learning
just as your
favourite thing

Linux
内核构建系统

（v1.0, 20 Nov 2010）

前言

作为 Linux 内核驱动程序开发者来说，虽说并不要求熟练掌握整个内核构建系统，而只是要求会写符合我们自己驱动程序需求的 Kbuild/Makefile 即可。但是，经常地，事情总是不会如我们所愿那样而一帆风顺的发展。

倘若有一天你使用内核构建系统所提供的接口来编译驱动程序却不成功。那怎么办呢？看书、查资料和在 mail list 上向别人请教固然可能解决问题，但是等问题解决了，那也可能是几天以后的事情了。

所以，这些方法对颇具探索精神的您来说，可能并非百分之百得完美和快截。 相反，您更喜欢或更习惯于直接去 hacking 那些提供这些接口的内核代码，带着你的问题，以您目前所掌握的知识为基础去 hacking……就像本文所做的这样。

本文旨在重现数年前我对内核构建系统的 hacking 过程，希望能对您嵌入式 Linux 的学习有所启发。作为 JulianTec 的建立者之一，我觉得或许 JulianTec 没办法教会您所有的知识，但是在教您用什么样的方法去获得这些知识的方面，我相信她一定能对您有所帮助。

本文内容来自我在 JulianTec 的技术博客，具体的入口地址在 http://yihect.juliantec.info/julblog//post/4/17，所讨论的构建系统对应着 2.6.30 的内核版本。您可以到 http://www.kernel.org 中下载该版本的内核对照着阅读。

为了学习目的，任何人或组织都可以自由阅读本文档，但若您要在自己待发表的文章里面引用本文的内容，还请注名出处。提及上面的来源 URL 或者 JulianTec 都是可以接受的。

由于自身水平有限，再加上写时时间匆忙，本文中或许藏着这样或那样的错误。还请读者您能够指出，我将JulianTec将努力确保文章质量。您可以发邮件到 yihct@juliantec.info 联系我们JulianTec。

Yihe Chen
Nov 2010

目录

1. 概论

在 xNIX 世界中，您若要想使用任何一款软件，通常来说都必须先从官方站点上获取源代码、继而编译安装、最后才是动手使用。概括起来，整个其过程一般可分成以下几个步骤：

a) 获取该软件的源代码；

源代码通常是以压缩包的形式由官方所发布出来的，所以到手后您必须解压。需要注意的是，如果官方代码中藏有 bugs，而正好有热心人修正了这些 bugs，并且发布出来一些补丁程序。那么您为了在后续使用时避免出现问题，您通常也需要下载这些补丁包，并进行 patch 操作。

b) 进入到源代码目录中进行配置；

对大型的软件项目来说，配置必不可少。该步骤的目的之一是为了确定编译阶段需要使用到的各种工具和编译过程所面对的系统架构。比如编译器在哪里？连接器又使用哪一个？搞清楚在什么机器架构上进行编译，编译出来的程序又运行在什么架构上？其目的之二，您通常需要在配置过程中指定哪些功能特征是您所需要的。要知道，一个较大型的软件项目，其中的功能可为多种多样，而您，通常只需要其中的一小部分，所以您需要在下面编译之前指定出来。

c) 编译该软件；

使用配置过程中所确定出来的工具，对实现您所指定那部分功能的代码模块进行编译，并经过连接，最后形成所需要的可执行程序或可执行映像。

d) 安装部署所编译出来的结果

可执行程序的安装，就是要把他们放在 PATH 环境变量所指定的目录中去，通常是 /bin, /usr/bin, /sbin, /usr/sbin 之类的目录。那么可执行映像呢，这通常和嵌入式应用相关，您需要将这些映像下载到非易失性存贮器中，比如 EEPROM 或者 FLASH 中去。

当然，上面这个过程只是一般的过程。但是，像 Windows 平台那样，现在有很多软件也都发布了针对不同架构编译好的可执行应用程序包，为用户省却了烦琐的配置和编译，这种情况我们不去考虑。Linux 内核作为一个大型的软件项目，其使用也遵循上面这样一个一般的过程。

针对 Linux 内核而言，上面的步骤 b 被称为 kconfig，而步骤 c 即为 kbuild。

kconfig 时，我们要使用类似 "make ARCH=arm CROSS_COMPILE=arm-linux-menuconfig" 的命令来选择我们所要的功能。这个命令结束时，我们所选择的结果会被记录到一个叫 ".config" 的隐藏文件中。注意，其中 ARCH=arm 规定了我们要为 arm 架构编译内核，而 CROSS_COMPILE=arm-linux-规定了编译内核时需要使用 arm-linux-打头的交叉工具链。

.config 文件生成后，我们就可以使用 "make ARCH=arm CROSS_COMPILE=arm-linux-" 之类的命令来做编译，即 kbuild 了。需要注意的是，对 Linux 内核做编译，其产生的可执行代码分布在两个部分中间：一个基本内核映像和数个可独立加载的模块。在使用 Linux 内核时，前者作为 Linux 操作系统的主体而常驻内存；而后者可按需要动态的加载到内存中或自内存中卸载，用以动态的修改内核所具有的功能。这种安排是作为整块式操作系统结构的 Linux 内核向微内核操作系统结构学习的结果。既然编译的出来的可执行代码可以以两种方

式存在，那么我们就必须预先告诉 kbuild 系统，哪部分放在基本内核里，哪部分又以模块的形式存在。所以显然，我们在配置过程中产生".config"文件的时候，除了需要记录我们所需要的功能外，还要记录这些功能都需要放在哪里。

Linux 内核编译完成后，接下来要做的同样也是安装。对于 x86 等架构的桌面机来说，我们需要将基本内核映像放到文件系统中，通常是/boot 目录下；对于嵌入式系统，我们要烧写进 FLASH 中。对于各种模块，我们也要装到文件系统的指定目录下面。另外值得一提的是为了让应用程序能正常使用 C 库，Linux 内核还要向 C 库提供一套头文件，所以我们也需要将内核提供的头文件安装到文件系统相应的目录下面。安装可以手动进行，也可以自动的用类似 "make modinstd"、"make install"、"make headerinst"之类的命令。

2. Linux内核构建系统所支持的目标

由前面的概述可以知道，不管是 kconfig 步骤、还是 kbuild 步骤、还是安装都可藉由"make targets"形式的命令来完成。所以，分析一下所有可能的 targets 是必要的。内核构建系统所支持的目标完整列表可由命令 "make help" 打印出来。这里仅简单的列出最重要的部分：

```
  Cleaning targets:
    clean           - Remove most generated files but keep the config and
                      enough build support to build external modules
    mrproper        - Remove all generated files + config + various backup file
    distclean       - mrproper + remove editor backup and patch files

  Configuration targets:
    %config         - 各种配置目标，会生成 .config

  Other generic targets:
    all             - Build all targets marked with [*]
  * vmlinux         - Build the bare kernel
  * modules         - Build all modules
    modules_install - Install all modules to INSTALL_MOD_PATH (default: /)

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

    modules_prepare - Set up for building external modules
    kernelrelease   - Output the release version string
    kernelversion   - Output the version stored in Makefile
    headers_install - Install sanitised kernel headers to INSTALL_HDR_PATH
                      (default: /home/yihect/linux-2.6.31/usr)

Architecture specific targets (arm):
  * zImage          - Compressed kernel image (arch/arm/boot/zImage)
    Image           - Uncompressed kernel image (arch/arm/boot/Image)
  * xipImage        - XIP kernel image, if configured (arch/arm/boot/xipImage)
    uImage          - U-Boot wrapped zImage
    bootpImage      - Combined zImage and initial RAM disk
                      (supply initrd image via make variable INITRD=<path>)
    install         - Install uncompressed kernel
    zinstall        - Install compressed kernel
                      Install using (your) ~/bin/installkernel or
                      (distribution) /sbin/installkernel or
                      install to $(INSTALL_PATH) and run lilo

    s3c2410_defconfig         - Build for s3c2410
```

这里咱们先澄清一个观念。很多人认为内核构建系统仅仅是 kbuild，其实不然。你由上面这些目标可以很清楚的看出来，整个内核构建系统实现了不仅 build，而且还包括 config 和 install。那些认为内核构建系统仅仅是 kbuild 的人，可能是受到这个词的中英文翻译的迷惑。

为了帮助大家对后面内容的理解，我们将内核构建系统所支持的所有目标分一下类。出现在 make 命令后面的内容总共有这么几个类别：

- 和 .config 完全无关的目标

这些目标既不会产生 .config 文件，也不需要使用 .config 文件。包括上面列出来的 clean,mrproper,distclean,headers_install,kernelrelease,kernelversion 等等。

- 和 .config 相关的目标

除了和.config 完全无关的那些目标外，其它目标皆是要么会产生.config 文件，要么需要使用.config 文件。前者又称之为配置目标(config targets)，后者为构建目标(build targets)。

前者主要是：config 和 %config。百分号是通配符，%config 指代了一系列的以 config 为结尾的目标，包括 menuconfig,oldconfig,silentoldconfig,s3c2410_defconfig 等等。

后者主要是：all,vmlinux,modules,zImage,uImage 等等。它们的产生需要以.config 文件中的配置结果作为依据。

- single targets

除了上面这几个外，上面列表中还有一类目标没有归纳进来。那就是 dir/,dir/file.[ois],dir/file.ko 等目标。这些目标不同于那些复合集成性的目标，它们以单个文件或目录的形式存在，比方源码树中的某一个小目录，某一个单独的对象文件，某一个外部模块文件等等。

值得一说的是，出现在 make 命令后面的内容，可以是上面说到的这些目标的单独形式，也可以是这些目标的混合形式。比方我使用一个混合形式的命令 ”make oldconfig all “来既做 config，又做 build。

另外，make 命令后面除跟目标外，还可以跟一些变量定义作为选项，主要有这么几个：

- V 变量

make v=0 [targets] 简化编译所输出的内容

make V=1 [targets] 详细输出

make V=2 [targets] 编译过程中会编译一系列子目标，用这个选项会列出之所以编译它们的原因

- O 变量

make O=dir [target]编译过程中，将所有中间文件和目标文件(包括.config 文件)存到 dir 目录中， 而不是像缺省的那样放在内核源代码树中。

- 编译外部模块时用的 M 变量和 SUBDIRS 变量

make -C KERNELDIR M=dir modules

make -C KERNELDIR SUBDIRS=dir modules

其中 KERNELDIR 为内核代码树目录，dir 为外部模块源代码所在目录。如果是给你自己机器上正在运行着的内核编译模块，那么 KERNELDIR 一般取值 /lib/modules/`uname -r`/build

3. 构成内核构建系统的文件

3.1. 共有五大类文件

知道内核构建系统所支持的目标之后，我们再来看看内核构建系统的组成。Linux kernel 无疑是复杂的，那构建内核的工具自然也一定是相对复杂的，事实上也是如此，它由下面这一系列的文件所组成：

- 顶层 Makefile;

  提供针对各种目标的接口，一般和实现无关。当我们要针对某个目标进行分析时，作为起点，总是尝试在此文件中找到对应的目标定义，然后沿着该定义深入挖掘。

- 平台相关的 Makefile;

  提供针对不同种架构的目标，变量和规则定义。其文件位置比较固定，通常位于 arch/*/ 目录下面，即 arch/*/Makefile 文件。

- 各类规则定义文件；

  在 script 目录下面，有很多定义了不同方面规则的文件。这些文件可视作对顶层 Makefile 内所设计接口的实现，它们包括：

  1) Kbuild.include

     通用的定义。这个文件被其他 Makefile 所包含，比方被顶层 Makeile，被 Makefile.build 等包含。

  2) Makefile.build

     这个文件定义了许多编译规则，用于编译 built-in.o, lib.a 等组件。后面会知道，正是这些组件构成了 Linux 内核映像。

  3) Makefile.lib

     这个文件给很多变量赋了值，比方 obj-y,obj-m,subdir-y,subdir-m 等等，这些变量所指代的一系列文件(目录)列表都是需要编译处理的。另外该文件还包括了很多 cmd 命令的定义，在编译内核的过程中搭配 if_changed 系列使用。

  4) Makefile.host

     在 Linux 内核的编译过程但中，不仅需要使用各种诸如 as,gcc,ld,tar 之类的工具，而且还需要用到很多其它程序来处理源代码。比方需要用 fixdep 来处理目标的依赖关系，需要使用 conf/gconf/mconf 等工具来解析众多 Kconfig 配置文件中的配置选项以生成.config 文件，需要使用 genksyms 来计算内核导出符号的校验和(checksum)等等。这些工具也以源代码的形式放在 script 的不同子目录下面，所以在内核编译过程中，内核构建系统需要先行将它们编译出来。那么文件 Makefile.host 内就定义了如何将它们编译成可执行程序的相关规则。

  5) Makefile.clean

     和一般应用程序的编译开发一样，内核编译后产生的结果需要经常被清除。这个文件内详细定义了做 clean 所需要的相关规则。

  6) Makefile.headerisnt

     定义了如何安装头文件的规则。

  7) Makefile.modinst

前面已经说过什么是模块。其实，在我们的实际开发中，通常包含两个部分的模块。一部分是内核源代码中实际上已经实现的模块，这部分在内核构建过程中会编译出来，我们且称之为内部模块；另外一部分是作为驱动程序开发者的我们自己写出来的，通常被放在内核源码树之外目录中的模块，我们称之为外部模块，需要我们自己单独编译。文件 Makefile.modinst 定义了如何安装内部模块到 INSTALL_MOD_PATH 所指定目录中去的相关规则。外部模块一般需要靠我们自己手动拷贝到那个目录下面去。

8) Makefile.modpost

在 2.6 内核中，模块的编译需要两个阶段。第一阶段需要将对应的模块源代码编译成.o 文件，第二阶段再将对应的.o 和另外产生的.mod.o 链接起来，形成.ko 内核模块文件。此文件内定义的规则负责完成这第二阶段的工作。

9) Makefile.fwinst

现在要使某些设备能够正常工作的话，单单有内核里面的驱动程序是不够的，它们还需要另外一种称之为 firmware 的东西，这其实是一段二进制数据。在使设备能正常工作之前，设备的驱动程序需要先负责将这段数据写入设备才行。大部分 firmware 代码都是私有的，其所使用的 License 和 GPL 相冲突，所以内核社区里一直有激烈的争论是否要将 fireware 从 GPLed 的内核代码中去除（http://lwn.net/Articles/284932/）。在 2.6.27 版本之前，firmware 和驱动程序代码一起编译，但是在 2.6.27 版本中，驱动程序目录 drivers/下的 firmware 被取出，单独放到 firmware/目录中。此 Makefile 的功能是将 firmware 映像安装到文件系统的特定目录下，以方便驱动程序在适单的时候找到并使用它们。

◆ 代码树下各目录中的 Makefile

内核的编译构建是一个不断递归进入不同子目录继续编译的过程。每个子目录下面均需要有一个 Makefile 负责设置该目录下需要编译哪些源文件，这种设置通常取决于 kconfig 配置的结果。这个 Makefile 的名字一般就是"Makefile"，但是名字“Kbuild”也可以使用，所以在内核文档 Documentation/kbuild/makefiles.txt 中将它们总称为"kbuild makefiles"。有时候，"Makefile"这个名字已经作为别的用途被使用掉了，那么就只好用"Kbuild"来命名。比方在 arch/x86/目录下面，已经有一个”Makefile“充当架构平台相关的 Makefile 来使用了，那么就需要换个名字"Kbuild"。

◆ 代码树各目录中的一系列配置文件 KCconfig

这些配置文件其实并不直接参与内核的构建。他们都是为了完成 kconfig 配置过程准备的。内核开发者事先将可以选择的选项及相关说明以一定的格式包括在这一系列 KConfig 中，然后在用户使用"make menuconfig"类似命令进行配置过程中，内核构建系统将这些选项从这系列配置文件中读出来呈现在屏幕上，之后用户进行操作以进行不同选项的设置。设置完最后的结果被保存在.config 文件中，以供后面的 kbuild 过程使用。

3.2. 内核构建系统中各文件之间的关系

在这众多文件中，最主要的角色自然是顶层目录下的 Makefile，其他的文件都或直接、或间接的和它相关联。这些文件之间的关联可以列出如下：

文件	直接包含	调用	变量定义
Makefile	I	Cb,Cc,Chi,Cmp,Cfi	$(clean),$(hdr-inst)
scripts/Kbuild.include			$(build)
scripts/Makefile.build	I K L H	Cb	
scripts/Makefile.lib			
scripts/Makefile.host			
scripts/Makefile.clean	K	Cc	$(clean)
scripts/Makefile.headerinst	I K	Chi	$(hdr-inst)
scripts/Makefile.modinst	I		
scripts/Makefile.modpost	I K L		
scripts/Makefile.fwinst	I H		
各子目录下的Kbuild/Makefile			

如上示，我们可以将这些关联分成两类：

1. 直接包含

在一个文件中，利用 include 来包含另外的文件。这是比较简单的，比如字母 I,L,H 就表示了对应的文件，这里比方 scripts/Makefile.build 直接包含了 scripts/Kbuild.include, scripts/Makefile.lib 和 scripts/Makefile.host。字母 K 表示了它还包含内核树中其他子目录下的 Kbuild/Makefile 文件。

2. 使用 make -f 来调用

-f 是使用不同 makefile 文件来进行 make 的选项，请参见我们的 JulWiki。这种方式在内核中用的挺多，比方上面的 Cmp, Cfi 表示顶层 Makefile 调用了 scripts/Makefile.modpost 和 scripts/Makefile.fwinst。其调用的代码比方为：

```
PHONY += modules
modules: $(vmlinux-dirs) $(if $(KBUILD_BUILTIN),vmlinux)
        $(Q)$(AWK) '!x[$$0]++' $(vmlinux-dirs:%=$(objtree)/%/modules.order) > $(objtree)/modules.order
        @$(kecho) '  Building modules, stage 2.';
        $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modpost
        $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.fwinst obj=firmware __fw_modbuild
```

有时候 "-f ... obj" 的部分被一个变量定义所取代。比方上面的 Cb,Cc,Chi 表示顶层 Makefile 调用了 scripts/Makefile.build, scripts/Makefile.clean 和 scripts/Makefile.headersinst，调用的代码分别举例为:

```
# Basic helpers built in scripts/
PHONY += scripts_basic
scripts_basic:
        $(Q)$(MAKE) $(build)=scripts/basic
```

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

```
PHONY += headers_install
headers_install: __headers
        $(if $(wildcard $(srctree)/$(hdr-dir)/Kbuild),, \
        $(error Headers not exportable for the $(SRCARCH) architecture))
        $(Q)$(MAKE) $(hdr-inst)=include
        $(Q)$(MAKE) $(hdr-inst)=$(hdr-dir) $(hdr-dst)
```

上面代码揭示了所使用的变量分别为 $(build), $(clean)和$(hdr-inst)。 而它们被定义在不同的文件里，比如 $(build) 被定义在 scripts/Kbuild.include 中，具体代码为:

```
###
# Shorthand for $(Q)$(MAKE) -f scripts/Makefile.build obj=
# Usage:
# $(Q)$(MAKE) $(build)=dir
build := -f $(if $(KBUILD_SRC),$(srctree)/)scripts/Makefile.build obj
```

这种调用的方式在整个内核构建系统中非常普遍，所以需要先有个了解。实际上，变量的等号后面常跟有一个目录，而这个目录中一般都有一个 Kbuild/Makefile，也就是上面所说的所谓“”各子目录下的 Kbuild/Makefile”。如果你继续追踪到 scripts/Makefile.build 文件里，你会发现是她直接包含了这些”各子目录下的 Kbuild/Makefile“。下面这些就是踪迹所在：

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

4. 顶层Makefile的总体框架结构

既然前面我们说过顶层 Makefile 最为重要，那么我们就先来研究一下这个文件。在你用 VI 编辑器打开这个文件时，千万别被它的复杂吓倒。这个文件虽然行数颇多，但其实里面也是有道道可寻的，我们可以抽出其中最重要的框架结构出来，列出如下(稍做整理和缩进)：

```
ifeq ($(KBUILD_SRC),)

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

    ifneq ($(KBUILD_OUTPUT),)
        ...                 //A部分-第二次调用顶层Makefile
        skip-makefile := 1  //指定要输出到另外目录时设置 skip-makefile 变量
    endif # ifneq ($(KBUILD_OUTPUT),)

endif # ifeq ($(KBUILD_SRC),)

ifeq ($(skip-makefile),)
    ....          //B1部分-处理Kconfig，Kbuild共用的目标scripts_basic和outputmakefile
    ....          //B2部分-设置 mixed-targets, config-targets, dot-config 等变量
    ifeq ($(mixed-targets),1)
        ...       //C部分-依序处理各个目标
    else
        ifeq ($(config-targets),1)
            ...   //D部分-处理不同的配置目标
        else
            ...
            ifeq ($(dot-config),1)
                # Read in config
                -include include/config/auto.conf

                ifeq ($(KBUILD_EXTMOD),)
                    # Read in dependencies to all Kconfig* files, make sure to run
                    # oldconfig if changes are detected.
                    -include include/config/auto.conf.cmd
                    ... //G1部分-处理 silientoldconfig 目标，重新生成 .config、auto.conf、auto.conf.cmd 以及 autoconf.h
                else
                    ... //G2部分-测试是否存在 auto.conf 以及 autoconf.h，如不存在则报错
                endif # KBUILD_EXTMOD
            else
                # Dummy target needed, because used as prerequisite
                include/config/auto.conf: ;
            endif # $(dot-config)
                ...   //E部分-对vmlinux、modules等构建目标以及和.config无关目标的处理
        endif #ifeq ($(config-targets),1)
    endif #ifeq ($(mixed-targets),1)
    ...   //F部分-single target的处理
endif   # skip-makefile
```

从上面的框架中可以看出，影响内核构建过程动作的有数个变量，分别是：KBUILD_SRC， KBUILD_OUTPUT， skip-makefile， mixed-targets， config-targets 和 dot-config。我们将它们分成两组，前三为一组，后三个为一组。显然，前一组影响着框架中最外面的两个 ifeq-endif 块，而后一组则决定了第二个 ifeq-endif 块内的逻辑。

对于第一组变量，其实是为了支持"make O=1 [Targets]"，也就是为了支持将输出文件放到另外目录(不同于内核源代码目录的其他目录)的功能而准备的。那到底是如何支持的呢？

我们说其实这是两次调用顶层 Makefile 的过程。

首先，当我们输入"make O=dir [Targets]"命令的时候，会第一次调用顶层 Makefile。而此时变量 KBUILD_SRC 没有定义，所以会进入到第一个 ifeq-endif 块。进去之后，条件判断 ifeq ("$(origin O)", "command line")会发现命令行里有变量 O 的定义，并且其值为 dir。所以接下来，make 会把变量 O 的值，也就是 dir 赋给变量 KBUILD_OUTPUT。

既然 KBUILD_OUTPUT 的值为 dir，所以条件判断 ifneq ($(KBUILD_OUTPUT),)就必定成立，所以 make 会去处理上面的 A 部分。处理完 A 部分后，会将变量 skip-makefile 设置为 1。因此显然，在退出第一个 ifeq-endif 块后，进入第二个 ifeq-endif 块的条件判断就得不到满足。所以我们说，其实到这里，第一次调用 Makefile 的过程就提前结束了。

那第二次调用顶层 Makefile 的过程又是在哪里呢？答案是在对 A 部分的处理上。我们列出 A 部分中关键的代码：

```
PHONY += $(MAKECMDGOALS) sub-make

$(filter-out _all sub-make $(CURDIR)/Makefile, $(MAKECMDGOALS)) _all: sub-make
        $(Q)@:

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

从这里，我们可以看出，不管命令"make O=dir [Targets]"中的 Targets 个数有多少个，它们都是要依赖于 sub-make。所以，对 A 部分的处理，其实就是执行 sub-make 规则的命令。在这个命令中，KBUILD_OUTPUT 值为 dir，CURDIR 值为当前的内核源码目录。因为 make 命令中没有设置 M 变量或者 SUBDIRS 变量，所以 KBUILD_EXTMOD 值为空。所以这个命令就简化为：

make -C dir KBUILD_SRC=`pwd` KBUILD_EXTMOD="" -f `pwd`/Makefile [Targets]

这样一个命令的执行就和执行没带 O=dir 的"make [Targets]"命令差不多了。其差别只在于将输出文件存到 dir 目录，而非内核源码树目录而已。

从上面的框架中看出来，如果只是简单的"make [Targets]"，那么就不会进去到第一个 ifeq-endif 块，而直接进到第二个 ifeq-endif 块中去处理了。这就要涉及到上面说到的第二组变量了。我们说，设置第二组变量的目的，就是为了适应 make 命令后面所跟 Targets 数目和类型上的多样性而已。

在详细分析第二个 ifeq-endif 块之前，让我们看看第二组变量的含义以及它们的初始值设置情况。由于我们之前已经对几乎所有 Targets 的作用及分类做了说明，所以理解它们不应该很难。如上面框架图中所示的那样，让我们先抽出 B2 部分中和这个相关的代码如下：

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

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

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

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

上面的代码一开始就设置了变量 no-dot-config-targets，它指代的是那些和.config 没有关系的目标，这已经在前面对目标进行分类的时候说过了。接下来，它将三个变量分别赋初值 0，0 和 1。再接下来，它会判断 make 命令中的[Targets]部分是否有且仅有变量 no-dot-config-targets 所指代的那些目标，如果是的话它就将变量 dot-config 的值设置为 0。这表明本次 make 命令所 make 的目标都是和.config 文件没有关联的，既不是那些会产生.config 的配置目标，也不是那些需要使用.config 文件内容的构建目标。

好，上面的条件判断 ifeq ($(KBUILD_EXTMOD),)指示了如果我们不是编译外部模块的话就进去到这最后一个 ifeq-endif 块里面。进去后，构建系统如果发现 make 命令中[Targets]部分包含有 config，或者有%config 目标的话，它就将变量 config-targets 设置为 1，这表明本次 make 需要处理配置目标。在此基础之上，如果它还发现有其他目标的话，它就将 mixed-targets 变量设置为 1。注意 mixed-targets 变量的确切含义，其确切含义是指配置目标和其他目标相混合，而不是仅仅指配置目标和构建目标相混合。换句话讲，对于"make s3c2410_defconfig kernelversion"这样的命令来说，配置目标 s3c2410_defconfig 和与.config 文件不相关的目标 kernelversion 相混合，此时变量 mixed-targets 也会被设置为 1。

4.1. 顶层Makefile如何处理混合目标

讨论完 B2 部分对三个变量的初始化，让我们再回到主框架结构上面来。框架结构中接下来就是针对三变量不同的取值搭配进行处理了。

首先，如果 mixed-targets 取值为 1，则表明是混合目标的情况，构建系统要处理框架中的 C 部分。我们取出其中代码如下：

```
# ===========================================================================
# We're called with mixed targets (*config and build targets).
# Handle them one by one.

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

从代码中可以看出，这里使用了一个双冒号的模式匹配规则。百分号代表任何目标都使用这个规则，其中$(srctree)为内核代码树所在目录，KBUILD_SRC 定义为空。所以如果 make 命令为：make s3c2410_defconfig all，那么构建系统就会分别执行下面两条命令：

```
make -C $(srctree) KBUILD_SRC= s3c2410_defconfig
make -C $(srctree) KBUILD_SRC= all
```

这其实和简单的用手动的输入两条连续命令(make s3c2410_defconfig 和 make all)是一样效果的。

4.2. 顶层Makefile如何处理配置目标

回到主框架，如果 make 命令的[Targets]部分不是混合目标，而是单个目标。那么构建系统会先用 ifeq ($(config-targets),1)判断是否是配置目标，如果是，则处理框架中的 D 部分，我们也抽出 D 部分的代码如下：

```
# ===========================================================================
# *config targets only - make sure prerequisites are updated, and descend
# in scripts/kconfig to make the *config target

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

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

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

观察上面 D 部分的代码，无论配置目标是 config，还是%config 形式的，都依赖于 script_basic 和 outputmakefile 两个目标。我们可以找到这两个依赖是定义在框架 B1 部分中的，抽出代码如下：

```
# ===========================================================================
# Rules shared between *config targets and build targets

# Basic helpers built in scripts/
PHONY += scripts_basic
scripts_basic:
        $(Q)$(MAKE) $(build)=scripts/basic

# To avoid any implicit rule to kick in, define an empty command.
scripts/basic/%: scripts_basic ;

PHONY += outputmakefile
# outputmakefile generates a Makefile in the output directory, if using a
# separate output directory. This allows convenient use of make in the
# output directory.
outputmakefile:
ifneq ($(KBUILD_SRC),)
        $(Q)ln -fsn $(srctree) source
        $(Q)$(CONFIG_SHELL) $(srctree)/scripts/mkmakefile
            $(srctree) $(objtree) $(VERSION) $(PATCHLEVEL)
endif
```

正如上面代码中的注释所说的一样，这两个目标对应的规则其实是 Kconfig 和 Kbuild 都要用到的。scripts_basic 规则的命令 (Q)(MAKE) $(build)=scripts/basic 做的工作就是：make -f scripts/Makefile.build obj=scripts/basic ，为什么？回过头去看看我们前面说的构建系统内各 makefile 的关联就知道了。

make -f scripts/Makefile.build obj=scripts/basic 命令由于没有指定目标，所以会在 script/Makefile.build 中处理默认目标__build，如下：

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

同时，别忘记在 scripts/Makefile.build 中会包含进 scripts/basic 目录下的 Kbuild/Makefile，所以该 make 命令的实际效果是去编译出 scripts/basic 目录下的三个 host program，也就是 fixdep docproc 和 hash。什么是 host program？一般认为是和内核无关，但是要在编译过程中使用的工具程序。关于这些程序的编译，请参见 scripts/Makefile.host 文件，以及 Documentation/kbuild/makefile.txt 文件中关于 host program 的这一节。请你留意这里的变量$(always)。

对于目标 outputmakefile，其实只在 make 命令带有 O 变量时才有用，因为这个时候变量 KBUILD_SRC 不会为空。这个时候中间文件不会存在内核源码树目录中，而是存在另外的一个目录。这个没什么困难的，所以我们这里不再做过多说明。

好，回到我们对配置目标的处理上面来。在处理完两个依赖后，构建系统将会创建两个目录: include/linux 和 include/config。接着执行 (Q)(MAKE) $(build)=scripts/kconfig $@ 。如果我们的 make 命令是 "make s3c2410_defconfig" 的话，这个时候执行的就是:

make -f scripts/Makefile.build obj=scripts/kconfig s3c2410_defconfig

又牵涉到文件 scripts/Makefile.build 了，我们先搜索一下这个文件里面有没有 s3c2410_defconfig 或 类似于 %config 之类的目标。没有，怎么办，该不会是弄错了吧？呵呵，你忘记了么？文件 scripts/Makefile.build 会包含 obj 变量所指代目录内的 Makefile 的，在这里就是 script/kconfig/Makefile。打开这个文件，果然能找到相关的规则：

```
%_defconfig: $(obj)/conf
        $(Q)$< -D arch/$(SRCARCH)/configs/$@ $(Kconfig)
```

在这里，s3c2410_defconfig 需要依赖于同目录下的 conf 程序。这其实就是 Linux 内核进行 Kconfig 操作的主程序之一了，类似的还有 mconf,qconf 和 gconf 等。他们其实都是 host program。关于它们是如何被编译出来的，还请参见 scripts/kconfig/Makefile 文件，主要是借助于 bison，flex 和 gperf 三个工具来生成 c 源程序文件，之后再编译出来的。由于这部分和我们 Linux 内核的构建主题关系不大，所以我在这里不再赘述。

由于变量 KBUILD_KCONFIG 在 arm 架构 Makefile 中没有被定义，所以 Kconfig 被定义成 arch/arm/kconfig，所以这个目标的规则就简化成:

$(obj)/conf -D arch/arm/configs/s3c2410_defconfig arch/arm/Kconfig

这个命令就是读取并解析以 arch/arm/Kconfig 为首的内核功能选项配置文件，并将文件 arch/arm/configs/s3c2410_defconfig 所设置的默认值分配给对应的所有选项，最终生成隐藏配置文件 .config 。你可以打开看一下 .config 文件的内容，都是这样的格式：

```
#
# Automatically generated make config: don't edit
# Linux kernel version: 2.6.31
# Fri Oct 29 18:27:42 2010
#
CONFIG_ARM=y
CONFIG_HAVE_PWM=y
CONFIG_SYS_SUPPORTS_APM_EMULATION=y
CONFIG_GENERIC_GPIO=y
CONFIG_MMU=y
CONFIG_NO_IOPORT=y
CONFIG_GENERIC_HARDIRQS=y
...
```

里面貌似都是一些变量的定义。稍后我们会看到在内核开始真正编译之前，构建系统会以 .config 文件为蓝本生成 include/config/auto.conf 文件，这个文件的格式和 .config 类似，这个文件会在顶层 以及 scripts/Makefile.build 文件中被直接包含进来，所以这些变量其实就成了 GNU Make 的变量。而内核各子目录中的 Kbuild/Makefile 就可以使用这些变量的定义，来决定是否将该目录下对应的代码功能直接编译到内核里面(这些变量取值为"y")、编译成模块(取值为"m")或者干脆不进行编译(取值为"空")。可以想见，如果选择不编译，那出来的 Linux 内核就不会有对应的功能。

前面之所以说是以 arch/arm/Kconfig 为首的，那就说明功能配置选项文件可能有多个。的确如此，你如果打开文件 arch/arm/Kconfig 就会看到它通过 source 来包含其他的选项配

置文件：

```
....
if ALIGNMENT_TRAP || !CPU_CP15_MMU
source "drivers/mtd/Kconfig"
endif

source "drivers/parport/Kconfig"

source "drivers/pnp/Kconfig"

source "drivers/block/Kconfig"

# misc before ide - BLK_DEV_SGIIOC4 depends on SGI_IOC4

source "drivers/misc/Kconfig"

source "drivers/ide/Kconfig"

source "drivers/scsi/Kconfig"

source "drivers/ata/Kconfig"

source "drivers/md/Kconfig"

source "drivers/message/fusion/Kconfig"
.....
```

不光是 arm 架构这样，其他所有的架构也都这样。在配置的时候，配置工具首先会解析架构平台目录下的 Kconfig，这就是所谓和平台相关的主 Kconfig。主 Kconfig 文件会包含其他目录的 Kcofnig 文件，而其他目录的 Kconfig 又会包含其他各子目录的 Kconfig。如此形成一个树型结构。

另外需要说一下的，如果你的 make 命令是 "make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig"，那用到的配置工具将会是 mconf，它同样会读取那些配置选项文件，只不过不同的是，它会 show 除一个菜单界面，并将这些可设置的选项一一呈现出来，如此我们就可以手动进行设置，而不是让 mconf 去读取 arch/arm/configs/s3c2410_defconfig 之类的文件进行默认设置了。关于 gconf 等其他配置工具以及 Kconfig 文件中具体的选项配置语法，我这里就不再详细叙述了，你可以参考目录 Documentation/kbuild/ 下的文件 kconfig.txt 以及 kconfig-language.txt 。关于配置目标，除了这里的 menuconfig 和 s3c2410_defconfig，我们后面还会说到另外一个很重要的配置目标: silentoldconfig 。

好，做个阶段性总结吧。作为内核构建系统对 kconfig 的支持，到这步就算是结束了，其根本目标是产生 .config 隐藏文件，用以记录我们所需要的配置结果。但是在 Linux 内核里面，仅仅把配置结果保存在像 .config 这样一个文件中是不够的。为什么这么说？我们说辛苦辛苦保存起来的东西在关键时候总是需要派上用场的。那么在 Linux 系统中，这些配置结果由谁去用，又是怎么去用呢？我们在后面会给大家一一道来，其实关于由谁去用，我们前面已经稍微提到过一点了。这里我们还是先不去管这些，休息片刻后把注意力转移到我们的主框架上面来吧。

4.3. 顶层Makefile如何处理构建目标及和.config无关目标

回到我们的主框架上面来，讨论完配置目标的处理后，就轮到框架中 "ifeq ($(config-targets),1)-endif" 块的 else 部分了。这部分是为了处理那些构建目标以及和.config 无关的目标，其对这些目标处理的代码都位于框架中的 E 部分中。在 E 部分之前，有一个不小的 "ifeq ($(dot-config),1)-endif" 块，我们暂先不去理会，且看这个 ifeq-endif 块之前有一小段注释：

```
# ===========================================================================
# Build targets only - this includes vmlinux, arch specific targets, clean
# targets and others. In general all targets except *config targets.
```

这段注释字面上的意思是说 "ifeq ($(config-targets),1)-endif" 块的 else 部分处理的都是 Build targets，也就是除了配置目标之外的其他目标。注意他这里对 Build targets 分类方法，其实和我们之前的分类方法是有差异的，他这里所谓的 Build targets ，除了包括我们之前分类中所说的构建目标外，还包括之前我们说的和 .config 文件无关的那些目标。这其实是对同个东西的两种不同分类罢了，不影响我们的分析。其实不管哪种分类，都改变不了在本 else 部分既处理真正的构建目标，又处理那些和 .config 文件无关目标的事实。

好，鉴于我们已有这样的事实认同。那接下来理解前面说的那个不小的 "ifeq ($(dot-config),1)-endif" 块就比较容易了。很显然这个时候如果变量 dot-config 等于 1，那说明针对的是那些真正的构建目标，因为他们需要文件 .config 来完成真正的构建。而如果这个变量不为 1，那么针对的就是那些和 .config 完全无关的目标了。

当 dot-config 等于 1 时，构建系统首先会尝试性的包含 include/config/auto.conf 文件。为什么说是尝试性的？这是 GNU make 做 "-include" 的特性。其意思和是否存在所包含的文件以及是否能根据所存在的规则去重新创建所包含文件有关系。

GNU Make 是这样一个大致的读取 Makefile 的流程：首先它读入主 Makefile，在读的过程中，如果碰到 "include" 或 "-include"，它就会包含对应的文件。如果对应的文件不存在，则暂时跳过做包含的地方，继续读入。待所有 makefie 都读完后。GNU Make 会考虑将每个 makefile 作为目标，在全局范围内查找是否有能生成这些目标的规则，如果发现有一个 makefile 可以被一条规则生成，那么 GNU Make 就会先生成这个 makefile。生成后，GNU Make 又会从零开始读入主 Makefile 以及所有被包含的 makefile，然后再检查是否有 makefile 可以被 remade….这样一次又一次，直到所有的 makefile 都不需要再次生成了，它才处理依赖规则链。它之这样做，是为了保证所有 makefile 都是 update-to-date 的。

那如果你的子 makefile 是被 "include" 所包含的，但是这个 makefile 本身不存在，且无法用一条规则去 Remake 出来，那么 GNU Make 就会报错并退出。相反，如果你用的是 "-include"，那么 GNU Make 就什么都不做，就好象什么也没发生过那样继续处理后面的事情。所以，我们说这里是尝试性的，通俗点就是”有则包含，没有也罢了“:)。

接下来回到主框架，假如你的 make 命令是 "make ARCH=arm CROSS_COMPILE=arm-linux- zImage"，那么 dot-config 等于 1，并且变量 KBUILD_EXTMOD 会等于空。构建系统又会先尝试性的包含文件

include/config/auto.conf.cmd，然后继续处理主框架中的 G1 部分。我们先看看 G1 部分的代码：

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

# If .config is newer than include/config/auto.conf, someone tinkered
# with it and forgot to run make oldconfig.
# if auto.conf.cmd is missing then we are probably in a cleaned tree so
# we execute the config step to be sure to catch updated Kconfig files
include/config/auto.conf: $(KCONFIG_CONFIG) include/config/auto.conf.cmd
	$(Q)$(MAKE) -f $(srctree)/Makefile silentoldconfig
```

:)，你看到这里也许会露出些许微笑，因为你还记得那个 "-include include/config/auto.conf"。没错，针对你的 make 命令，GNU Make 在处理任何目标之前，它一定会努力的为我们生成文件 include/cofig/auto.conf ，用的就是这里的这条对应规则。注意这里的细节，GNU Make 同样也会努力重新用上面这条规则去生成 include/config/auto.conf.cmd 文件，可是无奈上面这条规则既没有依赖，又没有命令，所以 GNU Make 是怎么也没办法生成 auto.conf.cmd。那么 auto.conf.cmd 文件又是从什么地方生成呢？答案是在生成 include/config/auto.conf 的时候。稍后，我们会看到，它同样在这个时候生成了另外一个文件 include/linux/autoconf.h。

既然已经说到这几个文件了，那我就预先来回答一下上面提出来的问题。什么问题？就是配置过程最后产生了用于记载配置结果的 .config ，那么其中的配置结果由谁使用，又是如何使用的问题。我们说在整个 Linux 内核系统中，有两方面的用户需要关注所记载的配置结果。一个自然是内核构建系统，它需要根据配置结果产生具有指定功能的内核映像；另外一个就是大部分代码为 C 语言代码的 Linux 内核本身，它也需要用户的配置结果，主要用来预处理 C 代码。前者使用配置结果，并不是直接通过 .config 文件来的，而是将其转换成两个文件：include/config/auto.conf 和 include/config/auto.conf.cmd。后者也没办法直接通过 .config 文件来使用配置结果，它需要将其转换成 C 语言头文件的形式使用，在这里就是文件 include/linux/autoconf.h 。关于这三个文件的内容，我们稍后叙述。

还是回到上面处理 auto.conf 的规则中来，变量 KCONFIG_CONFIG 指代的就是配置文件 .config 。目标 auto.conf 依赖于 .config 和 auto.conf.cmd 。当 GNU Make 使用这条规则来 remake auto.conf 的时候，这两个文件即使不存在也没有关系。因为这样的话，GNU Make 会因为有上面这条没有依赖也没有命令的规则而认为这两个依赖是最新的，所以此时 auto.conf 规则的命令总是会被得到执行。

这是有意思的地方，因为这是不是意味着我可以在一个刚刚解压出来的 Linux 内核目录中直接用命令 "make ARCH=arm CROSS_COMPILE=arm-linux- zImage"来完成构建呢？咱们试一下：

```
[yihect@juliantec linux-2.6.31]$ make ARCH=arm CROSS_COMPILE=arm-linux-  distclean
[yihect@juliantec linux-2.6.31]$ make ARCH=arm CROSS_COMPILE=arm-linux-  zImage
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/basic/docproc
  HOSTCC  scripts/basic/hash
  HOSTCC  scripts/kconfig/conf.o
  HOSTCC  scripts/kconfig/kxgettext.o
  SHIPPED scripts/kconfig/zconf.tab.c
  SHIPPED scripts/kconfig/lex.zconf.c
  SHIPPED scripts/kconfig/zconf.hash.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
scripts/kconfig/conf -s arch/arm/Kconfig
***
*** You have not yet configured your kernel!
*** (missing kernel config file ".config")
***
*** Please run some configurator (e.g. "make oldconfig" or
*** "make menuconfig" or "make xconfig").
***
make[2]: *** [silentoldconfig] Error 1
make[1]: *** [silentoldconfig] Error 2
make: *** [include/config/auto.conf] Error 2
```

从结果看前面都没问题，直到用 conf 进行最后一步的配置时出现了错误。至于错误原因呢，是因为找不到文件 .config 。我们可以找出出现错误的代码，就在文件 scripts/kconfig/conf.c 的 main 函数中，如下所示的代码片段：

```
   if (sync_kconfig) {
            name = conf_get_configname();
            if (stat(name, &tmpstat)) {
                    fprintf(stderr, _("***\n"
                            "*** You have not yet configured your kernel!\n"
                            "*** (missing kernel config file \"%s\")\n"
                            "***\n"
                            "*** Please run some configurator (e.g. \"make oldconfig\" or\n"
                            "*** \"make menuconfig\" or \"make xconfig\").\n"
                            "***\n"), name);
                    exit(1);
            }
    }
```

就像前面讲的，这个时候 GNU Make 会执行 auto.conf 规则的对应命令"(Q)(MAKE) -f $(srctree)/Makefile silentoldconfig"。而针对这个命令，配置程序 conf 最后会在 main 函数上面这个代码片段的前面，将上面这个代码片段中的 sync_kconfig 设置为 1。接下来取得配置文件的名称放在 name 变量里面，再使用 stat 函数查询这个文件的状态。结果该函数返回非 0 值，使得 GNU Make 认为配置文件.config 不存在而报错。

回到处理 auto.conf 的那条规则上来。考虑一下如果这个时候 auto.conf 文件和 .config 文件都存在，但是 .config 比较新的话会怎么样呢？想想，这会是一个怎么样的 Scenario 呢？如果你刚刚编译过内核并已生成映像文件，这个时候你从你朋友那里取来一个 .config 文件，并且将其放到了你的内核源码树下面，这个时候就产生了这种情况。这个时候你需要根据你的内核的版本来做处理。如果你的版本和你的朋友一样，那没问题，你大可以直接 "make ... zImage"。但是，如果你们的版本不一样，你最好做下 "make ... oldconfig"。这个命令的作用是针对原来版本内核的配置文件设置来配置新版本的内核。新版内核中没变的配置选项可以设置成原来的值，但是新添加的配置项就需要你自己手动设置了。

再次回到处理 auto.conf 的那条规则上来，我们看到它的命令 "(Q)(MAKE) -f $(srctree)/Makefile silentoldconfig"，这个命令最终会导致 GNU Make 执行文件 scripts/kconfig/Makefile 中针对目标 silentoldconfig 的命令：

$(obj)/conf -s arch/arm/Kconfig

conf 配置程序在前面已经有所提及，其对应的代码都在目录 scripts/kconfig/ 中。conf 的主函数 main 即定义在 conf.c 文件中。其实，目标 silentoldconfig 和 目标 oldconfig 类似，只不过它多了生成 auto.conf、auto.conf.cmd 以及 autoconf.h 等三个文件的任务。这是怎么做到的？答案就在 conf.c 文件中 main 函数最后的一段代码：

```
int main(int ac, char **av)
{
        int opt;
        const char *name;
        struct stat tmpstat;
        ....
        ....
        if (sync_kconfig) {
                /* silentoldconfig is used during the build so we shall update autoconf.
                 * All other commands are only used to generate a config.
                 */
                if (conf_get_changed() && conf_write(NULL)) {
                        fprintf(stderr, _("\n*** Error during writing of the kernel configuration.\n\n"));
                        exit(1);
                }
                if (conf_write_autoconf()) {
                        fprintf(stderr, _("\n*** Error during update of the kernel configuration.\n\n"));
                        return 1;
                }
        } else {
                if (conf_write(NULL)) {
                        fprintf(stderr, _("\n*** Error during writing of the kernel configuration.\n\n"));
                        exit(1);
                }
        }
        return 0;
}
```

前面我们已经介绍过，当用 conf 处理 silentoldconfig 时，变量 sync_kconfig 会被设置为 1。实际上，也只有处理此目标时，它才会被设置成 1，其他的目标都不会。对于 oldconfig、menuconfig 等目标来说，conf 程序最后会直接调用函数 conf_write 将配置结果写到配置文件 .config 中去。该函数的定义在同目录的另外一个文件 confdata.c 中，这里我们不再细究下去了，你可以自己探究。而对于 silentoldconfig 目标来说，conf 程序除了调用 conf_write 来写 .config 文件外，它还会调用 conf_write_autoconf 函数来完成 auto.conf、auto.conf.cmd 和 autoconf.h 三个文件的生成。我们且来看看同样定义在 confdata.c 文件中的 conf_write_autoconf 函数：

```
int conf_write_autoconf(void)
{
        ....
        file_write_dep("include/config/auto.conf.cmd");

        if (conf_split_config())
                return 1;

        out = fopen(".tmpconfig", "w");
        if (!out)
                return 1;

        out_h = fopen(".tmpconfig.h", "w");
        if (!out_h) {
                fclose(out);
                return 1;
        }

        sym = sym_lookup("KERNELVERSION", 0);
        sym_calc_value(sym);
        time(&now);
        fprintf(out, "#\n"
                     "# Automatically generated make config: don't edit\n"
                     "# Linux kernel version: %s\n"
                     "# %s"
                     "#\n",
                     sym_get_string_value(sym), ctime(&now));
        fprintf(out_h, "/*\n"
                       " * Automatically generated C config: don't edit\n"
                       " * Linux kernel version: %s\n"
                       " * %s"
                       " */\n"
                       "#define AUTOCONF_INCLUDED\n",
                       sym_get_string_value(sym), ctime(&now));

        for_all_symbols(i, sym) {
                sym_calc_value(sym);
                if (!(sym->flags & SYMBOL_WRITE) || !sym->name)
                        continue;
                ....
        }
        fclose(out);
        fclose(out_h);

        name = getenv("KCONFIG_AUTOHEADER");
        if (!name)
                name = "include/linux/autoconf.h";
        if (rename(".tmpconfig.h", name))
                return 1;
        name = conf_get_autoconfig_name();
        /*
         * This must be the last step, kbuild has a dependency on auto.conf
         * and this marks the successful completion of the previous steps.
         */
        if (rename(".tmpconfig", name))
                return 1;

        return 0;
}
```

该函数一开始就写 include/config/auto.conf.cmd 文件，然后将对应的内容写入到两个临时文件 .tmpconfig 和 .tmpconfig.h 中，并在最后将这两个文件分别重新命名为 include/config/auto.conf 和 include/linux/autoconf.h。注意上面代码中对 conf_split_config 函数的调用，其目的是在目录 include/config 中产生一系列的头文件，至于这些头文件如何产

生的、以及它们的作用，我们这里先留下一个伏笔，回头再来探究。

知道了这几个文件是如何产生的。我们再来注意一下它们的内容。文件 auto.conf.cmd 里面记录的是 auto.conf 目标的相关依赖，而文件 auto.conf 和文件 autoconf.h 的内容和 .config 文件的内容直接相关。举例来说，如果你找到有个定义在 .config 文件里的变量形式：CONFIG_MMU=y 表示要将虚拟内存管理的功能编译进内核，那么在 auto.conf 里面也会有完全相同的定义形式：CONFIG_MMU=y。而在文件 autoconf.h 文件中，则会有一个这样形式的宏定义：#define CONFIG_MMU 1。假如 .config 中的形式是 CONFIG_IKCONFIG=m，那么在 auto.conf 中的形式也会是：CONFIG_IKCONFIG=m ，而在文件 autoconf.h 中的定义形式则变成：#define CONFIG_IKCONFIG_MODULE 1 。

前面说的是当 KBUILD_EXTMOD 为空的时候，那么当这个变量取值不为空(也就是我们尝试在用 "make ... -M=..." 之类的命令来编译外部模块)时，GNU Make 只是简单的处理上面框架中的 G2 部分。编译外部模块时并不需要关心 auto.conf/autoconf.h 是不是最新的。它只是检查他们是否存在，如果不存在，它就报错并退出。

之前我们说过构建目标以及和.config 无关的目标是混在一块处理的。那当 dot-config 等于 1 时，处理的是构建目标，而在 dot-config 等于 0 时，构建系统处理的是那些和文件 .config 没有关系的目标。注意看我们的框架，处理和.config 无关目标的时候，构建系统只是简单的针对 auto.conf 目标定义了一个既没有依赖又没有命令的规则：

```
# Dummy target needed, because used as prerequisite
include/config/auto.conf: ;
```

这样做的意思是因为 auto.conf 在后面会为多个其他目标所依赖。我们在这里只是登记一下说：“嘿，兄弟我(auto.conf)已经是最新的了，你们别再管我了，只管继续做你们自己的事情就好。”

关于对上面框架代码 E 部分中和 .config 文件无关目标的处理，这里限于篇幅，我们就不再多讲，您可自己研究。

5. 用两个例子深入讨论如何处理构建目标

对另外构建目标的处理，我们使用两个例子来讲述，那就是配置内核后用来编译内核的命令:"make ARCH=arm CROSS_COMPILE=arm-linux- "和编译外部模块的命令:"make ARCH=arm CROSS_COMPILE=arm-linux- -C KERNELDIR M=dir"。之所以选取这两个 make 命令 来作为例子讲述，是因为它所涉及到的关于构建系统的知识比较多，覆盖比较完整。

在开始着手讲例子之前，我们先来关注一下框架中的 E 部分。其实，E 部分本身又可以分成两个小部分，其中之一是为了处理基本内核以及内部模块等的；其中之二是为了处理外部模块的。这两部分就像下面这样按照变量 KBUILD_EXTMOD 的取值分开：

```
ifeq ($(KBUILD_EXTMOD),)
  ... //E1部分-对基本内核vmlinux、内部模块modules等的处理
else # KBUILD_EXTMOD
  ... //E2部分-对外部模块modules等的处理
endif # KBUILD_EXTMOD
```

值得先说一说的是，第一个例子所涉及的代码均分布在 E1 部分中，而第二个例子所涉及的代码都在 E2 部分中。

5.1. make ARCH=arm CROSS_COMPILE=arm-linux-

好，咱们先讲第一个例子中用来编译内核的命令。在这个 make 命令中，因为没有明确指出所要 make 的目标，所以实际上这个命令要处理顶层 Makefile 中的的缺省目标 _all。

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

而在刚进入 "ifeq ($(skip-makefile),)-endif" 块后，就会有这样的代码：

```
# 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
```

这说明什么？如果我们处理的不是外部模块，那目标 _all 即依赖于 all，否则目标 _all 依赖于 modules。所以前面的 make 命令实际上就等价于：

"make ARCH=arm CROSS_COMPILE=arm-linux- all"。

构建系统中关于目标 all 的处理，我们可以从被顶层 Makefile 所包含的架构 Makefile 中找到：

```
# Default target when executing plain make
ifeq ($(CONFIG_XIP_KERNEL),y)
KBUILD_IMAGE := xipImage
else
KBUILD_IMAGE := zImage
endif

all:    $(KBUILD_IMAGE)
```

同时我们还可以在顶层 Makefile 中找到关于 all 目标的规则(两条)：

```
# 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
```

以及

```
ifdef CONFIG_MODULES
# By default, build modules as well
all: modules
...
else # CONFIG_MODULES
...
endif # CONFIG_MODULES
```

因为对于 s3c2410_defconfig 的默认配置来说，CONFIG_XIP_KERNEL 变量没有被定义，而变量 CONFIG_MODULES 被定义为 y 。所以，实际上构建系统对 all 目标的处理，就是先后处理这样三个目标：vmlinux zImage 和 modules。

5.1.1. 对目标 vmlinux 的处理

首先，我们来看看对目标 vmlinux 的处理。在顶层 Makefile 中，在框架代码的 E1 部分中可以找到：

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

从上面的代码可见，vmlinux 依赖于 $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o $(kallsyms.o)， 而从下面的代码中又可以看出 $(vmlinux-lds)、$(vmlinux-init)和

$(vmlinux-main)等三个目标又依赖于$(vmlinux-dirs)。

```
# The actual objects are generated when descending,
# make sure no implicit rule kicks in
$(sort $(vmlinux-init) $(vmlinux-main)) $(vmlinux-lds): $(vmlinux-dirs) ;
```

在继续讨论之前，我们先看看前面 vmlinux 对应规则中所碰到过的变量定义。我们一起把它们列出来：

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

变量 vmlinux-lds 指代的是 arch/arm/kernel/ 目录中的 vmlinux.lds 文件，大家知道 lds 文件是用来指导连接器做连接的脚本了。关于它的目的我们就这里不做具体描述，我们仅说说内核构建系统是如何处理 .lds 文件的。在内核里面，它是由同目录下的同名 .lds.S 文件所预处理出来的。其对应的规则定义在文件 scripts/Makefile.build 中:

```
# Linker scripts preprocessor (.lds.S -> .lds)
# ---------------------------------------------------------------------------
quiet_cmd_cpp_lds_S = LDS     $@
      cmd_cpp_lds_S = $(CPP) $(cpp_flags) -D__ASSEMBLY__ -o $@ $<

$(obj)/%.lds: $(src)/%.lds.S FORCE
        $(call if_changed_dep,cpp_lds_S)
```

观察上面规则中的命令，就是以 cpp_lds_S 为第一个参数，调用了函数 if_changed_dep，该函数(其实就是一个 make 变量)定义在文件 scripts/Kbuild.include 中：

```
# Execute command if command has changed or prerequisite(s) are updated.
#
if_changed = $(if $(strip $(any-prereq) $(arg-check)),                          \
        @set -e;                                                                \
        $(echo-cmd) $(cmd_$(1));                                                \
        echo 'cmd_$@ := $(make-cmd)' > $(dot-target).cmd)

# Execute the command and also postprocess generated .d dependencies file.
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)

# Usage: $(call if_changed_rule,foo)
# Will check if $(cmd_foo) or any of the prerequisites changed,
# and if so will execute $(rule_foo).
if_changed_rule = $(if $(strip $(any-prereq) $(arg-check) ),                    \
        @set -e;                                                                \
        $(rule_$(1)))
```

从上面代码可以看出，该函数连同另外两个函数：if_changed 和 if_changed_rule 一起构成一个系列。他们的用法一致，都是 $(call if_XXXX_XX, YYYY) 的形式，调用的结果都是将函数 if_XXXX_XX 中的$(1)部分用 YYYY 去取代而构成的 if 判断式，这种调用形式一般放在一个规则的命令部分里面。

其中最简单的就是 if_changed，当发现规则的依赖有被更新了、或者编译该规则对应目标的命令行发生改变了，它就先用 $(echo-cmd) 回显出新的命令$(cmd_$(1))，接着执行命令$(cmd_$(1))，最后再将该命令写到一个叫做 $(dot-target).cmd 的临时文件中去，以方便

下一次检查命令行是否有变的时候用。变量 dot-target 定义成 .targetname 的形式，如下：

```
###
# Name of target with a '.' as filename prefix. foo/bar.o => foo/.bar.o
dot-target = $(dir $@).$(notdir $@)
```

那如何去检查规则的依赖文件被更新了，以及检查编译该规则对应目标的命令行发生改变了的呢？答案就是下面的两个定义：

```
# Find any prerequisites that is newer than target or that does not exist.
# PHONY targets skipped in both cases.
any-prereq = $(filter-out $(PHONY),$?) $(filter-out $(PHONY) $(wildcard $^),$^)
```

```
ifneq ($(KBUILD_NOCMDDEP),1)
# Check if both arguments has same arguments. Result is empty string if equal.
# User may override this check using make KBUILD_NOCMDDEP=1
arg-check = $(strip $(filter-out $(cmd_$(1)), $(cmd_$@)) \
                    $(filter-out $(cmd_$@),   $(cmd_$(1))) )
endif
```

在 any-prereq 变量定义中，$(filter-out $(PHONY),$?) 指代的是那些比目标还新的依赖文件，而 $(filter-out $(PHONY) $(wildcard $^),$^) 指的是那些当前还不存在的依赖文件。另外注意 arg-check 变量定义中比较新老命令的方式。假设我们现在有下面这样一条规则调用了函数 if_changed：

```
target: prereq1 prereq2
        $(call if_changed,link_target)
```

那么上面比较的是变量 cmd_link_target 所指代的新命令和变量 cmd_target 所指代的老命令。而这个老命令就是被 if_changed 写入文件 .target.cmd 的。可以想见，内核构建系统必定在某个地方将这些包含老命令的 .*.cmd 读入进来。没错，读入代码可以找到在顶层 Makefile 中：

```
# read all saved command lines

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

ifneq ($(cmd_files),)
  $(cmd_files): ;       # Do not try to update included dependency files
  include $(cmd_files)
endif
```

注意了，上面只包含了处理那些列在变量 targets 中的目标的老命令。所以如果你想让构建系统也帮你比较新老命令，并若发现其中有区别就帮你处理的话，你需要将你的目标也列入 targets 变量中。另外，因为构建系统中目录 scripts 下的很多 Makfile 和顶层 Makefile 是独立运行的，所以在目录 scripts 下面，像在 Makefile.build、Makefile.headersinst、Makefile.modpost 以及 Makefile.fwinst 等文件中，你也可以找到类似的读入代码。

if_changed_dep 函数和 if_changed 差不多，所不同的是它用 fixdep 工具程序处理了依赖文件 *.d ，并将依赖信息也一并写入到文件 .targetname.cmd 文件中去。可以说依赖处理是整个内核构建系统中最难理解的部分，我们后面会花一点专门的篇幅来讨论它。if_changed_rule 其实也和 if_changed 差不多，只不过它直接调用了命令 rule_$(1)，而不是 cmd_$(1) 所指代的命令。if_changed_XXX 系列在内核构建系统中用的比较多，还请注意掌握。

回过头去看看变量 vmlinux-init 和 vmlinux-main 的定义。其中变量 head-y 通常被定义在架构相关的 Makefile 中，它通常包含那些需要放在编译出来的内核映像文件前面的对象文件。比方在 arch/arm/Makefile 中定义的 head-y：

head-y := arch/arm/kernel/head$(MMUEXT).o arch/arm/kernel/init_task.o

除了 head-y 之外，其他的变量最开始的时候都是作为目录定义在顶层 Makefile 中：

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

# Objects we will link into vmlinux / subdirs we need to visit
init-y          := init/
drivers-y       := drivers/ sound/ firmware/
net-y           := net/
libs-y          := lib/
core-y          := usr/
endif # KBUILD_EXTMOD
....
core-y          += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
```

注意，在架构相关的 Makefile 中，通常都会往这些变量中追加架构相关的代码目录。这些目录中包含架构相关的需要编译进内核的代码，而顶层 Makefile 中的这些都是架构无关的代码。举例说来，比方在 arch/arm/Makefile 中有这样的代码：

```
# If we have a machine-specific directory, then include it in the build.
core-y                          += arch/arm/kernel/ arch/arm/mm/ arch/arm/common/
core-y                          += $(machdirs) $(platdirs)
core-$(CONFIG_FPE_NWFPE)        += arch/arm/nwfpe/
core-$(CONFIG_FPE_FASTFPE)      += $(FASTFPE_OBJ)
core-$(CONFIG_VFP)              += arch/arm/vfp/

drivers-$(CONFIG_OPROFILE)      += arch/arm/oprofile/

libs-y                          := arch/arm/lib/ $(libs-y)
```

接下来，内核构建系统又会在顶层 Makefile 中重新设置这些指代不同目录的变量，如下：

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

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

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

设置的结果，就是原先指代目录的这些变量都变成了指代文件的变量。这些文件的名称，大部分是 built-in.o。你可以看到几乎任何一个目录中都应该会有这样的文件。另外，还有一部分名为 lib.a 的文件。你看到后面会知道，内核构建系统会一一产生这些对象文件和库代

码，连同前面的变量 head-y 指代的对象文件，还有其他的对象文件，构建系统会将他们连接起来，构成基本的 Linux 内核映像 vmlinux。

注意看，在上面的代码中，在设置这些目录之前，构建系统会将所有可能的代码目录保存在变量 vmlinux-dirs 。之所以会有 -y 和 -m 的区别，那是因为将这些目录赋值给这些目录变量的时候，常会使用配置选项 CONFIG_XXXXX 之类的。比方前面在 arch/arm/Makefile 中给 core-$(CONFIG_VFP) 赋值的形式。

由于我们在前面已经看到 $(vmlinux-lds)、$(vmlinux-init)、$(vmlinux-main)等目标依赖于 $(vmlinux-dirs)。所以对目标$(vmlinux-dirs)处理的规则就成为关键。我们从顶层 Makefile 中把它找出来：

```
# Handle descending into subdirectories listed in $(vmlinux-dirs)
# Preset locale variables to speed up the build process. Limit locale
# tweaks to this spot to avoid wrong language settings when running
# make menuconfig etc.
# Error messages still appears in the original language

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

在看这条规则的命令之前，我们先看看它有两个依赖：prepare 和 scripts。下面是顶层 Makefile 中 prepare 相关的代码：

```
# Things we need to do before we recursively start building the kernel
# or the modules are listed in "prepare".
# A multi level approach is used. prepareN is processed before prepareN-1.
# archprepare is used in arch Makefiles and when processed asm symlink,
# version.h and scripts_basic is processed / created.

# Listed in dependency order
PHONY += prepare archprepare prepare0 prepare1 prepare2 prepare3

# prepare3 is used to check if we are building in a separate output directory,
# and if so do:
# 1) Check that make has not been executed in the kernel src $(srctree)
# 2) Create the include2 directory, used for the second asm symlink
prepare3: include/config/kernel.release
ifneq ($(KBUILD_SRC),)
        @$(kecho) '  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;                                          \
            ln -fsn $(srctree)/include/asm-$(SRCARCH) include2/asm;     \
        fi
endif

# prepare2 creates a makefile if using a separate output directory
prepare2: prepare3 outputmakefile

prepare1: prepare2 include/linux/version.h include/linux/utsrelease.h \
                   include/asm include/config/auto.conf
        $(cmd_crmodverdir)

archprepare: prepare1 scripts_basic

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

# All the preparing..
prepare: prepare0
```

构建系统处理 prepare 及相关目标的目的是为了后面真正进入各子目录编译内核或模块做准备。

在这些目标的处理中，最重要的莫过于对 prepare0 目标的处理了。注意目标 prepare0 对应规则的命令，它们会调用 scripts/Makefile.build，并且在其中包含顶层目录中的 Kbuild 文件，其功能分别是由 arch/arm/kerel/asm-offset.c 文件生成 include/asm-arm/asm-offset.h 文件以及使用 scripts/checksyscalls.sh 来检查是否还有未实现的系统调用(检查时，以 i386 所实现的系统调用为比较依据)。

至于 prepare 相关处理的其他部分，限于本文的篇幅过于庞大，这里就先略去不讲了。有兴趣的朋友可以参与我们的 mail list 进行讨论。

至于 $(vmlinux-dires) 所依赖的另外一个目标 scripts。它就是为了在真正进入各子目录编译内核或者模块之前，在目录 scripts 中准备好若干工具。其规则如下：

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

回到我们对 $(vmlinux-dirs) 目标进行处理的命令上来。 命令 "(Q)(MAKE) $(build)=$@" 其实就是调用 scripts/Makefile.Build 文件，并依次在其中包含变量 $(vmlinux-dirs) 所对应各目录的 Kbuild/Makefile，最终在各目录中编译出不同的对象文件来(一系列的.o 文件和.a 文件)。这到底是如何实现的？我们先看命令 "(Q)(MAKE) $(build)=$@"，简化出来就是：

make -f scripts/Makefile.build obj=$@

由于上面这个命令中并没有指定要处理什么具体的目标，所以此 make 命令实际上是在处理 scripts/Makefile.build 中的默认目标：__build，我们列出具体的规则：

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

这条规则没有什么命令，但是却有很多依赖。我们说正是这些依赖指明了内核构建系统进入到各个子目录(由 vmlinux-dirs 中列出)后所要编译处理的各种目标。我们通过分析这些依赖来搞清楚这些目标有哪些。在此之前，我们观察到两个变量 KBUILD_BUILTIN 和 KBUILD_MODULES，它们是在顶层 Makefile 中就初始化好了并导出来的，其代码如下：

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

KBUILD_MODULES :=
KBUILD_BUILTIN := 1

#       If we have only "make modules", don't compile built-in objects.
#       When we're building modules with modversions, we need to consider
#       the built-in objects during the descend as well, in order to
#       make sure the checksums are up to date before we record them.

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

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

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

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

export KBUILD_MODULES KBUILD_BUILTIN
```

这两个变量用于记录在用内核构建系统进行 make 过程中是否处理基本内核(basic kernel,namely vmlinux)，以及是否处理包含在内核代码树中的内部模块(为了编译内部模块，你需要在配置的时候选择 m，而不是 y)。如果要处理基本内核，那变量 KBUILD_BUILTIN 被设置为 1；如果要编译内部模块，那变量 KBUILD_MODULES 被设置为 1。对于我们所举的例子："make ARCH=arm CROSS_COMPILE=arm-linux-"，由于既要处理基本内核又要处理内部模块(all 既依赖于 vmlinux,又依赖于 modules)，所以此两变量均取 1 的值。

好，回到我们前面处理 __build 的那条规则上面来，我们可以开始分析构建系统进入各子目录后，到底要处理哪些目标了，总共有这么几项：

- 由于有变量 lib-target 的定义：

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

所以构建系统有可能要编译出各子目录下的 lib.a。注意其要不要编译，是看在各子目录下的 Makefile 中有无"lib-*"之类变量的定义。

对于 lib-target 的处理规则，定义在 scripts/Makefile.build 中，列出如下：

```
#
# 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) rcs $@ $(lib-y)

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

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

上面的规则表明构建系统会使用归档工具将变量 lib-y 中列出的所有对象文件归档成 lib.a 文件。而在整个 Linux 内核代码树中，只有两类(个)目录下的 Kbuild/Makefile 包含有对 lib-y 的定义，一个是内河代码树下的 lib/目录，另外一个 arch/$(ARCH)/lib/目录。我们等会会举目录 arch/arm/lib/下的 Makefile 来说明，这里先看看构建系统在进行归档之前，会对变量 lib-y 做何种处理。在 scripts/Makefile.lib 中，有这样的代码：

```
# 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))
```

上面代码将列在 lib-y/lib-m 中，但同时又定义在 obj-y 中的文件过滤掉；并在 lib-y 所包含文件名前面加上目录名。好，我们拿 arch/arm/lib/Makefile 中的 lib-* 定义进行举例：

```
lib-y           := backtrace.o changebit.o csumipv6.o csumpartial.o   \
                   csumpartialcopy.o csumpartialcopyuser.o clearbit.o \
                   delay.o findbit.o memchr.o memcpy.o                \
                   memmove.o memset.o memzero.o setbit.o              \
                   strncpy_from_user.o strnlen_user.o                 \
                   strchr.o strrchr.o                                 \
                   testchangebit.o testclearbit.o testsetbit.o        \
                   ashldi3.o ashrdi3.o lshrdi3.o muldi3.o             \
                   ucmpdi2.o lib1funcs.o div64.o sha1.o               \
                   io-readsb.o io-writesb.o io-readsl.o io-writesl.o

mmu-y   := clear_user.o copy_page.o getuser.o putuser.o

....  # mmu-y 变量的其他定义

# using lib_ here won't override already available weak symbols
obj-$(CONFIG_UACCESS_WITH_MEMCPY) += uaccess_with_memcpy.o

lib-$(CONFIG_MMU) += $(mmu-y)

ifeq ($(CONFIG_CPU_32v3),y)
  lib-y += io-readsw-armv3.o io-writesw-armv3.o
else
  lib-y += io-readsw-armv4.o io-writesw-armv4.o
endif

lib-$(CONFIG_ARCH_RPC)          += ecard.o io-acorn.o floppydma.o
lib-$(CONFIG_ARCH_L7200)        += io-acorn.o
lib-$(CONFIG_ARCH_SHARK)        += io-shark.o
```

上面代码中，对 lib-y 进行很多 *.o 的赋值。构建系统会编译对应的同名 C 程序文件或同名汇编程序文件编译成对象文件，并将这些对象文件链接成 lib.a。

- 由于有变量 builtin-target 的定义：

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

所以构建系统有可能要编译出各个子目录下的 built-in.o。同样的，要不要真的编译，得看在各子目录下的 Makefile 中有无"obj-*"之类变量的定义。

built-in.o 文件和 lib.a 文件都会被连接到基本内核映像 vmlinux 中去。而和 lib.a 不同的是，built-in.o 文件会存在于内核代码树的大多数目录，所以其会依据内核代码树的组织结构构成一颗树。不像 lib.a ，其只存在于少数的几个目录中。我们先看看构建系统对 builtin-target 的处理，在 scripts/Makefile.build 中有：

```
#
# Rule to compile a set of .o files into one .o file
#
ifdef builtin-target
quiet_cmd_link_o_target = LD      $@
# If the list of objects to link is empty, just create an empty built-in.o
cmd_link_o_target = $(if $(strip $(obj-y)),\
                      $(LD) $(ld_flags) -r -o $@ $(filter $(obj-y), $^) \
                      $(cmd_secanalysis),\
                      rm -f $@; $(AR) rcs $@)

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

targets += $(builtin-target)
endif # builtin-target
```

由上面的对 cmd_link_o_target 的定义可知道， builtin-target 是由变量 obj-y 所指定的一系列对象文件所连接而成的。而在连接之前，构建系统会在文件 scripts/Makefile.lib 中对变量 obj-y 做一些处理：

```
obj-y             := $(patsubst %/, %/built-in.o, $(obj-y))
obj-y             := $(addprefix $(obj)/,$(obj-y))
```

我们先说一下，在内核各子目录的 Kbuild/Makefile 中，通常会给变量 obj-y 赋予三种类型的值：

1) 单一对象文件；

   这种对象文件由一个单一的 C 程序文件或者汇编程序文件编译而来。

2) 复合对象文件；

   这种对象文件由多个对象文件连接而成，这些对象文件均由 C 程序文件或汇编程序文件编译而来。

3) 包含在该子目录系统下的二级子目录：

   在内核构建系统中，各个目录下的 Kbuild/Makefile 只负责本目录下源程序的编译，本目录下二级子目录中的代码由二级子目录中的 Kbuild/Makefile 负责。但是构建系统需要知道要进去到哪些二级子目录下进行编译，所以我们需要在上层 Kbuild/Makefile 中将这些二级子目录的名称赋值给变量 obj-y。

上面处理代码中的第一行就是将变量 obj-y 中的二级子目录名称后面添加 built-in.o 文件名。前面已经看到，构建系统正是用 cmd_link_o_target 将这些 built-in.o ，连同 obj-y 中的其他单一对象文件以及复合对象文件，连接成一个新的 built-in.o 对象文件。接着用同样的方法编译出更上层目录中的 built-in.o，循环递规直到编译出内核代码树顶层目录各子目录中的 built-in.o ，最后再行基本内核映像 vmlinux 的链接。上面处理代码中的第二行是给各对象文件增加目录前缀。

变量 obj-y 中的单一对象文件编译比较简单，我们略去不说。这里我们讨论一下复合对象文件是如何连接产生的。我们能在 scripts/Makefile.build 找到这样的代码：

```
link_multi_deps =                       \
$(filter $(addprefix $(obj)/,           \
$($(subst $(obj)/,,$(@:.o=-objs)))      \
$($(subst $(obj)/,,$(@:.o=-y)))), $^)

quiet_cmd_link_multi-y = LD      $@
cmd_link_multi-y = $(LD) $(ld_flags) -r -o $@ $(link_multi_deps) $(cmd_secanalysis)
....
$(multi-used-y) : %.o: $(multi-objs-y) FORCE
        $(call if_changed,link_multi-y)
```

其中变量 multi-used-y 和 multi-objs-y 又定义在 scripts/Makefile.lib 中：

```
multi-used-y := $(sort $(foreach m,$(obj-y), $(if $(strip $($(m:.o=-objs)) $($(m:.o=-y))), $(m))))
multi-objs-y := $(foreach m, $(multi-used-y), $($(m:.o=-objs)) $($(m:.o=-y)))
```

构建系统是如何知道去链接哪些对象文件，从而构成复合对象文件的。我们说通过在 Kbuild/Makefile 中用变量 *-objs 或者 *-y 来指定的。举个例子，假如我们在一个目录的 Kbuild/Makefile 给 obj-y 这样赋了值：

obj-y = ASingle.o BComposite.o CSubDir1/

那经过前述讨论过的对 obj-y 的处理，构建系统会将 obj-y 处理成：
obj-y = PrefixDir/ASingle.o PrefixDir/BComposite.o PrefixDir/CSubDir1/built-in.o

假如我们在同一个 Kbuild/Makefile 中又像下面那样设置了变量 BComposite-objs 或者 BComposite-y，那构建系统就会认为 BComposite.o 是一个由本目录下 A.o, B.o, C.o, D.o 四个对象文件复合链接而来。

```
BComposite-objs = A.o B.o C.o
BComposite-$(CONFIG-XXX) = D.o
```

当然，复合文件中包含 D.o 的前提是 COFIG-XXX 配置选项被设置为 y。

知道这些，理解前面的命令就比较容易了。前面设置变量 multi-used-y 的时候，就是依次检查 obj-y 中列出的变量(假设每个为 mexample.o)，看是否存在名为 mexample-objs 或者 mexample-y 的变量定义，如果有存在，那就认为 mexample 为复合对象。接着，将构成这些复合对象的所有单一对象文件，也就是变量 XXX-objs 和 XXX-y 的值赋给变量 multi-objs-y，最后动用一条静态匹配规则将这些单一对象文件链接成复合对象文件，请注意上面变量 link_multi_deps 的定义。

- 对于变量 extra-y ，它是用来指定在该目录下有没有额外的目标需要编译创建。如果有的话，构建系统也要编译出来。

一个目录下需要编译的对象文件通常都列在 obj-y, obj-m 等变量中，但也存在其他一些需要编译的对象文件，它们并不编译到 built-in.o 中，也不参与构成内部模块，这样的对象文件我们就用 extra-y 指定出来。一个比较明显的例子就是在目录 arch/arm/kernel 下的 Makefile 内这样的代码：

```
.... // obj-y/obj-m 的赋值
head-y                  := head$(MMUEXT).o
obj-$(CONFIG_DEBUG_LL)  += debug.o

extra-y := $(head-y) init_task.o vmlinux.lds
```

上面代码中除了对 obj-y/obj-m 的赋值外，还把 head.o init_task.o vmlinux.lds 等赋给 extra-y。变量 MMUEXT 像下面这样定义在文件 arch/arm/Makefile 中，对于 s3c2410_defconfig 配置来说，配置选项 CONFIG_MMU 被设置为 y，所以 MMUEXT 被定义为空。head.o 和 init_task.o 都是需要直接放在基本内核前面，而非被构建系统链接进 built-in.o 中去的，所以这里将它们列在 extra-y 下面。另外，vmlinux.lds 也是需要被额外预处理的，它是基本内核的链接器脚本，这个我们已经在前面说过。

```
# defines filename extension depending memory manement type.
ifeq ($(CONFIG_MMU),)
MMUEXT          := -nommu
endif
```

- 对于变量 obj-m ，它是用来指定在内核代码树中要编译成为内核内部模块的对象文件。这样的对象文件可能是从一个源文件编译而来，也可能是多个源文件编译成多个对象文件后链接得到。

和变量 obj-y 一样，变量 obj-m 中所包含的东西也有三种：单一对象文件，复合对象文件和子目录。子目录的处理我们统一在下面的第 f 项中说明。这里先看看构建系统是如何处理变量 obj-m 的。在文件 scripts/Makefile.build 中有：

```
# When an object is listed to be built compiled-in and modular,
# only build the compiled-in version

obj-m := $(filter-out $(obj-y),$(obj-m))
....
obj-m           := $(filter-out %/, $(obj-m))
....
multi-used-m := $(sort $(foreach m,$(obj-m), $(if $(strip $($(m:.o=-objs)) $($(m:.o=-y))), $(m))))
....
single-used-m := $(sort $(filter-out $(multi-used-m),$(obj-m)))
....
multi-objs-m := $(foreach m, $(multi-used-m), $($(m:.o=-objs)) $($(m:.o=-y)))
...
obj-m           := $(addprefix $(obj)/,$(obj-m))
single-used-m   := $(addprefix $(obj)/,$(single-used-m))
multi-used-m    := $(addprefix $(obj)/,$(multi-used-m))
multi-objs-m    := $(addprefix $(obj)/,$(multi-objs-m))
```

处理过的变量 obj-m 中过滤掉了子目录部分，只包含带所在目录的单一对象文件以及复合对象文件。同时，请注意构建系统将复合对象文件列在变量 multi-used-m，而将单一对象文件列在 single-used-m 中。变量 multi-objs-m 中列出了那些构成符合对象文件的所有单个成员。我们继续看看在构建系统如何处理 single-used-m 和 multi-used-m 所对应的目标，在 scripts/Makefile.build 中：

```
# Single-part modules are special since we need to mark them in $(MODVERDIR)

$(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)
....
quiet_cmd_link_multi-m = LD [M]  $@
cmd_link_multi-m = $(cmd_link_multi-y)
....
$(multi-used-m) : %.o: $(multi-objs-m) FORCE
        $(call if_changed,link_multi-m)
        @{ echo $(@:.o=.ko); echo $(link_multi_deps); } > $(MODVERDIR)/$(@F:.o=.mod)
```

如你所见，和对 multi-used-y 的处理不同，构建系统在处理 single-used-m 及 multi-used-m 的时候，除了要编译出对应的对象文件之外，它还使用了这样的命令：

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

其中变量 MODVERDIR 定义在顶层 Makefile 中：

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

这个定义先看编译的是不是外部模块(区别于内部模块，内部模块的代码通常都已经合并在内核代码树里去了。而外部模块则是我们开发设备驱动程序的最常见模式，外部模块的代码放在内核树以外的独立目录中。)，如果是就在外部模块所在代码目录中定义隐藏目录 .tmp_versions，否则在内核代码树中定义。

所以，假如我们要创建的模块由多个对象文件(MPa.o/Mpb.o/Mpc.o 等三个文件)构成，且其名称为 MyModule.ko，那前面额外使用的那条命令的功能就是在这个隐藏目录中新建名为 MyModule.mod 的文件，并在此文件中写入这样的内容：

```
Directories/MyModule.ko
Directories/MPa.o Directories/MPb.o Directories/MPc.o
```

对于内部模块和外部模块，虽然它们略有差异，但是它们的产生都要经历两个步骤：1，生成构成模块的.o 对象文件以及.mod 文件。2，调用 scripts/Makefile.post 由.o 和.mod 生成.ko 模块文件。这里讨论的正是第一个步骤，我们在后面讨论步骤二。

- 对于变量 modorder-target，其指代的是本目录下的文件 modules.order。该文件记录的是构建系统构建内部模块的先后次序。内核在加载内部模块的时候，基本上会按照这个次序来加载。

我们先看看 modorder-target 目标在 scripts/Makefile.build 中是如何处理的：

```
#
# Rule to create modules.order file
#
# Create commands to either record .ko file or cat modules.order from
# a subdirectory
modorder-cmds =                                         \
        $(foreach m, $(modorder),                       \
                $(if $(filter %/modules.order, $m),     \
                        cat $m;, echo kernel/$m;))

$(modorder-target): $(subdir-ym) FORCE
        $(Q)(cat /dev/null; $(modorder-cmds)) > $@
```

从上面代码中可以看出，本目录下的文件 modules.order 中的内容其实是由执行变量 modorder-cmds 所代表的命令而造成的输出结果。modorder-cmds 变量中所用到的 modorder 定义在 scripts/Makefile.lib 中：

```
# Determine modorder.
# Unfortunately, we don't have information about ordering between -y
# and -m subdirs.  Just put -y's first.
modorder        := $(patsubst %/,%/modules.order, $(filter %/, $(obj-y)) $(obj-m:.o=.ko))
....
modorder        := $(addprefix $(obj)/,$(modorder))
```

这个定义表示，包含在 modorder 变量中的，要么是 subdir/modules.order 的形式，要么就是 builtin.ko 一样的模块名称。注意，前者中的 subdir 是包含在 obj-y 中的目录名称。所以，本目录中文件 modules.order 的内容就是各子目录中 modules.order 的内容加上本目录中要编译出的内部模块的名称。最终，modules.order 文件中就列出了各子目录及本目录

中所包含的 .ko 的列表，这个列表所隐含的次序就是构建系统构建这些模块的前后次序。

后面我们在讨论处理内部模块的时候，我们会看到构建系统会将这些 modules.order 的内容合并到内核顶层目录下，形成一个完整的 modules.order，也就是所有内部模块的列表。关于这个列表的作用，我们在后面讨论内部模块处理的时候再行叙述。

- 对于变量 subdir-ym，其指代的是本目录下的一系列子目录。内核构建系统在处理本目录之前，需要先行完成对本目录下各子目录的处理。

我们前面在讨论 obj-y 的时候，已经知道了如何让构建系统明白要进去哪些二级子目录去编译出 built-in.o，也已经知道了构建系统会用什么样的命令去编译 built-in.o，那么构建系统是如何进去到这些二级子目录中去进行编译的呢？另外在讨论 obj-m 的时候，也知道构建系统将包含在其中的二级子目录过滤掉了，那构建系统又是如何进去到这些二级子目录中去编译出对象文件来生成内核模块的呢？在内核构建系统中，有一个专门的名词：Descending，来指示进去到二级子目录中去编译出 built-in.o 或者 *.o。带着这两个问题，我们先在 script/Makefile.lib 中看看 subdir-ym 的定义：

```
__subdir-y      := $(patsubst %/,%,$(filter %/, $(obj-y)))
subdir-y        += $(__subdir-y)
__subdir-m      := $(patsubst %/,%,$(filter %/, $(obj-m)))
subdir-m        += $(__subdir-m)
....
# Subdirectories we need to descend into

subdir-ym       := $(sort $(subdir-y) $(subdir-m))
....
subdir-ym       := $(addprefix $(obj)/,$(subdir-ym))
```

从上面的代码中可以看到，subdir-y/subdir-m/subdir-ym 等变量包含的就是那些 obj-y/obj-m 中包含的二级子目录名称了。构建系统对 subdir-ym 的处理代码包含在 scripts/Makefile.build 中：

```
# Descending
# --------------------------------------------------------------------------

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

这就是实现 Descending 的关键所在。构建系统调用 scripts/Makefile.build，并在其中包含各个二级子目录的 Kbuild/Makefile，最后 make 出其中定义的缺省目标。Descending 之后，二级子目录中的 built-in.o 或者生成*.ko 的 *.o 也都编译出来了。

- 对于变量 always，其指代的是在各子目录下总是需要构建的目标。

当你需要让构建系统进入子目录后总是处理某些目标的时候，你就可以将它们赋给变量 always 。一个使用 always 的好例子就是在编译 host program 的时候，如果你没有显式的将这个 host program 作为依赖放在一条规则里面，那你就需要将它赋给 always，以让构建系统帮你编译出它来。

以上，讨论了构建系统处理 scripts/Makefile.build 中默认目标__build 的所有依赖。由于__build 对应规则没有命令，所以当这些依赖处理完成后，__build 目标也就处理完毕。回到处理 变量 vmlinux-dirs 的规则上来，就意味着该变量所对应的目录中，各种需要产生的 built-in.o，lib.a，以及用于产生*.ko 模块的各种*.o 和*.mod 也都产生了，在这之后，构建系统就可做下一步的动作了。

我们前面说过，$(vmlinux-lds)、$(vmlinux-init)和$(vmlinux-main)等三个目标依赖于 $(vmlinux-dirs)。所以有的同学可能会觉得接下来要处理这三个目标了。但是我们说这三个目标所指示的 vmlinux.lds，顶层目录各子目录中的 built-in.o/lib.a 等也都已构建完毕(由 vmlinux-init 和 vmlinux-main 指示)，架构相关目录中的对象文件也已经编译完成(由 vmlinux-init 所指示)。

回到处理 vmlinux 的那条规则上面来，在处理好 $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main)等目标后，构建系统接下来要处理的就是 vmlinux.o 和 $(kallsyms.o) 了。

内核构建系统之所以要在链接 vmlinux 之前，去链接出 vmlinux.o。其原因并不是要将 vmlinux.o 链接进 vmlinux，而是要在链接 vmlinux.o 的过程中做完两个动作：

a) elf section 是否 mis-match 的检查；
b) 生成内核导出符号文件 Module.symvers(Symbol version dump 文件)；

其中第一个动作，由于明白它需要很多 elf 文件格式的知识，而本文重在解释内核构建系统的原理与实现，所以要在这里详细讲解就显得不妥，等以后有机会了再写文章来讨论。有兴趣的同学，可以先行查看 scripts/mod/modpost.c 中的代码。而第二个动作中提到的 Modules.symvers 文件，其中包含内核及内部模块所导出的各种符号及其相关 CRC 校验值。构建系统会在编译 vmlinux.o 的过程中将基本内核(vmlinux)所导除的符号记录在这个文件中，在后面讨论内部模块的第二步处理的时候，我们会发现它也会将各内部模块所导出的符号记录在这个文件中。

这里提到的所谓符号，你可以先将其理解成是基本内核或者内部模块导出来的，供其他人(通常是驱动程序)使用的函数。等你以后开发驱动程序进行调试的时候，你会发现有这么一个 dump 文件会有多么方便。构建系统在编译外部模块的时候也要使用这个文件。

为了不脱离本文主题，我在下面讨论 vmlinu.o 编译的过程中，特意将上面两个动作中和内核构建无关的部分略去，以后我们会专门写另外的文章来讨论。

对于 vmlinux.o 目标，在 顶层 Makefile 中，有这样的规则定义：

```
modpost-init := $(filter-out init/built-in.o, $(vmlinux-init))
vmlinux.o: $(modpost-init) $(vmlinux-main) FORCE
        $(call if_changed_rule,vmlinux-modpost)
```

该规则命令中的 if_changed_rule，我们前面已经介绍过。这里构建系统要调用 rule_vmlinux-modpost 变量所定义的命令。变量 rule_vmlinux-modpost 定义在顶层 Makefile 文件中:

```
# Do modpost on a prelinked vmlinux. The finally linked vmlinux has
# relevant sections renamed as per the linker script.
quiet_cmd_vmlinux-modpost = LD      $@
      cmd_vmlinux-modpost = $(LD) $(LDFLAGS) -r -o $@                          \
         $(vmlinux-init) --start-group $(vmlinux-main) --end-group             \
         $(filter-out $(vmlinux-init) $(vmlinux-main) FORCE ,$^)
define rule_vmlinux-modpost
        :
        +$(call cmd,vmlinux-modpost)
        $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modpost $@
        $(Q)echo 'cmd_$@ := $(cmd_vmlinux-modpost)' > $(dot-target).cmd
endef
```

变量 rule_vmlinux-modpost 中定义的命令中，一开始就是调用 cmd_vmlinux-modpost 变量所定义的命令来链接出 vmlinux.o 目标。紧接着，构建系统就用 scripts/Makefile.modpost 来调用 make。最后，它将构建 vmlinux.o 目标的命令保存到 .vmlinux.o.cmd 文件中。注意看第二条 make 命令中的目标为 $@，也就是说要 make vmlinux.o。我们看 Makefile.modpost 中对 vmlinux.o 的定义：

```
quiet_cmd_kernel-mod = MODPOST $@
      cmd_kernel-mod = $(modpost) $@

vmlinux.o: FORCE
        @rm -fr $(kernelmarkersfile)
        $(call cmd,kernel-mod)
```

而 modpost 变量被定义成这样：

```
modpost = scripts/mod/modpost                     \
 $(if $(CONFIG_MODVERSIONS),-m)                   \
 $(if $(CONFIG_MODULE_SRCVERSION_ALL),-a,)        \
 $(if $(KBUILD_EXTMOD),-i,-o) $(kernelsymfile)    \
 $(if $(KBUILD_EXTMOD),-I $(modulesymfile))       \
 $(if $(KBUILD_EXTRA_SYMBOLS), $(patsubst %, -e %,$(KBUILD_EXTRA_SYMBOLS))) \
 $(if $(KBUILD_EXTMOD),-o $(modulesymfile))       \
 $(if $(CONFIG_DEBUG_SECTION_MISMATCH),,-S)       \
 $(if $(CONFIG_MARKERS),-K $(kernelmarkersfile)) \
 $(if $(CONFIG_MARKERS),-M $(markersfile))        \
 $(if $(KBUILD_EXTMOD)$(KBUILD_MODPOST_WARN),-w) \
 $(if $(cross_build),-c)
```

可以看出，其中包含有很多条件的判断。由于编译 vmlinux.o 时，CONFIG_MODVERSIONS，CONFIG_MODULE_SRCVERSION_ALL，KBUILD_EXTMOD，KBUILD_EXTRA_SYMBOLS，CONFIG_MARKERS，CONFIG_DEBUG_SECTION_MISMATCH 等等都没定义，而我们做的是交叉编译，也就是 cross_build 则被赋值 1。所以在 scripts/Makefile.modpost 中，处理 vmlinux.o 的命令就是：

scripts/mod/modpost -o /home/yihect/linux-2.6.31/Module.symvers -S -c vmlinux.o

这个命令做的就是用工具 modpost 来解析 vmlinux.o 对象文件，并将基本内核导出的所有符号都记录到文件 Module.symvers 中去。当然，这个命令附带完成的，还有前面所说到的检查 sections 是否 mis match 的工作。你可以先浏览看看 scripts/mod/modpost.c 中的代码。打开文件 Module.symvers，你会发现针对每个内核导出的符号(symbol)，都会有一行，其格式和举例如下：

```
The syntax of the Module.symvers file is:
        <CRC>       <Symbol>          <module>              <Symbol_exported_macro>
Sample:
        0x2d036834  scsi_remove_host   drivers/scsi/scsi_mod  EXPORT_SYMBOL
```

等一下，你此时打开该文件后看到的符号 CRC 值都是 0x00000000，那是因为你在配置的时并没有设置 CONFIG_MODVERSIONS。一旦设置过这个配置选项，就意味着打开了内核的 Module versioning 功能。Module versioning 功能应用在我们使用模块的场合。如果 Module versioning 功能被打开的话，它会以每个导出符号的 C 原型声明作为输入，计算出对应的 CRC 校验值，保存在文件 Module.symvers 中。如此一来，内核在后面要加载使用模块的时候，会两相比较模块中的 CRC 值和保存下来的 CRC 值，如果发现不相等，内核就拒绝加载这个模块。

在编译完 vmlinux.o 后，在链接 vmlinux 之前，构建系统还要处理目标 $(kallsyms.o)。这里先说说所谓 kallsyms 是做什么用的。

在 2.6 版的内核中，为了更方便的调试内核代码，开发者考虑将内核代码中所有函数以及所有非栈变量的地址抽取出来，形成是一个简单的数据块(data blob)，并将此链接进 vmlinux 中去。如此，在需要的时候，内核就可以将符号地址信息以及符号名称都显示出来，方便开发者对内核代码的调试。完成这一地址抽取+数据快组织封装功能的相关子系统就称之为 kallsyms。反之，如果没有 kallsyms 的帮助，内核只能将十六进制的符号地址呈现给外界，因为它能理解的只有符号地址，而并不包括人类可读的符号名称。

经常使用 Windows 的人都知道所谓的蓝屏是怎么回事，那是系统出了致命问题，而不能继续运行下去而 show 出的一个蓝色屏。那么对 Linux 来说，也会有致命错误的出现，如果这种错误使得 Linux 不可继续运行，那么 Linux 就会显示类似下面这样的屏幕内容(以运行在 PowerPC 架构下的 Linux 来举例，其他架构也差不多)以 dump 出出现错误那时刻的系统状态。这种现象就是所谓的 Oops，也就 Linux 下的蓝屏。

```
$ modprobe loop
Oops: kernel access of bad area, sig: 11 [#1]
NIP: C000D058 LR: C0085650 SP: C7787E80 REGS: c7787dd0 TRAP: 0300   Not tainted
MSR: 00009032 EE: 1 PR: 0 FP: 0 ME: 1 IR/DR: 11
DAR: 00000000, DSISR: 22000000
TASK = c7d187b0[323] 'modprobe' THREAD: c7786000
Last syscall: 128
GPR00: 0000006C C7787E80 C7D187B0 00000000 C7CD25CC FFFFFFFF 00000000 80808081
GPR08: 00000001 C034AD80 C036D41C C034AD80 C0335AB0 1001E3C0 00000000 00000000
GPR16: 00000000 00000000 00000000 100170D8 100013E0 C9040000 C903DFD8 C9040000
GPR24: 00000000 C9040000 C9040000 00000940 C778A000 C7CD25C0 C7CD25C0 C7CD25CC
NIP [c000d058] strcpy+0x10/0x1c
LR [c0085650] register_disk+0xec/0xf0
Call trace:
 [c00e170c] add_disk+0x58/0x74
 [c90061e0] loop_init+0x1e0/0x430 [loop]
 [c002fc90] sys_init_module+0x1f4/0x2e0
 [c00040a0] ret_from_syscall+0x0/0x44
Segmentation fault
```

上面出现的 Oops 消息中，显示了出错时的 CPU 各寄存器的值，以及以 "Call trace:" 一行开始的 C 函数调用栈。注意其中除了显示在中括号内的地址，还显示了函数的名称，这就是受助于 kallsyms 的结果，否则它只能显示地址信息。要在一个内核中启用 kallsyms 功能，你必须设置 CONFIG_KALLSYMS 选项为 y；如果你要在 kallsyms 中包含全部符号信息，必须设置 CONFIG_KALLSYMS_ALL 为 y。

我们既然明白了 kallsyms 的大致作用，那么内核是如何抽取函数和非堆栈变量的地址，以及如何将它们连同对应的名称保存到一个 data blog 中去以便最后链接到 vmlinux 中

去的。抽取函数和非堆栈变量的地址对内核构建系统来说比较简单，只需要使用 nm 工具即可，我们后面会看到。内核使用这个工具的输出作为输入，来调用主机工具程序 scripts/kallsyms，从而生成一个汇编程序文件。在这个汇编程序文件的数据段中，定义有若干个标号(你可以将其理解成用来存储数据的 C 数组或数据结构)。在这些标号下面就存储有前面取到的函数/变量地址和对应的名称。所以，很自然的，前面所谓的 data blog 其实就是这个汇编文件编译后的对象文件。构建系统将其链接到 vmlinux 基本内核中。

上面是一般的原理。在内核构建系统中，真正得到正确的汇编文件是一个两遍的过程。第一遍得到的名为 ./.tmp_kallsyms1.S，第二遍为 ./.tmp_kallsyms2.S。两个文件格式完全一样，不同的时其中包含的函数/变量地址和名称等等。我们先来看看这个汇编文件的格式，大致是这样的(我这里重在列出那些标号，具体数据都省略掉)：

```
#include <asm/types.h>
#if BITS_PER_LONG == 64
#define PTR .quad
#define ALGN .align 8
#else
#define PTR .long
#define ALGN .align 4
#endif
        .section .rodata, "a"
.globl kallsyms_addresses
        ALGN
kallsyms_addresses:
        PTR     _text - 0x27000
        ......
        PTR     _text + 0x332a38
        PTR     0xc0362000

.globl kallsyms_num_syms
        ALGN
kallsyms_num_syms:
        PTR     24163

.globl kallsyms_names
        ALGN
kallsyms_names:
        .byte 0x06, 0x25, 0x3d, 0x62, 0x65, 0x67, 0xf4
        .......
        .byte 0x08, 0x52, 0xff, 0x1d, 0xcb, 0xff, 0x5f, 0x1f, 0x6d
        .byte 0x05, 0x41, 0x5f, 0x65, 0xbe, 0x09

.globl kallsyms_markers
        ALGN
kallsyms_markers:
        PTR     0
        PTR     2394
        ......
        PTR     257959
        PTR     260879

.globl kallsyms_token_table
        ALGN
kallsyms_token_table:
        .asciz  "3_"
        ......
        .asciz  "ta"
        .asciz  "__"

.globl kallsyms_token_index
        ALGN
kallsyms_token_index:
        .short  0
        .......
```

从上面的汇编代码中可以看出，有六个汇编标号(也是全局变量)的定义，分别是 kallsyms_addresses ， kallsyms_num_syms ， kallsyms_names ， kallsyms_markers ， kallsyms_token_table 和 kallsyms_token_index。这些变量的具体用法，我们这里先不予关心，因为已经超出本文的讨论范围。你目前只需要知道 Linux 内核通过它们来保存函数/变量地

址和对应名称之间的 mapping 即可。因为在这里定义的时候，它们都是全局的。所以在 C 代码里面，只需要做一下 extern 声明就可以直接引用它们，这些 extern 声明放在文件 kernel/kallsyms.c 中，具体代码如下：

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

/*
 * Tell the compiler that the count isn't in the small data section if the arch
 * has one (eg: FRV).
 */
extern const unsigned long kallsyms_num_syms
__attribute__((weak, section(".rodata")));

extern const u8 kallsyms_token_table[] __attribute__((weak));
extern const u16 kallsyms_token_index[] __attribute__((weak));

extern const unsigned long kallsyms_markers[] __attribute__((weak));
```

这里需要引起注意的是，上面声明中都使用了 __attribute__((weak))。这个作何解释，我博客的其他文章中已有所介绍，这里就不再叙述。好，我们还是来看看构建系统是如何处理 kallsyms 的，我们先看看变量 kallsyms.o 的定义：

```
ifdef CONFIG_KALLSYMS_EXTRA_PASS
last_kallsyms := 3
else
last_kallsyms := 2
endif

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

默认情况下，CONFIG_KALLSYMS_EXTRA_PASS 是不会被配置的，因此 last_kallsyms 被默认设为 2，给它赋值为 3 只是为了更方便调试 kallsyms 系统代码来的。所以 kallsyms.o 变量指代的就是.tmp_kallsyms2.o。好，知道 kallsyms.o 的定义后，我们由此出发构造一个目标依赖关系链表(a-->b，表示目标 a 依赖于目标 b)，你可以据此来理解：

.tmp_kallsyms2.o --> .tmp_kallsyms2.S --> .tmp_vmlinux2 --> .tmp_kallsyms1.o --> .tmp_kallsyms1.S --> .tmp_vmlinux1

由这个依赖链表出发，可以很明显的看出是一个两遍的过程。只不过前后两遍的过程顺序是逆着依赖关系来的，右边为第一遍，左边为第二遍。每一遍都是以先生成一个内核映像(.tmp_vmlinux*)出发，用 nm/kallsyms 来生成汇编程序文件(.tmp_kallsyms*.S)，并最终以编译此汇编文件产生对象文件为结束。

好了，知道这些后，我们再列出顶层 Makefile 中的代码来就比较好懂了：

```
# Update vmlinux version before link
# Use + in front of this rule to silent warning about make -j1
# 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

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

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

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

# .tmp_vmlinux1 must be complete except kallsyms, so update vmlinux version
.tmp_vmlinux1: $(vmlinux-lds) $(vmlinux-all) FORCE
        $(call if_changed_rule,ksym_ld)

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

.tmp_vmlinux3: $(vmlinux-lds) $(vmlinux-all) .tmp_kallsyms2.o FORCE
        $(call if_changed,vmlinux__)
```

注意，处理目标 .tmp_vmlinux1 的命令并非直接调用 cmd_vmlinux__ ，而是调用了 rule_ksym_ld。在 rule_ksym_ld 中，它先用 cmd_vmlinux_version 去更新基本内核的链接次数，也就是 init/version.o 中的版本号，具体代码为：

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

在第一遍生成的内核基本映像 .tmp_vmlinux1 中，实际上已经有对上面提到的六个 kallsyms_*变量的的引用，只不过那是 weak 链接，意味着在链接时这些变量即使没有没有定义也没有关系。.tmp_vmlinux1 生成后，就可以生成 .tmp_kallsyms1.S 了，所用的命令为： cmd_kallsyms 。假设定义了 CONFIG_KALLSYMS_ALL，所以简化一下，生成 .tmp_kallsyms1.S 的命令就是：

arm-linux-nm -n .tmp_vmlinux1 | scripts/kallsyms --all-symbols > .tmp_kallsyms1.S

生成 .tmp_kallsyms1.S 后，内核中所有函数和非堆栈变量的地址及名称也都已经保存

在汇编程序中了(也就是上面汇编程序中省略掉的部分)。将这个汇编文件编译成对象文件后，勾建系统就着手进行第二个阶段，开始链接第二个基本内核映像 .tmp_vmlinux2 了。注意和 .tmp_vmlinux1 不同的是，.tmp_vmlinux2 将 .tmp_kallsyms1.o 也链接进去了。

注意，链接成功 .tmp_vmlinux2 后，其中包含的部分函数和部分非堆栈变量的地址就发生了变化。为什么？很简单，在一个排好的队伍中间插进去几个人，那后面原有那些人的序号就会因增加不同数目而发生改变。这个时候，这些新地址与记录在 .tm_kallsyms1.o 中的对应地址就不一样。那么以哪个为准？自然是这些新地址，别忘了，它们是因为链接进 kallsyms 而发生改变的，我们就是要链接在一起的效果。

既然发生了地址改变，我们就必须想办法重新生成一次汇编程序 .tmp_kallsyms2.S。这个汇编程序和前面的那个 .tmp_kallsyms1.S 相比。在文件尺寸上没有差别，所不同的只是部分地址罢了。所以倘若拿新生成的 .tmp_kallsyms2.S 编译后的对象文件.tmp_kallsyms2.o 去编译一个新的基本内核，那这个基本内核中所具有的函数 / 变量地址就将不会再发生变化。所以结论就是，这个对象文件 .tmp_kallsyms2.o 就是我们最后要得到的 data blog，即 $(kallsyms.o) 目标。上面构建系统的代码中也准备了 .tmp_kallsyms3.S，.tmp_kallsyms3.o 和.tmp_vmlinux3 等等，但实际上，这是为了调试 kallsyms 系统代码用的。如果真的用到了这第三遍，那就证明 scripts/kallsyms.c 文件内的代码出问题了。

回到处理 vmlinux 的规则上面来。至此，目标 vmlinux 的所有依赖都处理完毕了，接下来构建系统就会执行该规则内如下的命令：

```
ifdef CONFIG_HEADERS_CHECK
        $(Q)$(MAKE) -f $(srctree)/Makefile headers_check
endif
ifdef CONFIG_SAMPLES
        $(Q)$(MAKE) $(build)=samples
endif
ifdef CONFIG_BUILD_DOCSRC
        $(Q)$(MAKE) $(build)=Documentation
endif
        $(call vmlinux-modpost)
        $(call if_changed_rule,vmlinux__)
        $(Q)rm -f .old_version
```

对于 s3c2410_defconfig 的默认配置来说，CONFIG_HEADERS_CHECK/CONFIG_SAMPLES/CONFIG_BUILD_DOCSRC 等都没有设置，所以这些命令中，最重要的就是 $(call if_changed_rule,vmlinux__) 一句了。这之前的 $(call vmlinux-modpost)，因为我始终都找不到变量 vmlinux-modpost 的定义，所以我认为 $(call vmlinux-modpost) 是条多余的没用的命令。在内核代码里，这样的多余现象是时常有的。在实际查找过程中，我甚至还像下面这样插入一条 echo 测试过，结果显示 vmlinux-modpost 没有被定义。

```
$(call vmlinux-modpost)
echo 'vmlinux-modpost := $(vmlinux-modpost)' > vmlinux-modpost.cmd
$(call if_changed_rule,vmlinux__)
$(Q)rm -f .old_version
```

命令 $(call if_changed_rule,vmlinux__) 会调用 rule_vmlinux__ 变量所定义的命令，在

顶层 Makefile 中找到 rule_vmlinux__ 的定义：

```
# Generate System.map
quiet_cmd_sysmap = SYSMAP
      cmd_sysmap = $(CONFIG_SHELL) $(srctree)/scripts/mksysmap

# Link of vmlinux
# If CONFIG_KALLSYMS is set .version is already updated
# Generate System.map and verify that the content is consistent
# Use + in front of the vmlinux_version rule to silent warning with make -j2
# First command is ':' to allow us to use + in front of the rule
define rule_vmlinux__
        :
        $(if $(CONFIG_KALLSYMS),,+$(call cmd,vmlinux_version))

        $(call cmd,vmlinux__)
        $(Q)echo 'cmd_$@ := $(cmd_vmlinux__)' > $(@D)/.$(@F).cmd

        $(Q)$(if $($(quiet)cmd_sysmap),                                      \
          echo '  $($(quiet)cmd_sysmap)  System.map' &&)                     \
        $(cmd_sysmap) $@ System.map;                                         \
        if [ $$? -ne 0 ]; then                                               \
                rm -f $@;                                                    \
                /bin/false;                                                  \
        fi;
        $(verify_kallsyms)
endef
```

在 rule_vmlinux__ 变量一开始，构建系统会检查是否有定义过 CONFIG_KALLSYMS，如果没有定义过，它就使用 cmd_vmlinux_version 来递增链接版本。还记得么，如果定义过 CONFIG_KALLSYS，它又是在哪里递增版本的？rule_vmlinux__ 接下来才会用 cmd_vmlinux__ 去把 vmlinux 链接出来。在这里，我们倒是可以将链接时所使用的命令贴出来看一下：

```
arm-linux-ld -EL  -p --no-undefined -X --build-id -o vmlinux -T arch/arm/kernel/vmlinux.lds
  arch/arm/kernel/head.o arch/arm/kernel/init_task.o  init/built-in.o --start-group
  usr/built-in.o  arch/arm/kernel/built-in.o  arch/arm/mm/built-in.o  arch/arm/common/built-in.o
  arch/arm/mach-s3c2410/built-in.o arch/arm/mach-s3c2400/built-in.o  arch/arm/mach-s3c2412/built-in.o
  arch/arm/mach-s3c2440/built-in.o  arch/arm/mach-s3c2442/built-in.o  arch/arm/mach-s3c2443/built-in.o
  arch/arm/plat-s3c24xx/built-in.o  arch/arm/plat-s3c/built-in.o  arch/arm/nwfpe/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  arch/arm/lib/lib.a  lib/lib.a
  arch/arm/lib/built-in.o  lib/built-in.o  drivers/built-in.o  sound/built-in.o
  firmware/built-in.o  net/built-in.o --end-group .tmp_kallsyms2.o
```

从上面链接 vmlinux 的命令可以看出，其使用的链接脚本是 .../arch/arm/kernel/vmlinux.lds。关于链接脚本的相关知识，我们会另外写一篇文章来讨论，我们这里不予详细说明。注意上面的链接将把 .../arch/arm/kernel/head.o 放在映像文件 vmlinux 的最前面，这是由链接器脚本所规定的，后续其他文章中的分析可能会告诉你 head.o 正是整个 Linux 开始的地方。

从 rule_vmlinux__ 的定义看，接下来，它使用 cmd_sysmap 所定义的命令来生成基本内核符号表文件 System.map。在其中包含有所有内核符号以及它们的地址，实际上前面用 kallsyms 包含在基本内核映像中的函数/变量地址及名称信息都等同于 System.map 中的内容，我们以后对内核代码的分析过程会经常引用这个文件。

rule_vmlinux__ 最后会额外做一步，用 verify_kallsyms 来确认前面的 kallsyms 是否工作正常。确认的方法是先拿前面的 .tmp_vmlinux2 重新生成一份新的 map: .tmp_System.map。接着构建系统会比较 .tmp_System.map 和 System.map，如果不一致，那说明 kallsyms 子系统代码工作不正常，所以构建系统会建议你设置

CONFIG_KALLSYMS_EXTRA_PASS 来重新 make vmlinux。verify_kallsyms 的定义为：

```
define verify_kallsyms
        $(Q)$(if $($(quiet)cmd_sysmap),                                  \
          echo '  $($(quiet)cmd_sysmap)  .tmp_System.map' &&)            \
          $(cmd_sysmap) .tmp_vmlinux$(last_kallsyms) .tmp_System.map
        $(Q)cmp -s System.map .tmp_System.map ||                         \
                (echo Inconsistent kallsyms data;                        \
                 echo Try setting CONFIG_KALLSYMS_EXTRA_PASS;            \
                 rm .tmp_kallsyms* ; /bin/false )
endef
```

5.1.2. 对目标 modules 的处理

前面有说当在 make 命令中不明确指定目标时，其使用的缺省目标是_all，而_all 又依赖于 all。构建系统对 all 的处理又是按顺序处理三个目标：vmlinux，zImage 和 modules。那我们这里已经把 vmlinux 的处理讨论完毕，其结果是产生两个输出文件：vmlinux 和 System.map，前者是基本内核的 ELF 映像，后者是基本内核符号表文件。接下来，我们先将对 zImage 的讨论搁置起来，先来看构建系统对 modules 的处理。

通过前面的分析，我们已经知道，在 Linux 中，区分有两种模块：内部模块和外部模块。我们这里说的对目标 modules 的处理指的就是要编译出那些内部模块，对外部模块的处理我们将在后面叙述。我们还知道，不管是内部模块，还是外部模块，其编译都要分两个阶段进行。阶段一生成组成模块的对应 .o 文件和 .mod 文件，阶段二要用 scripts/mod/modpost 来生成 .mod.c 文件，并将其编译成 .mod.o 对象文件，最后将 .mod.o 连同前面的 .o 一起链接成 .ko 模块文件。另外我们还知道，在生成 vmlinux 的过程中，会在内核顶层目录中生成一个 Modules.symvers，里面存放基本内核导出的、供模块使用的符号以及 CRC 校验和。通过前面的讨论所得到的这些知识，或许对你来说还不十分清楚，没关系，我们再行深入继续对内部模块目标 modules 的讨论，它将让你有个较为清楚的认识。

好，先在顶层 Makefile 中(框架中的 E1 部分)找到处理 modules 目标的规则：

```
#       Build modules
#
#       A module can be listed more than once in obj-m resulting in
#       duplicate lines in modules.order files.  Those are removed
#       using awk while concatenating to the final file.

PHONY += modules
modules: $(vmlinux-dirs) $(if $(KBUILD_BUILTIN),vmlinux)
        $(Q)$(AWK) '!x[$$0]++' $(vmlinux-dirs:%=$(objtree)/%/modules.order) > $(objtree)/modules.order
        @$(kecho) '  Building modules, stage 2.';
        $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modpost
        $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.fwinst obj=firmware __fw_modbuild
```

上面显示 modules 目标依赖于 $(vmlinux-dirs)。这种依赖就意味着内部模块处理的第一阶段就已经在处理 vmlinux-dirs 的过程中完成了。前面对 vmlinux-dirs 的讨论过程也说了如何编译出构成模块的那些 .o 对象文件，以及如何生成 .mod 文件。

很显然，既然内部模块的第一阶段已经完成，那处理 modules 目标规则的命令部分就是来完成内部模块的第二阶段了。

命令部分中的第一行用一个 awk 调用来将各子目录中 modules.order 文件内容归集到

顶层目录的 modules.order 文件中。该文件列出了构建系统构建内部模块的次序。如果你深入学习，你会知道 package module-init-tools 中包含有一个工具：depmod。该工具解析出各个内核模块的依赖关系，它会将依赖关系保存在文件 modules.dep 中。所谓模块之间的依赖，举个例子比方说模块 A 的代码中用到了模块 B 所导出来的函数，那么我们就说模块 A 是依赖于模块 B 的。很显然，既然模块之间有依赖，那模块之间的加载次序就得规定好。如我们的例子中，必须先加载模块 B，后加载模块 A，否则就会出错。老版本的 depmod 只是单纯的依赖内部模块之间的依赖来决定内部模块的加载顺序。但是先版本的 depmod 还会考虑内核构建系统构建各内核模块的顺序。关于更多 modules.orders 的使用信息，你可以参考内核开发邮件列表中的内容，在这里可以看到：http://lkml.org/lkml/2007/12/7/96。另外你也可以查看 module-init-tools 包 git 的修订记录：

http://git.kernel.org/?p=utils/kernel/module-init-tools/module-init-tools.git&a=search&h=HEAD&st=commit&s=modules.order 。

上面命令部分中最关键的就是接下来那一行：(Q)(MAKE) -f $(srctree)/scripts/Makefile.modpost。由于该命令没有指定 make 的目标，所以它会构建 Makefile.modpost 中的缺省目标 _modpost。而在同一个文件中查看一下 _modpost 的相关规则：

```
PHONY := _modpost
_modpost: __modpost
....
# Stop after building .o files if NOFINAL is set. Makes compile tests quicker
_modpost: $(if $(KBUILD_MODPOST_NOFINAL), $(modules:.ko:.o),$(modules))
```

上面代码表明，_modpost 依赖于 __modpost。同时，如果有定义过 KBUILD_MODPOST_NOFINAL，那么它还依赖于那些和模块名称对应的 .o 文件。打个比方，如果有两个对象文件 part1.o 和 part2.o 组成一个模块 MyModule.ko，那么它就依赖于 MyModule.o 对象文件。另外如果没有定义过，那它还依赖于所有的内部模块。所以变量 KBUILD_MODPOST_NOFINAL 的定义就意味着我们只是生成 MyModule.o，而不要再继续从 MyModule.o 出发生成 MyModule.ko 模块。变量 modules 被这样定义：

```
# Step 1), find all modules listed in $(MODVERDIR)/
__modules := $(sort $(shell grep -h '\.ko' /dev/null $(wildcard $(MODVERDIR)/*.mod)))
modules   := $(patsubst %.o,%.ko, $(wildcard $(__modules:.ko=.o)))
```

这个定义用 grep 搜索目录 $(MODVERDIR)/ 中的所有 *.mod 文件，找出其中包含模块文件名称后缀 .ko 的那些行。效果上也就是等价于找出所有的内部模块名称，组成列表赋给 modules。还记得么？前面提到过，目录$(MODVERDIR)就是 .../.tmp_version/，其中存有模块处理第一阶段中生成的所有 .mod 文件。

我们回来看一下 __modpost 目标的处理，找出代码如下：

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

仔细看该规则的命令部分，它调用了 cmd_modpost，我们来看看它的定义：

```
# Step 2), invoke modpost
#  Includes step 3,4
modpost = scripts/mod/modpost                     \
 $(if $(CONFIG_MODVERSIONS),-m)                   \
 $(if $(CONFIG_MODULE_SRCVERSION_ALL),-a,)        \
 $(if $(KBUILD_EXTMOD),-i,-o) $(kernelsymfile)    \
 $(if $(KBUILD_EXTMOD),-I $(modulesymfile))       \
 $(if $(KBUILD_EXTRA_SYMBOLS), $(patsubst %, -e %,$(KBUILD_EXTRA_SYMBOLS))) \
 $(if $(KBUILD_EXTMOD),-o $(modulesymfile))       \
 $(if $(CONFIG_DEBUG_SECTION_MISMATCH),,-S)       \
 $(if $(CONFIG_MARKERS),-K $(kernelmarkersfile)) \
 $(if $(CONFIG_MARKERS),-M $(markersfile))        \
 $(if $(KBUILD_EXTMOD)$(KBUILD_MODPOST_WARN),-w) \
 $(if $(cross_build),-c)

quiet_cmd_modpost = MODPOST $(words $(filter-out vmlinux FORCE, $^)) modules
      cmd_modpost = $(modpost) -s
```

似曾相识对吧？没错，我们在前面讨论 vmlinux.o 的处理的时候，就已经碰到工具程序 .../scripts/mod/modpost 的使用了。只不过，那时候使用它的是变量 cmd_kernel-mod，而非 cmd_modpost。当时，构建系统用来完成两个动作：mis-match section 的检查和生成基本内核导出符号文件 Module.symvers，其中包含基本内核所导出的所有符号及 CRC 校验。那此处调用 .../scripts/mod/modpost 来做何用途呢？我们且先来看 modpost 工具程序的调用方式。

由于我们的配置(s3c2410_defcofig)中，并没有设置 CONFIG_MODVERSIONS，CONFIG_MODULE_SRCVERSION_ALL，CONFIG_DEBUG_SECTION_MISMATCH 以及 CONFIG_MARKERS。同时我们也没设置 KBUILD_EXTRA_SYMBOLS，而当前我们是在处理内部模块的第二阶段，所以上面处理 __modpost 规则中的命令实际上就是：

scripts/mod/modpost -o /home/yihect/linux-2.6.31/Module.symvers -S -c -s vmlinux MyModule.o YouModule.o HisModule.o ....

其中，命令后半部分包括省略号所表示的，是与各内部模块名称对应的 .o 文件。这个命令在这里主要也是要完成两项工作：

a) 解析出 vmlinux 以及各对应的 .o 文件内的符号，并重新将它们连同各自的 CRC 校验写入到顶层目录中的文件 Modules.symvers 内。所以最后该文件内不仅包含基本内核的符号及 CRC 校验，还包括各内部模块所导出的符号及 CRC 校验，在结果上是前面处理 vmlinux.o 时所生成的 Modules.symvers 的超集；

b) 针对各个内部模块，生成对应的 *.mod.c 文件。生成 *.mod.c 文件的代码在 modpost.c 文件的 main 函数中：

```
int main(int argc, char **argv)
{
        struct module *mod;
        struct buffer buf = { };

        ....
        for (mod = modules; mod; mod = mod->next) {
                char fname[strlen(mod->name) + 10];

                if (mod->skip)
                        continue;

                buf.pos = 0;

                add_header(&buf, mod);
                add_staging_flag(&buf, mod->name);
                err |= add_versions(&buf, mod);
                add_depends(&buf, mod, modules);
                add_moddevtable(&buf, mod);
                add_srcversion(&buf, mod);

                sprintf(fname, "%s.mod.c", mod->name);
                write_if_changed(&buf, fname);
        }
        ....

        return err;
}
```

具体的生成代码就分布在不同的 add_* 函数当中，由于超出本问主题范围，我们在这里不对它们详加阐述。你可自行查看代码，并参与我们在 mail list 中的讨论。我们这里看下它生成的 *.mod.c 文件的内容，我们以目录 .../net/wireless/ 下的模块 cfg80211.ko 为例(由于 s3c2410_defconfig 的默认配置将变量 CONFIG_CFG80211 设置为 M，所以根据该目录下 Makefile 的内容，构建系统会生成模块 cfg80211.ko)。为了完整的说明 *.mod.c 文件的内容，我们特意修改了 .../.config 配置文件，将 CONFIG_MODVERSIONS 及 CONFIG_MODULE_SRCVERSION_ALL 两变量设置为 y。也就是打开了内核的 Module versioning 功能。我们列出文件 cfg80211.mod.c 的内容(有删减):

```
#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[]
__used
__attribute__((section("__versions"))) = {
        { 0x89b672ff, "module_layout" },
        { 0x12da5bb2, "__kmalloc" },
        ......
        { 0xb859f38b, "krealloc" },
        { 0xe914e41e, "strcpy" },
};

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

MODULE_INFO(srcversion, "DF83F7FEC7262012BB15A0E");
```

该文件大部分的代码是定义一些变量，并将其放在三个不同的 elf section 内(后面构建系统会编译这个 .mod.c 形成对象文件，链接进 .ko)：

a) 定义 struct module 结构变量 __this_module，并将其放在 .gnu.linkonce.this_module section 中。在将模块加载进运行着的内核时，内核负责将这个对象加到内部的 modules list 中。modules list 是内核维护所有已加载模块的一个双向链表(更多请看:http://lkml.org/lkml/2002/12/27/16)。

b) 定义 struct modversion_info 结构数组 ____versions，并将其放到 __versions sectiong 中。该数组中存放的都是该模块中使用到，但没被定义的符号，也就是所谓的 unresolved symbol，它们或在基本内核中定义，或在其他模块中定义，内核使用它们来做 Module versioning。注意其中的 module_layout 符号，这是一个 dummy symbol。内核使用它来跟踪不同内核版本关于模块处理的相关数据结构的变化。当一个模块在 A 版本的内核中编译后，又在另外一个 B 版本的内核中加载，如果两个内核中处理 modules 的那些数据结构体定义发生变化了，那内核就拒绝继续做其他 Module versioning 工作，也就是拒绝加载模块。

符号 module_layout 在文件 .../kernel/module.c 中被定义成一个函数:

```
#ifdef CONFIG_MODVERSIONS
/* Generate the signature for all relevant module structures here.
 * If these change, we don't want to try to parse the module. */
void module_layout(struct module *mod,
                   struct modversion_info *ver,
                   struct kernel_param *kp,
                   struct kernel_symbol *ks,
                   struct marker *marker,
                   struct tracepoint *tp)
{
}
EXPORT_SYMBOL(module_layout);
#endif
```

该函数函数体为空，但却有很多的参数类型。为什么？就是因为内核要用它来跟踪 module/modversion_info/kernel_param/kernel_symbol/marker/tracepoint 等结构体定义变化。那如何跟踪这种变化呢？内核会和处理其他的符号一样，用这个函数原型做一次 CRC 校验， 产生校验和。将其放如 *.mod.c 的__versions section 中，待在模块加载时，拿其与保存在正运行的内核中的 CRC 进行比较，如果不同，就拒绝进一步加载模块。加载模块时内核对此项的检查代码在 .../kernel/module.c，如下：

```
static inline int check_modstruct_version(Elf_Shdr *sechdrs,
                                          unsigned int versindex,
                                          struct module *mod)
{
        const unsigned long *crc;

        if (!find_symbol(MODULE_SYMBOL_PREFIX "module_layout", NULL,
                         &crc, true, false))
                BUG();
        return check_version(sechdrs, versindex, "module_layout", mod, crc);
}
```

函数 check_version 就做真正的检查工作，检查不通过，内核会报出著名的错误信息：disagrees about version of symbol module_layout。

c) 最后，.mod.c 中会将很多信息塞进 .modinfo section 中，包括：vermagic 字符串，模块依赖信息，srcversion 信息等等(还有其他很多信息)。我们以 vermagic 来举例分析。

宏 MODULE_INFO 定义在 .../include/linux/module.h 中：

```
/* Generic info of form tag = "info" */
#define MODULE_INFO(tag, info) __MODULE_INFO(tag, tag, info)
```

而 __MODULE_INFO 则定义在 .../include/linux/moduleparam.h 中：

```
#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__)[]                         \
  __used                                                                \
  __attribute__((section(".modinfo"),unused)) = __stringify(tag) "=" info
```

其中 __stringify 又是定义在 .../include/linux/stringify.h 文件中的宏：

```
#define __stringify_1(x...)     #x
#define __stringify(x...)       __stringify_1(x)
```

所以，综上所述，当你在一个 c 程序文件的第 21 行写下这样一条语句后：MODULE_INFO(pppp, "qqq"); 那么经过 C 预处理，就会展开成：

static const char __mod_pppp21[] __used __attribute__((section(".modinfo"),unused)) = "pppp" "=" "qqq";

实际上，就是定义了一个名为 __mod_pppp21 的字符数组，将其初始化成字符串 "pppp=qqq" 形式后放入 .modinfo section 中。再来看宏 VERMAGIC_STRING 的定义：它定义在文件.../include/linux/vermagic.h 中：

```
/* Simply sanity version stamp for modules. */
#ifdef CONFIG_SMP
#define MODULE_VERMAGIC_SMP "SMP "
#else
#define MODULE_VERMAGIC_SMP ""
#endif
#ifdef CONFIG_PREEMPT
#define MODULE_VERMAGIC_PREEMPT "preempt "
#else
#define MODULE_VERMAGIC_PREEMPT ""
#endif
#ifdef CONFIG_MODULE_UNLOAD
#define MODULE_VERMAGIC_MODULE_UNLOAD "mod_unload "
#else
#define MODULE_VERMAGIC_MODULE_UNLOAD ""
#endif
#ifdef CONFIG_MODVERSIONS
#define MODULE_VERMAGIC_MODVERSIONS "modversions "
#else
#define MODULE_VERMAGIC_MODVERSIONS ""
#endif
#ifndef MODULE_ARCH_VERMAGIC
#define MODULE_ARCH_VERMAGIC ""
#endif

#define VERMAGIC_STRING                                                 \
        UTS_RELEASE " "                                                 \
        MODULE_VERMAGIC_SMP MODULE_VERMAGIC_PREEMPT                     \
        MODULE_VERMAGIC_MODULE_UNLOAD MODULE_VERMAGIC_MODVERSIONS       \
        MODULE_ARCH_VERMAGIC
```

可见，机器是否为 SMP，内核是否配置为抢占式，是否不允许模块卸载以及 Module versioning 功能是否开启等等，都影响着 VERMAGIC_STRING 的取值。当加载模块到内核中时，内核也会检查 vermagic 是否有变化。如果不一样，内核照样不允许该模块的加载。检查代码在 .../kernel/module.c 中：

```
modmagic = get_modinfo(sechdrs, infoindex, "vermagic");
/* This is allowed: modprobe --force will invalidate it. */
if (!modmagic) {
        err = try_to_force_load(mod, "bad vermagic");
        if (err)
                goto free_hdr;
} else if (!same_magic(modmagic, vermagic, versindex)) {
        printk(KERN_ERR "%s: version magic '%s' should be '%s'\n",
               mod->name, modmagic, vermagic);
        err = -ENOEXEC;
        goto free_hdr;
}
```

内核先取得存储在模块中的 magic 字符串(由 .mod.c 编译连接到模块中)放在 modmagic 中，再用 same_magic 函数去和内核中保存好的 magic 字符串比较。比较不一致时，内核就会报错，从而拒绝该模块的加载。

*.mod.c 文件生成之后，如果 KBUILD_MODPOST_NOFINAL 定义过，那对 _modpost 的处理就算结束了，否则因为 _modpost 要依赖于 $(modules)，构建系统还要负责构建出各个内部模块(.ko)。从 .../scripts/Makefile.modpost 中找到处理 $(modules) 的相关规则：

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

这是一条静态匹配规则，可以看出内部模块文件 *.ko 要依赖于同名的 *.o 和 同名的 *.mod.o 。同名*.o 已经在第一阶段处理 vmlinux-dirs 时准备妥当，但是同名 *.mod.o 还未生成，所以构建系统必须用下面的规则来生成它：

```
quiet_cmd_cc_o_c = CC      $@
      cmd_cc_o_c = $(CC) $(c_flags) $(CFLAGS_MODULE)     \
                   -c -o $@ $<

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

由处理 $(modules) 的规则看出，生成 *.mod.o 后，构建系统使用变量 cmd_ld_ko_o 定义的命令来将同名*.o 和同名*.mod.o 链接成*.ko：

```
# Step 6), final link of the modules
quiet_cmd_ld_ko_o = LD [M]  $@
      cmd_ld_ko_o = $(LD) -r $(LDFLAGS) $(LDFLAGS_MODULE) -o $@         \
                    $(filter-out FORCE,$^)
```

5.1.3. 对目标 zImage 的处理

好，至此，所有内部模块均已构建完毕。是时候讨论 zImage 的处理了。关于另外一种模块，即外部模块，我们在讨论完 zImage 的处理后再来讨论。

在编译 vmlinux 以及 modules 的过程中，构建系统已经产生了很多输出：vmlinux elf 映像、System.map 符号表文件、各种内部模块等等。这些东西是编译支持任何架构的 Linux 内核过程中都要产生的，但是如果要想真正去使用 Linux 内核，光用前面产生的 vmlinux 是不行的。因为前面产生的 vmlinux 是一个 ELF 映像，不能拿来直接执行。这个不像我们平常所开发出来的上层应用程序，比方一个 hello 程序，代码编写完毕后，编译成 ELF 可执行文件，然后直接放到 Linux 操作系统下面去执行(具体执行之前，需先由放在内核中的加载器加载)。为什么不能直接拿 vmlinux ELF 映像来使用？就是因为不存在另外一个已经运行好的 Linux 环境(加载器)来帮助我们加载。我们真正需要的是能放到内存里面直接执行的二进制指令序列，也就是不带任何格式的纯粹二进制文件。通常情况下，在使用这个二进制内核时，它总是会被压缩存储的。业内有人将前面过程中产生的 vmlinux 称为 the kernel

proper，为了后面叙述的方便，我们也使用这样的称呼。

虽然存在有很少的架构和 bootload 能直接引导启动这个二进制内核文件，但是更通常的情况是：我们需要在这一二进制文件的基础之上，加上其他工具性的代码以完成加载和引导 Linux 内核。加上这些工具性的代码后，最终出来一个完整的内核映像。不幸的是，这个内核映像名字也叫 vmlinux，为了区分清楚，内核开发者通常称之为 composite kerel image。最后，为了让 bootloader 能真正的去引导启动 Linux 内核，构建系统还会将 composite kernel image 做一下处理，形成诸如 zImage、bzImage、xipImage、uImage 等等众多的映像。注意在嵌入式应用中用的最多的是 zImage 和 xipImage。在 x86 平台中使用最多的 bzImage。而 uImage 是 zImage 经过 mkimage 工具处理过，搭配 uboot 来使用的内核映像。

说了这么多，也许你还没怎么明白，不要紧，以下我们以 arm 平台(s3c2410_defconfig 缺省配置)上的 zImage 来分析构建系统的构建动作。熟悉这个后，其他的你自己就可以进行分析了。前面已经说过在 arm 架构的对应的架够 Makefile 中规定了 all 要依赖于 zImage，所以我们在 .../arch/arm/Makefile 中找到 zImage 的对应规则：

```
zImage Image xipImage bootpImage uImage: vmlinux
        $(Q)$(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $(boot)/$@
```

上面规则中说 zImage 依赖于 vmlinux。这个 vmlinux 就是我们前面构建好的在顶层目录中的 vmlinux，也就是所谓的 kernel proper。由于 boot 变量被定义成 arch/arm/boot，而且 MACHINE 变量被定义成 arch/arm/mach-s3c2410，所以该规则的命令部分可以简化成：

make -f scripts/Makefile.build obj=arch/arm/boot MACHINE=arch/arm/mach-s3c2410 arch/arm/boot/zImage。

我们遵循以前的做法，到 .../arch/arm/boot/Makefile 中找到关于 zImage 的规则：

```
$(obj)/zImage:  $(obj)/compressed/vmlinux FORCE
        $(call if_changed,objcopy)
        @echo '  Kernel: $@ is ready'
```

由上可知，zImage 的构建依赖于 .../arch/arm/boot/compressed/ 下的 vmlinux。:)别急，是的，很不幸，这个也叫做 vmlinux，名字与我们的 kernel proper 同名。其实这个就是我们刚才讲到的 composite kernel image。我们在 .../arch/arm/boot/Makefile 下找出它的处理规则：

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

从中看出它要依赖于 .../arch/arm/boot/ 下面的 Image，我们继续追踪找到 .../arch/arm/boot/Image 的构建规则：

```
$(obj)/Image: vmlinux FORCE
        $(call if_changed,objcopy)
        @echo '  Kernel: $@ is ready'
```

呵呵，坚持住别抓狂！这里又出现了一个 vmlinux。其实，这就是我们之前所说的 kernel proper，它已经在内核源码树的顶层目录中生成了。看到这里我们终于可以松一口气了，因为到这里为止，目标之间的依赖关系终于可以告一个段落，剩下的就是要沿着规则的命令部分倒着回去分析。这个规则的命令部分其实就是调用变量 cmd_obcopy 所定义的命令。变

量 cmd_objcopy 定义在 .../scripts/Makefile.lib 中：

```
# Objcopy
# ---------------------------------------------------------------------------

quiet_cmd_objcopy = OBJCOPY $@
cmd_objcopy = $(OBJCOPY) $(OBJCOPYFLAGS) $(OBJCOPYFLAGS_$(@F)) $< $@
```

而变量 OBJCOPYFLAGS 的定义在文件 .../arch/arm/Makefile 中：

OBJCOPYFLAGS :=-O binary -R .note -R .note.gnu.build-id -R .comment -S

变量 OBJCOPYFLAGS_$(@F) 可以扩展成 OBJCOPYFLAGS_Image，而该变量在构建系统中没有定义，所以 cmd_objcopy 所定义的命令就简化成：

arm-linux-objcopy -O binary -R .note -R .note.gnu.build-id -R .comment -S vmlinux arch/arm/boot/Image

这个命令的作用是用 objcopy 工具从 ELF 格式的 vmlinux(kernel proper) 中取出纯粹的二进制指令生成 Image 文件。做这个过程的同时，使用 -R 选项去掉 vmlinux 中的一些 elf section：.note 和 .note.gnu.build-id；另外使用 -S 选项将其中的调试符号信息去掉。

既然已经生成了 .../arch/arm/boot/Image，那接下来，便可用命令 (Q)(MAKE) $(build)=$(obj)/compressed $@ 来生成 composite kernel image 了。该命令可以简化成：

make -f scripts/Makefile.build obj=arch/arm/boot/compressed/ arch/arm/boot/compressed/vmlinux。

按照我们一贯的做法，从 .../arch/arm/boot/compressed/Makefile 中找出构造 vmlinux(composite kernel image) 的规则来：

```
$(obj)/vmlinux: $(obj)/vmlinux.lds $(obj)/$(HEAD) $(obj)/piggy.o \
                $(addprefix $(obj)/, $(OBJS)) FORCE
        $(call if_changed,ld)
        @:
```

从该规则可以看出，vmlinux(composite kernel image)的处理需要依赖于其他一些东西，咱们分别来看：

- ✓ .../arch/arm/boot/compressed/vmlinux.lds

  对于 composite kernel image 的处理，实际上是一个链接的过程。既然是链接，就得有一个连接脚本来指示如何来链接。前面我们在讨论处理 vmlinux 的时候已经接错过一个 vmlinux.lds，它是由 .../arch/arm/kernel/ 目录下的 vmlinux.lds.S 预处理而来的。但是这里的 vmlinux.lds 则是用来规范 composite kernel image 的链接过程的，它是由流编辑器处理 .../arch/arm/boot/compressed/ 目录下的 vmlinux.lds.in 文件得来的，希望同学们不要将其混淆。在 .../arch/arm/boot/compressed/Makefile 中找到对应的处理规则如下：

```
$(obj)/vmlinux.lds: $(obj)/vmlinux.lds.in arch/arm/boot/Makefile .config
      @sed "$(SEDFLAGS)" < $< > $@
```

✓ $(HEAD)

变量 HEAD 被定义成 head.o，它是构建系统使用 .../script/Makefile.build 文件中下面的规则从 head.S 生成的。后面你会知道，构建系统将 head.o 放在整个 composite kernel image 的最前面，并将 head.S 中定义的标号 start 作为整个 composite kernel image 的入口。你如果去看 .../arch/arm/boot/compressed/vmlinux.ds 的话，会看到，head.o 中定义的 .start section 将会被放在整个 composite image kernel 的最前面。

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

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

head.S 里面完成了从 bootloader 到内核启动(第一条内核代码开始运行)的所有工作，包括 kernel proper 的解压缩和重定位，我们会在后面的文章中详细分析这个文件。

✓ piggy.o

这是在构建 zImage 整个过程中最有意思的地方。piggy 一词，原指小猪、贪心的之意。但在这里，强调的是 piggy-back 的意思，也即背负的意思。这很好理解，长征－2F 运载火箭背负着神州七号飞向太空，神州七号是负载。按照同样的理解，这里也有负载。那是什么？是前面的 kernel proper。那这里的长征火箭又是什么呢？后面会看到，那就是所谓的 bootstrap loader。我们先来看看构建系统是如何帮助实现这一 piggy-back 的，.../arch/arm/boot/compressed/Makefile 中有这样的规则：

```
$(obj)/piggy.gz: $(obj)/../Image FORCE
$(call if_changed,gzip)

$(obj)/piggy.o:  $(obj)/piggy.gz FORCE
```

从这些规则上看出，有一个目标 piggy.gz，是经由压缩 kernel proper 产生的，具体使用的命令是：cmd_gzip。定义在 .../scripts/Makefile.lib 中:

```
# Gzip
# ---------------------------------------------------------------------------

quiet_cmd_gzip = GZIP    $@
cmd_gzip = (cat $(filter-out FORCE,$^) | gzip -f -9 > $@) || \
        (rm -f $@ ; false)
```

另外你也看到，piggy.o 要依赖于 piggy.z。这个地方，有些同学一下找不到构建 piggy.o 的规则就以为它是 make 的内嵌规则所致。实际上，正和前面构造 head.o 一样，构建系统也是用同样的规则来将 piggy.S 生成 piggy.o 的。打开 piggy.S，你会发现这里正是实现背负的关键地方：

```
        .section .piggydata,#alloc
        .globl  input_data
input_data:
        .incbin "arch/arm/boot/compressed/piggy.gz"
        .globl  input_data_end
input_data_end:
```

可以发现，正是这段代码将前面压缩得到的 piggy.gz 当做一段数据镶嵌到编译 piggy.S 后生成的 piggy.o 对象文件的 .piggydata ELF section 中去的。

✓ $(OBJS)

变量 OBJS 被定义成 misc.o。在其他一些 ARM 架构的芯片中，它还有可能包含 big-endian.o 对象文件，但是就 s3c2410_defconfig 来说，因为没有定义 CONFIG_CPU_ENDIAN_BE32，所以 OBJS 变量并不包含 big-endia.o。构建系统使用下面 .../scripts/Makefile.build 中的规则从 misc.c 生成 misc.o。misc.o 里面包含的主要是用来解压缩 kernel proper 的函数代码。这些函数由 head.o 中的代码来调用。

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

在前面我们已经介绍过类似的规则，所以这里就不再细讲。

好，回到构建 composite kernel image 的规则上来，既然所有的依赖都已经准备妥当，那就该用最后的链接命令 cmd_ld 来生成 composite kernel image 了。cmd_ld 定义在 .../scripts/Makefile.lib 中：

```
# Linking
# ---------------------------------------------------------------------------

quiet_cmd_ld = LD      $@
cmd_ld = $(LD) $(LDFLAGS) $(ldflags-y) $(LDFLAGS_$(@F)) \
               $(filter-out FORCE,$^) -o $@
```

实际上执行的这个命令简化一下就是：

```
arm-linux-ld -EL    --defsym zreladdr=0x30008000 --defsym params_phys=0x30000100 -p --no-undefined
  -X /home/yihect/eldk/usr/bin/../lib/gcc/arm-linux-gnueabi/4.2.2/soft-float/libgcc.a
  -T arch/arm/boot/compressed/vmlinux.lds arch/arm/boot/compressed/head.o arch/arm/boot/compressed/piggy.o
  arch/arm/boot/compressed/misc.o -o arch/arm/boot/compressed/vmlinux
```

为了让您更明白 composite kernel image 的构建，在这里用一个图来描述它的构建过程(原图取自 elp，稍做修改)：

注意，如该图所示，上面这些 head.o、misc.o 等构成了 bootstrap loader。在 composite kernel image 中，正是由 bootstrap loader 背负(piggy-back)了真正的负载，也即 kernel

proper。

最后，内核构建系统根据下面 .../arch/arm/boot/Makefile 中的规则，使用 objcopy 工具将没用的 ELF section 从 composite image kernel 中去除掉，形成最后可供 bootloader 引导的内核映像 zImage。关于这个规则中涉及到的 cmd_objcopy 命令，我们前面已经讨论过，这里不再讨论。

```
$(obj)/zImage:  $(obj)/compressed/vmlinux FORCE
        $(call if_changed,objcopy)
        @echo '  Kernel: $@ is ready'
```

好，至此，内部模块已经产生出来，可供引导的内核映像也已经产生出来了。所以我们为讲述构建目标而举的第一个例子："make ARCH=arm CROSS_COMPILE=arm-linux-"已全部介绍完毕。

5.2. make ARCH=arm CROSS_COMPILE=arm-linux- -C KERNELDIR M=dir

5.2.1. 编译外部模块的先决条件

接下来，让我们来看看另外一个例子，也就是编译外部模块的命令:"make ARCH=arm CROSS_COMPILE=arm-linux- -C KERNELDIR M=dir"。

关于包含在该命令中的两个选项 "-C" 和 "M"，我们在前面已经有所介绍了。"-C"用来使 make 工具进入某个目录下面去 make。这里是让 make 工具进入到内核源码所在目录 KERNELDIR 下面，去利用该目录下的 Makefile 进行 make 处理。如果我们是为当前正在运行着的内核继续外部模块的编译，那我们需要设置 KERNELDIR 为：/lib/modules/`uname -r`/build。因为该/lib/modules/`uname -r`/build 通常会在内核安装过程中软连接到内核代码所在目录。

选项 M 是用来指定外部模块代码所在目录的。所谓外部模块，就是指代码并非包含在内核源码树中的那些可加载模块。构建系统允许我们使用两种手段来指定外部模块代码所在目录。除了在命令行里使用 M 选项外，还可以在命令行中使用另外一个选项 SUBDIR，但是这是过时的做法。如果你在命令行内既指定了 M 选项，又使用了 SUBDIR 选项，那内核构建系统会优先考虑使用 M 选项所指定的那个值。这可以从顶层 Makefile 中的下面代码中看出来：

```
# Use make M=dir to specify directory of external module to build
# Old syntax make ... SUBDIRS=$PWD is still supported
# Setting the environment variable KBUILD_EXTMOD take precedence
ifdef SUBDIRS
  KBUILD_EXTMOD ?= $(SUBDIRS)
endif

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

上面代码中，将外部模块所在目录赋值给 make 中的变量：KBUILD_EXTMOD。所以，如果我们愿意，我们也可以直接在 make 命令行中使用 KBUILD_EXTMOD 来设置。

需要注意的是，在进行外部模块的编译之前，内核源代码树中必须存在有 .../include/config/auto.conf 文件，因为我们在框架的 G2 部分中看到这样的代码：

```
# external modules needs include/linux/autoconf.h and include/config/auto.conf
# but do not care if they are up-to-date. Use auto.conf to trigger the test
PHONY += include/config/auto.conf

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

我们可以在一个配置过的内核树中执行命令："make ... prepare" 来生成 auto.conf 文件。另外编译外部模块也要求使用一些工具程序，譬如 .../scripts/mod 目录下面的 modpost 程序，你可以在一个配置过的内核树中执行命令："make ... scripts" 来生成这些工具。一般情况下，如果你已经在你的内核树中编译过内核，那上面这两个条件都已经得到满足。因为编译内核通常需要比较长的时间，假如在编译外部模块之前等不及，或者老板在狠命的催你，那你就用这样的命令来准备一下你的内核树环境："make ... modules_prepare"。你看 makefile 中代码的话，你会注意到这个命令先后 make 了 prepare 和 scripts 两个目标，所以能有一次执行上述两个命令的效果。当然，这个命令也有不好的地方，那就是它不会帮我们在顶层目录下生成 Modules.symvers 文件，即使设置了 CONFIG_MODVERSIONS 选项，也是如此。所以，你如果想起用 Module Versioning 功能的话，你最好还是花点时间先编译一下内核。

5.2.2. 如何实现这例子二中的外部模块构建

好，接下来注意看看我们例子二中的命令："make ARCH=arm CROSS_COMPILE=arm-linux- -C KERNELDIR M=dir"。并没有指定所要 make 的目标是什么，但是构建系统会给它准备一个默认的目标 _all，这我们前面已经有所接触。但是由于设置了 KBUILD_EXTMOD，构建系统又会在下面的代码中让 _all 依赖于 modules。所以命令 "make ARCH=arm CROSS_COMPILE=arm-linux- -C KERNELDIR M=dir" 其实就相当于 "make ARCH=arm CROSS_COMPILE=arm-linux- -C KERNELDIR M=dir modules"：

```
# 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
```

这在前面已经看到过，modules 目标的处理也已经讨论过。但是注意，这里的这个 modules 并非之前的那个 modules。之前的那个 modules 是构建系统为了处理内部模块而准备的，它位于顶层 Makefile 的 E1 部分中。而此处的这个 modules 则是为了处理外部模块而准备的，它位于 E2 部分中。前面已经说过 E1 部分和 E2 部分就是因为 KBUILD_EXTMOD 的取值而分开的。我们把 E2 部分中的那个 modules 规则取出来：

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

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

比较这里处理外部模块的 modules 规则和前面处理内部模块的 modules 规则。差别是显而易见的，但是也有共同点。这些异同主要体现在以下几个方面：

- 从整体上可以看出，无论内部模块，还是外部模块的处理。都需要经过两个阶段的处理。
- 内部模块依赖于 vmlinux-dirs 目标的处理，来完成第一阶段；而外部模块的处理则依赖于 modules-dirs 目标的处理。
- 内部模块的 modules 目标处理所有内部模块；而外部模块的 modules 目标只处理 KBUILD_EXTMOD 目录中的外部模块。
- 都会生成 .mod 文件，并将其放在 MODVERDIR 目录中，也就是 .tmp_version 目录；只是 .tmp_version 目录的位置不同。
- 都会使用 .../scripts/mod/modpost 来生成 .mod.c 文件，以及处理 Module.symvers 文件。

还是让我们带着这些异同查看代码吧，也许，你会在看代码过程中概括出来更多的异同之处。modules 依赖于 $(modules-dirs)，也就是依赖于 _module_dir(dir 为外部模块代码所在的目录名称)。$(modules-dirs)目标的第一个依赖是 cromodverdir 目标，它的处理规则是：

```
PHONY += crmodverdir
crmodverdir:
        $(cmd_crmodverdir)
```

而 cmd_crmodverdir 变量的定义为：

```
# Create temporary dir for module support files
# clean it up only when building all modules
cmd_crmodverdir = $(Q)mkdir -p $(MODVERDIR) \
                  $(if $(KBUILD_MODULES),; rm -f $(MODVERDIR)/*)
```

再追踪下去，变量 MODVERDIR 的定义我们前面已经见过。当定义有 KBUILD_EXTMOD 时，它的取值是 dir/.tmp_versions(其中 dir 为外部模块代码所在目录)。

$(modules-dirs)目标的第二个依赖是 $(objtree)/Module.symvers。因为变量 objtree 被定义成了 $(CURDIR)，也就是 make 命令"-C"选项带进去的内核源代码树目录 KERNELDIR。所以我们说外部模块的处理，最好有一个东西作为保证，那就是在内核顶层目录下面存在文件 Modules.symvers。否则构建系统会按照下面规则而报出警告：

```
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 )
```

回到处理 $(module-dirs) 的规则上面来。接下来，构建系统要处理命令："(Q)(MAKE) $(build)=$(patsubst _module_%,%,$@)"。因为此时 $@ 为_module_dir，所以该命令可简化成："make -f scripts/Makefile.build obj=dir"。这其实和我们前面处理 $(vmlinux-dirs) 目标的命令是等同的，其最终效果是要用下面这条规则去处理 .../scripts/Makefile.build 文件中的 __build 目标：

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

这条规则的细节，我这里就不再关注了。隐藏在其背后的本质是构建系统像对待内核代码树中的子目录那样去对待一个代码树之外的目录。这条规则执行的最后结果，是在外部模块代码所在的目录中构建出组成外部模块所需的 .o 文件以及 .mod 文件。注意 .mod 文件的存放位置，是 dir/.tmp_versions/，而非内核源码树顶层目录中的 .tmp_versions/。

回到处理外部模块 modules 的规则上面来，既然 modules 的依赖 $(module-dirs) 处理完毕，构建系统就会使用命令：(Q)(MAKE) -f $(srctree)/scripts/Makefile.modpost 来完成外部模块处理的第二阶段。其大体流程和内部模块的处理差不多，只是在调用 .../scripts/mod/modpost 的时候，所使用的参数有所不同，我们列出 .../scripts/Makefile.modpost 中的相关代码来看：

```
# Step 2), invoke modpost
#  Includes step 3,4
modpost = scripts/mod/modpost                     \
 $(if $(CONFIG_MODVERSIONS),-m)                   \
 $(if $(CONFIG_MODULE_SRCVERSION_ALL),-a,)        \
 $(if $(KBUILD_EXTMOD),-i,-o) $(kernelsymfile)    \
 $(if $(KBUILD_EXTMOD),-I $(modulesymfile))       \
 $(if $(KBUILD_EXTRA_SYMBOLS), $(patsubst %, -e %,$(KBUILD_EXTRA_SYMBOLS))) \
 $(if $(KBUILD_EXTMOD),-o $(modulesymfile))       \
 $(if $(CONFIG_DEBUG_SECTION_MISMATCH),,-S)       \
 $(if $(CONFIG_MARKERS),-K $(kernelmarkersfile)) \
 $(if $(CONFIG_MARKERS),-M $(markersfile))        \
 $(if $(KBUILD_EXTMOD)$(KBUILD_MODPOST_WARN),-w) \
 $(if $(cross_build),-c)

quiet_cmd_modpost = MODPOST $(words $(filter-out vmlinux FORCE, $^)) modules
      cmd_modpost = $(modpost) -s

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

由于额外定义了 KBUILD_EXTMOD，所以构建系统处理 __modpost 的实际命令便是：

```
scripts/mod/modpost   -i /home/yihect/linux-2.6.31/Module.symvers -I /home/yihect/helko/Module.symvers
  -o /home/yihect/helko/Module.symvers -S   -w -c -s MyModule.o
```

和之前构建内部模块时所用的命令(见下)相比，最重要的区别就在于其多了两个选项: -i 和 -I，并且选项 -o 的参数也不同了。

```
scripts/mod/modpost   -o /home/yihect/linux-2.6.31/Module.symvers    -S    -c -s
  vmlinux MyModule.o YouModule.o HisModule.o ....
```

这是因为前面构建内部模块时，modpost 是取出 vmlinux 及各内部模块中符号(包括 CRC 值)，写入到顶层内核源码目录中的文件 Modules.symvers 中去的。而此时，构建系统构建的是外部模块，它却是要读入顶层内核源码目录中的 Module.symvers。这是因为构建系统在构建外部模块的时候，它会经常碰到一些在该外部模块中没有定义的符号，它要去检查这些符号是否为内核或某个内部模块所导出。如果是的话，那 OK 没问题，否则如果有的符号既没从基本内核中导出，又没从任一内部模块中导出，构建系统就会报错说 unresolve symbol。

modpost 在读入内核顶层目录中的 Module.symvers 的同时，它也在外部模块所在目录中输出一个新的 Module.symvers，里面记载该外部模块自己本身所导出的符号(包括 CRC 值)。注意，在构建外部模块的时候，外部模块本身所在目录下就有一个 Module.symvers 的话，它也会读入。这是为了 module stacking 而准备的。因为一个外部模块 A 中，除了使用基本内核以及内部模块所导出来的符号外，还有可能使用另外的外部模块 B 所导出的符号。这样的话，我们在编译构建外部模块 A 的时候， 就可以把构建外部模块 B 所产生的那个 Module.symvers 文件放到模块 A 所在目录下面。

好，至此，我们的第二个例子，即构建系统如何处理外部模块已经讨论完毕。也就意味着框架中："E 部分-对 vmlinux、modules 等构建目标以及和.config 无关目标的处理"讨论完毕。

6. Linux内核构建系统如何处理单一目标

接下来，我们还是来看框架中的 G 部分。在内核开发过程中，需要的经常不是整个内核的编译，而是经常需要重新编译生成某个单一的对象文件、某一个单一的目录、某一个单一的内核内部模块。这个时候，我们开发者就没有必要去动用 "make vmlinux" 之类的牛刀去重新编译整个内核，所以内核构建系统为支持此目的实现了对这些 Single target 的处理。作为内核构建系统的用户来讲，你只需要使用这些命令去构建：

```
make ARCH=arm CROSS_COMPILE=arm-linux- init/main.o      //编译内核树中单个对象文件
make ARCH=arm CROSS_COMPILE=arm-linux- lib/              //编译内核树中的一个子目录
make ARCH=arm CROSS_COMPILE=arm-linux- drivers/spi/spi_s3c24xx.ko   //编译内核树中的一个内部模块
```

当然，在做这些构建的时候，必须先完成内核的配置(kconfig)，因为它们的执行至少要求有 .config 的存在。另外，构建系统甚至还允许编译外部模块中的某个对象文件，比方这样：

make ARCH=arm CROSS_COMPILE=arm-linux- -C ~/linux-2.6.31 M=`pwd` hello.o

G 部分中和 Single target 相关的这些代码比较简单，这里就不再详细介绍，你自己就可以看懂。

7. Linux内核构建系统对依赖关系的处理

到目前为止，内核构建系统的大部分重要的地方都已讨论完毕，惟独有一个很关键的方面还没讨论完全，那就是依赖关系的处理。熟悉 Linux 内应用程序开发的人都知道，要想用 make 工具来自动化的管理他们的应用项目工程，就必须正确处理所要编译的目标和生成这些目标所需文件之间的依赖关系。举个例子，比方你要编译一个对象文件 hello.o，那么你就需要告诉 make 工具生成该对象文件所需要的依赖有哪些后，make 才能帮你正确的管理它的编译。这些所依赖的文件通常有可能是：对应的 C 程序文件 hello.c、你自己写的头文件、函数库内的头文件等等。

在实际应用项目开发的时候，我们开发者往往不会用手动的方式把这些所有依赖关系都列在 Makefile 中。更常见的方法是在 Makefile 中使用 gcc 的选项 -M(或者其他 -MD 之类的)，让它帮我们生成包含如下形式依赖性规则的 *.d 文件，然后在 Makefile 的后面用 -include 之类的来包含这个 *.d 文件。这样编译的时候，如果其中某个依赖文件被修改了，那对应的这个目标就一定会被 make 更新。

```
hello.o: hello.c /usr/include/linux/init.h /usr/include/linux/config.h \
  /usr/include/linux/module.h /usr/include/linux/list.h
```

那对于 Linux kernel，其实也需要处理一系列的依赖关系。只不过，它将应用程所常用的那种依赖关系扩大化了。具体来说，内核构建系统在构建的时候，需要根据下列情况来决定一个目标是否需要被构建：

a) 如果存在有该目标对应的依赖文件被改动，或者当前还未被生成，那内核构建系统肯定要生成该目标；
b) 如果生成该目标的命令行与之前保存下来的命令行不一致，比方改了一下 C 编译器或汇编器的标志之类的，那该目标也要被重新生成；
c) 如果某个 CONFIG_XXX 配置选项被改了。比方原先是 =y 的，现在改成 =m；原先没设置过的，现在设置成 =m 之类的。那所有在源代码中使用过这个选项的，比方在 hello.c(也有可能是另外某个头文件) 里面用到了这个选项，那对应的目标 hello.o 就要被重新生成。

我们来看看这些是如何实现的，有些东西在之前的讨论中我们已经接触到过，这里只是总结性的提一下。也有些没讨论到的，这里会列出一些代码来分析。在内核构建系统的众多 makefile 中都会使用到字符串 "-MD,$(depfile)"，比方是在像文件 .../scripts/Makefile.lib 中，将它赋给了 c_flags/a_flags/cpp_flags 等标志，这意味着构建系统在使用 c 编译器/汇编器/c 预处理器等过程中，就会产生依赖规则文件。变量 depfile 定义在 .../scripts/Kbuild.include 文件中：

```
###
# Name of target with a '.' as filename prefix. foo/bar.o => foo/.bar.o
dot-target = $(dir $@).$(notdir $@)

###
# The temporary file to save gcc -MD generated dependencies must not
# contain a comma
depfile = $(subst $(comma),_,$(dot-target).d)
```

另外，也会经常使用到字符串 "$(LINUXINCLUDE)"，比方在同样的文件中赋给这些标志变量。而下面变量 LINUXINCLUDE 的定义中(定义在顶层 Makefile 文件中)会使用 "-include include/linux/autoconf.h"。

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

所以这意味着，在产生的依赖规则文件中，也会包含文件 .../include/linux/autoconf.h，并且这种包含是全局迷漫性的。也就是这个文件会被内核大多数目标所依赖。

这里会存在一个问题。假如我前后两次的内核编译过程中，区别只在于第一次配置选项 CONFIG_MY_DRIVER 被设置为 =m，而第二次被设置为 =y。那你想在这第二次编译过程中是不是因为更新了 .../include/linux/autoconf.h 文件而去全部编译依赖这个文件的所有目标呢？显然是这样的，因为 .../include/linux/autoconf.h 文件会因为配置选项的改变而变得更新，所以，自然而然，依赖它的所有目标会被重新更新。

这是没有必要的严重浪费。因为我只是改变了我自己所写内部模块对应的那个那个配置选项 CONFIG_MY_DEVICE_DRIVER。换句话讲，这个选项只影响到我自己写的那个内部模块，而其他任何内部模块或者基本内核代码都没有使用这个选项，所以在第二次编译过程中，没有必要去重新 remake 差不多全部目标，而只需要重新编译我自己写的那个内部模块即可。

为了避免这种不必要的额外负担，而达到只编译那些因为配置选项变更而确实受到影响的目标。内核构建系统使用 .../scripts/basic/fixdep 做了一个小动作。该动作修改依赖规则文件，从中删除对 .../include/linux/autoconf.h 文件的直接依赖，而代之以对 .../include/config/ 目录下的空的头文件的依赖。下面，我们慢慢来解释这是怎么做到的。

在前面我们讨论 kconfig 的时候说过，配置工具 .../scripts/kconfig/conf 在产生 auto.conf、auto.conf.cmd 和 autoconf.h 等三个文件。与此同时，它会在函数 conf_write_autoconf 中调用 conf_split_config 函数。当时并没有说这个函数的确切工作过程，而只是留下一个伏笔说它会负责在 .../include/config 产生一系列的头文件。那这里我们详细来看看它内部是如何产生这些文件。先列出该函数的框架：

```
int conf_split_config(void)
{
        const char *name;
        char path[128];
        char *s, *d, c;
        ...
        name = conf_get_autoconfig_name();
        conf_read_simple(name, S_DEF_AUTO);

        if (chdir("include/config"))
                return 1;

        res = 0;
        for_all_symbols(i, sym) {
                ...//读取 ".../include/config/auto.conf" 文件中的每一个 CONFIG_XXX 定义

                /* Replace all '_' and append ".h" */
                s = sym->name;
                d = path;
                while ((c = *s++)) {
                        c = tolower(c);
                        *d++ = (c == '_') ? '/' : c;
                }
                strcpy(d, ".h");

                ...//用 open 系统调用搭配 O_CREAT flag 创建名为 config/xxx.h 的空的头文件
        }
out:
        if (chdir("../.."))
                return 1;

        return res;
}
```

该函数先使用 conf_get_autoconfig_name 取得文件 ".../include/config/auto.conf" 的名称，然后用 for 循环处理每一个 CONFIG_XXX 的定义。比方针对我们之前的那个 CONFIG_MY_DEVICE_DRIVER 配置选项，处理的时候，它在字符数组 path 中存储这样的字符串：my/device/driver.h。接下来，它会在后面用 open 系统调用在 .../include/config 目录下去创建名为此字符串的空头文件。从这里可以看出，针对每个配置过的配置选项，都会有这样的头文件产生。

前面说过构建系统会将对 .../include/linux/autoconf.h 文件的依赖转换为对这里的这些头文件的依赖。这是怎样的一个机制？比方说，在我自己的内部模块中，因为使用到了配置选项 CONFIG_MY_DEVICE_DRIVER，所以 fixdep 会将我对 autoconf.h 文件的依赖转换成对 .../include/config/my/device/driver.h 头文件的依赖。而对其他内部模块或者基本内核的目标来说，在代码中没有使用到这个配置选项，所以 fixdep 对它们的处理，只是简单的从它们对应的依赖规则中删除对 autoconf.h 文件的依赖。

这个过程现在不明白不要紧，我们先来看看 fixdep 在构建系统代码中是如何被使用的。在这之后，我们来举例说明这个过程。在内核构建系统中，你在变量 if_changed_dep 以及宏 rule_cc_o_c 的定义中都能发现对 scripts/basic/fixdep 的调用：

```
# Execute the command and also postprocess generated .d dependencies file.
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)

define rule_cc_o_c
        $(call echo-cmd,checksrc) $(cmd_checksrc)                              \
        $(call echo-cmd,cc_o_c) $(cmd_cc_o_c);                                 \
        $(cmd_modversions)                                                     \
        $(cmd_record_mcount)                                                   \
        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
```

概括一下，其中对 fixdep 的调用语法是：fixdep <depfile> <target> <cmdline>。fixdep 会读入依赖规则文件和命令行，并输出一些内容送向标准输出。那么前面这两段代码中对 fixdep 的调用就会将这些标准输出中输出的内容重重定向到文件 $(dot-target).tmp。最后此文件被重命名为 $(dot-target).cmd。

还是让我们举例来说明吧，现在假设我们要编译的目标是 .../drivers/mydriver.o，它是构成我们前面内部模块 .ko 的对应对象文件。那么根据前面我们讨论的记过，构建系统会调用 rule_cc_o_c 来生成 mydriver.o。假设前面针对 mydriver.o 生成出来的依赖规则文件 .mydriver.o.d 是这样的：

```
mydriver.o: drivers/mydriver.c include/linux/autoconf.h include/linux/types.h \
  /home/yihect/linux-2.6.31/arch/arm/include/asm/types.h \
  include/asm-generic/int-ll64.h \
  /home/yihect/linux-2.6.31/arch/arm/include/asm/bitsperlong.h \
  .....
```

并且，生成 mydriver.o 的命令行是：

arm-linux-gcc -Wp,-MD,drivers/.mydriver.o.d -nostdinc ....

那么 fixdep 在处理的时候，会首先向标准输出输出内容:

cmd_drivers/mydriver.o := arm-linux-gcc -Wp,-MD,drivers/.mydriver.o.d -nostdinc ...

然后，它遍历处理依赖规则中的所有依赖文件，用 GREP 检查这些文件中是否包含有对配置选项诸如 CONFIG_MY_DEVICE_DRIVER 之类的使用。如果有的话，它将 .../include/config/my/device/driver.h 文件也加到依赖文件列表中。注意在遍历处理的过程中，fixdep 会把 autoconf.h 从依赖列表中过滤掉。遍历完之后，它又会向标准输出输出下面这样的内容：

```
deps_drivers/mydriver.o := \
  drivers/mydriver.c \
    $(wildcard include/config/my/device/driver.h) \
    $(wildcard include/config/smp.h) \
    $(wildcard include/config/proc/fs.h) \
    $(wildcard include/config/constructors.h) \
  include/linux/types.h \
    $(wildcard include/config/uid16.h) \
    $(wildcard include/config/lbdaf.h) \
    $(wildcard include/config/phys/addr/t/64bit.h) \
    $(wildcard include/config/64bit.h) \
  /home/yihect/linux-2.6.31/arch/arm/include/asm/types.h \
  .....


drivers/mydriver.o: $(deps_drivers/mydriver.o)

$(deps_drivers/mydriver.o):
```

最后，所有的输出内容会被重定向到文件 .mydriver.o.tmp 中。接着该文件被重命名为 .mydriver.o.cmd。再最后这个命令文件会连同其它所有命令文件被包含进 makefile 规则链体系，这在前面已经看到过，这里不再论述。正是通过这样的小动作，内核构建系统移除了对 autoconf.h 的直接依赖，从而达到避免那种额外重新处理其他目标的负担的。