

KConfig、Makefile详解以及ARM平台Linux内核的编译

本文主要介绍Linux2.6的内核配置系统。

如果你浏览一下源代码目录，就可以发现源码目录及其子目录中有很多的KConfig文件和Makefile文件。这些文件什么作用呢？正是这些文件组成了Linux2.6的内核配置系统。

一、make menuconfig的背后----- KConfig文件的组织

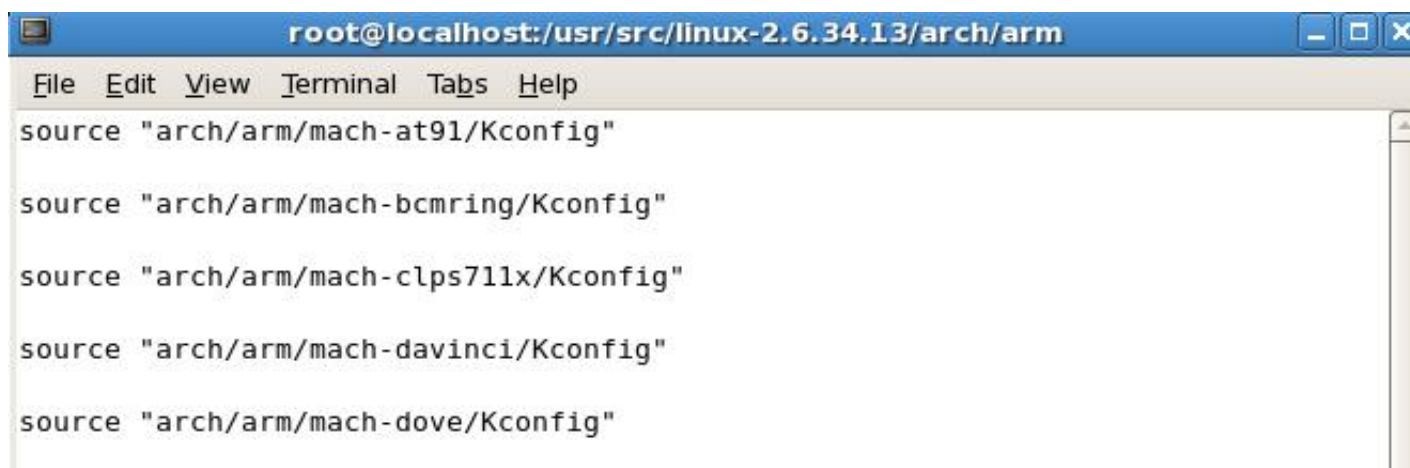
有没有想过，我们make menuconfig后，显示的那个菜单列表是怎么来的？

带着这个疑问，我们先来简单学一下Kconfig文件的“语法”。

source 关键字：

用法：source <filename>

这个关键字相当于C语言里的“include”关键字，source后面跟一个文件名，相当于把该文件的内容复制到当前位置。下面是源码目录的arch/arm目录下Kconfig文件的部分内容。



```
root@localhost:/usr/src/linux-2.6.34.13/arch/arm
File Edit View Terminal Tabs Help
source "arch/arm/mach-at91/Kconfig"
source "arch/arm/mach-bcmring/Kconfig"
source "arch/arm/mach-clps711x/Kconfig"
source "arch/arm/mach-davinci/Kconfig"
source "arch/arm/mach-dove/Kconfig"
```

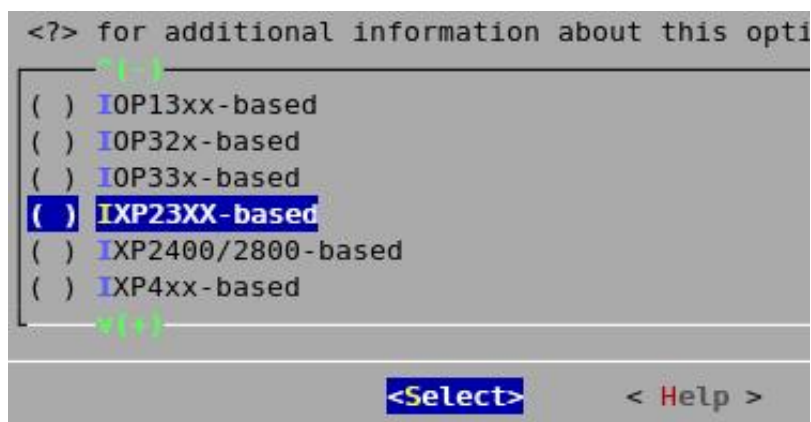
通过这种source引用，可以引入很多其他子目录中的Kconfig文件，而且引入的Kconfig文件中，还可以继续通过source来引入下一级的Kconfig文件。这样的结构就可以将所有的Kconfig文件包含进来。

一个菜单项（或叫配置项）的基本组成：config、bool(tristate)、default、prompt、help

一个简单的菜单项：

```
config ARCH_IXP23XX
    bool "IXP23XX-based"
    depends on MMU
    select CPU_XSC3
    select PCI
    help
        Support for Intel's IXP23xx (XScale) family of processors.
```

其中，config关键字表示新定义一个菜单项，后面跟的是这个菜单项的名字(ARCH_IXP23XX)。bool标识这个菜单项是bool类型，也就是这一项只能有两个值Y和N，此外还有一种最常用的类型，tristate三态型，这种类型的可以有三个值，Y/M/N，这三个值的意义在（上）篇中已经说过了。后面的“IXP23XX-based”是这个菜单项的描述，就是在make menuconfig时我们能看到的，如下图：



在菜单列表里我们并看不到一个菜单项的名字，而只能看到它的描述，因为看它的描述更便于我们理解这个菜单项的意义，方便我们配置。关于菜单项的描述，还有一个prompt关键字，举例说明其用法。比如下面两段是等效的

```
-----
CONFIG MY_MENU
```

```
bool
```

```
prompt "this is my menu"
-----
```

```
CONFIG MY_MENU
```

```
bool "this is my menu"
-----
```

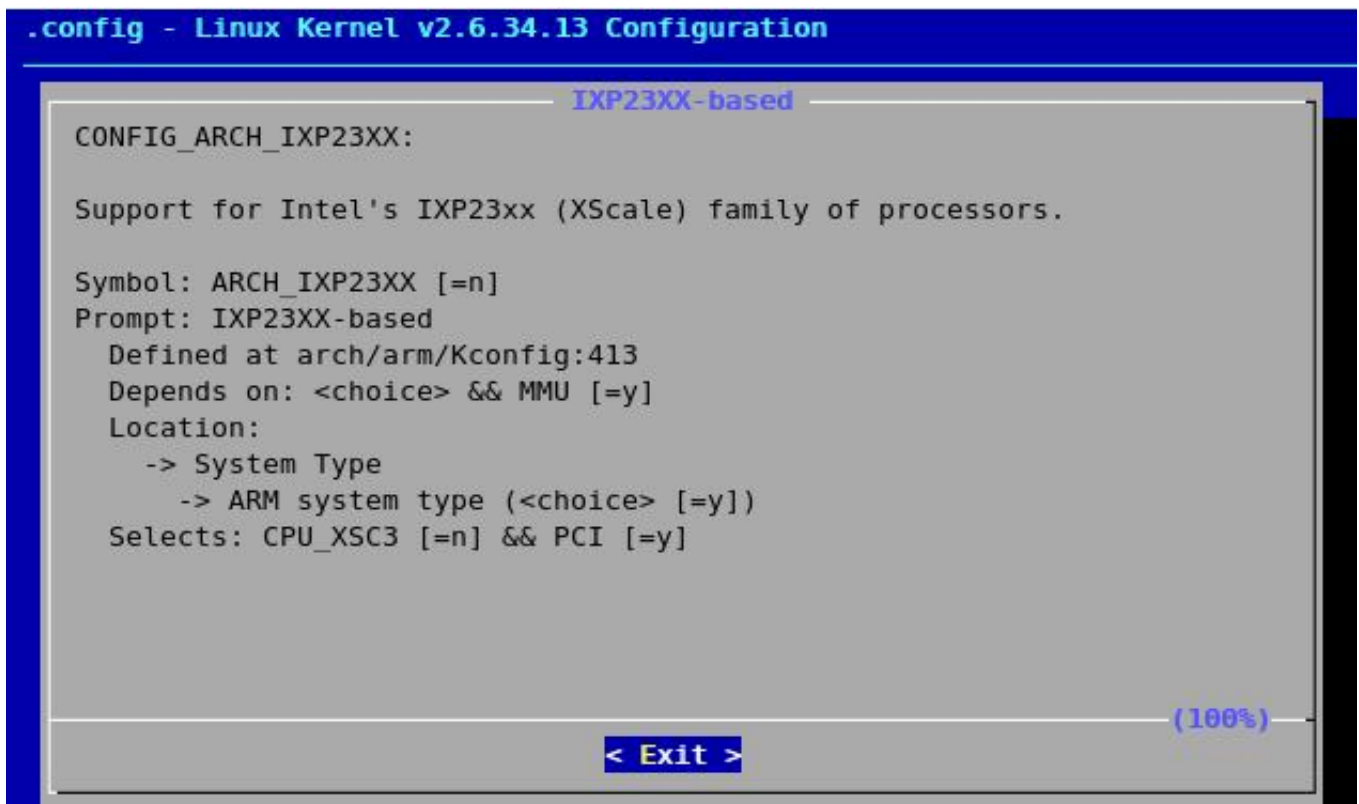
就是说，一个菜单项的描述，可以直接跟在其类型(bool)的后面来进行声明，也可以由prompt关键字声明。

关于default关键字，截图中并未出现，但也是很常用的，它表明一个菜单项的默认值。如

default y

在进入菜单列表时，可以发现很多菜单项都有默认值，这些默认值就是通过Kconfig文件里的default定义的。

还有一个help关键字，help关键字后面的内容是帮助信息，就是我们点击右下角的help时显示的关于这个菜单项的帮助信息。下面是关于上图所示的菜单项的帮助信息：



```
.config - Linux Kernel v2.6.34.13 Configuration

                                IXP23XX-based

CONFIG_ARCH_IXP23XX:

Support for Intel's IXP23xx (XScale) family of processors.

Symbol: ARCH_IXP23XX [=n]
Prompt: IXP23XX-based
Defined at arch/arm/Kconfig:413
Depends on: <choice> && MMU [=y]
Location:
  -> System Type
    -> ARM system type (<choice> [=y])
Selects: CPU_XSC3 [=n] && PCI [=y]

(100%)

< Exit >
```

菜单项间的依赖关系：select和depends on

还拿上面的例子来说明，第三行"depends on MMU"。这一行是说，现在定义的"ARCH_IXP23XX"这个菜单项的值(Y/N)依赖于MMU这个菜单项的值。当MMU这个菜单项为N时，ARCH_IXP23XX只能为N。ARCH_IXP23XX的值必须“小于”MMU的值。（对于bool型，Y>N；对于tristate型，Y>M>N）。

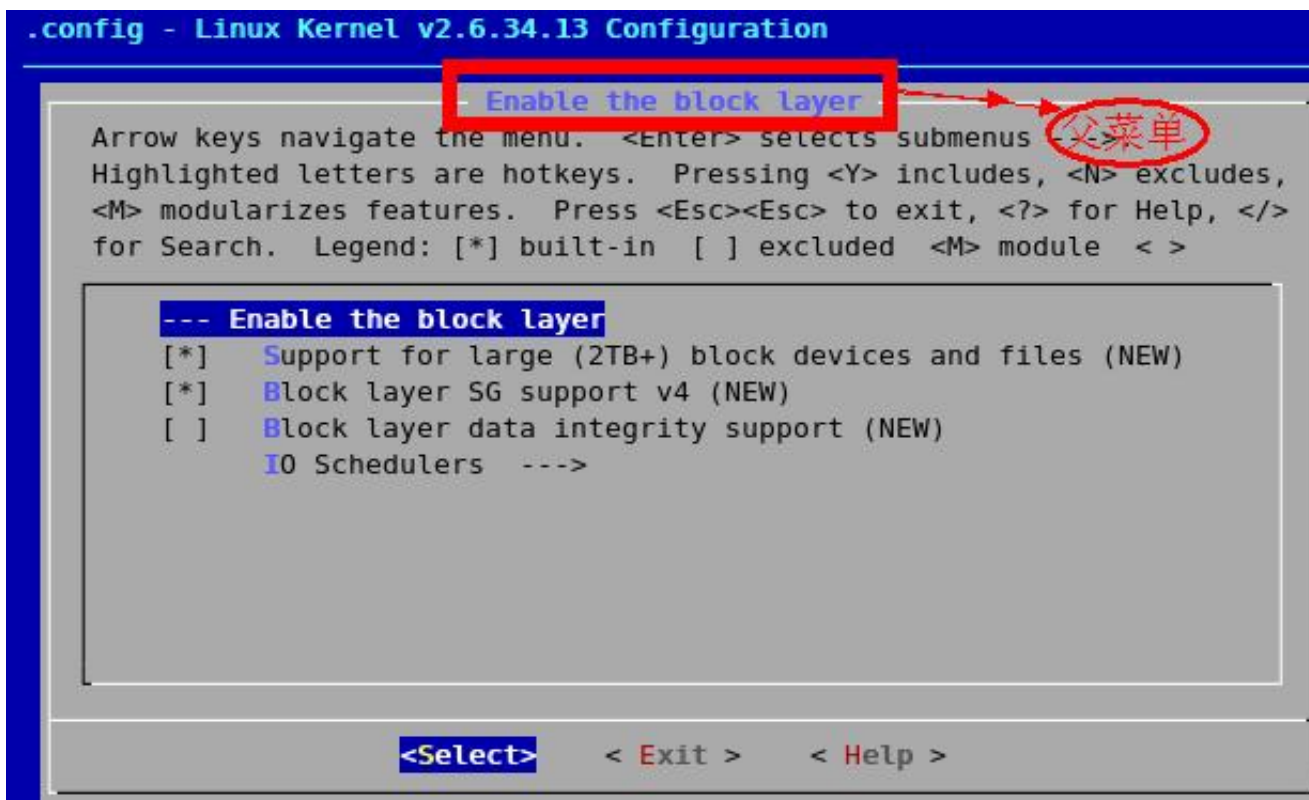
select关键字的作用恰与depends on相反，它描述了一个反依赖的关系。以第五行"select PCI"为例，PCI的值依赖于ARCH_IXP23XX。在定义PCI这个菜单项时，也要加上这样一句："depends on ARCH_IXP23XX"。

根据各菜单项之间的依赖关系，在make menuconfig时，系统会自动将这些相关联的菜单项整理成菜单项与子菜单项的形式，如下图

```

General setup --->
[*] Enable loadable module support
--*-- Enable the block layer --->
System Type --->

```



第二张图中的菜单项都依赖于"Enable the block layer"对应的菜单项，所以系统将它们整理成子菜单项。只有"Enable the block layer"对应的菜单项不为N时，这些子菜单项才可以配置。

menu与endmenu关键字

这个关键字主要是为了给菜单项分组，使菜单结构看起来更有条理。menu用来定义一个子菜单，这个子菜单里包括一些相关的菜单项，在menu和endmenu关键字之间定义的菜单项都属于这个子菜单。还以那上面两张图为例，"Enable the block layer----->"菜单项下面的"system type---->"就是一个子菜单的名称。将这个子菜单展开就可以看到这个子菜单包含的菜单项了。

menu "System type"

config

.....

config.....

.....

config.....

.....

.....

endmenu

这里再额外解释一下，在上面的图中，“Enable the block layer--->”和“system type-->”，这两个虽然看起来很像，都可以展开，但其性质是不同的。前者是根据各菜单项间的依赖关系建立起来的，“Enable the block layer”本身就对应一个菜单项或者说配置项，它也有自己的值(Y/M/N)，而“system type”则只是一个子菜单的名称，它下面包含了一些相关的配置项，但他本身不对应某个配置项，因而没有值（所以菜单列表中“system type--->”的前面没有*或M这些符号）。

choices与endchoice关键字

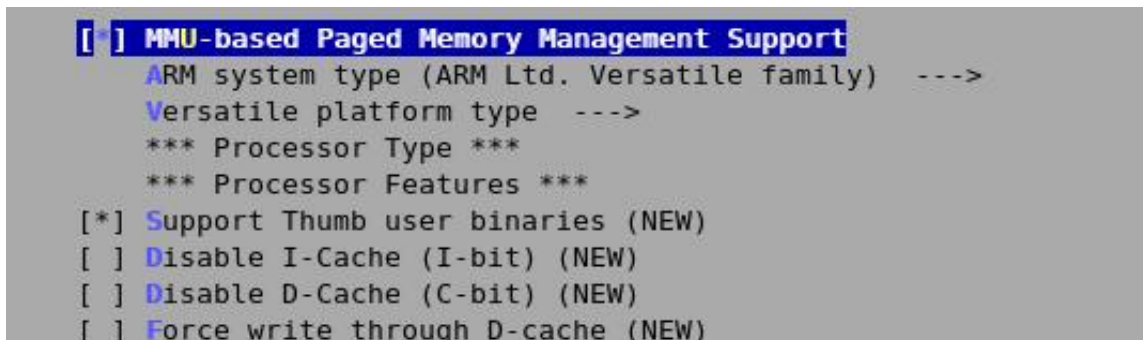
跟menu与endmenu用法基本一样，唯一的区别在于，choices定义的“子菜单”（应该叫选项表）中的多个菜单项只能有一个被选中，相当于menu定义一个可多选的子菜单，choices定义一个单选的子菜单。篇幅限制，不再截图详述。

if与endif关键字

这两个真心不用解释，原谅我直接略过。

comment关键字

用来在菜单列表中插入一行文字，也是为了优化菜单结构。



```
[*] MMU-based Paged Memory Management Support
  ARM system type (ARM Ltd. Versatile family) --->
  Versatile platform type --->
  *** Processor Type ***
  *** Processor Features ***
  [*] Support Thumb user binaries (NEW)
  [ ] Disable I-Cache (I-bit) (NEW)
  [ ] Disable D-Cache (C-bit) (NEW)
  [ ] Force write through D-cache (NEW)
```

如上图中的第四、五行，就是通过 comment "Processor Type" 和 comment "Processor Features"插入的。这两行既不能展开，也不能被配置，他们只是为菜单列表分段的一行文字。

终于把Kconfig中的关键字连图带字的解说完了，好累啊。（很多教材和博客上介绍这些关键字的时候都是只有文字描述，而不结合make menuconfig后出现的菜单界面联系着说明，读起来不够直观，现在我就把这些整理出来，供新手们快速掌握和理解这些关键字的作用）。大家现在可以试着去阅读一个Kconfig文件了。

好了，对这些关键字有了认识之后，我们来说一下这个菜单列表的形成过程。运行make menuconfig后，

系统的配置工具先分析与体系结构对应的/arch/ARCH/Kconfig文件（这里出现的ARCH参数在本文最后会讲到。它其实指的就是所用的<cpu>），/arch/ARCH/Kconfig文件中定义了一个主菜单mainmenu，它除了包含一些配置项和配置菜单以外，还通过source语句引用了一系列其他子目录中的Kconfig文件，被引用进来的Kconfig文件内部可能再次通过source引入下一级目录中的Kconfig，系统就根据所有这些Kconfig文件中包含的菜单项（配置项）形成了菜单列表，然后根据用户对各个菜单项设置的值，最后生成.config文件。

二、另一个重要角色-----kbuild Makefile的介绍

Kconfig文件帮助用户完成配置过程，而真正编译内核则是在各个子目录中的一系列Makefile共同完成的。Makefile中的重要语法就三个，比较好理解，这里直接引用书中的内容。我把需要注意的地方用粗体标示。

引自 华清远见嵌入式培训中心 宋宝华 编著的《Linux设备驱动开发详解》（这本书真的很好，需要电子版的同仁可以直接联系我）：

（1）目标定义。

目标定义用来定义哪些内容要作为模块编译，哪些要编译并连接进内核。

例如：

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

表示要由foo.c 或者foo.s 文件编译得到foo.o 并连接进内核，而obj-m 则表示该文件要作为模块编译。除了y、m以外的obj-x形式的目标都不会被编译。

而更常见的做法是根据.config 文件的CONFIG_变量来决定文件的编译方式，如下所示：

```
obj-$(CONFIG_ISDN) += isdn.o
```

```
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
```

除了obj-形式的目标以外，还有lib-y library库、hostprogs-y 主机程序等目标，但是基本都应用在特定的目录和场合下。

（2）多文件模块的定义。

如果一个模块由多个文件组成，这时候应采用模块名加-objs后缀或者-y后缀的形式来定义模块的组成文件。如下面的例子所示：

```
obj-$(CONFIG_EXT2_FS) += ext2.o
```

```
ext2-y := balloc.o bitmap.o
```

```
ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o
```

模块的名字为ext2，由balloc.o和bitmap.o两个目标文件最终连接生成ext2.o 直至ext2.ko 文件，是否包括xattr.o 取决于内核配置文件的配置情况。如果

CONFIG_EXT2_FS 的值是y 也没有关系，在此过程中生成的 ext2.o 将被连接进built-in.o最终连接进内核。这里需要注意的一点是，**该kbuild Makefile所在的目录中不能再包含和模块名相同的源文件如ext2.c/ext2.s。**

或者写成如-objs的形式：

```
obj-$(CONFIG_ISDN) += isdn.o
```

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

（3）目录层次的迭代。

示例：

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

当CONFIG_EXT2_FS 的值为y或m时，kbuild将会把ext2 目录列入向下迭代的目标中，具体ext2 目录下的文件是要作为模块编译还是链入内核由ext2 目录下的Makefile文件的内容决定。

引用结束。

上面所述的（3）特别关键，目录层次的迭代使得整个Makefile系统呈现一个树状结构，条理清晰，加载新的编译目录也很方便。

三、牛刀初试

假如我们自己开发了一个新的模块，要如何才能把我们自己的模块加到配置系统中？上面已经介绍了Kconfig和Makefile文件的基本语法，为了确保我们掌握了这两种文件的配置方法，我们最好做一下练习，。这里我要再次偷懒，直接COPY书中内容了。

引用开始：

下面讲解一个综合实例，假设我们要在内核源代码drivers目录下为ARM体系结构新增如下用于test driver 的树型目录：

```
|--test
|-- cpu
|  -- cpu.c
|-- test.c
|-- test_client.c
|-- test_ioctl.c
|-- test_proc.c
|-- test_queue.c
```

在内核中增加目录和子目录，我们需为相应的新增目录创建Kconfig 和Makefile 文件，而新增目录的父目录中的Kconfig 和Makefile 文件也需要修改，以便新增的Kconfig和Makefile文件能被引用。

在新增的test目录下，应该包含如下Kconfig文件：

```
#
# TEST driver configuration
#
menu "TEST Driver "
comment " TEST Driver"
config CONFIG_TEST
bool "TEST support "
config CONFIG_TEST_USER
tristate "TEST user-space interface"
depends on CONFIG_TEST
endmenu
```


由于TEST driver 对于内核来说是新的功能，所以首先需要创建一个菜单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。为了使这个Kconfig文件能起作用，需要修改arch/arm/Kconfig文件，增加以下内容：

```
source "drivers/test/Kconfig"
```

脚本中的source意味着引用新的Kconfig文件。

在新增的test目录下，应该包含如下Makefile文件：

```
# drivers/test/Makefile
```

```
#
```

```
# Makefile for the TEST.
```

```
#
```

```
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
```

```
obj-$(CONFIG_TEST_CPU) += cpu/
```

该脚本根据配置变量的取值构建obj-*列表。由于test目录中包含一个子目录cpu，当CONFIG_TEST_CPU=y时，需要将cpu目录加入列表。

test目录中的cpu子目录也需包含如下的Makefile文件：

```
# drivers/test/test/Makefile
```

```
#
```

```
# Makefile for the TEST CPU
```

```
#
```

```
obj-$(CONFIG_TEST_CPU) += cpu.o
```

为了使得整个test 目录能够被编译命令作用到，test 目录父目录中的Makefile 文件也需新增如下脚本：

```
obj-$(CONFIG_TEST) += test/
```

在drivers/Makefile中加入obj-\$(CONFIG_TEST) += test/，使得用户在进行内核编译时能够进入test目录。

增加了Kconfig和Makefile文件之后的新的test树型目录如下所示：

```
|--test
```

```
|-- cpu
```

```
| -- cpu.c
```

```
| -- Makefile
```

```
|-- test.c
```

```
|-- test_client.c
```

```
|-- test_ioctl.c
```

```
|-- test_proc.c
```

```
|-- test_queue.c
```

```
|-- Makefile
```

```
|-- Kconfig
```


附：（ 顶层Makefile中的ARCH参数、编译ARM平台Linux内核的方法 ）

源代码目录顶层目录中的Makefile文件被称作顶层Makefile，它里面涉及到一个ARCH参数，还有一个CROSS_COMPILE参数，分别对应处理器内核架构和交叉编译器。一般情况下，ARCH默认为x86，CROSS_COMPILE默认也是x86平台上的交叉编译器。在执行make menuconfig,make bzImage,make modules等命令时，系统都会首先分析ARCH的值，根据选择的平台来执行相应操作。因此，如果想编译ARM平台的Linux内核，在输入相应命令时要加上架构和交叉编译器选项，如

```
make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig
```

```
make ARCH=arm CROSS_COMPILE=arm-linux- bzImage
```

```
make ARCH=arm CROSS_COMPILE=arm-linux- modules
```

如果直接将

```
ARCH=arm
```

```
CROSS_COMPILE=arm-linux-
```

这两行添加到顶层Makefile的开头，就可以将ARCH和CROSS_COMPILE的默认值改成"arm"和"arm-linux-"了。以后再编译ARM平台的Linux内核时就可以直接使用make menuconfig/bzImage/modules等命令，而不用再加"ARCH=arm CROSS_COMPILE=arm-linux-"。