China 搜索: 在 dW 全部内容 搜 索
只用 + - " " ( )
IBM 主页 | 产品与服务 | 支持与下载 | 个性化服务
developerWorks 中国网站 : Linux : 所有的文章 developerWorks

Linux 内核配置系统浅析

汤凯 (tangk73@hotmail.com)

2003 年 2 月

内容：

配置系统的基本结构
Makefile
配置文件
实例
参考资料
关于作者

在 Linux 专区还有：

教程
工具与产品
代码与组件
文章

随着 Linux 操作系统的广泛应用，特别是 Linux 在嵌入式领域的发展，越来越多的人开始投身到 Linux 内核级的开发中。面对日益庞大的 Linux 内核源代码，开发者在完成自己的内核代码后，都将面临着同样的问题，即如何将源代码融入到 Linux 内核中，增加相应的 Linux 配置选项，并最终被编译进 Linux 内核。这就需要了解 Linux 的内核配置系统。

众所周知，Linux 内核是由分布在全球的 Linux 爱好者共同开发的，Linux 内核每天都面临着许多新的变化。但是，Linux 内核的组织并没有出现混乱的现象，反而显得非常的简洁，而且具有很好的扩展性，开发人员可以很方便的向 Linux 内核中增加新的内容。原因之一就是 Linux 采用了模块化的内核配置系统，从而保证了内核的扩展性。

本文首先分析了 Linux 内核中的配置系统结构，然后，解释了 Makefile 和配置文件的格式以及配置语句的含义，最后，通过一个简单的例子--TEST Driver，具体说明如何将自行开发的代码加入到 Linux 内核中。在下面的文章中，不可能解释所有的功能和命令，只对那些常用的进行解释，至于那些没有讨论到的，请读者参考后面的参考文献。

1． 配置系统的基本结构

Linux内核的配置系统由三个部分组成，分别是：

1. Makefile：分布在 Linux 内核源代码中的 Makefile，定义 Linux 内核的编译规则；
2. 配置文件（config.in）：给用户提供配置选择的功能；
3. 配置工具：包括配置命令解释器（对配置脚本中使用的配置命令进行解释）和配置用户界面（提供基于字符界面、基于 Ncurses 图形界面以及基于 Xwindows 图形界面的用户配置界面，各自对应于 Make config、Make menuconfig 和 make xconfig）。

这些配置工具都是使用脚本语言，如 Tcl/TK、Perl 编写的（也包含一些用 C 编写的代码）。本文并不是对配置系统本身进行分析，而是介绍如何使用配置系统。所以，除非是配置系统的维护者，一般的内核开发者无须了解它们的原理，只需要知道如何编写 Makefile 和配置文件就可以。所以，在本文中，我们只对 Makefile 和配置文件进行讨论。另外，凡是涉及到与具体 CPU 体系结构相关的内容，我们都以 ARM 为例，这样不仅可以将讨论的问题明确化，而且对内容本身不产生影响。

2． Makefile

2.1 Makefile 概述

Makefile 的作用是根据配置的情况，构造出需要编译的源文件列表，然后分别编译，并把目标代码链接到一起，最终形成 Linux 内核二进制文件。

由于 Linux 内核源代码是按照树形结构组织的，所以 Makefile 也被分布在目录树中。Linux 内核中的 Makefile 以及与 Makefile 直接相关的文件有：

1. Makefile：顶层 Makefile，是整个内核配置、编译的总体控制文件。
2. .config：内核配置文件，包含由用户选择的配置选项，用来存放内核配置后的结果（如 make config）。
3. arch/*/Makefile：位于各种 CPU 体系目录下的 Makefile，如 arch/arm/Makefile，是针对特定平台的 Makefile。
4. 各个子目录下的 Makefile：比如 drivers/Makefile，负责所在子目录下源代码的管理。
5. Rules.make：规则文件，被所有的 Makefile 使用。

用户通过 make config 配置后，产生了 .config。顶层 Makefile 读入 .config 中的配置选择。顶层 Makefile 有两个主要的任务：产生 vmlinux 文件和内核模块（module）。为了达到此目的，顶层 Makefile 递归的进入到内核的各个子目录中，分别调用位于这些子目录中的 Makefile。至于到底进入哪些子目录，取决于内核的配置。在顶层 Makefile 中，有一句：include arch/$(ARCH)/Makefile，包含了特定 CPU 体系结构下的 Makefile，这个 Makefile 中包含了平台相关的信息。

位于各个子目录下的 Makefile 同样也根据 .config 给出的配置信息，构造出当前配置下需要的源文件列表，并在文件的最后有 include $(TOPDIR)/Rules.make。

Rules.make 文件起着非常重要的作用，它定义了所有 Makefile 共用的编译规则。比如，如果需要将本目录下所有的 c 程序编译成汇编代码，需要在 Makefile 中有以下的编译规则：

```
%.s: %.c
$(CC) $(CFLAGS) -S $< -o $@
```

有很多子目录下都有同样的要求，就需要在各自的 Makefile 中包含此编译规则，这会比较麻烦。而 Linux 内核中则把此类的编译规则统一放置到 Rules.make 中，并在各自的 Makefile 中包含进了 Rules.make（include Rules.make），这样就避免了在多个 Makefile 中重复同样的规则。对于上面的例子，在 Rules.make 中对应的规则为：

```
%.s: %.c
$(CC) $(CFLAGS) $(EXTRA_CFLAGS) $(CFLAGS_$(*F)) $(CFLAGS_$@) -S $< -o $@
```

2.2 Makefile 中的变量

顶层 Makefile 定义并向环境中输出了许多变量，为各个子目录下的 Makefile 传递一些信息。有些变量，比如 SUBDIRS，不仅在顶层 Makefile 中定义并且赋初值，而且在 arch/*/Makefile 还作了扩充。

常用的变量有以下几类：

1） 版本信息
版本信息有：VERSION，PATCHLEVEL, SUBLEVEL, EXTRAVERSION，KERNELRELEASE。
版本信息定义了当前内核的版本，比如 VERSION=2，PATCHLEVEL=4，SUBLEVEL=18，EXATAVERSION=-rmk7，它们共同构成内核的发行版本KERNELRELEASE：2.4.18-rmk7

2） CPU 体系结构：ARCH
在顶层 Makefile 的开头，用 ARCH 定义目标 CPU 的体系结构，比如 ARCH:=arm 等。许多子目录的 Makefile 中，要根据 ARCH 的定义选择编译源文件的列表。

3） 路径信息：TOPDIR, SUBDIRS
TOPDIR 定义了 Linux 内核源代码所在的根目录。例如，各个子目录下的 Makefile 通过 $(TOPDIR)/Rules.make 就可以找到 Rules.make 的位置。
SUBDIRS 定义了一个目录列表，在编译内核或模块时，顶层 Makefile 就是根据 SUBDIRS 来决定进入哪些子目录。SUBDIRS 的值取决于内核的配置，在顶层 Makefile 中 SUBDIRS 赋值为 kernel drivers mm fs net ipc lib；根据内核的配置情况，在 arch/*/Makefile 中扩充了 SUBDIRS 的值，参见4）中的例子。

4） 内核组成信息：HEAD, CORE_FILES, NETWORKS, DRIVERS, LIBS
Linux 内核文件 vmlinux 是由以下规则产生的：

```
vmlinux: $(CONFIGURATION) init/main.o init/version.o linuxsubdirs
        $(LD) $(LINKFLAGS) $(HEAD) init/main.o init/version.o \
                --start-group \
                $(CORE_FILES) \
                $(DRIVERS) \
                $(NETWORKS) \
                $(LIBS) \
                --end-group \
                -o vmlinux
```

可以看出，vmlinux 是由 HEAD、main.o、version.o、CORE_FILES、DRIVERS、NETWORKS 和 LIBS 组成的。这些变量（如 HEAD）都是用来定义连接生成 vmlinux 的目标文件和库文件列表。其中，HEAD在arch/*/Makefile 中定义，用来确定被最先链接进 vmlinux 的文件列表。比如，对于 ARM 系列的 CPU，HEAD 定义为：

```
HEAD            := arch/arm/kernel/head-$(PROCESSOR).o \
                   arch/arm/kernel/init_task.o
```

表明 head-$(PROCESSOR).o 和 init_task.o 需要最先被链接到 vmlinux 中。PROCESSOR 为 armv 或 armo，取决于目标 CPU。 CORE_FILES，NETWORK，DRIVERS 和 LIBS 在顶层 Makefile 中定义，并且由 arch/*/Makefile 根据需要进行扩充。 CORE_FILES 对应着内核的核心文件，有 kernel/kernel.o，mm/mm.o，fs/fs.o，ipc/ipc.o，可以看出，这些是组成内核最为重要的文件。同时，arch/arm/Makefile 对 CORE_FILES 进行了扩充：

```
# arch/arm/Makefile

# If we have a machine-specific directory, then include it in the build.
MACHDIR         := arch/arm/mach-$(MACHINE)
ifeq ($(MACHDIR),$(wildcard $(MACHDIR)))
SUBDIRS         += $(MACHDIR)
```

```
CORE_FILES      := $(MACHDIR)/$(MACHINE).o $(CORE_FILES)
endif

HEAD            := arch/arm/kernel/head-$(PROCESSOR).o \
                   arch/arm/kernel/init_task.o
SUBDIRS         += arch/arm/kernel arch/arm/mm arch/arm/lib arch/arm/nwfpe
CORE_FILES      := arch/arm/kernel/kernel.o arch/arm/mm/mm.o $(CORE_FILES)
LIBS            := arch/arm/lib/lib.a $(LIBS)
```

5） 编译信息：CPP, CC, AS, LD, AR，CFLAGS，LINKFLAGS
在 Rules.make 中定义的是编译的通用规则，具体到特定的场合，需要明确给出编译环境，编译环境就是在以上的变量中定义的。针对交叉编译的要求，定义了 CROSS_COMPILE。比如：

```
CROSS_COMPILE   = arm-linux-
CC              = $(CROSS_COMPILE)gcc
LD              = $(CROSS_COMPILE)ld
......
```

CROSS_COMPILE 定义了交叉编译器前缀 arm-linux-，表明所有的交叉编译工具都是以 arm-linux- 开头的，所以在各个交叉编译器工具之前，都加入了 $(CROSS_COMPILE)，以组成一个完整的交叉编译工具文件名，比如 arm-linux-gcc。
CFLAGS 定义了传递给 C 编译器的参数。
LINKFLAGS 是链接生成 vmlinux 时，由链接器使用的参数。LINKFLAGS 在 arm/*/Makefile 中定义，比如：

```
# arch/arm/Makefile

LINKFLAGS       :=-p -X -T arch/arm/vmlinux.lds
```

6） 配置变量CONFIG_*
.config 文件中有许多的配置变量等式，用来说明用户配置的结果。例如 CONFIG_MODULES=y 表明用户选择了 Linux 内核的模块功能。
.config 被顶层 Makefile 包含后，就形成许多的配置变量，每个配置变量具有确定的值：y 表示本编译选项对应的内核代码被静态编译进 Linux 内核；m 表示本编译选项对应的内核代码被编译成模块；n 表示不选择此编译选项；如果根本就没有选择，那么配置变量的值为空。

2.3 Rules.make 变量

前面讲过，Rules.make 是编译规则文件，所有的 Makefile 中都会包括 Rules.make。Rules.make 文件定义了许多变量，最为重要是那些编译、链接列表变量。

O_OBJS，L_OBJS，OX_OBJS，LX_OBJS：本目录下需要编译进 Linux 内核 vmlinux 的目标文件列表，其中 OX_OBJS 和 LX_OBJS 中的 "X" 表明目标文件使用了 EXPORT_SYMBOL 输出符号。

M_OBJS，MX_OBJS：本目录下需要被编译成可装载模块的目标文件列表。同样，MX_OBJS 中的 "X" 表明目标文件使用了 EXPORT_SYMBOL 输出符号。

O_TARGET，L_TARGET：每个子目录下都有一个 O_TARGET 或 L_TARGET，Rules.make 首先从源代码编译生成 O_OBJS 和 OX_OBJS 中所有的目标文件，然后使用 $(LD) -r 把它们链接成一个 O_TARGET 或 L_TARGET。O_TARGET 以 .o 结尾，而 L_TARGET 以 .a 结尾。

2.4 子目录 Makefile

子目录 Makefile 用来控制本级目录以下源代码的编译规则。我们通过一个例子来讲解子目录 Makefile 的组成：

```
#
# Makefile for the linux kernel.
#
# All of the (potential) objects that export symbols.
# This list comes from 'grep -l EXPORT_SYMBOL *.[hc]'.

export-objs       := tc.o

# Object file lists.

obj-y             :=
obj-m             :=
obj-n             :=
obj-              :=

obj-$(CONFIG_TC) += tc.o
obj-$(CONFIG_ZS) += zs.o
obj-$(CONFIG_VT) += lk201.o lk201-map.o lk201-remap.o

# Files that are both resident and modular: remove from modular.

obj-m             := $(filter-out $(obj-y), $(obj-m))

# Translate to Rules.make lists.

L_TARGET := tc.a

L_OBJS            := $(sort $(filter-out $(export-objs), $(obj-y)))
LX_OBJS           := $(sort $(filter     $(export-objs), $(obj-y)))
M_OBJS            := $(sort $(filter-out $(export-objs), $(obj-m)))
MX_OBJS           := $(sort $(filter     $(export-objs), $(obj-m)))

include $(TOPDIR)/Rules.make
```

a) 注释

对 Makefile 的说明和解释，由#开始。

b) 编译目标定义

类似于 obj-$(CONFIG_TC) += tc.o 的语句是用来定义编译的目标，是子目录 Makefile 中最重要的部分。编译目标定义那些在本子目录下，需要编译到 Linux 内核中的目标文件列表。为了只在用户选择了此功能后才编译，所有的目标定义都融合了对配置变量的判断。

前面说过，每个配置变量取值范围是：y，n，m 和空，obj-$(CONFIG_TC) 分别对应着 obj-y，obj-n，obj-m，obj-。如果 CONFIG_TC 配置为 y，那么 tc.o 就进入了 obj-y 列表。obj-y 为包含到 Linux 内核 vmlinux 中的目标文件列表；obj-m 为编译成模块的目标文件列表；obj-n 和 obj- 中的文件列表被忽略。配置系统就根据这些列表的属性进行编译和链接。

export-objs 中的目标文件都使用了 EXPORT_SYMBOL() 定义了公共的符号，以便可装载模块使用。在 tc.c 文件的最后部分，有 "EXPORT_SYMBOL(search_tc_card);"，表明 tc.o 有符号输出。

这里需要指出的是，对于编译目标的定义，存在着两种格式，分别是老式定义和新式定义。老式定

义就是前面 Rules.make 使用的那些变量，新式定义就是 obj-y，obj-m，obj-n 和 obj-。Linux 内核推荐使用新式定义，不过由于 Rules.make 不理解新式定义，需要在 Makefile 中的适配段将其转换成老式定义。

c) 适配段
适配段的作用是将新式定义转换成老式定义。在上面的例子中，适配段就是将 obj-y 和 obj-m 转换成 Rules.make 能够理解的 L_TARGET，L_OBJS，LX_OBJS，M_OBJS，MX_OBJS。
L_OBJS := $(sort $(filter-out $(export-objs), $(obj-y))) 定义了 L_OBJS 的生成方式：在 obj-y 的列表中过滤掉 export-objs（tc.o），然后排序并去除重复的文件名。这里使用到了 GNU Make 的一些特殊功能，具体的含义可参考 Make 的文档（info make）。

d) include $(TOPDIR)/Rules.make

3． 配置文件

3.1 配置功能概述

除了 Makefile 的编写，另外一个重要的工作就是把新功能加入到 Linux 的配置选项中，提供此项功能的说明，让用户有机会选择此项功能。所有的这些都需要在 config.in 文件中用配置语言来编写配置脚本，
在 Linux 内核中，配置命令有多种方式：

配置命令	解释脚本
Make config, make oldconfig	scripts/Configure
Make menuconfig	scripts/Menuconfig
Make xconfig	scripts/tkparse

以字符界面配置（make config）为例，顶层 Makefile 调用 scripts/Configure， 按照 arch/arm/config.in 来进行配置。命令执行完后产生文件 .config，其中保存着配置信息。下一次再做 make config 将产生新的 .config 文件，原 .config 被改名为 .config.old

3.2 配置语言

1） 顶层菜单
mainmenu_name /prompt/ /prompt/ 是用'或"包围的字符串，'与"的区别是'…'中可使用$引用变量的值。mainmenu_name 设置最高层菜单的名字，它只在 make xconfig 时才会显示。

2） 询问语句

```
bool        /prompt/ /symbol/
hex         /prompt/ /symbol/ /word/
int         /prompt/ /symbol/ /word/
string      /prompt/ /symbol/ /word/
tristate    /prompt/ /symbol/
```

询问语句首先显示一串提示符 /prompt/，等待用户输入，并把输入的结果赋给 /symbol/ 所代表的配置变量。不同的询问语句的区别在于它们接受的输入数据类型不同，比如 bool 接受布尔类型（ y 或 n ），hex 接受 16 进制数据。有些询问语句还有第三个参数 /word/，用来给出缺省值。

3） 定义语句

```
define_bool     /symbol/ /word/
define_hex      /symbol/ /word/
define_int      /symbol/ /word/
define_string   /symbol/ /word/
define_tristate /symbol/ /word/
```

不同于询问语句等待用户输入，定义语句显式的给配置变量 /symbol/ 赋值 /word/。

4） 依赖语句

```
dep_bool        /prompt/ /symbol/ /dep/ ...
dep_mbool       /prompt/ /symbol/ /dep/ ...
dep_hex         /prompt/ /symbol/ /word/ /dep/ ...
dep_int         /prompt/ /symbol/ /word/ /dep/ ...
dep_string      /prompt/ /symbol/ /word/ /dep/ ...
dep_tristate    /prompt/ /symbol/ /dep/ ...
```

与询问语句类似，依赖语句也是定义新的配置 变量。不同的是，配置变量/symbol/的取值范围将依赖于配置变量列表/dep/ …。这就意味着：被定义的配置变量所对应功能的取舍取决于依赖列表所对应功能的选择。以dep_bool为例，如果/dep/ …列表的所有配置变量都取值y，则显示/prompt/，用户可输入任意的值给配置变量/symbol/，但是只要有一个配置变量的取值为n，则/symbol/被强制成n。
不同依赖语句的区别在于它们由依赖条件所产生的取值范围不同。

5） 选择语句

```
choice          /prompt/ /word/ /word/
```

choice 语句首先给出一串选择列表，供用户选择其中一种。比如 Linux for ARM 支持多种基于 ARM core 的 CPU，Linux 使用 choice 语句提供一个 CPU 列表，供用户选择：

```
 choice 'ARM system type' \
"Anakin                 CONFIG_ARCH_ANAKIN \
 Archimedes/A5000       CONFIG_ARCH_ARCA5K \
 Cirrus-CL-PS7500FE     CONFIG_ARCH_CLPS7500 \
                  ......
 SA1100-based           CONFIG_ARCH_SA1100 \
 Shark                  CONFIG_ARCH_SHARK" RiscPC
```

Choice 首先显示 /prompt/，然后将 /word/ 分解成前后两个部分，前部分为对应选择的提示符，后部分是对应选择的配置变量。用户选择的配置变量为 y，其余的都为 n。

6） if语句

```
if [ /expr/ ] ; then
  /statement/
  ...
fi

if [ /expr/ ] ; then
  /statement/
  ...
```

```
else
  /statement/
  ...
fi
```

if 语句对配置变量（或配置变量的组合）进行判断，并作出不同的处理。判断条件 /expr/ 可以是单个配置变量或字符串，也可以是带操作符的表达式。操作符有：=，!=，-o，-a 等。

7） 菜单块（menu block）语句

```
mainmenu_option next_comment
comment '…..'
…
endmenu
```

引入新的菜单。在向内核增加新的功能后，需要相应的增加新的菜单，并在新菜单下给出此项功能的配置选项。Comment 后带的注释就是新菜单的名称。所有归属于此菜单的配置选项语句都写在 comment 和 endmenu 之间。

8） Source 语句

source /word/

/word/ 是文件名，source 的作用是调入新的文件。

3.3 缺省配置

Linux 内核支持非常多的硬件平台，对于具体的硬件平台而言，有些配置就是必需的，有些配置就不是必需的。另外，新增加功能的正常运行往往也需要一定的先决条件，针对新功能，必须作相应的配置。因此，特定硬件平台能够正常运行对应着一个最小的基本配置，这就是缺省配置。

Linux 内核中针对每个 ARCH 都会有一个缺省配置。在向内核代码增加了新的功能后，如果新功能对于这个 ARCH 是必需的，就要修改此 ARCH 的缺省配置。修改方法如下（在 Linux 内核根目录下）：

1. 备份 .config 文件
2. cp arch/arm/deconfig .config
3. 修改 .config
4. cp .config arch/arm/deconfig
5. 恢复 .config

如果新增的功能适用于许多的 ARCH，只要针对具体的 ARCH，重复上面的步骤就可以了。

3.4 help file

大家都有这样的经验，在配置 Linux 内核时，遇到不懂含义的配置选项，可以查看它的帮助，从中可得到选择的建议。下面我们就看看如何给给一个配置选项增加帮助信息。

所有配置选项的帮助信息都在 Documentation/Configure.help 中，它的格式为：

<description>

```
<variable name>
<help file>
```

<description> 给出本配置选项的名称，<variable name> 对应配置变量，<help file> 对应配置帮助信息。在帮助信息中，首先简单描述此功能，其次说明选择了此功能后会有什么效果，不选择又有什么效果，最后，不要忘了写上"如果不清楚，选择 N（或者）Y"，给不知所措的用户以提示。

4． 实例

对于一个开发者来说，将自己开发的内核代码加入到 Linux 内核中，需要有三个步骤。首先确定把自己开发代码放入到内核的位置；其次，把自己开发的功能增加到 Linux 内核的配置选项中，使用户能够选择此功能；最后，构建子目录 Makefile，根据用户的选择，将相应的代码编译到最终生成的 Linux 内核中去。下面，我们就通过一个简单的例子--test driver，结合前面学到的知识，来说明如何向 Linux 内核中增加新的功能。

4.1 目录结构

test driver 放置在 drivers/test/ 目录下：

```
$cd drivers/test
$tree
.
|-- Config.in
|-- Makefile
|-- cpu
|   |-- Makefile
|   `-- cpu.c
|-- test.c
|-- test_client.c
|-- test_ioctl.c
|-- test_proc.c
|-- test_queue.c
`-- test
    |-- Makefile
    `-- test.c
```

4.2 配置文件

1） drivers/test/Config.in

```
#
# TEST driver configuration
#
mainmenu_option next_comment
comment 'TEST Driver'

bool 'TEST support' CONFIG_TEST
if [ "$CONFIG_TEST" = "y" ]; then
  tristate 'TEST user-space interface' CONFIG_TEST_USER
  bool 'TEST CPU ' CONFIG_TEST_CPU
fi

endmenu
```

由于 test driver 对于内核来说是新的功能，所以首先 创建一个菜单 TEST Driver。然后，显示 "TEST support"，等待用户选择；接下来判断用户是否选择了 TEST Driver，如果是（CONFIG_TEST=y），则进一步显示子功能：用户接口与 CPU 功能支持；由于用户接口功能可以被编译成内核模块，所以这里的询问语句使用了 tristate（因为 tristate 的取值范围包括 y、n 和 m，m 就是对应着模块）。

2） arch/arm/config.in

在文件的最后加入：source drivers/test/Config.in，将 TEST Driver 子功能的配置纳入到 Linux 内核的配置中。

4.3 Makefile

1）drivers/test/Makefile

```
#       drivers/test/Makefile
#
#       Makefile for the TEST.
#

SUB_DIRS     :=
MOD_SUB_DIRS := $(SUB_DIRS)
ALL_SUB_DIRS := $(SUB_DIRS) cpu

L_TARGET := test.a
export-objs := test.o test_client.o

obj-$(CONFIG_TEST)              += test.o test_queue.o test_client.o
obj-$(CONFIG_TEST_USER)         += test_ioctl.o
obj-$(CONFIG_PROC_FS)           += test_proc.o

subdir-$(CONFIG_TEST_CPU)       += cpu

include $(TOPDIR)/Rules.make

clean:
        for dir in $(ALL_SUB_DIRS); do make -C $$dir clean; done
        rm -f *.[oa] .*.flags
```

drivers/test 目录下最终生成的目标文件是 test.a。在 test.c 和 test-client.c 中使用了 EXPORT_SYMBOL 输出符号，所以 test.o 和 test-client.o 位于 export-objs 列表中。然后，根据用户的选择（具体来说，就是配置变量的取值），构建各自对应的 obj-* 列表。由于 TEST Driver 中包一个子目录 cpu，当 CONFIG_TEST_CPU=y（即用户选择了此功能）时，需要将 cpu 目录加入到 subdir-y 列表中。

2）drivers/test/cpu/Makefile

```
#       drivers/test/test/Makefile
#
#       Makefile for the TEST CPU
#

SUB_DIRS     :=
MOD_SUB_DIRS := $(SUB_DIRS)
ALL_SUB_DIRS := $(SUB_DIRS)
```

```
L_TARGET := test_cpu.a

obj-$(CONFIG_test_CPU)        += cpu.o


include $(TOPDIR)/Rules.make

clean:
        rm -f *.[oa] .*.flags
```

3）drivers/Makefile

```
……
subdir-$(CONFIG_TEST)                 += test
……
include $(TOPDIR)/Rules.make
```

在 drivers/Makefile **中加入** subdir-$(CONFIG_TEST)+= test，**使得在用户选择** TEST Driver **功能后，内核编译时能够进入** test **目录。**

4）Makefile

```
……
DRIVERS-$(CONFIG_PLD) += drivers/pld/pld.o
DRIVERS-$(CONFIG_TEST) += drivers/test/test.a
DRIVERS-$(CONFIG_TEST_CPU) += drivers/test/cpu/test_cpu.a

DRIVERS := $(DRIVERS-y)
……
```

在顶层 Makefile **中加入** DRIVERS-$(CONFIG_TEST) += drivers/test/test.a **和** DRIVERS-$(CONFIG_TEST_CPU) += drivers/test/cpu/test_cpu.a。**如何用户选择了** TEST Driver，**那么** CONFIG_TEST **和** CONFIG_TEST_CPU **都是** y，test.a **和** test_cpu.a **就都位于** DRIVERS-y **列表中，然后又被放置在** DRIVERS **列表中。在前面曾经提到过，**Linux **内核文件** vmlinux **的组成中包括** DRIVERS，**所以** test.a **和** test_cpu.a **最终可被链接到** vmlinux **中。**

5． 参考

1. Document/kbuild/makefiles.txt，Linux Kernel Source code
2. Document/kbuild/config-language.txt，Linux Kernel Source code
3. Contributing to the Linux Kernel--The Linux Configuration System，Linux Journal，http://www.linuxjournal.com/categories.php?op=newindex&catid=178
4. Unreliable Guide To Hacking The Linux Kernel，Paul Rusty Russell，rusty@rustcorp.com.au

关于作者

汤凯，通过tangk73@hotmail.com**可以跟他联系。**

到页首

您对这篇文章的看法如何？

○ **真棒！**(5) ○ **好材料** (4) ○ **一般；尚可** (3) ○ **需提高** (2) ○ **太差！** (1)

建议？

提交反馈意见

(c) Copyright IBM Corp. 2001, (c) Copyright IBM China 2001, All Right Reserved
关于 IBM | 隐私条约 | 使用条款 | 联系 IBM