmd_fore_check_src 为空，第一次执行 init/main.o 不存在时， rule_cc_o_c 肯定会执行且执行顺序为:

- 接着调用 cmd_cc_o_c ，生成 .o ， 同时生成 init/.main.o.d(-MD)

```
gcc -m32 -Wp,-MD,init/.main.o.d  -nostdinc -isystem \
    /usr/lib/gcc/i486-linux-gnu/4.2.4/include -D__KERNEL__
    -Iinclude  -include include/linux/autoconf.h \
    -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs
```

```
-fno-strict-aliasing -fno-common \
-Werror-implicit-function-declaration -O2 -pipe
-msoft-float -mregparm=3 -freg-struct-return \
-mpreferred-stack-boundary=2  -march=i586 \
-mtune=generic -ffreestanding -maccumulate-outgoing-args \
-DCONFIG_AS_CFI=1 -DCONFIG_AS_CFI_SIGNAL_FRAME=1 \
-Iinclude/asm-i386/mach-default -fomit-frame-pointer \
-g  -fno-stack-protector -Wdeclaration-after-statement
-Wno-pointer-sign    -D"KBUILD_STR(s)=#s" \
-D"KBUILD_BASENAME=KBUILD_STR(main)"  \
-D"KBUILD_MODNAME=KBUILD_STR(main)" \
-c -o init/.tmp_main.o init/main.c
```

- 由于 CONFIG_MODVERSIONS=y, $(cmd_modversions) 会由 init/.tmp_main.o 生成 init/main.o .其实现细节将在7.2.2详述。
- 由 scripts/basic/fixdep 生成 init/.main.o.cmd

```
scripts/basic/fixdep init/.main.o.d init/main.o \
     'gcc -m32 -Wp,-MD,init/.main.o.d  -nostdinc -isystem \
     /usr/lib/gcc/i486-linux-gnu/4.2.4/include -D__KERNEL__ \
     -Iinclude  -include include/linux/autoconf.h -Wall -Wundef \
     -Wstrict-prototypes -Wno-trigraphs -fno-strict-aliasing \
     -fno-common -Werror-implicit-function-declaration -O2 -pipe\
     -msoft-float -mregparm=3 -freg-struct-return
     -mpreferred-stack-boundary=2  -march=i586 -mtune=generic \
     -ffreestanding -maccumulate-outgoing-args -DCONFIG_AS_CFI=1\
     -DCONFIG_AS_CFI_SIGNAL_FRAME=1 -Iinclude/asm-i386/mach-default\
     -fomit-frame-pointer -g  -fno-stack-protector \
     -Wdeclaration-after-statement -Wno-pointer-sign \
     -D"KBUILD_STR(s)=\#s" -D"KBUILD_BASENAME=KBUILD_STR(main)" \
     -D"KBUILD_MODNAME=KBUILD_STR(main)" \
     -c -o init/.tmp_main.o init/main.c' > init/.main.o.tmp; \
     rm -f init/.main.o.d; mv -f init/.main.o.tmp init/.main.o.cmd'
```

fixdep的使用规则为:

```
fixdep <depfile> <target> <cmdline>
```

由于基本上每个 kernel .c 都间接包含了 include/autoconf.h (编译时 -include include/autoconf.h)， 如够你只改了 .config 中的某一项 CONFIG_THIS_DRIVER 时， make *config 会重新更性 include/autoconf.h ，这时问题就产生了：

只修改了某一项，却要影响几乎所有的文件，这种实现就十分低效。 kbuild 采用的是和 "mkdep" 一样的策略，当系统调用 make -f scripts/Makefile.build obj=silentoldconfig 时，系统会根据 .config 的内容为每个选项创建一个单独的 include/config/*.h . fixedep ，所要做得就是将所编译的源文件中包含的所有 CONFIG_XXX ，转化为依赖头文件 include/config/*.h 而不是一整个 include/autoconf.h ，当 .config 中的某一项改变时，系统会根据依赖关系只更新受影响的部分，而不是全部。同时， fixdep 也会将 cmd_init/main.o 写入依赖文件，当某个选项改变时，系统也会重新编译， fixdep 产生的最终结果为：

init/.main.o.cmd

```
cmd_init/main.o := \
    gcc -m32 -Wp,-MD,init/.main.o.d -nostdinc -isystem \
        /usr/lib/gcc/i486-linux-gnu/4.2.4/include \
        -D__KERNEL__ -Iinclude  -include \
        include/linux/autoconf.h -Wall -Wundef \
        -Wstrict-prototypes -Wno-trigraphs \
        -fno-strict-aliasing -fno-common \
        -Werror-implicit-function-declaration -O2 -pipe \
        -msoft-float -mregparm=3 -freg-struct-return \
        -mpreferred-stack-boundary=2  -march=i586 -mtune=
           generic\
        -ffreestanding -maccumulate-outgoing-args \
        -DCONFIG_AS_CFI=1 -DCONFIG_AS_CFI_SIGNAL_FRAME=1 \
        -Iinclude/asm-i386/mach-default -fomit-frame-pointer
        -g  -fno-stack-protector -Wdeclaration-after-statement
           \
        -Wno-pointer-sign \
        -D"KBUILD_STR(s)=\#s" \
        -D"KBUILD_BASENAME=KBUILD_STR(main)" \
        -D"KBUILD_MODNAME=KBUILD_STR(main)" \
        -c -o init/.tmp_main.o init/main.c

deps_init/main.o := \
  init/main.c \
    $(wildcard include/config/x86/local/apic.h) \
    $(wildcard include/config/acpi.h) \
    $(wildcard include/config/debug/rodata.h) \
    ......
  include/linux/types.h \
    ......
  include/video/edid.h \

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

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

而这些首次生成的 *.cmd 也会被随后重新编译时包含进 Makefile.build 成为规则的一部分:

```
targets := $(wildcard $(sort $(targets)))
cmd_files := $(wildcard $(foreach f,$(targets),$(dir $(f)).$(
   notdir $(f)).cmd))

ifneq ($(cmd_files),)
  include $(cmd_files)
endif
```

总体而言 kbuild 的 Dependency tracing 实现相对复杂，但是带来的好处有：

- kbuild 不仅能够依据普通的 .o : %c %.h 依赖规则更新，而且命令行的改变也能导致重新编译 .o 。
- kbuild 改变了由文件直接包含 include/linux/autoconf.h 的做法，现在改由编译命令行直接加上 -include include/linux/autoconf.h 。这样每个编译的源文件不管是否包含 CONFIG_XXX 选项， include/linux/autoconf.h 都会起作用而且加入其依赖头文件中，当然如果确实没有用到 CONFIG_XXX ， fixdep 会帮你去掉没用的依

赖。而老的做法的坏去是:假设某个源文件用到了 CONFIG_XXX ，却没有包含 autoconf.h ，那么就会当成什么都没有定义，无声无息的产生你很难以察觉的错误。

- fixdep 的引入将 include/linux/autoconf.h 进一步细化为只把源文件用到的 CONFIG_XXX 选项作为空的 include/config/XXX.h 加入依赖关系链.也就是说其他无关的 CONFIG_YYY 改动并不会导致包含 CONFIG_XXX 但不包含 CONFIG_YYY 的源文件重新编译，这就极大地提高了编译的效率。

7.1.2 普通的dependency tracing

一般普通的 dependency tracing 并不像 kbuild 实现的那样复杂，其中一种常见做法就是:

```
%.d: %.c
        set -e; rm -f $@; \
        $(CC) -MM $(CPPFLAGS) $< > $@.$$$$; \
        sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
        rm -f $@.$$$$
-include $(sources:.c=.d)
```

由于 %.d 被 Makefile 包含，而且 %.d 又是命令的目标，当 .d 不存在时， %.d:%.c 规则会生成 .d ，这个 .d 被更新(又是 include 文件)，从而导致上面提到的 Makefile 执行过程中的 Remaking .

这种做法的好处是: 所有的 .d 会由于 include 的关系首先生成，和 %.o:%.c 的规则不是互相混合的。但是如果模个依赖的头文件找不到(依赖路径不对)，对不起， Makefile 的 Remaking 就会无限循环下去.

7.2 Moduleversion

7.2.1 编译单个.o中的Moduleversion

Makefile.build 中对关于 Moduleversion (模块版本)的命令定义如下:

```

ifndef CONFIG_MODVERSIONS
cmd_cc_o_c = $(CC) $(c_flags) -c -o $@ $<

else
cmd_cc_o_c = $(CC) $(c_flags) -c -o $(@D)/.tmp_$(@F) $<
cmd_modversions =
                        \
        if $(OBJDUMP) -h $(@D)/.tmp_$(@F) | grep -q __ksymtab;
             then      \
                 $(CPP) -D__GENKSYMS__ $(c_flags) $<
                                       \
                 | $(GENKSYMS) $(if $(KBUILD_SYMTYPES),
                                       \
                                    -T $(@D)/$(@F:.o=.symtypes)) -a
                                       $(ARCH)    \
                 > $(@D)/.tmp_$(@F:.o=.ver);
                                       \
                                                                    \
```

```
14                  $(LD) $(LDFLAGS) -r -o $@ $(@D)/.tmp_$(@F)
                                    \
15                          -T $(@D)/.tmp_$(@F:.o=.ver);
                                                \
16                  rm -f $(@D)/.tmp_$(@F) $(@D)/.tmp_$(@F:.o=.ver
                        );          \
17          else
                                    \
18                  mv -f $(@D)/.tmp_$(@F) $@;
                                          \
19          fi;
20  endif
```

当CONFIG_MODVERSIONS未定义时:

- cmd_modversions命令体为空
- cmd_cc_o_c会直接将.c编译成.o而不做任何中间处理.

而当 CONFIG_MODVERSIONS=y 时， cmd_cc_o_c 会将 file.c 编译成 .tmp_file.o 而不是 file.o 。 cmd_modversions 会检查 .tmp_file.o 是否包含 __ksymtab ，也就是说 file.c 是否包含 EXPORT_SYMBOL(xxx); 如果没有 __ksymtab ， cmd_modversions 会将 .tmp_file.o 直接更名为file.o 。如果确实包含 __ksymtab ， cmd_modversions 会通过 genksyms 产生 xxxx (export symbol)的符号签名(checksum)，然后调用 linker 重新把这些符号以及这些符号的 checksum 链接进 file.o 。

我们还是以上面的 init/main.o 为例， cmd_modversions 执行如下:

```
if objdump -h init/.tmp_main.o | grep -q __ksymtab; \
then gcc -m32 -E -D__GENKSYMS__ -Wp,-MD,init/.main.o.d  \
  -nostdinc -isystem /usr/lib/gcc/i486-linux-gnu/4.2.4/include \
  -D__KERNEL__ -Iinclude  -include include/linux/autoconf.h \
  -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs \
  -fno-strict-aliasing -fno-common \
  -Werror-implicit-function-declaration -O2 -pipe \
  -msoft-float -mregparm=3 -freg-struct-return \
  -mpreferred-stack-boundary=2  -march=i586 \
  -mtune=generic -ffreestanding -maccumulate-outgoing-args \
  -DCONFIG_AS_CFI=1 -DCONFIG_AS_CFI_SIGNAL_FRAME=1 \
  -Iinclude/asm-i386/mach-default -fomit-frame-pointer -g  \
  -fno-stack-protector -Wdeclaration-after-statement \
  -Wno-pointer-sign    \
  -D"KBUILD_STR(s)=#s" -D"KBUILD_BASENAME=KBUILD_STR(main)"\
  -D"KBUILD_MODNAME=KBUILD_STR(main)" init/main.c | \
  scripts/genksyms/genksyms  -a i386 > init/.tmp_main.ver; \
  ld -m elf_i386 -m elf_i386 -r -o init/main.o init/.tmp_main.o
  -T init/.tmp_main.ver; rm -f init/.tmp_main.o init/.tmp_main.ver;
  else mv -f init/.tmp_main.o init/main.o; fi;;
```

注意 gcc -m32 -E -D__GENKSYMS__ ... init/main.c | scripts/genksyms/genksyms -a i386 > init/.tmp_main.ver 。 genksyms 将 init/main.c 预编译 (Preprocess) 的结果作为输入，其目的是将每个 export symbol 涉及到的数据结构，函数类型等转化会一个 checksum 作为 export symbol 的值。以 init/main.c 为例:

init/main.c中包含三个export symbol:

```
1 EXPORT_SYMBOL(system_state);
2 EXPORT_SYMBOL(reset_devices);
3 EXPORT_SYMBOL(loops_per_jiffy);
```

genksyms产生的文件.tmp.main.ver, 上述命令产生的结果为：

```
__crc_system_state = 0x2288378f ;
__crc_reset_devices = 0xc2e587d1 ;
__crc_loops_per_jiffy = 0xba497f13 ;
__crc___per_cpu_offset = 0x6b95785b ;
```

以init/.tmp.main.o中system_state输出符号为例：

```
$nm init/.tmp.main.o|grep system_state
         w __crc_system_state
0000000c r __kcrctab_system_state
0000002f r __kstrtab_system_state
00000018 r __ksymtab_system_state
00000000 B system_state
```

cmd_modversions执行命令：

```
ld -m elf_i386 -m elf_i386 -r -o init/main.o \
   init/.tmp_main.o -T init/.tmp_main.ver
init/main.o中的system_state为
2288378f A __crc_system_state
0000000c r __kcrctab_system_state
0000002f r __kstrtab_system_state
00000018 r __ksymtab_system_state
00000000 B system_state
```

此时 __crc_sysmte_state 的值为绝对值(类型A)，此绝对值即为 genksyms 计算出来的 checksum ，这个值在以后整个链接过程中都不会改变。

当内核加载模块时会比对模块所用符号checksum和内核相应符号 checksum ， 如果不符，则会报出 "kernel-module version mismatch " 错误.

7.2.2 编译模块.ko中的Moduleversion

上面的章节中已经提到一个模块的编译是分两个阶段的:

阶段一 <module>.c 编译成 <module>.o ，同时会生成一个 .tmp_versions/<module>.mod 的文件，以 fs/ext2/ext2.ko 为例:

```
fs/ext2/ext2.ko
fs/ext2/balloc.o fs/ext2/dir.o fs/ext2/file.o \
fs/ext2/fsync.o fs/ext2/ialloc.o fs/ext2/inode.o \
fs/ext2/ioctl.o fs/ext2/namei.o fs/ext2/super.o \
fs/ext2/symlink.o fs/ext2/xattr.o fs/ext2/xattr_user.o \
fs/ext2/xattr_trusted.o fs/ext2/acl.o fs/ext2/xattr_security.o
```

阶段二调用make -f scripts/Makefile.modpost 生成.ko Makefile.modpos会调用modpost

```
scripts/mod/modpost -m -a -o Module.symvers -s fs/ext2/ext2.o
```

产生 fs/ext2/ext2.mod.c , 把 module 中的 EXPORT_SYMBOL 符号写入 Module.symvers ，本例中 ext2.ko 没有输出符号，所以也就没有记录。在 Module.symvers 中会看到:

```
0x691c7f4f usb_hcd_pci_shutdown drivers/usb/core/usbcore EXPORT_SYMBOL
```

生成的fs/ext2/ext2.mod.c具有如下格式:

```
include <linux/module.h>
#include <linux/vermagic.h>
#include <linux/compiler.h>

MODULE_INFO(vermagic, VERMAGIC_STRING);

struct module __this_module
__attribute__((section(".gnu.linkonce.this_module"))) = {
 .name = KBUILD_MODNAME,
 .init = init_module,
#ifdef CONFIG_MODULE_UNLOAD
 .exit = cleanup_module,
#endif
 .arch = MODULE_ARCH_INIT,
};

static const struct modversion_info ____versions[]
__attribute_used__
__attribute__((section("__versions"))) = {
        { 0xc3a2c096, "struct_module" },
        { 0xa7e702c9, "mb_cache_entry_find_next" },
        { 0xdac163d0, "kmalloc_caches" },
        { 0x12da5bb2, "__kmalloc" },
        ... ...
        { 0xb45578b8, "memscan" },
        { 0x3a206eaf, "truncate_inode_pages" },
        { 0xdf929370, "fs_overflowgid" },
};

static const char __module_depends[]
__attribute_used__
__attribute__((section(".modinfo"))) =
"depends=mbcache";

MODULE_INFO(srcversion, "AFA0ADB02731C706BFF6A4C");
```

在 include/linux/moduleparam.h 中 MODULE_INFO 的定义如下:

```

#ifdef MODULE
#define ___module_cat(a,b) __mod_ ## a ## b
#define __module_cat(a,b) ___module_cat(a,b)
#define __MODULE_INFO(tag, name, info)
                      \
static const char __module_cat(name,__LINE__)[]
                     \
  __attribute_used__
                         \
  __attribute__((section(".modinfo"),unused)) = __stringify(
     tag) "=" info
#else  /* !MODULE */
```

```
10 #define __MODULE_INFO(tag, name, info)
11 #endif
12
13 include/linux/vermagic.中h
14 #define VERMAGIC_STRING
                        \
15         UTS_RELEASE " "
                              \
16         MODULE_VERMAGIC_SMP MODULE_VERMAGIC_PREEMPT
                              \
17         MODULE_VERMAGIC_MODULE_UNLOAD MODULE_ARCH_VERMAGIC
```

ext2.mod.c中:

- version, srcversion, __module_depends 存在于 __attribute__ 指定的 .modinfo section 中.
- __this_module 存在于 .gnu.linkonce.this_module section 中.
- __versions 存在于 __versions section 中.

事实上除了上面 ext2.mod.c 提到的几种符号外，还存在: drivers/net/8139too.mod.c

```
1 MODULE_ALIAS("pci:v000010ECd00008139sv*sd*bc*sc*i*")这个也存在
    于;.modinfo 中section.
```

从编译出来的fs/ext2/ext2.mod.o可以看出

```
$ nm fs/ext2/ext2.mod.o
00000000 r _____versions
00000000 r __mod_srcversion180
00000040 r __mod_vermagic5
00000023 r __module_depends
00000000 D __this_module
         U cleanup_module
         U init_module
$readelf -S fs/ext2/ext2.mod.o
  [ 8] .gnu.linkonce.thi PROGBITS        00000000 002f80 000600 \
                         00  WA  0   0 128
  [10] .modinfo          PROGBITS        00000000 003580 000066 \
                         00   A  0   0 32
  [11] __versions        PROGBITS        00000000 003600 002600 \
                         00   A  0   0 32
```

ext2.mod.o 含有上述的3个 section ，.gnu.linkoncewthis_module, .modinfo, __versions 。

当 ext2.o 和 ext2.mod.o 链接后形成的 ext2.ko ，就包含了这些信息:

```
1  #   Version magic (see include/vermagic.h for full details)
2  #     - Kernel release
3  #     - SMP is CONFIG_SMP
4  #     - PREEMPT is CONFIG_PREEMPT
5  #     - GCC Version
6  #   Module info
7  #     - Module version (MODULE_VERSION)
8  #     - Module alias'es (MODULE_ALIAS)
9  #     - Module license (MODULE_LICENSE)
10 #     - See include/linux/module.h for more details
```

当 CONFIG_MODVERSIONS=y 打开时编译的 kernel 装载模块时，就会利用模块的版本信息以决定是否符合要求。

7.3 kallsyms

7.3.1 kallsyms简介

在2.6内核中，为了更好地调试内核，引入了 kallsyms 。 kallsyms 抽取内核用到的所有函数地址(全局的，静态的)和非栈数据变量地址，生成一个数据小块(data blog)，作为只读数据链接进 kernel image 。相当于内核中存在了一份 System.map .内核可以根据符号名查出函数的地址。当系统发生 oops 时，

```
EIP; c01475ca <cache_alloc_refill+b2/2a4>
Call Trace:
[<c016f3b3>] permission+0x2f/0x49
[<c0147a65>] kmem_cache_alloc+0x4a/0x4c
[<c01610c2>] get_empty_filp+0x48/0xe9
[<c015f34e>] dentry_open+0x16/0x221
[<c01476d9>] cache_alloc_refill+0x1c1/0x2a4
[<c015f336>] filp_open+0x5d/0x5f
[<c015f8a8>] sys_open+0x55/0x85
[<c010924f>] syscall_call+0x7/0xb
```

EIP和函数调用栈都以可读的形式显示出来.

上面的 Ooops 用到了kallsyms的 print_symbol 函数，除此之外， include/linux/kallsym.h 还提供了很多有用的接口，如: kernel/kprobe.c 就用到了 kallsyms_lookup_name 以及 kallsyms_lookup 函数。当然，将 kallsyms 链接进内核会导致 kernel image 增大约300K左右。 kernel 中可以通过选项决定是否将 kallsyms 链接进内核以及 KALLSYMS 是否包含全部符号:

```
CONFIG_KALLSYMS=y
CONFIG_KALLSYMS_ALL=y
```

当 CONFIG_KALLSYMS_ALL=n 时， scripts/kallsyms 并不使用 –all-symbol 选项，所有非 text/inittext section 外的符号都会被简单丢弃。

7.3.2 scripts/kallsym

当.config中CONFIG_KALLSYMS=y时，kallsyms.o会直接链接进kernel image，

```
kallsyms.o := .tmp_kallsyms$(last_kallsyms).o
```

在讨论 kallsyms.o 的链接生成过程之前，我们有必要看看 scripts/kallsyms 产生的文件格式。

我们首先来看 .tmp_kallsyms1.S 的格式:

```
1 # Generate .S file with all kernel symbols
2 quiet_cmd_kallsyms = KSYM    $@
3       cmd_kallsyms = $(NM) -n $< | $(KALLSYMS) \
4                      $(if $(CONFIG_KALLSYMS_ALL),--all-symbols
                           ) > $@
5
6 .tmp_kallsyms%.S: .tmp_vmlinux% $(KALLSYMS)
```

```
7         $(call cmd,kallsyms)
8
9  # .tmp_vmlinux1 must be complete except kallsyms, so update
       vmlinux version
10 .tmp_vmlinux1: $(vmlinux-lds) $(vmlinux-all) FORCE
11        $(call if_changed_rule,ksym_ld)
```

本例中.tmp_kallsyms1.S的生成规则如下:

```
.tmp_kallsyms1.S: .tmpvmlinux1 scripts/kallsyms
nm -n .tmp_vmlinux1 | scripts/kallsyms \
        --all-symbols > .tmp_kallsyms1.S
```

nm -n .tmp_vmlinux1会将.tmp_vmlinux1中的所有符号按地址从小到大输出:

```
         w kallsyms_addresses
         w kallsyms_markers
         w kallsyms_names
         w kallsyms_num_syms
         w kallsyms_token_index
         w kallsyms_token_table
00000001 a CF_MASK
00000004 a ALLOCATOR_SLOP
00000100 a TF_MASK
 ... ...
bfbc04b8 A __crc_ethtool_op_set_sg
c003c637 A __crc___strncpy_from_user
c007a6af A __crc_leds_list_lock
c0100000 T _text
c0100000 T startup_32
c01000c0 T startup_32_smp
 ... ...
c0468e50 b wireless_nlevent_queue
c0468e60 B __bss_stop
c0468e60 B _end
c0469000 B pg0
c0580937 A __crc_rb_erase
c06e5af3 A __crc_flush_signals
c0798606 A __crc_tcf_unregister_action
... ...
ffa94a2f A __crc_tc_classify
ffbe21ea A __crc_sk_wait_data
ffd5a395 A __crc_default_wake_function
```

最前面的6个未定义weak符号在kernel/kallsyms.c中定义如下:

```
1  /* These will be re-linked against their real values during
       the second link stage */
2  extern const unsigned long kallsyms_addresses[] __attribute__
       ((weak));
3  extern const unsigned long kallsyms_num_syms __attribute__((
       weak));
4  extern const u8 kallsyms_names[] __attribute__((weak));
5  extern const u8 kallsyms_token_table[] __attribute__((weak));
```

```
6 extern const u16 kallsyms_token_index[] __attribute__((weak));
7 extern const unsigned long kallsyms_markers[] __attribute__((
      weak));
```

scripts/kallsyms 程序会自动过滤绝对符号(A,a)和未定义符号。 scripts/kallsyms 产生的 .tmp_kallsyms1.S 格式如下:

```
1  #include <asm/types.h>
2  #if BITS_PER_LONG == 64
3  #define PTR .quad
4  #define ALGN .align 8
5  #else
6  #define PTR .long
7  #define ALGN .align 4
8  #endif
9          .section .rodata, "a"
10 .globl kallsyms_addresses
11         ALGN
12 kallsyms_addresses:
13         PTR     _text + 0
14         PTR     _text + 0
15         PTR     _text + 0xc0
16         PTR     _text + 0x140
17         ... ...
18
19 .globl kallsyms_num_syms
20         ALGN
21 kallsyms_num_syms:
22         PTR     26561
23
24 .globl kallsyms_names
25         ALGN
26 kallsyms_names:
27         .byte 0x05, 0x54, 0x5f, 0xb2, 0x78, 0x74
28         .byte 0x06, 0x54, 0xb7, 0x3f, 0xac, 0x33, 0x32
29         .byte 0x09, 0x54, 0xb7, 0x3f, 0xac, 0x33, 0x32, 0x5f,
               0x73, 0x83
30         ... ...
31
32 .globl kallsyms_markers
33         ALGN
34 kallsyms_markers:
35         PTR     0
36         PTR     2574
37         PTR     5519
38         ... ...
39
40 .globl kallsyms_token_table
41         ALGN
42 kallsyms_token_table:
43         .asciz  "put_"
44         .asciz  "ops"
45         .asciz  "tim"
46         ... ...
47
48 .globl kallsyms_token_index
49         ALGN
50 kallsyms_token_index:
```

```
51          .short  0
52          .short  5
53          .short  9
54          ... ...
```

.section .rodata, "a" , 表示所有的这些 kallsyms_xxxx 都是编在 .rodata section 中，属性为 allocable 。这可以从汇编后的 .o 得到验证:

```
$ nm .tmp_kallsyms1.o
         U _text
00000000 R kallsyms_addresses
00060fe4 R kallsyms_markers
00019f08 R kallsyms_names
00019f04 R kallsyms_num_syms
000614fc R kallsyms_token_index
00061184 R kallsyms_token_table
```

下面逐个解释这些变量的意义:

kallsyms_addresses :

全局变量存放了 kallsyms_num_syms (26561）个函数或非栈数据的地址, 而且是以相对于 _text section (x86 为 c0100000 ,为内核虚拟地址 VMA)的形式显示.

kallsyms_names :

相应的函数或数据的 name ,总数也为 kallsyms_num_syms (26561). 其格式为

```
1           .byte LEN, TYPE,  INDX1, ... INDXLEN-1
```

以 .byte 0x06, 0x54, 0xb7, 0x3f, 0xac, 0x33, 0x32 为例: 0x06 为符号名所占长度, 0x54为符号类型，后5个字节都是 kallsyms_token_table 中的索引。0x06并不意味着 Name 的长度就是6，因为有的 token 可能包含好几个字符。

```
0x54, 84,  .asciz "T"
0xb7, 183, .asciz       "sta"
0x3f, 63,  .asciz       "rt"
0xac, 172, .asciz       "up_"
0x33, 51,  .asciz       "3"
0x32, 50,  .asciz       "2"
```

从中可以看出符号类型为 "T" ,名字: startup_32 ,对应于.S中的

```
c0100000 T startup_32
```

kallsyms_markers :

上面的 .S 含有 26561 个符号，如果顺序查找函数名是一个很费时的工作。 kallsyms_markers 的作用就是以256个符号为一组，记录每组第一个符号在 kallsyms_names 中的 offset 。例如我要查第257个符号的位置，就是以 257/256=1 作为索引值在 kallsyms_markers 中查找一个 offset 值，再从 offset 位置顺序查找。

kallsyms_token_table :

token_table 总长为256(正好是一个 ascii 表)， kallsyms 依据使用频度，抽取出常用的 token ，存于 ascii 中非打印字符中，这样就实现了字符串的压缩，例如， ascii 0x00 位置存的是 “put_” ，从而节省了3个字节。这种方法据说能够实现 50% 的压缩率(作者 Kai Germaschewski 如是说)。而且为了兼容 ascii 可见字符，每个 ascii 中可见字符只代表它本身，并没有压缩，如 0x33， 51， .ascizi “3” 。

kallsyms_token_index :

kallsyms_token_table 同样也是一个大字节数组。 kallsyms_token_index 为每个 token (反正才256个)都设一个 kallsyms_token_table 的索引值，从而实现快速查找。

7.3.3 kernel image和kallsyms的链接

当生成vmlinux时

```
# vmlinux image - including updated kernel symbols
vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) $(
    kallsyms.o) vmlinux.o FORCE
ifdef CONFIG_HEADERS_CHECK
        $(Q)$(MAKE) -f $(srctree)/Makefile headers_check
endif
        $(call vmlinux-modpost)
        $(call if_changed_rule,vmlinux__)
        $(Q)rm -f .old_version
```

kallsyms.o作为vmlinux的依赖目标之一.

当CONFIG_KALLSYMS=y时，

```
ifdef CONFIG_KALLSYMS_EXTRA_PASS
last_kallsyms := 3
else
last_kallsyms := 2
endif
kallsyms.o := .tmp_kallsyms$(last_kallsyms).o
```

大多数情况下 last_kallsyms＝2 ， 意味着你只需要编2遍.只有当 kallsyms 程序不能正确工作，前两步生成了不一致的 kallsyms 数据，从而导致编译失败，这时是你报 bug 的时候了，只有这时 CONFIG_KALLSYMS_EXTRA_PASS=y 才会编译出一个稳定版的 kallsyms.o ，也就是说，需要编3遍，意味着你发现了 kallsyms 程序的 bug 。下面我们来详细阐述 kallsyms.o 的两次编译过程: 第一遍 .tmp_kallsyms1.o :

```
.tmp_kallsyms1.o .tmp_kallsyms2.o .tmp_kallsyms3.o: %.o: %.S
    scripts FORCE
        $(call if_changed_dep,as_o_S)
```

这是一个静态模式规则 .tmp_kallsyms1.o 依赖于目标 .tmp_kallsyms1.S 。

```
.tmp_kallsyms%.S: .tmp_vmlinux% $(KALLSYMS)
        $(call cmd,kallsyms)
```

.tmp_kallsyms1.S 依赖于 .tmp_vmlinux1 。

链接ksym的规则为cmd_ksym_ld:

```
# First command is ':' to allow us to use + in front of this
    rule
cmd_ksym_ld = $(cmd_vmlinux__)
define rule_ksym_ld
        :
        +$(call cmd,vmlinux_version)
        $(call cmd,vmlinux__)
        $(Q)echo 'cmd_$@ := $(cmd_vmlinux__)' > $(@D)/.$(@F).
            cmd
endef

# .tmp_vmlinux1 must be complete except kallsyms, so update
    vmlinux version
```

```
11 .tmp_vmlinux1: $(vmlinux-lds) $(vmlinux-all) FORCE
12         $(call if_changed_rule,ksym_ld)
```

注意： .tmp_vmlinux1 调用的是 cmd_ksym_ld ，它首先会调用 cmd_vmlinux_version 更新 init/version.o ，然后调用 cmd_vmlinux 进行链接。

cmd_vmlinux_version 则用于更新版本号，这个可以体现在 uname -v 命令里:

```
$ uname -v
#1 SMP Tue Jun 30 20:28:53 UTC 2009
```

#1 表示系统中 kernel 只是链接了1次，也就是说 CONFIG_KALLSYMS=n 。 cmd_vmlinux_version 会更新 .version ，每次加1， init/linux/compile.h 和 init/version.o 会更新，从而更新 init/built-in.o ：

```
1  # Generate new vmlinux version
2  quiet_cmd_vmlinux_version = GEN     .version
3        cmd_vmlinux_version = set -e;                    \
4          if [ ! -r .version ]; then                     \
5            rm -f .version;                              \
6            echo 1 >.version;                            \
7          else                                           \
8            mv .version .old_version;                    \
9            expr 0$$(cat .old_version) + 1 >.version;    \
10         fi;                                            \
11         $(MAKE) $(build)=init
```

.tmp_vmlinux1中包含的kallsyms_xxx符号:

```
$ nm .tmp_vmlinux1 |grep kallsyms|grep " w"
          w kallsyms_addresses
          w kallsyms_markers
          w kallsyms_names
          w kallsyms_num_syms
          w kallsyms_token_index
          w kallsyms_token_table
```

可以看出这几个变量都未定义，属性都是弱符号。当.tmp_vmlinux1生成后，

```
1  # Generate .S file with all kernel symbols
2  quiet_cmd_kallsyms = KSYM    $@
3        cmd_kallsyms = $(NM) -n $< | $(KALLSYMS) \
4                       $(if $(CONFIG_KALLSYMS_ALL),--all-symbols
                           ) > $@
```

```
LD      .tmp_vmlinux1
KSYM    .tmp_kallsyms1.S
nm -n .tmp_vmlinux1 | scripts/kallsyms --all-symbols > .tmp_kallsyms1.S
AS      .tmp_kallsyms1.o
```

kallsyms.o 第一阶段完成，此时 .tmp_kallsyms1.o 已经包含了所有函数和非栈数据符号，当把 .tmp_kallsyms1.o 链接进去以后，新生成的 kernel image (.tmp_vmlinux2)中的函数和非栈数据地址已经发生了偏移，和 .tmp_kallsyms1.o 中的地址已经不一致了。幸运的是， .tmp_kallsyms1.o 所有符号的总长不会再改变了，因为它已经包含了所有需要链接的符号.

kallsyms.o第二遍:

```
1 .tmp_vmlinux2: $(vmlinux-lds) $(vmlinux-all) .tmp_kallsyms1.o
     FORCE
2         $(call if_changed,vmlinux__)
```

把 .tmp_kallsyms1.o 链接进 .tmp_vmlinux2 ，由于 kallsyms 中所有符号总长已经固定不变了，所以 .tmp_vmlinux2 中的函数和非栈数据符号地址也就固定了，只是和链接进来的 .tmp_kallsyms1.o 中的符号地址不匹配，所以需要重新生成 .tmp_kallsyms2.o ，更新那些符号的地址。再次重新链接就OK了.

注意： .tmp_vmlinux2 调用的 cmd_vmlinux__ 命令，而 .tmp_vmlinux1 调用的是 rule_ksym_ld ， .tmp_vmlinux2 没有更新版本号的 cmd_vmlinux_version 命令.

生成 .tmp_vmlinux2 后，就可以生成 .tmp_kallsyms2.o .

```
  LD       .tmp_vmlinux2
  KSYM     .tmp_kallsyms2.S
nm -n .tmp_vmlinux2 | scripts/kallsyms --all-symbols > .tmp_kallsyms2.S
  AS       .tmp_kallsyms2.o
  LD       vmlinux
```

生成 .tmp_kallsyms2.o 后，这个 .tmp_kallsyms2.o 就是 $(kallsyms.o) 会最终链接进 vmlinux ，链接的命令为 rule_vmlinux__ ， rule_vmlinux__ 大致可分为如下几步:

- 调用cmd_vmlinux_version更新版本号.
- 调用 cmd_vmlinux__ 进行 $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) $(kallsyms.o) vmlinux.o 的链接.
- 生成.vmlinux.cmd
- 调用命令生成 scripts/mksysmap vmlinux System.map .
- 调用 verify_kallsyms 命令，用 System.map 和 .tmp_vmlinux2 生成的 .tmp_System.map 比较，如果不一致，说明 kallsyms 有 bug (Inconsisten kallsys data，Try setting CONFIG_KALLSYS_EXTRA_PASS .)。

7.3.4 kallsyms应用

本节主要侧重于 kbuild 的编译过程，毕竟不是关于 kallsyms 的专题，所有关于 kernel/kallsyms.c 函数接口和实现分析都忽略不谈。关于 kallsyms 提供的接口函数，可参见 include/linux/kallsyms.h 。 kallsyms 函数实现，可参见 kernel/kallsyms.c 。

7.4 i386 bzImage

7.4.1 bzImage概述

bzImage 由于和 i386 平台紧密关联，实现相对特殊，这也是将 bzImage 作为一个主题单独列出来的原因。

前面提到 vmlinux 的入口地址为 system_32 ，该函数是工作在 32-bit 段寻址的保护模式，但是问题是系统自加电那一刻起，就运行于 16-bit 实模式。所以我们需要一些辅助程序从 16-bit 实模式转到 32-bit 保护模式，设置好必须的参数后才能开启分页模式转到 32-bit 分页保护模式. 前半部分是由 arch/i386/boot/setup.bin 实现的，后半部分则是由 arch/i386/kernel/head.o 实现的。 setup.bin 出了模式的转换外，还有很多其它的事情要做。

从 vmlinux 的链接过程来看 vmlinux 是一个已编好的 bare kernel 。和普通的 ELF 可执行文件没有什么区别。

```
$ file vmlinux
vmlinux: ELF 32-bit LSB executable, Intel 80386, \
        version 1 (SYSV), statically linked, not stripped
```

我们知道，C和汇编源文件被编译成 ELF 文件后，通常会包含有 Linker 或 Loader 所需要的 ELF header， Program header table 或 Section header table 。这些 ELF header 和一些 section 的作用是告诉内核 ELF Loader 如何载入 ELF 可执行文件。但是， Linux 内核作为一种特殊的 ELF 文件，没有 ELF Loader 的帮助，需要特殊辅助程序去装载它。往往它的装载地址是固定的。这时，为了保证通用性而存在的 ELF header 和一些 section 对内核的装载就没有意义了。为了使内核尽可能小，可以把这些信息去掉。所以通常采用 objcopy 来去掉原来 ELF 文件中的 header 和 section ，同时转化为 raw binary 格式的文件。即便如此，通过 objcopy 处理过的 vmlinux 一般需要压缩以后在重新链接，当然必须把解压缩的程序也同时链接进来。这个压缩的过程着实奇怪:压缩后，然后再解压缩，岂不是浪费启动时间?这还是因为当 X86 系列处理器启动初期，处于实模式状态，可以寻得的地址空间十分有限，如果内核过大，就无法加载。更新的内核(2.6.30)甚至提供了更高压缩率的格式: bzip 和 LZMA 。

由上面的分析可以看出， bzImage 至少包含 kernel booting 辅助程序和压缩的 vmlinux 。这可以从 bzImage 的规则链得到验证:
arch/i386/Makefile中关于bzImage的规则

```
1 zImage bzImage: vmlinux
2         $(Q)$(MAKE) $(build)=$(boot) $(KBUILD_IMAGE)
```

调用命令为: make -f scripts/Makefile.build obj=arch/i386/boot arch/i386/boot/bzImage 。注意:这个命令为 Makefile.build 指定了目标 arch/i386/boot/bzImage ，而不是使用缺省的 __build 。

arch/i386/boot/Makefile中bzImage的规则如下:

```
1  hostprogs-y     := tools/build
2
3  quiet_cmd_image = BUILD   $@
4  cmd_image = $(obj)/tools/build $(BUILDFLAGS) $(obj)/setup.bin
       \
5              $(obj)/vmlinux.bin $(ROOT_DEV) > $@
6
7  $(obj)/zImage $(obj)/bzImage: $(obj)/setup.bin \
8                                $(obj)/vmlinux.bin $(obj)/tools/
                                    build FORCE
9          $(call if_changed,image)
10         @echo 'Kernel: $@ is ready' ' (#'`cat .version`')'
```

arch/i386/boot/bzImage 依赖的目标有： setup.bin vmlinux.bin tools/build, tools/build 。 tools/build 已加到 hostprogs-y 中去了，由 Makefile.lib 负责生成相应的规则。 tools/build 将 arch/i386/boot/setup.bin 和 arch/i386/boot/vmlinux.bin 拼接在一起。下面我们将分别讲述 setup.bin 和 vmlinux.bin 的生成过程:

7.4.2 arch/i386/boot/setup.bin

PC机上kernel的booting流程一般大致如下:

1. BIOS确定启动设备(HardDisk，CD-ROM，Floppy之类).

2. BIOS从启动设备(如 Hard Disk 的MBR)中装载 bootsector :简单的 bootloader ， 如 bootsect.S 只占一个扇区; 复杂的 bootloader 如 LiLo，grub 除了启动扇区外，可能还有很多扇区，要分好几个阶段才能完全装入。

3. bootsector ，装载 setup.bin ，然后跳入 setup.bin 的入口函数 _start(arch/i386/boot/head.S)。

4. setup.bin 负责获取系统参数，转到 32-bit 保护模式，解压缩 kernel 。跳入 vmlinux 的入口点 startup_32 (arch/i386/kernel/head.S)。

5. startup_32设置IDT，GDT 打开分页，跳入start_kernel。

同老版本kernel实现稍有不同的是:

- 没有 bootsect.S ，所以就不能支持从 floppy 直接启动，必须借助 bootloader 程序。
- 以前完全由汇编书写的setup.S，也大部分由c文件代替。

setup.bin主要作用就是完成流程4的工作，其函数调用顺序为:

```
_start->start_of_setup, 这两个函数都定义在arch/i386/boot/head.S
                      ->main(arch/i386/boot/main.c)
                      ->goto_protect_mode(arch/i486/boot/pm.c)
                      ->protect_mode_jump(arch/i386/pmjump.S)
```

main.c 主要功能为检测系统参数如: Detect memory layout, set video mode 等，最后调用 goto_protect_mode ， 设置 32-bit 保护段式寻址模式。注意: main.o 编译为16位 real mode 程序。 goto_protect_modea 则会调用 protected_mode_jump 。

arch/i386/pmjump.S

```
1  /*
2   * void protected_mode_jump(u32 entrypoint, u32 bootparams);
3   */
4  protected_mode_jump:
5          xorl    %ebx, %ebx              # Flag to indicate
               this is a boot
6          movl    %edx, %esi              # Pointer to
               boot_params table
7          movl    %eax, 2f                # Patch ljmpl
               instruction
8          jmp     1f                      # Short jump to flush
               instruction q.
9
10 1:
11         movw    $__BOOT_DS, %cx
12
13         movl    %cr0, %edx
14         orb     $1, %dl                 # Protected mode (PE)
               bit
15         movl    %edx, %cr0
16
17         movw    %cx, %ds
18         movw    %cx, %es
19         movw    %cx, %fs
20         movw    %cx, %gs
21         movw    %cx, %ss
22
```

```
23          # Jump to the 32-bit entrypoint
24          .byte   0x66, 0xea              # ljmpl opcode
25  2:      .long   0                       # offset
26          .word   __BOOT_CS               # segment
27
28          .size   protected_mode_jump, .-protected_mode_jump
```

protected_mode_jump 最后几行就相当于 ljmpl __BOOT_CS:0，实际地址为 0x00000000，也就是 vmlinux.bin 中，arch/i386/boot/compressed/head.S 的入口函数 startup_32， startup_32 会解压缩 kernel :

```
1   /*
2    * Do the decompression, and jump to the new kernel..
3    */
4           movl output_len(%ebx), %eax
5           pushl %eax
6           pushl %ebp      # output address
7           movl input_len(%ebx), %eax
8           pushl %eax      # input_len
9           leal input_data(%ebx), %eax
10          pushl %eax      # input_data
11          leal _end(%ebx), %eax
12          pushl %eax      # end of the image as third argument
13          pushl %esi      # real mode pointer as second arg
14          call decompress_kernel
15          addl $20, %esp
16          popl %ecx
17
18          忽略了CONFIG_RELOCATABLE部分
19  /*
20   * Jump to the decompressed kernel.
21   */
22          xorl %ebx,%ebx
23          jmp *%ebp
```

其中 output_len, input_len, input_data 都是定义于 arch/i386/boot/compressed/vmlinux.scr。这将在 vmlinux.bin 中详述。最后两条命令会跳入 kernel 入口函数： startup_32 (arch/i386/kernel/head.S)。

setup.bin的目标生成规则链为:

```
1   SETUP_OBJS = $(addprefix $(obj)/,$(setup-y))
2
3   LDFLAGS_setup.elf       := -T
4   $(obj)/setup.elf: $(src)/setup.ld $(SETUP_OBJS) FORCE
5           $(call if_changed,ld)
6
7   OBJCOPYFLAGS_setup.bin  := -O binary
8
9   $(obj)/setup.bin: $(obj)/setup.elf FORCE
10          $(call if_changed,objcopy)
```

其中cmd_ld，cmd_objcopy都是定义于Makefile.lib中.

最终链接命令为:

```
ld -m elf_i386 -m elf_i386  \
   -T arch/i386/boot/setup.ld arch/i386/boot/a20.o \
```

```
   arch/i386/boot/apm.o arch/i386/boot/cmdline.o \
   arch/i386/boot/copy.o arch/i386/boot/cpu.o \
   arch/i386/boot/cpucheck.o arch/i386/boot/edd.o \
   arch/i386/boot/header.o arch/i386/boot/main.o \
   arch/i386/boot/mca.o arch/i386/boot/memory.o \
   arch/i386/boot/pm.o arch/i386/boot/pmjump.o \
   arch/i386/boot/printf.o arch/i386/boot/string.o
   arch/i386/boot/tty.o arch/i386/boot/video.o \
   arch/i386/boot/version.o arch/i386/boot/voyager.o\
   arch/i386/boot/video-vga.o arch/i386/boot/video-vesa.o
   arch/i386/boot/video-bios.o -o arch/i386/boot/setup.elf
objcopy -O binary -R .note -R .comment -S -O binary \
   arch/i386/boot/setup.elf arch/i386/boot/setup.bin
```

其中 objcopy 去除了 .note 和 .comment section ， 同时-S去除了重定位(relocation)和符号，并输出为 raw binary 。

值得一提的是arch/i386/boot/setup.ld：

```
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)

SECTIONS
{
        . = 0;
        .bstext         : { *(.bstext) }
        .bsdata         : { *(.bsdata) }

        . = 497;
        .header         : { *(.header) }
        .inittext       : { *(.inittext) }
        .initdata       : { *(.initdata) }
        .text           : { *(.text*) }

        . = ALIGN(16);
        .rodata         : { *(.rodata*) }

        .videocards     : {
                video_cards = .;
                *(.videocards)
                video_cards_end = .;
        }

        . = ALIGN(16);
        .data           : { *(.data*) }

        .signature      : {
                setup_sig = .;
                LONG(0x5a5aaa55)
        }


        . = ALIGN(16);
        .bss            :
        {
                __bss_start = .;
```

```
39                  *(.bss)
40                  __bss_end = .;
41          }
42          . = ALIGN(16);
43          _end = .;
44
45          /DISCARD/ : { *(.note*) }
46
47          . = ASSERT(_end <= 0x8000, "Setup too big!");
48          . = ASSERT(hdr == 0x1f1, "The setup header has the
               wrong offset!");
49  }
```

setup.ld 定义了输出文件的 section 组织顺序: 由于 head.o 位于 setup.elf 最前部，只有 arch/i386/boot/head.o 定义了这些 section :

```
.bstext,  .bsdata,  .header,  .inittext .initdata. setup.bin
```

0-496为 .bstext 和 .bsdata section 。497开始为 .header，.inittext，.initdata，.text section 数据。 497 - 511 的15个字节为缺省参数， tools/build 在生成 bzImage 的过程中会改变某些缺省值。

```
1  hdr:
2  setup_sects:    .byte SETUPSECTS
3  root_flags:     .word ROOT_RDONLY
4  syssize:        .long SYSSIZE
5  ram_size:       .word RAMDISK
6  vid_mode:       .word SVGA_MODE
7  root_dev:       .word ROOT_DEV
8  boot_flag:      .word 0xAA55
```

hexdump setup.bin可以看出497-511的值:

```
$ hexdump arch/i386/boot/setup.bin|grep "0000200" -C 1
00001f0 0400 0001 7f00 0000 0000 ffff 0000 aa55
0000200 3aeb 6448 5372 0206 0000 0000 1000 278c
0000210 0100 8000 0000 0010 0000 0000 0000 0000
```

_start函数进入点正好在512偏移地址处。

```
$ nm arch/i386/boot/setup.elf |grep " _start"
00000200 R _start
```

arch/i386/boot/video-*.c 都定义有__videocard，其最终定义为:

```
1  #define __videocard struct card_info __attribute__((section(".
      videocards")))
```

所有以__videocard属性修饰的数据都会放入.videocards section.

.bss段并不会占用实际存储空间，setup.bin的.signature位于文件末尾:

```
hexdump -C arch/x86/boot/setup.bin | tail -2
0002ac0 aa55 5a5a
0002ac4
```

7.4.3 arch/i386/boot/vmlinux.bin

vmlinux.bin的目标生成规则链为:

```
arch/i386/boot/中，Makefile
$(obj)/vmlinux.bin: $(obj)/compressed/vmlinux FORCE
        $(call if_changed,objcopy)

$(obj)/compressed/vmlinux: FORCE
        $(Q)$(MAKE) $(build)=$(obj)/compressed IMAGE_OFFSET=$(
            IMAGE_OFFSET) $@

arch/i386/boot/compressed/中Makefilevmlinux.规则为bin:
$(obj)/vmlinux: $(src)/vmlinux.lds $(obj)/head.o $(obj)/misc.o
    $(obj)/piggy.o FORCE
        $(call if_changed,ld)
        @:
```

其中，$(obj)/piggy.o比较复杂一些，可大致分为几步:

1. objcopy除掉ELF header和无用信息生成raw binary格式的vmlinux.bin。

```
objcopy -O binary -R .note -R .comment -S  \
        vmlinux arch/i386/boot/compressed/vmlinux.bin
```

2. vmlinux.bin.gz 由于 CONFIG_RELOCATABLE=y (关于 CONFIG_RELOCATABLE，将在下一小节简要讲述)，所以必须把 relocable 的符号信息附在 vmlinux.bin 后，这大概会增加%10的大小。

```
gcc -Wp,-MD,arch/i386/boot/compressed/.relocs.d  \
    -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer \
    -o arch/i386/boot/compressed/relocs \
    arch/i386/boot/compressed/relocs.c
arch/i386/boot/compressed/relocs vmlinux > \
    arch/i386/boot/compressed/vmlinux.relocs;\
arch/i386/boot/compressed/relocs --abs-relocs vmlinux
cat arch/i386/boot/compressed/vmlinux.bin \
    arch/i386/boot/compressed/vmlinux.relocs \
    > arch/i386/boot/compressed/vmlinux.bin.all
gzip -f -9 < arch/i386/boot/compressed/vmlinux.bin.all \
    > arch/i386/boot/compressed/vmlinux.bin.gz
```

3. 将vmlinux.bin.gz链接成piggy.o

```
ld -m elf_i386 -m elf_i386  -r --format binary \
   --oformat elf32-i386 \
   -T arch/i386/boot/compressed/vmlinux.scr \
   arch/i386/boot/compressed/vmlinux.bin.gz \
   -o arch/i386/boot/compressed/piggy.o
```

其中的vmlinux.scr很重要:

```
SECTIONS
{
```

```
3   .data.compressed : {
4         input_len = .;
5         LONG(input_data_end - input_data) input_data = .;
6         *(.data)
7         output_len = . - 4;
8         input_data_end = .;
9         }
10  }
```

vmlinux.scr只定义了一个 .data.compressed section, vmlinux.bin.gz 作为 .data 包含于其中，而且定义了以后解压缩数据所需的重要变量： input_len, input_data, output_len 。

4. head.o，misc.o，piggy.o 链接生成 vmlinux.bin 。

```
ld -m elf_i386 -m elf_i386 \
   -T arch/i386/boot/compressed/vmlinux.lds \
   arch/i386/boot/compressed/head.o \
   arch/i386/boot/compressed/misc.o \
   arch/i386/boot/compressed/piggy.o \
   -o arch/i386/boot/compressed/vmlinux
objcopy -O binary -R .note -R .comment -S  \
   arch/i386/boot/compressed/vmlinux \
   arch/i386/boot/vmlinux.bin
```

head.o和用于解压缩的misc.o以及piggy.o链接在一起形成新的vmlinux。

```
1   SECTIONS
2   {
3           /* Be careful parts of head.S assume startup_32 is
                at
4            * address 0.
5            */
6           . =  0  ;
7           .text.head : {
8                   _head = . ;
9                   *(.text.head)
10                  _ehead = . ;
11          }
12          .data.compressed : {
13                  *(.data.compressed)
14          }
15          ... ...
16  }
```

arch/i386/boot/compressed/vmlinux.lds 链接脚本中 .text.head section 后就是 .data.compressed section ， 而实际上也只有 head.o 定义了 .text.head 。

7.4.4 bzImage的链接

tools/build 是用于构建最终 bzimage 的工具，他的作用就是吧 setup.bin, vmlinux.bin 连接到一起: setup.bin 按照512字节对齐，同时负责把 rootdev ， 内核 crc ， 以及 setup 和 kernel 的大小， patch 到 setup.bin 开头的 arch/x86/boot/head.S 中。

```
arch/i386/boot/tools/build -b arch/i386/boot/setup.bin \
      arch/i386/boot/vmlinux.bin CURRENT > arch/i386/boot/bzImage
```

7.5 Relocatable kernel

7.5.1 Relocatable kernel简介

bzImage 链接时指定了 kernel 必须运行于 0x100000 (1MB)处， kernel 装载到其他地址就无法工作，因此要想让 bzImage 运行于多个其它的的地址，就必须多次重新链接，生成多个不同链接地址的 bzImage 。对于普通终端用户来说， kernel 装载到固定地址运行并不会带来什么麻烦，再不济，重编一次 bzImage 就是了。但是，对于 kernel 开发者来说，可重定位的 kernel 支持(Relocatable kernel，CONFIG_RELOCATABLE=y)对于某些开发者来说却非常有用。例如 kdump(2.6.13 引入)，当 kernel crash 后， kdump 会调用 kexec boot 另一个 kernel(dump-capture kernel) 把前一个 kernel(primary kernel) 的内存印像 (memory image) 存储起来，类似于 BSD, solaris 等 unix 内核的 crash-dump 机制(详见 Documentation/kdump/kdump.txt)，这种机制大有凤凰涅磐浴火重生之势。这个 dum-capture kernel 通常运行于 primary kernel 预留的地址空间内。可以通过命令行指定，因而 dump-capture kernel 必须是一个 relocatable kernel ，能够从不同地址启动运行。

7.5.2 相关配置选项

Relocatable kernel 的支持是从 2.6.20 开始引入的，几个相关的配置选项简介如下:

```
CONFIG_PHYSICAL_START,  kernel装载并开始运行的物理地址，
                        缺省为0x100000(bzImage)。
CONFIG_PHYSICAL_ALIGN,  CONFIG_PHYSICAL_START的对齐要求，
                        缺省值为0x100000，range 0x2000 0x400000。
CONFIG_RELOCATABLE ,    编译一个可重定位的kernel(Y/n)。
```

7.5.3 实现

Relocatable kernel的实现分两部分:编译时的支持和运行时的支持。

编译时的支持: arch/i386/Makefile

```
1
2 ifdef CONFIG_RELOCATABLE
3 LDFLAGS_vmlinux := --emit-relocs
4 endif
```

–emit-relocs 的作用就是:在一个链接后的可执行文件中保存 relocation section 和内容。以 vmlinux 为例:

```
ld -m elf_i386 -m elf_i386 --emit-relocs --build-id \
  -o vmlinux -T arch/i386/kernel/vmlinux.lds \
  arch/i386/kernel/head.o arch/i386/kernel/init_task.o \
  init/built-in.o --start-group  *built-in.o --end-group \
  .tmp_kallsyms2.o
```

```
$ readelf -S vmlinux
Section Headers:
[Nr] Name              Type     Addr     Off    Size   ES Flg Lk Inf Al
[ 0]                   NULL     00000000 000000 000000 00      0   0  0
[ 1] .text.head        PROGBITS c0100000 001000 000391 00  AX  0   0 16
[ 2] .rel.text.head    REL      00000000 274cc34 0001a8 08     81   1  4
```

```
[ 3] .text             PROGBITS c0101000 002000 209957 00  AX  0   0 4096
[ 4] .rel.text         REL      00000000 274cddc 07c360 08      81   3  4
[ 5] __ex_table        PROGBITS c030a960 20b960 000d70 00   A  0   0  8
[ 6] .rel__ex_table    REL      00000000 27c913c 001ae0 08      81   5  4
[ 7] .notes            NOTE     c030b6d0 20c6d0 000024 00  AX  0   0  4
[ 8] __bug_table       PROGBITS c030b6f8 20c6f8 003564 00   A  0   0  1
[ 9] .rel__bug_table   REL      00000000 27cac1c 004730 08      81   8  4
[10] .tracedata        PROGBITS c030ec5c 20fc5c 000018 00   A  0   0  1
[11] .rel.tracedata    REL      00000000 27cf34c 000020 08      81  10  4
[12] .rodata           PROGBITS c030f000 210000 09f268 00   A  0   0 32
[13] .rel.rodata       REL      00000000 27cf36c 0414c0 08      81  12  4
[14] .pci_fixup        PROGBITS c03ae268 2af268 000770 00   A  0   0  4
[15] .rel.pci_fixup    REL      00000000 281082c 000770 08      81  14  4
[16] __ksymtab         PROGBITS c03ae9d8 2af9d8 004d58 00   A  0   0  4
[17] .rel__ksymtab     REL      00000000 2810f9c 009ab0 08      81  16  4
[18] __ksymtab_gpl     PROGBITS c03b3730 2b4730 0014a0 00   A  0   0  4
[19] .rel__ksymtab_gpl REL      00000000 281aa4c 002940 08      81  18  4
[20] __kcrctab         PROGBITS c03b4bd0 2b5bd0 0026ac 00   A  0   0  4
[21] .rel__kcrctab     REL      00000000 281d38c 004d58 08      81  20  4
[22] __kcrctab_gpl     PROGBITS c03b727c 2b827c 000a50 00   A  0   0  4
[23] .rel__kcrctab_gpl REL      00000000 28220e4 0014a0 08      81  22  4
[24] __ksymtab_strings PROGBITS c03b7ce0 2b8ce0 00d76c 00   A  0   0 32
[25] __param           PROGBITS c03c544c 2c644c 0003c0 00   A  0   0  4
[26] .rel__param       REL      00000000 2823584 0005f0 08      81  25  4
[27] .data             PROGBITS c03c6000 2c7000 020b50 00  WA  0   0 128
[28] .rel.data         REL      00000000 2823b74 009f70 08      81  27  4
  ... ...
[71] .debug_line       PROGBITS 00000000 219a02e 19d2cb 00       0   0  1
[72] .rel.debug_line   REL      00000000 35530bc 002a80 08      81  71  4
[73] .debug_frame      PROGBITS 00000000 23372fc 06af58 00       0   0  4
[74] .rel.debug_frame  REL      00000000 3555b3c 02d820 08      81  73  4
[75] .debug_str        PROGBITS 00000000 23a2254 0c7bbb 01  MS   0   0  1
[76] .debug_loc        PROGBITS 00000000 2469e0f 25dbbf 00       0   0  1
[77] .rel.debug_loc    REL      00000000 358335c 1b0cc0 08      81  76  4
[78] .debug_ranges     PROGBITS 00000000 26c79ce 084288 00       0   0  1
[79] .rel.debug_ranges REL      00000000 373401c 084300 08      81  78  4
[80] .shstrtab         STRTAB   00000000 274bc56 0002e6 00       0   0  1
[81] .symtab           SYMTAB   00000000 37b831c 077930 10      82 20351  4
[82] .strtab           STRTAB   00000000 382fc4c 09d806 00       0   0
```

基本上每个section都有对应的relocation section，其目的的保留每个符号在各个section的偏移位置。

```
arch/i386/boot/compressed/relocs vmlinux > \
        arch/i386/boot/compressed/vmlinux.relocs; \
arch/i386/boot/compressed/relocs --abs-relocs vmlinux
```

relocs的作用就是将所有的符号偏移位置保存到vmlinux.relocs arch/i386/boot/compressed/relocs.c

```
1 static void collect_reloc(Elf32_Rel *rel, Elf32_Sym *sym)
2 {
3         /* Remember the address that needs to be adjusted. */
```

```
4         relocs[reloc_idx++] = rel->r_offset;
5 }
```

随后vmlinux.reloc被放到vmlinux.bin后，一起压缩成vmlinux.bin.gz

```
cat arch/i386/boot/compressed/vmlinux.bin \
    arch/i386/boot/compressed/vmlinux.relocs \
    > arch/i386/boot/compressed/vmlinux.bin.all
gzip -f -9 < arch/i386/boot/compressed/vmlinux.bin.all \
    > arch/i386/boot/compressed/vmlinux.bin.gz
```

运行的支持: arch/i386/boot/compressed/head.S

- ebp开始值为vmlinux.bin.gz解压缩后的起始地址.
- ecx开始值为vmlinux.bin.gz解压缩后的长度.
- edi值即为vmlinux.bin.gz解压缩后的末尾地址.
- ebx为实际装载地址和编译指定地址之差.

```
1  #if CONFIG_RELOCATABLE
2  /* Find the address of the relocations.
3   */
4          movl %ebp, %edi
5          addl %ecx, %edi
6
7  /* Calculate the delta between where vmlinux was compiled to
       run
8   * and where it was actually loaded.
9   */
10         movl %ebp, %ebx
11         subl $LOAD_PHYSICAL_ADDR, %ebx
12         jz   2f         /* Nothing to be done if loaded at
               compiled addr. */
13 /*
14  * Process relocations.
15  */
16
17 1:       subl $4, %edi
18         movl 0(%edi), %ecx
19         testl %ecx, %ecx
20         jz 2f
21         addl %ebx, -__PAGE_OFFSET(%ebx, %ecx)
22         jmp 1b
23 2:
24 #endif
```

__PAGE_OFFSET的定义如下: include/asm-i386/page.h

```
1  #ifdef __ASSEMBLY__
2  #define __PAGE_OFFSET           CONFIG_PAGE_OFFSET
3  #else
4  #define __PAGE_OFFSET           ((unsigned long)
       CONFIG_PAGE_OFFSET)
5  #endif
```

```
6
7 #define PAGE_OFFSET             ((unsigned long)__PAGE_OFFSET)
```

include/linux/autoconf.h

```
1 #define CONFIG_PAGE_OFFSET 0xC0000000
```

所以

```
1 addl %ebx, -__PAGE_OFFSET(%ebx, %ecx)
```

就是根据vmlinux.relocs的符号offset值(%ecx)，更正解压缩后的kernel image中相应符号实际地址值.

从标号1开始的循环是从 vmlinux.relocs 末尾开始，以 %ecx = 0 结束，这也可以从relocs.c代码看出:

arch/i386/boot/compressed/relocs.c

```
1  static void emit_relocs(int as_text)
2  {
3          ... ...
4          /* Print the relocations */
5          if (as_text) {
6                  ... ...
7          }
8          else {
9                  unsigned char buf[4];
10                 buf[0] = buf[1] = buf[2] = buf[3] = 0;
11                 /* Print a stop */
12                 printf("%c%c%c%c", buf[0], buf[1], buf[2], buf
                       [3]);
13                 /* Now print each relocation */
14                 for(i = 0; i < reloc_count; i++) {
15                         buf[0] = (relocs[i] >>  0) & 0xff;
16                         buf[1] = (relocs[i] >>  8) & 0xff;
17                         buf[2] = (relocs[i] >> 16) & 0xff;
18                         buf[3] = (relocs[i] >> 24) & 0xff;
19                         printf("%c%c%c%c", buf[0], buf[1], buf
                               [2], buf[3]);
20                 }
21         }
22 }
```

输出的第一行作为结束标记.

```
$$ hexdump arch/i386/boot/compressed/vmlinux.relocs |head -2
0000000 0000 0000 0004 c010 0018 c010 001d c010
0000010 0029 c010 0037 c010 0058 c010 0064 c010
```

前4个字节为0

综上所述， relocatable kernel 的实现并没有增加额外的运行时负担(runtime overload)，而且也没有导致内存中的 kernel image 大小的增大(objcopy 丢弃了所有的 –emit-reloc 产生的 relocation)，而且 vmlinux.relocs 在使用完后被完全丢弃，不会占用任何额外内存，唯一的缺点就是 vmlinux.relocs 导致 bzImage 又增大了不少。

Part 5

8 kbuild总结

8.1 kbuild中的设计思想

kbuild 是很多设计需求之间平衡的结果，总体来说， kbuild 提供了一个 framework ：绝大部分开发者只需要知道如何去做(how to do it)，而不需要了解为什么这么做。kbuild 对绝大部分开发者基本上就是透明的，隐藏了大部分实现细节，可以通过接口和用户交互，这就有效地降低了 kernel 开发者的学习曲线和负担，能使他们专注于自己要做的事，而不至于花费无谓的时间和精力做与自己无关的事。如果我们以一个程序设计的角度来看， kbuild 的实现同样体现了很多优秀程序共有的设计思想:

- 接口和实现的分离。 kbuild 中的顶层Makefile相当于实现了很多接口，而kbuild的规则相当于各种接口的具体实现。这就使得对于具体实现的改动并不会影响接口的使用。例如 vmlinux 依赖于 make -f scripts/Makefile.build obj=<dir> ， Makefile.build 的变动并不会影响 make vmlinux 的使用。
- 模块化和单一职责在 kbuild rules 的实现中表现得非常明显。例如 Makefile.build 负责 kernel Makefile 中多目标 (built-in.o和*.ko) 的生成，而 Makefile.clean 和 Makefile.modpost 则分别负责 clean 和模块第二阶段的编译。这样实现的好处是减少了各个模块间的耦合，从而避免了不必要的复杂性.设想 Makefile.build 和 Makefile.clean 两个规则合在一起， Makefile.build 需要 include 每个依赖文件，而这对于 Makefile.clean 则是完全没必要的，所以可能的写法就是:

```
1  /* dependency 文件 */
2  cmd_files := $(wildcard $(foreach f,$(targets),$(dir $(f)).$(
       notdir $(f)).cmd))
3
4  ifeq "$(findstring clean,$(MAKECMDGOALS))" ""
5  ifneq "$(strip $(cmd_files))" ""
6  ifeq "$(NODEPENDS)" ""
7  ifeq "$(CLEAN)" ""
8  -include $(DEPENDS)
9  endif                # CLEAN
10 endif                # NODEPENDS
11 endif                # strip cmd_files
12 endif                # findstring MAKECMDGOALS
```

8.2 kbuild同2.4相比的改进

同2.4的相比，kbuild主要有了如下改进：

1. 自动建立depency graph，不需要额外的make dep.
2. 可以直接从只读的source编译，不需要修改source 目录，Make O=<dir>.
3. 所有的平台都有一个default的target，只需要一个make.
4. 增加了make help.
5. 顶层Makefile更加简洁，2.4中需要 built-in 的各个目录都由 Makefile.build 统一编译成 built-init.o 然后链接.

6. Makefile语法更加简单:去除了export-objs.
7. 为 arch makefile 提供了一个 framework ，并且所有的arch都已更新到了这一新的 framework .
8. No-verbose mode-使警告更加明显.

虽然 kbuild 同2.4相比有了很大的改进，但是一个明显的缺点依然存在：顶层 Makefile 充斥着大量 ifdef ，使得 Makefile 很难以阅读。

9 后记

“高山仰止，景行行止。虽不能至，然心往之”。这句话用来形容读 Linux kernel 的感觉是再合适不过了。 Linux kernel 作为一个非常出色的 open source 软件为我们树立了一个很难企及的高度。但是，除了“景仰”之外，我们依然能从阅读和分析 Linux kernel 源代码中学习到很多优秀的设计思想和编程技巧，而把这些学到的设计思想和编程技巧融合到自己的项目中则能极大地提高代码质量和项目的成功率。

参考文献

[1] GNU make info page version 3.81或http://www.gnu.org/software/make/manual/html_node/index.html
[2] GNU make info page v3.80,中文翻译http://www.linuxsir.org/main/doc/gnumake/GNUmake_v3.80-zh_CN_html/index.html
[3] Managing Projects with GNU make, 3rd Edition , By Robert Mecklenburg, O’Reilly, 2004
[4] Linux Kernel Documentation: Documentation/kbuild/
[5] Linux kernel internals, Trigran Aviazian, 2000