KBUILD 系统原理分析

kbuild，即 kernel build，用于编译 Linux 内核文件。kbuild 对 makefile 进行了功能上的扩充，使其在编译内核文件时更加高效，简洁。大部分内核中的 Makefile 都是使用 Kbuild 组织结构的 kbuild Makefile。

下面将分两部分介绍，首先介绍 Linux 的命令工具 make 及其所操作的 makefile，它负责将源代码编译成可执行文件；然后介绍 kbuild makefile 对 makefile 做了哪些扩充，以及 kbuild makefile 的工作原理。

Chapter 1. MAKE 概述

1.1 准备知识

一般来说，无论是 C、C++、还是 pas，首先要把源文件编译成中间代码文件，在 Windows 下也就是 .obj 文件，UNIX 下是 .o 文件，即 Object File，这个动作叫做编译（compile）。然后再把大量的 Object File 合成执行文件，这个动作叫作链接（link）。

编译：把高级语言书写的代码转换为机器可识别的机器指令。编译高级语言后生成的指令虽然可被机器识别，但是还不能被执行。编译时，编译器检查高级语言的语法、函数与变量的声明是否正确。只有所有的语法正确、相关变量定义正确编译器就可以编译出中间目标文件。通常，一个高级语言的源文件都可对应一个目标文件。目标文件在 Linux 中默认后缀为“.o”（如“hello.c”的目标文件为“hello.o”）。

链接：将多个.o 文件，或者.o 文件和库文件链接成为可被操作系统执行的可执行程序。链接器不检查函数所在的源文件，只检查所有.o 文件中的定义的符号。将.o 文件中使用的函数和其它.o 或者库文件中的相关符号进行合并，最后生成一个可执行的程序。“ld”是 GNU 的链接器。

静态库：又称为文档文件（Archive File）。它是多个.o 文件的集合。Linux 中静态库文件的后缀为“.a”。静态库中的各个成员（.o 文件）没有特殊的存在格式，仅仅是一个.o 文件的集合。使用“ar”工具维护和管理静态库。

共享库：也是多个.o 文件的集合，但是这些.o 文件时有编译器按照一种特殊的方式生成。对象模块的各个成员的地址（变量引用和函数调用）都是相对地址。因此在程序运行时，可动态加载库文件和执行共享的模块（多个程序可以共享使用库中的某

一个模块)。

1.2 makefile 简介

make 在执行时，需要一个命名为 Makefile 的文件。这个文件告诉 make 以何种方式编译源代码和链接程序，即告诉 make 需要做什么（完成什么任务），该怎么做。典型地，可执行文件可由一些.o 文件按照一定的顺序生成或者更新。如果在你的工程中已经存在一个或者多个正确的 Makefile。当对工程中的若干源文件修改以后，需要根据修改来更新可执行文件或者库文件，正如前面提到的你只需要在 shell 下执行“make”。make 会自动根据修改情况完成源文件的对应.o 文件的更新、库文件的更新、最终的可执行程序的更新。

make 通过比较对应文件（规则的目标和依赖，）的最后修改时间，来决定哪些文件需要更新、那些文件不需要更新。对需要更新的文件 make 就执行数据库中所记录的相应命令（在 make 读取 Makefile 以后会建立一个编译过程的描述数据库。此数据库中记录了所有各个文件之间的相互关系，以及它们的关系描述）来重建它，对于不需要重建的文件 make 什么也不做。

而且可以通过 make 的命令行选项来指定需要重新编译的文件。

在执行 make 之前，需要一个命名为 Makefile 的特殊文件（本文的后续将使用 Makefile 作为这个特殊文件的文件名）来告诉 make 需要做什么（完成什么任务），该怎么做。通常，make 工具主要被用来进行工程编译和程序链接。

当使用 make 工具进行编译时，工程中以下几种文件在执行 make 时将会被编译（重新编译):

1.所有的源文件没有被编译过，则对各个 C 源文件进行编译并进行链接，生成最后的可执行程序;

2.每一个在上次执行 make 之后修改过的 C 源代码文件在本次执行 make 时将会被重新编译;

3.头文件在上一次执行 make 之后被修改。则所有包含此头文件的 C 源文件在本次执行 make 时将会被重新编译。

后两种情况是 make 只将修改过的 C 源文件重新编译生成.o 文件，对于没有修改的文件不进行任何工作。重新编译过程中，任何一个源文件的修改将产生新的对应的.o 文件，新的.o 文件将和以前的已经存在、此次没有重新编译的.o 文件重新连接生成最后的可执行程序。

首先让我们先来看一些 Makefile 相关的基本知识。

1.3 Makefile 规则介绍

一个简单的 Makefile 描述规则组成:

```
TARGET: PREREQUISITES
[TAB] COMMAND
```

target: 规则的目标。通常是程序中间或者最后需要生成的文件名。可以是.o 文件、也可以是最后的可执行程序的文件名。另外，目标也可以是一个 make 执行的动作的名称，如目标“clean”，成这样的目标是“伪目标”。

prerequisites: 规则的依赖。生成规则目标所需要的文件名列表。通常一个目标依赖于一个或者多个文件。

command: 规则的命令行。是 make 程序所有执行的动作（任意的 shell 命令或者可在 shell 下执行的程序）。

一个规则可以有多个命令行，每一条命令占一行。注意: 每一个命令行必须以[Tab]字符开始，[Tab]字符告诉 make 此行是一个命令行。make 按照命令完成相应的动作。这也是书写 Makefile 中容易产生，而且比较隐蔽的错误。

命令就是在任何一个目标的依赖文件发生变化后重建目标的动作描述。一个目标可以没有依赖而只有动作（指定的命令）。比如 Makefile 中的目标“clean”，此目标没有依赖，只有命令。它所指定的命令用来删除 make 过程产生的中间文件（清理工作）。

在 Makefile 中“规则”就是描述在什么情况下、如何重建规则的目标文件，通常规则中包括了目标的依赖关系（目标的依赖文件）和重建目标的命令。make 执行重建目标的命令，来创建或者重建规则的目标（此目标文件也可以是触发这个规则的上一个规则中的依赖文件）。规则包含了目标和依赖的关系以及更新目标所要求的命令。

1.4 简单的示例

此例子由 3 个头文件和 8 个 C 文件组成。写一个简单的 Makefile，创建最终的可执行文件“edit”，此可执行文件依赖于 8 个 C 源文件和 3 个头文件。Makefile 文件的内容如下:

```
#sample Makefile
edit : main.o kbd.o command.o display.o \
insert.o search.o files.o utils.o
	cc -o edit main.o kbd.o command.o display.o \
insert.o search.o files.o utils.o

main.o : main.c defs.h
	cc -c main.c
```

```
kbd.o : kbd.c defs.h command.h
	cc -c kbd.c

command.o : command.c defs.h command.h
	cc -c command.c

display.o : display.c defs.h buffer.h
	cc -c display.c

insert.o : insert.c defs.h buffer.h
	cc -c insert.c

search.o : search.c defs.h buffer.h
	cc -c search.c

files.o : files.c defs.h buffer.h command.h
	cc -c files.c

utils.o : utils.c defs.h
	cc -c utils.c

clean :
	rm edit main.o kbd.o command.o display.o \
insert.o search.o files.o utils.o
```

在书写时，一个较长行可以使用反斜线（\）分解为多行，这样做可以使 Makefile 清晰、容易阅读。注意：反斜线之后不能有空格（这也是大家最容易犯的错误，而且错误比较隐蔽）。大家在书写 Makefile 时，推荐者中将较长行分解为使用反斜线连接得多个行的方式。当我们完成了这个 Maekfile 以后；创建可执行程序“edit”，你所要做的就是在包含此 Makefile 的目录（当然也在代码所在的目录）下输入命令“make”。删除已经本目录下生成的文件和所有的.o 文件，只需要输入命令“make clean”就可以了。

在这个 Makefile 中，目标（target）包含：可执行文件“edit”和.o 文件（main.o,kbd.o….），依赖（prerequisites）就是冒号后面的那些 .c 文件和 .h 文件。所有的.o 文件既是依赖（相对于可执行程序 edit）又是目标（相对于.c 和.h 文件）。命令包括 “cc –c maic.c”、“cc –c kbd.c”……

目标是一个文件时，当它的任何一个依赖文件被修改以后，这个目标文件将会被重新编译或者重新连接。当然，此目标的任何一个依赖文件如果有必要则首先会被重新编译。在这个例子中，“edit”的依赖为 8 个.o 文件；而“main.o”的依赖文件为“main.c”

和“defs.h”。当“main.c”或者“defs.h”被修改以后，再次执行“make”时“main.o”就会被更新（其它的.o 文件不会被更新），同时“main.o” 的更新将会导致“edit”被更新。

在描述目标和依赖之下的 shell 命令行，它描述了如何更新目标文件。命令行必需以[Tab]键开始，以和 Makefile 其他行区别。就是说所有的命令行必需以[Tab] 字符开始，但并不是所有的以[Tab]键出现行都是命令行。但 make 程序会把出现在第一条规则之后的所有的以[Tab]字符开始的行都作为命令行来处理。（要记住：make 程序不关心命令是如何工作的，对目标文件的更新需要你在规则的描述中提供正确的命令。“make”程序所做的就是当目标程序需要更新时执行规则所定义的命令）。

目标“clean”不是一个文件，它仅仅代表了执行一个动作的标识。通常情况下，不需要执行这个规则所定义的动作，因此目标“clean”没有出现在其它规则的依赖列表中。在执行 make 时，它所指定的动作不会被执行。除非执行 make 时明确地指定它作为重建目标。而且目标“clean”没有任何依赖文件，它只有一个目的，就是通过这个目标名来执行它所定义的命令。Makefile 中把那些没有任何依赖只有执行动作的目标称为“伪目标”（phony targets）。执行“clean”目标所定义的命令，可在 shell 下输入：make clean。

1.5 make 如何工作

默认的情况下，make 执行 Makefile 中的第一个规则，此规则的第一个目标称之为“最终目的”或者“终极目标”（就是一个 Makefile 最终需要更新或者创建的目标）。

上例的 Makefile，目标“edit”在 Makefile 中是第一个目标，因此它就是 make 的“终极目标”。当修改了任何 C 源文件或者头文件后，执行 make 将会重建终极目标“edit”。

当在 shell 提示符下输入“make”命令以后。make 读取当前目录下的 Makefile 文件，并将 Makefile 文件中的第一个目标作为其“终极目标”，开始处理第一个规则（终极目标所在的规则）。在我们的例子中，第一个规则就是目标“edit”所在的规则。规则描述了“edit”的依赖关系，并定义了链接.o 文件生成目标“edit”的命令； make 在处理这个规则之前，首先将处理目标“edit”的所有的依赖文件（例子中的那些.o 文件）的更新规则；对包含这些.o 文件的规则进行处理。对.o 文件所在的规则的处理有下列三种情况：

1.目标.o 文件不存在，使用其描述规则创建它；

2.目标.o 文件存在，目标.o 文件所依赖的.c 源文件、.h 文件中的任何一个比目标.o 文件“更新”（在上一次 make 之后被修改）。则根据规则重新编译生成它；

3.目标.o 文件存在，目标.o 文件比它的任何一个依赖文件（的.c 源文件、.h 文件）“更新”（它的依赖文件在上一次 make 之后没有被修改），则什么也不做。

这些.o 文件所在的规则之所以会被执行，是因为这些.o 文件出现在“终极目标”的依赖列表中。如果在 Makefile 中一个规则所描述的目标不是“终极目标”所依赖的（或者“终极目标”的依赖文件所依赖的），那么这个规则将不会被执行。除非明确指定这个

规则（可以通过 make 的命令行指定重建目标，那么这个目标所在的规则就会被执行，例如 “make clean”）。在编译或者重新编译生成一个.o 文件时，make 同样会去寻找它的依赖文件的重建规则（是这样一个规则：这个依赖文件在规则中作为目标出现），就是.c 和.h 文件的重建规则。在上例的 Makefile 中没有哪个规则的目标是.c 或者.h 文件，所以没有重建.c 和.h 文件的规则。

完成了对.o 文件的创建（第一次编译）或者更新之后，make 程序将处理终极目标“edit”所在的规则，分为以下三种情况:

1.目标文件“edit”不存在，则执行规则创建目标“edit”。

2.目标文件“edit”存在，其依赖文件中有一个或者多个文件比它“更新”，则根据规则重新链接生成“edit”。

3.目标文件“edit”存在，它比它的任何一个依赖文件都“更新”，则什么也不做。

上例中，如果更改了源文件“insert.c”后执行 make，“insert.o”将被更新，之后终极目标“edit”将会被重生成；如果我们修改了头文件“command.h”之后运行“make”，那么“kbd.o”、“command.o”和“files.o”将会被重新编译，之后同样终极目标“edit”也将被重新生成。

以上我们通过一个简单的例子，介绍了 Makefile 中目标和依赖的关系。对于 Makefile 中的目标。在执行“make”时首先执行终极目标所在的规则，接下来一层层地去寻找终极目标的依赖文件所在的规则并执行。当终极目标的规则被完全的展开以后，make 将从最后一个被展开的规则处开始执行，之后处理倒数第二个规则，……依次回退。最后一步执行的就是终极目标所在的规则。整个过程就类似于 C 语言中的递归实现一样。在更新（或者创建）终极目标的过程中，如果出现错误 make 就立即报错并退出。整个过程 make 只是负责执行规则，而对具体规则所描述的依赖关系的正确性、规则所定义的命令的正确性不做任何判断。就是说，一个规则的依赖关系是否正确、描述重建目标的规则命令行是否正确，make 不做任何错误检查。

因此，需要正确的编译一个工程。需要在提供给 make 程序的 Makefile 中来保证其依赖关系的正确性、和执行命令的正确性。

1.6 总结

make 的执行过程如下:

1. 依次读取变量“MAKEFILES”定义的 makefile 文件列表
2. 读取工作目录下的 makefile 文件（根据命名的查找顺序“GNUmakefile”，“makefile”，“Makefile”，首先找到那个就读取那个）
3. 依次读取工作目录 makefile 文件中使用指示符“include”包含的文件
4. 查找重建所有已读取的 makefile 文件的规则（如果存在一个目标是当前读取的某一个 makefile 文件，则执行此规则重建此 makefile 文件，完成以后从第一步开始重新执行）

5. 初始化变量值并展开那些需要立即展开的变量和函数并根据预设条件确定执行分支
6. 根据“终极目标”以及其他目标的依赖关系建立依赖关系链表
7. 执行除“终极目标”以外的所有的目标的规则(规则中如果依赖文件中任一个文件的时间戳比目标文件新，则使用规则所定义的命令重建目标文件)
8. 执行“终极目标”所在的规则

Chapter 2. KBUILD MAKE 原理介绍

2.1 概述

Linux 内核的 Makefile 分为 5 个部分:

Makefile	顶层 Makefile
.config	内核配置文件
arch/$(ARCH)/Makefile	具体架构的 Makefile
scripts/Makefile.*	通用的规则等，面向所有的 Kbuild Makefiles。
kbuild Makefiles	内核源代码中大约有 500 个这样的文件

顶层 Makefile 阅读的.config 文件，而该文件是由内核配置程序生成的。顶层 Makefile 负责制作: vmlinux(内核文件)与模块(任何模块文件)。制作的过程主要是通过递归向下访问子目录的形式完成。并根据内核配置文件确定访问哪些子目录。顶层 Makefile 要原封不动的包含一具体架构的 Makefile，其名字类似于 arch/$(ARCH)/Makefile。该架构 Makefile 向顶层 Makefile 提供其架构的特别信息。

每一个子目录都有一个 Kbuild Makefile 文件，用来执行从其上层目录传递下来的命令。Kbuild Makefile 从.config 文件中提取信息，生成 Kbuild 完成内核编译所需的文件列表。

scripts/Makefile.*包含了所有的定义、规则等信息。这些文件被用来编译基于 kbuild Makefile 的内核。

2.2 Kbuild 文件

大部分内核中的 Makefile 都是使用 Kbuild 组织结构的 Kbuild Makefile。这章介绍了 Kbuild Makefile 的语法。Kbuild 文件倾向于"Makefile"这个名字，"Kbuild"也是可以用的。但如果"Makefile" "Kbuild"同时出现的话，使用的将会是"Kbuild"文件。

3.1 节 目标定义是一个快速介绍，以后的几章会提供更详细的内容以及实例。

2.2.1 目标定义

目标定义是 Kbuild Makefile 的主要部分，也是核心部分。主要是定义了要编译的文件，所有的

选项，以及到哪些子目录去执行递归操作。

最简单的 Kbuild makefile 只包含一行:

```
obj-y += foo.o
```

该例子告诉 Kbuild 在这目录里，有一个名为 foo.o 的目标文件。foo.o 将从 foo.c 或 foo.S 文件编译得到。

如果 foo.o 要编译成一模块，那就要用 obj-m 了。所采用的形式如下:

```
obj-$(CONFIG_FOO) += foo.o
```

$(CONFIG_FOO)可以为 y(编译进内核) 或 m(编译成模块)。如果 CONFIG_FOO 不是 y 和 m，那么该文件就不会被编译联接了。

2.2.2 编译进内核 - obj-y

Kbuild Makefile 规定所有编译进内核的目标文件都存在$(obj-y)列表中。而这些列表依赖内核的配置。

Kbuild 编译所有的$(obj-y)文件。然后，调用"$(LD) -r"将它们合并到一个 build-in.o 文件中。稍后，该 build-in.o 会被其父 Makefile 联接进 vmlinux 中。

$(obj-y)中的文件是有顺序的。列表中有重复项是可以的：当第一个文件被联接到 built-in.o 中后，其余文件就被忽略了。

联接也是有顺序的，那是因为有些函数(module_init()/__initcall)将会在启动时按照他们出现的顺序进行调用。所以，记住改变联接的顺序可能改变你 SCSI 控制器的检测顺序，从而导致你的硬盘数据损害。

```
#drivers/isdn/i4l/Makefile
# Makefile for the kernel ISDN subsystem and device drivers.
# Each configuration option enables a list of files.
obj-$(CONFIG_ISDN) += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
```

2.2.3 编译可装载模块 - obj-m

$(obj-m) 列举出了哪些文件要编译成可装载模块。

一个模块可以由一个文件或多个文件编译而成。如果是一个源文件，KbuildMakefile 只需简单的将其加到$(obj-m)中去就可以了。

```
#drivers/isdn/i4l/Makefile
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
```

注意：此例中 $(CONFIG_ISDN_PPP_BSDCOMP) 的值为'm'

如果内核模块是由多个源文件编译而成，那你就要采用上面那个例子一样的方法去声明你所要编译的模块。

Kbuild 需要知道你所编译的模块是基于哪些文件，所以你需要通过变量$(<module_name>-objs)来告诉它。

```
#drivers/isdn/i4l/Makefile
obj-$(CONFIG_ISDN) += isdn.o
```

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

在这个例子中，模块名将是 isdn.o,Kbuild 将编译在$(isdn-objs)中列出的所有文件，然后使用"$(LD) -r"生成 isdn.o。

Kbuild 能够识别用于组成目标文件的后缀-objs 和后缀-y。这就让 KbuildMakefile 可以通过使用 CONFIG_ 符号来判断该对象是否是用来组合对象的。

```
#fs/ext2/Makefile
obj-$(CONFIG_EXT2_FS) += ext2.o
ext2-y    := balloc.o bitmap.o
ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o
```

在这个例子中，如果 $(CONFIG_EXT2_FS_XATTR) 是 'y',xattr.o 将是复合对象 ext2.o 的一部分。

注意: 当然，当你要将其编译进内核时，上面的语法同样适用。所以，如果你的 CONFIG_EXT2_FS=y，那 Kbuild 会按你所期望的那样，生成 ext2.o 文件，然后将其联接到 built-in.o 中。

2.2.4 目标库文件 - lib-y

在 obj-* 中所列文件是用来编译模块或者是联接到特定目录中的 built-in.o。同样，也可以列出一些将被包含在 lib.a 库中的文件。在 lib-y 中所列出的文件用来组成该目录下的一个库文件。

在 obj-y 与 lib-y 中同时列出的文件，因为都是可以访问的，所以该文件是不会被包含在库文件中的。同样的情况， lib-m 中的文件就要包含在 lib.a 库文件中。

注意，一个 Kbuild makefile 可以同时列出要编译进内核的文件与要编译成库的文件。所以，在一个目录里可以同时存在 built-in.o 与 lib.a 两个文件。

```
#arch/i386/lib/Makefile
lib-y := chechsum.o delay.o
```

这将由 checksum.o 和 delay.o 两个文件创建一个库文件 lib.a。为了让 Kbuild 真正认识到这里要有一个库文件 lib.a 要创建，其所在的目录要加到 libs-y 列表中。还可参考"6.3 递归下向时要访问的目录列表" lib-y 使用一般限制在 lib/ 和 arch/*/lib 中。

2.2.5 递归向下访问目录

一个 Makefile 只对编译所在目录的对象负责。在子目录中的文件的编译要由其所在的子目录的 Makefile 来管理。只要你让 Kbuild 知道它应该递归操作，那么该系统就会在其子目录中自动的调用 make 递归操作。

这就是 obj-y 和 obj-m 的作用。

ext2 被放的一个单独的目录下，在 fs 目录下的 Makefile 会告诉 Kbuild 使用下面的赋值进行向下递归操作。

```
#fs/Makefile
obj-$(CONFIG_EXT2_FS) += ext2/
```

如果 CONFIG_EXT2_FS 被设置为 'y'(编译进内核)或是'm'(编译成模块)，相应的 obj- 变量就会被设置，并且 Kbuild 就会递归向下访问 ext2 目录。Kbuild 只是用这些信息来决定它是否需要访问该目录，而具体怎么编译由该目录中的 Makefile 来决定。

将 CONFIG_ 变量设置成目录名是一个好的编程习惯。这让 Kbuild 在完全忽略那些相应的 CONFIG_ 值不是'y'和'm'的目录。

2.2.6 编辑标志

EXTRA_CFLAGS, EXTRA_AFLAGS, EXTRA_LDFLAGS, EXTRA_ARFLAGS

所有的 EXTRA_ 变量只在所定义的 Kbuild Makefile 中起作用。EXTRA_ 变量可以在 Kbuild Makefile 中所有命令中使用。

$(EXTRA_CFLAGS) 是用 $(CC) 编译 C 源文件时的选项。

```
# drivers/sound/emu10kl/Makefile
EXTRA_CFLAGS += -I$(obj)
ifdef DEBUG
    EXTRA_CFLAGS += -DEMU10KL_DEBUG
endif
```

该变量是必须的，因为顶层 Makefile 拥有变量 $(CFLAGS) 并用来作为整个源代码树的编译选项。

$(EXTRA_AFLAGS) 也是一个针对每个目录的选项，只不过它是用来编译汇编源代码的。

```
#arch/x86_64/kernel/Makefile
EXTRA_AFLAGS := -traditional
```

$(EXTRA_LDFLAGS) 和 **$(EXTRA_ARFLAGS)**分别与 $(LD)和 $(AR)类似，只不过，他们是针对每个目录的。

```
#arch/m68k/fpsp040/Makefile
EXTRA_LDFLAGS := -x
```

CFLAGS_$@, AFLSGA_$@

CFLAGS_$@ 和 AFLAGS_$@ 只能在当前 Kbuild Makefile 中的命令中使用。

$(CFLAGS_$@) 是 $(CC) 针对每个文件的选项。$@ 表明了具体操作的文件。

```
# drivers/scsi/Makefile
CFLAGS_aha152x.o =  -DAHA152X_STAT -DAUTOCONF
CFLAGS_gdth.o    = # -DDEBUG_GDTH=2 -D__SERIAL__ -D__COM2__ \
-DGDTH_STATISTICS
CFLAGS_seagate.o =  -DARBITRATE -DPARITY -DSEAGATE_USE_ASM
```

以上三行分别设置了 aha152x.o,gdth.o 和 seagate.o 的编辑选项。

$(AFLAGS_$@) 也类似，只不是是针对汇编语言的。

```
# arch/arm/kernel/Makefile
AFLAGS_head-armv.o := -DTEXTADDR=$(TEXTADDR) -traditional
AFLAGS_head-armo.o := -DTEXTADDR=$(TEXTADDR) -traditional
```

2.2.7 跟踪依赖

Kbuild 跟踪在以下方面依赖:

1) 所有要参与编译的文件(所有的.c 和.h 文件)

2) 在参与编译文件中所要使用的 CONFIG_ 选项

3) 用于编译目标的命令行

因此，如果你改变了 $(CC) 的选项，所有受影响的文件都要重新编译。

2.2.8 特殊规则

特殊规则就是那 Kbuild 架构不能提供所要求的支持时，所使用的规则。一个典型的例子就是在构建过程中生成的头文件。另一个例子就是那些需要采用特殊规则来准备启动镜像。

特殊规则的写法与普通 Make 规则一样。Kbuild 并不在 Makefile 所在的目录执行，所以所有的特殊规则都要提供参与编译的文件和目标文件的相对路径。

在定义特殊规则时，要使用以下两个变量:

$(src)

$(src) 表明 Makefile 所在目录的相对路径。经常在定位源代码树中的文件时，使用该变量。

$(obj)

$(obj) 表明目标文件所要存储目录的相对路径。经常在定位所生成的文件时，使用该变量。

```
#drivers/scsi/Makefile
$(obj)/53c8xx_d.h: $(src)/53c7,8xx.scr $(src)/script_asm.pl
$(CPP) -DCHIP=810 - < $< | ... $(src)/script_asm.pl
```

这就是一个特殊规则，遵守着 make 所要求的普通语法。目标文件依赖于两个源文件。用$(obj)来定位目标文件，用$(src)来定位源文件(因为它们不是我们生成的文件)。

2.2.9 $(CC) 支持的函数

内核可能由多个不同版本的$(CC)编译，而每个版本都支持一不同的功能集与选项集。Kbuild 提供了检查 $(CC) 可用选项的基本功能。$(CC)一般情况下是 gcc 编译器，但也可以使用其它编译器来代替 gcc。

as-option

as-option，当编译汇编文件(*.S)时，用来检查 $(CC) 是否支持特定选项。如果第一个选项不支持的话，可选的第二个选项可以用来指定。

```
#arch/sh/Makefile
cflags-y += $(call as-option,-Wa$(comma)-isa=$(isa-y),)
```

在上面的例子里，如果 $(CC) 支持选项 -Wa$(comma)-isa=$(isa-y)，cflags-y 就会被赋予该值。第二个参数是可选的，当第一个参数不支持时，就会使用该值。

ld-option

ld-option，当联接目标文件时，用来检查 $(CC) 是否支持特定选项。如果第

一个选项不支持的话，可选的第二个选项可以用来指定。

```
#arch/i386/kernel/Makefile
vsyscall-flags += $(call ld-option, -Wl$(comma)--hash-style=sysv)
```

在上面的例子中，如果 $(CC)支持选项 -Wl$(comma)--hash-style=sysv，ld-option 就会被赋予该值。第二个参数是可选的，当第一个参数不支持时，就会使用该值。

cc-option

cc-option，用来检查 $(CC) 是否支持特定选项,并且不支持使用可选的第二项。

```
#arch/i386/Makefile
cflags-y += $(call cc-option,-march=pentium-mmx,-march=i586)
```

在上面的例子中，如果 $(CC)支持选项 -march=pentium-mmx，cc-option 就会被赋予该值，否则就赋 -march-i586。cc-option 的第二个参数是可选的。如果忽略的话，当第一个选项不支持时，cflags-y 不会被赋值。

cc-option-yn

cc-option-yn,用来检查 gcc 是否支持特定选项，返回'y'支持，否则为'n'。

```
#arch/ppc/Makefile
biarch  := $(call cc-option-yn, -m32)
aflags-$(biarch) += -a32
cflags-$(biarch) += -m32
```

在上面的例子里，当 $(CC) 支持 -m32 选项时，$(biarch)设置为 y。当 $(biarch) 为 y 时，扩展的 $(aflags-y) 和 $(cflags-y)变量就会被赋值为-a32 和 -m32。

cc-option-align

gcc 版本大于 3.0 时，改变了函数，循环等用来声明内存对齐的选项。当用到对齐选项时，$(cc-option-align) 用来选择正确的前缀:

gcc < 3.00

cc-option-align = -malign

gcc >= 3.00

cc-option-align = -falign

例子:

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

在上面的例子中，选项 -falign-funcions=4 被用在 gcc >= 3.00 的时候。对于小于 3.00 时， 使用 -malign-funcions=4 。

cc-version

cc-version 以数学形式返回 $(CC) 编译器的版本号。其格式是: <major><minor>，二者都是数学。比如，gcc 3.41 会返回 0341。当某版本的 $(CC) 在某方面有缺陷时,cc-version 就会很有用。比如，选项-mregparm=3 虽然会被 gcc 接受，但其实现是有问题的。

```
#arch/i386/Makefile
cflags-y += $(shell \
if [ $(call cc-version) -ge 0300 ] ; then \
echo "-meregparm=3";
 fi ;
```

在上面的例子中，-mregparm=3 只会在 gcc 的版本号大于等于 3.0 的时候使用。

cc-ifversion

cc-ifversion 测试 $(CC) 的版本号，如果版本表达式为真，就赋值为最后的参数。

```
#fs/reiserfs/Makefile
EXTRA_CFLAGS := $(call cc-ifversion, -lt, 0402, -O1)
```

在这个例子中，如果 $(CC) 的版本小于 4.2,EXTRA_CFLAGS 就被赋值 -O1。cc－ifversion 可使用所有的 shell 操作符：-eq,-ne,-lt,-le,-gt,和-ge。第三个参数可以像上面例子一样是个文本，但也可以是个扩展的变量或宏。

2.3 本机程序支持

Kbuild 支持编译那些将在编译阶段使用的可执行文件。为了使用该可执行文件，要将编译分成二个阶段。

第一阶段是告诉 Kbuild 存在哪些可执行文件。这是通过变量 hostprogs-y 来完成的。

第二阶段是添加一个对可执行文件的显性依赖。有两种方法:增加依赖关系到一个规则中，或是利用变量 $(always)。以下是详细叙述.

2.3.1 单的本机程序

在编译内核时，有时会需要编译并运行一个程序。下面这行就告诉了 kbuild，程序 bin2hex 应该在本机上编译。

```
hostprogs-y := bin2hex
```

在上面的例子中，Kbuild 假设 bin2hex 是由一个与其在同一目录下，名为 bin2hex.c 的 C 语言源文件编译而成的。

2.3.2 复合的本机程序

本机程序可以由多个文件编译而成。所使用的语法与内核的相应语法很相似。$(<executeable>-objs) 列出了联接成最后的可执行文件所需的所有目标文件。

```
#scripts/lxdialog/Makefile
hostprogs-y := lxdialog
lxdialog-objs := checklist.o lxdialog.o
```

扩展名为.o 的文件是从相应的.c 文件编译而来的。在上面的例子中，checklist.c 编译成了 checklist.o，lxdialog.c 编译成了 lxdialog.o。最后，两个.o 文件联接成了一可执行文件，lxdialog。

注意：语法 <executable>-y 不是只能用来生成本机程序。

2.3. 定义共享库

扩展名为 so 的文件称为共享库，被编译成位置无关对象。Kbuild 也支持共享库，但共享库的使用很有限。在下面的例子中，libconfig.so 共享库用来联接到可执行文件 conf 中。

```
#scripts/kconfig/Makefile
hostprogs-y := conf
conf-objs := conf.o libkconfig.so
libkcofig-objs := expr.o type.o
```

共享库文件经常要求一个相应的 -objs，在上面的例子中，共享库 libkconfig 是由 expr.o 和 type.o 两个文件组成的。expr.o 和 type.o 将被编译成位置无关码，然后联接成共享库文件 libkconfig.so。C++并不支持共享库。

2.3.4 使用用 C++编写的本机程序

kbuild 也支持用 C++编写的本机程序。在此专门介绍是为了支持 kconfig，并且在一般情况下不推荐使用。

```
#scripts/kconfig/Makefile
hostprogs-y := qconf
qconf-cxxobjs := qconf.o
```

在上面的例子中，可执行文件是由 C++文件 qconf.cc 编译而成的，由$(qconf-cxxobjs)来标识。

如果 qconf 是由.c 和.cc 一起编译的，那么就需要专门来标识这些文件了。

```
#scripts/kconfig/Makefile
hostprogs-y := qconf
qconf-cxxobjs := qconf.o
qconf-objs := check.o
```

2.3.5 控制本机程序的编译选项

当编译本机程序时，有可能使用到特殊选项。程序经常是利用$(HOSTCC)编译，其选项在 $(HOSTCFLAGS)变量中。可通过使用变量 HOST_EXTRACFLAGS，影响所有在 Makefile 文件中要创建的主机程序。

```
#scripts/lxdialog/Makefile
HOST_EXTRACFLAGS += -I/usr/include/ncurses
```

为一单个文件设置选项，可按形式进行:

```
#arch/ppc64/boot/Makefile
HOSTCFLAGS_pinggyback.o := -DKERNELBASE=$(KERNELBASE)
```

同样也可以给联接器声明一特殊选项。

```
#scripts/kconfig/Makefile
HOSTLOADLIBES_qconf := -L$(QTDIR)/lib
```

当联接 qconf 时，将会向联接器传递附加选项 "-L$(QTDIR)/lib"。

2.3.6 编译主机程序时

Kbuild 只在需要时编译主机程序。有两种方法:

(1) 在一具体的规则中显性列出所需要的文件，例子:

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

目标 $(obj)/devlist.h 是不会在 $(obj)/gen-devlist 更新之前编译的。注意在该规则中所有有关主机程序的命令必须以$(obj)开头。

(2) 使用 $(always)

当 Makefile 要编译主机程序，但没有适合的规则时，使用 $(always)。例子:

```
#scripts/lxdialog/Makefile
hostprogs-y := lxdialog
always := $(hostprogs-y)
```

这就是告诉 Kbuild，即使没有在规则中声明，也要编译 lxdialog。

2.3.7 使用 hostprogs-$(CONFIG_FOO)

一个典型的 Kbuild 模式如下:

```
#scripts/Makefile
hostprogs-$(CONFIG_KALLSYMS) += kallsyms
```

Kbuild 知道 'y' 是编译进内核，而 'm' 是编译成模块。所以，如果配置符号是'm'，Kbuild 仍然会编译它。换句话说，Kbuild 处理 hostprogs-m 与 hostprogs-y 的方式是完全一致的。只是，如果不用 CONFIG，最好用 hostprogs-y。

2.4 build 清理(clean)

"make clean"删除几乎所有的在编译内核时生成的文件，包括了主机程序在内。

Kbuild 通过列表 $(hostprogs-y)，$(hostprogs-m)，$(always)，$(extra-y) 和$(targets) 知道所要编译的目标。这些目标文件都会被 "make clean" 删除。另外，在"make clean"还会删除匹配 "*.[oas]"，"*.ko" 的文件，以及由 Kbuild 生成的辅助文件。辅助文件由 Kbuild Makefile 中的 $(clean-files) 指明。例子:

```
#drivers/pci/Makefile
clean-files  := devlist.h classlist.h
```

当执行 "make clean" 时，"devlist.h classlist.h"这两个文件将被删除。如果不使用绝对路径(路径以'/'开头)的话，Kbuild 假设所要删除的文件与 Makefile 在同一个相对路径上。

要删除一目录:

```
#scripts/package/Makefile
clean-dirs := $(objtree)/debian/
```

这就会删除目录 debian，包括其所有的子目录。如果不使用绝对路径(路径以'/'开头)的话，Kbuild 假设所要删除的目录与 Makefile 在同一个相对路径上。

一般情况下，Kbuild 会根据 "obj-* := dir/" 递归访问其子目录，但有的时候，Kbuild 架构还不足以描述所有的情况时，还要显式的指明所要访问的子目录。例子:

```
#arch/i386/boot/Makefile
subdir-  := compressed/
```

上面的赋值命令告诉 Kbuild，当执行"make clean"时，要递归访问目录 compressed/。

为了支持在最终编译完成启动镜像后的架构清理工作，还有一可选的目标 archclean:

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

当"make clean"执行时，make 会递归访问并清理 arch/i386/boot。在 arch/i386/boot 中的 Makefile 可以用来提示 make 进行下一步的递归操作。

注意 1: arch/$(ARCH)/Makefile 不能使用"subdir-"，因为该 Makefile 被包含在顶层的 Makefile 中，Kbuild 是不会在此处进行操作的。

注意 2: "make clean" 会访问在 core-y，libs-y，drivers-y 和 net-y 列出的所有目录。

2.5 架构 Makefile

在递归访问目录之前，顶层 Makefile 要完成设置环境变量以及递归访问的准备工作。顶层 Makefile 包含的公共部分，而 arch/$(ARCH)/Makefile 包含着针对某一特定架构的配置信息。所以，要在 arch/$(ARCH)/Makefile 中设置一部分变量，并定义一些目标。

Kbuild 执行的几个步骤(大致):

1) 根据内核配置生成文件 .config

2) 将内核的版本号存储在 include/linux/version.h

3) 生成指向 include/asm-$(ARCH) 的符号链接

4) 更新所有编译所需的文件: 附加的文件由 arch/$(ARCH)/Makefile 指定。

5) 递归向下访问所有在下列变量中列出的目录: init-* core* drivers-* net-* libs-*，并编译生成目标文件。这些变量的值可以在 arch/$(ARCH)/Makefile 中扩充。

6) 联接所有的目标文件，在源代码树顶层目录中生成 vmlinux。最先联接是在 head-y 中列出的文件，该变量由 arch/$(ARCH)/Makefile 赋值。

7) 最后完成具体架构的特殊要求，并生成最终的启动镜像。

-- 包含生成启动指令

-- 准备 initrd 镜像或类似文件

2.5.1 调整针对某一具体架构生成的镜像

LDFLAGS 一般是 $(LD) 选项

该选项在每次调用联接器时都会用到。

一般情况下，只用来指明模拟器。例子:

```
#arch/s390/Makefile
LDFLAGS := -m elf_s390
```

注意：EXTRA_LDFLAGS 和 LDFLAGS_$@ 可用来进一步自定义选项。

LDFLAGS_MODULE 联接模块时的联接器的选项

LDFLAGS_MODULE 所设置的选项将在联接器在联接模块文件 .ko 时使用。默认值为 "-r"，指定输出文件是可重定位的。

LDFLAGS_vmlinux 联接 vmlinux 时的选项

LDFLAGS_vmlinux 用来传递联接 vmlinux 时的联接器的选项。

LDFLAGS_vmlinux 需 LDFLAGS_$@ 支持。例子:

```
#arch/i386/Makefile
LDFLAGS_vmlinux := -e stext
```

OBJCOPYFLAGS objcopy 选项

当用 $(call if_changed,objcopy) 来转换(translate)一个.o 文件时，该选项就会被使用。

$(call if_changed,objcopy) 经常被用来为 vmlinux 生成原始的二进制代码。例子:

```
#arch/s390/Makefile
OBJCOPYFLAGS := -O binary
#arch/s390/boot/Makefile
$(obj)/image: vmlinux FORCE
$(call if_changed,objcopy)
```

在此例中，二进制文件 $(obj)/image 是 vmlinux 的一个二进制版本。

$(call if_chagned,xxx)的用法稍后描述。

AFLAGS $(AS) 汇编编译器选项

默认值在顶层 Makefile

扩充或修改在各具体架构的 Makefile，例子:

```
#arch/sparc64/Makefile
AFLAGS += -m64 -mcpu=ultrasparc
```

CFLAGS $(CC) 编译器选项

默认值在顶层 Makefile

扩充或修改在各具体架构的 Makefile。

一般，CFLAGS 要根据内核配置设置。例子:

```
#arch/i386/Makefile
cflags-$(CONFIG_M386)  += -march=i386
CFLAGS += $(cflags-y)
```

许多架构 Makefile 都通过调用所要使用的 C 编译器，动态的检查其所支持的选项:

```
#arch/i386/Makefile
...
cflags-$(CONFIG_MPENTIUMII) += $(call cc-option,\
-march=pentium2,-march=i686)
...
# Disable unit-at-a-time mode ...
CFLAGS += $(call cc-option,-fno-unit-at-a-time)
...
```

第一个例子利用了一个配置选项，当其为'y'时，扩展。

```
CFLAGS_KERNEL :
#arch/i386/Makefile
...
cflags-$(CONFIG_MPENTIUMII) += $(call cc-option,\
-march=pentium2,-march=i686)
...
# Disable unit-at-a-time mode ...
CFLAGS += $(call cc-option,-fno-unit-at-a-time)
...
```

第一个例子利用了一个配置选项，当其为'y'时，扩展。

CFLAGS_KERNEL 编译进内核时，$(CC) 所用的选项

$(CFLAGS_KERNEL) 包含了用于编译常驻内核代码的附加编译器选项。

CFLAGS_MODULE 编译成模块时，$(CC)所用的选项

$(CFLAGS_MODULE) 包含了用于编译可装载模块的附加编译器选项。

2.5.2 将所需文件加到 archprepare 中:

archprepare 规则在递归访问子目录之前，列出编译目标文件所需文件。

一般情况下，这是一个包含汇编常量的头文件。(assembler constants)

例子:

```
#arch/arm/Makefile
archprepare: maketools
```

此例中，目标文件 maketools 将在递归访问子目录之前编译。在 TODO 一章可以看到，Kbuild 是如何支持生成分支头文件的。(offset header files)

2.5.3 递归下向时要访问的目录列表

如何生成 vmlinux，是由架构 makefile 和顶层 Makefile 一起来定义的。注意，架构 Makefile 是不会定义与模块相关的内容的，所有构建模块的定义是与架构无关的。

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

$(head-y) 列出了最先被联接进 vmlinux 的目标文件。

$(libs-y) 列出了生成的所有 lib.a 所在的目录。

其余所列的目录，是 built-in.o 所在的目录。

$(init-y) 在 $(head-y) 之后所要使用的文件。

然后，剩下的步骤如下:

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

顶层 makefile 定义了通用的部分，arch/$(ARCH)/Makefile 添加了架构的特殊要求。

例子:

```
#arch/sparc64/Makefile
core-y += arch/sparc64/kernel/
libs-y += arch/sparc64/prom/ arch/sparc64/lib/
drivers-$(CONFIG_OPROFILE)  += arch/sparc64/oprofile/
```

2.5.4 具体架构的启动镜像

一具体架构 Makefile 的具体目的就是，将生成并压缩 vmlinux 文件，写入启动代码，并将其拷贝到正确的位置。这就包含了多种不同的安装命令。该具体目的也无法在各个平台间进行标准化。

一般，附加的处理命令入在 arch/$(ARCH)/下的 boot 目录。

Kbuild 并没有为构造 boot 所指定的目标提供任何更好的方法。所以，arch/$(ARCH)/Makefile 将会调用 make 以手工构造 boot 的目标文件。

比较好的方法是，在 arch/$(ARCH)/Makefile 中包含快捷方式，并在 arch/$(ARCH)/boot/Makefile 中使用全部路径。例子:

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

当在子目录中调用 make 时，推荐使用 "(Q)(MAKE) $(build)=<dir>" 。

并没有对架构特殊目标的命名规则，但用命令 "make help" 可以列出所有的相关目标。

为了支持 "make help"，$(archhelp) 必须被定义。例子:

```
#arch/i386/Makefile
define archhelp
    echo  '* bzImage - Image (arch/$(ARCH)/boot/bzImage)'
endef
```

当 make 没带参数执行时，所遇到的第一个目标将被执行。在顶层，第一个目标就是 all:

每个架构 Makefile 都要默认构造一可启动的镜像文件。

在 "make help"中，默认目标就是被加亮的'*'。

添加一新的前提文件到 all:，就可以构造出一不同的 vmlinux。例子:

```
#arch/i386/Makefile
```

```
all: bzImage
```

当 make 没有参数时，bzImage 将被构造。

2.5.5 构造非 Kbuild 目标

extra-y

extra-y 列出了在当前目录下，所要创建的附加文件，不包含任何已包含在 obj-* 中的文件。

用 extra-y 列目标，主要是两个目的:

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

例子:

```
#arch/i386/kernel/Makefile
extra-y := head.o init_task.o
```

在此例子中，extra-y 用来列出所有只编译，但不联接到 built-in.o 的目标文件。

2.5.6 构建启动镜像的命令

Kbuild 提供了几个用在构建启动镜像时的宏。

if_changed

if_changed 为下列命令的基础。使用方法:

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

当执行该规则时，就检查是否有文件需要更新，或者在上次调用以后，命令行发生了改变。如果有选项发生了改变，后者会导致重新构造。只有在 $(targets)列出的的目标文件，才能使用 if_changed，否则命令行的检查会失败，并且目标总会被重建。给 $(targets)的赋值没有前缀 $(obj)/ 。 if_changed 可用来联接自定义的 Kbuild 命令，关于 Kbuild 自定义命令请看 6.7 节。

注意：忘记 FORCE 是一种典型的错误。还有一种普遍的错误是，空格有的时候是有意义的;比如。下面的命令就会错误(注意在逗号后面的那个多余的空格):

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

ld 联接目标

经常是使用 LDFLAGS_$@来设置 ld 的特殊选项。

objcopy

拷贝二进制代码。一般是在 arch/$(ARCH)/Makefile 中使用 OBJCOPYFLAGS。

OBJCOPYFLAGS_$@ 可以用来设置附加选项。

gzip

压缩目标文件。尽可能的压缩目标文件。例子:

#arch/i386/boot/Makefile

```
LDFLAGS_bootsect  := -Ttext 0x0 -s --oformat binary
LDFLAGS_setup    := -Ttext 0x0 -s --oformat binary -e begtext
targets += setup setup.o bootsect bootsect.o
$(obj)/setup $(obj)/bootsect: %: %.o FORCE
$(call if_changed,ld)
```

在这个例子中，有两个可能的目标文件，分别要求不同的联接选项。定义联接器的选项使用的是 LDFLAGS_$@ 语法，每个潜在的目标一个。$(targets) 被分配给所有的潜在目标，因此知道目标是哪些，并且还会:

1) 检查命令行是否改变

2) 在 "make clean" 时，删除目标文件前提部分中的 ": %: %.o" 部分使我们不必在列出文件 setup.o 和

bootsect.o 。

注意：一个普遍的错误是忘记了给 "target"赋值，导致在 target 中的文件总是无缘无故的被重新编译。

2.5.7 Kbuild 自定义命令

当 Kbuild 的变量 KBUILD_VERBOSE 为 0 时，只会显示命令的简写。

如果要为自定义命令使用这一功能，需要设置 2 个变量:

quiet_cmd_<command> - 要显示的命令

cmd_<command> - 要执行的命令

例子:

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

当用"make KBUILD_VERBOSE=0"更新 $(obj)/bzImage 目标时显示:

BUILD arch/i386/boot/bzImage

2.5.8 联接器预处理脚本

当构造 vmlinux 镜像时，使用联接器脚本: arch/$(ARCH)/kernel/vmlinux.lds

该脚本是由在同一目录下的 vmlinux.lds.S 生成的。

Kbuild 认识.lds 文件，并包含由*.lds.S 文件生成*.lds 文件的规则。例子:

```
#arch/i386/kernel/Makefile
```

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

$(always)的值是用来告诉 Kbuild，构造目标 vmlinux.lds。

$(CPPFLAGS_vmlinux.lds)，Kbuild 在构造目标 vmlinux.lds 时所用到的特殊选项。

当构造 *.lds 目标时，Kbuild 要用到下列变量:

CPPFLAGS：在顶层目录中设置

EXTRA_CPPFLAGS：可以在 Kbuild Makefile 中设置

CPPFLAGS_$(@F)：目标特别选项

注意，此处的赋值用的完整的文件名。

针对*.lds 文件的 Kbuild 构架还被用在许多具体架构的文件中。(***不通***)

2.6 Kbuild 变量

顶层 Makefile 输出以下变量:

VERSION, PATCHLEVEL, SUBLEVEL, EXTRAVERSION

这些变量定义了当前内核的版本号。只有很少一部分 Makefile 会直接用到这些变量；可使用$(KERNELRELEASE)代替。

$(VERSION),$(PATCHLEVEL),和$(SUBLEVEL) 定义了最初使用的三个数字的版本号，比如"2""4"和"0"。这三个值一般是数字。

$(EXTRAVERSION) 为了补丁定义了更小的版本号。一般是非数字的字符串，比如"-pre4"，或就空着。

KERNELRELEASE

$(KERNELRELEASE) 是一个字符串，类似"2.4.0-pre4"，用于安装目录的命名或
显示当前的版本号。一部分架构 Makefile 使用该变量。

ARCH

该变量定义了目标架构，比如"i386","arm" 或"sparc"。有些 Kbuild Makefile 根据 $(ARCH) 决定编译哪些文件。默认情况下，顶层 Makefile 将其设置为本机架构。如果是跨平台编译，用户可以用下面的命令覆盖该值:

make ARCH=m68k ...

INSTALL_PATH

该变量为架构 Makefile 定义了安装内核镜像与 System.map 文件的目录。主要用来指明架构特殊的安装路径。

INSTALL_MOD_PATH,MODLIB

$(INSTALL_MOD_PATH) 为了安装模块，给 $(MODLIB) 声明了前缀。该变量不能在 Makefile

中定义，但可以由用户传给 Makefile。

$(MODLIB) 具体的模块安装的路径。顶层 Makefile 将 $(MODLIB) 定义为 $(INSTALL_MOD_PATH)/lib/modules/$(KERNELRELEASE)。用户可以通过命令行参数的形式将其覆盖。

INSTALL_MOD_STRIP

如果该变量有定义，模块在安装之前，会被剥出符号表。如果 INSTALL_MOD_STRIP 为 "1"，就使用默认选项 --strip-debug。否则，INSTALL_MOD_STRIP 将作为命令 strip 的选项使用。

2.7 Makefile 语言

内核的 Makefile 使用的是 GNU Make。该 Makefile 只使用 GNU Make 已注明的功能，并使用了许多 GNU 的扩展功能。GNU Make 支持基本的显示处理过程的函数。内核 Makefile 使用了一种类似小说的方式，显示"if"语句的构造、处理过程。GNU Make 有 2 个赋值操作符，":="和"="。":="，将对右边的表达式求值，并将所求的值赋给左边。"="更像是一个公式定义，只是将右边的值简单的赋值给左边，当左边的表达式被使用时，才求值。有时使用"="是正确的。但是，一般情况下，推荐使用":="。