


“黑色经典”系列之《嵌入式 Linux 系统开发技术详解——基于 ARM》




第 7 章 配置编译内核

本章目标

本章介绍了 Linux 2.6 内核的特点和配置编译。通过学习本章，可以了解 Linux 2.6 内核的 kbuild 编译管理方式，掌握基本的配置编译过程。

Linux 内核特点 

配置编译内核源码 

7.1 Linux 内核特点

7.1.1 Linux 内核版本介绍

Linux 内核的版本号分为主版本号、次版本号和扩展版本号等。根据稳定版本、测试版本和开发版本定义不同版本序列。

稳定版本的主版本号用偶数表示，例如：2.2、2.4、2.6。每隔 2~3 年启动一个 Linux 稳定主版本号。

紧接着是次版本号，例如：2.6.13、2.6.14、2.6.15。次版本号不分奇偶数，顺序递增。每隔 1~2 个月发布一个稳定版本。

然后是升级版本号，例如：2.6.14.3、2.6.14.4、2.6.14.5。升级版本号不分奇偶数，顺序递增。每周几次发布升级版本号，修正最新的稳定版本的问题。

另外一种测试版本。在下一个稳定版本发布之前，每个月发布几个测试版本，例如：2.6.12-rc1。通过测试，可以使内核正式发布的时候更加稳定。

还有一类是开发版本。开发版本的主版本号用奇数表示，例如：2.3、2.5。也有次版本号，例如：2.5.32、2.5.33。开发版本是不稳定的，适合内核开发者在新的稳定的主版本发布之前使用。

7.1.2 Linux 内核特点

(1) Linux 内核的重要特点可移植性 (Portability)，支持硬件平台广泛，在大多数体系结构上都可以运行。

可量测性 (Scalability)，即可以运行在超级计算机上，也可以运行在很小的设备上 (4MB RAM 就能满足)。

标准化和互用性 (Interoperability)，遵守标准化和互用性规范。

完善的网络支持。

安全性，开放源码使缺陷暴露无疑，它的代码也接受了许多专家的审查。

稳定性 (Stability) 和可靠性 (Reliability)。

模块化 (Modularity)，运行时可以根据系统的需要加载程序。

编程容易，可以学习现有的代码，还可以从网络上找到很多有用的资源。

(2) Linux 内核支持的处理器体系结构

Linux 内核能够支持的处理器器的最小要求：32 位处理器，带或者不带 MMU。需要说明的是，不带 MMU 的处理器过去是 uClinux 支持的。Linux 2.6 内核采纳了 m68k 等不带 MMU 的部分平台，Linux 支持的绝大多数处理器还是带 MMU 的。

Linux 内核既能支持 32 位体系结构，又能支持 64 位体系结构。

每一种体系结构在内核源码树的 arch/目录下有子目录。各种体系结构的详细内容可以查看源码 Documentation/<arch>/目录下的文档。

(3) Linux 内核遵守的软件许可

Linux 内核全部源代码是遵守 GPL 软件许可的免费软件，这就要求在发布 Linux 软件的时候免费开放源码。

对于 Linux 等自由软件，必须对最终用户开放源代码，但是没有义务向其他任何人开放。在商业 Linux 公司中，通常会要求客户签署最终用户的使用许可。

私有的模块是允许使用的。只要不被认定为源自 GPL 的代码，就可以按照私有许可使用。但是，私有的驱动程序不能静态链接到内核中去，可以作为动态加载的模块使用。

(4) 开放源码驱动程序的优点

基于庞大的 Linux 社区和内核源码工程，有各种各样的驱动程序和应用程序可以利用，而没有必要从头写程序。

开发者可以免费得到社区的贡献、支持、检查代码和测试。驱动程序可以免费发布给其他人，可以静态编译进内核。

对 Linux 公司来说，用户和社区的正面印象可以使他们更容易聘请到有才能的开发者。

以源码形式发布驱动程序，可以不必为每一个内核版本和补丁版本都提供二进制的程序。另外通过分析源代码，可以保证它没有安全隐患。

7.1.3 Linux 2.6 内核新特性

Linux 2.6 内核吸收了一些新技术，在性能、可量测性、支持和可用性方面不断提高。这些改进多数是添加支持更多的体系结构、处理器、总线、接口和设备；也有一些是标准化内部接口，简化扩展添加新设备和子系统的支持。

与 Linux 2.4 版本相比，Linux 2.6 版本具有许多新特性，内核也有很大修改。其中一些修改只跟内核或者驱动开发者有关，另外一些修改则会影响到系统启动、系统管理和应用程序开发。

Linux 2.6 内核重要的新特性如下。

(1) 新的调度器

Linux 2.6 版本的 Linux 内核使用了新的调度器算法，它是由 Ingo Molnar 开发的 O(1) 调度器算法。它在高负载的情况下极其出色，并且对多处理器调度有很好的扩展。

Linux 2.4 版本的标准调度器中，使用时间片重算的算法。这种算法要求在所有的进程都用尽时间片以后，重新计算下一次运行的时间片。这样每次任务调度的花销不确定，可能因为计算比较复杂，产生较大调度延迟。特别是多处理器系统，可能由于调度的延迟，导致大部分处理器处于空闲状态，影响系统性能。

新的调度器采用 O(1) 的调度算法，通过优先级数组的数据结构来实现。优先级数组可以使每个优先级都有相应的任务队列，还有一个优先级位图。每个优先级对应位图中一位，通过位图可以快速执行最高优先级任务。因为优先级个数是固定的，所以查找的时间也固定，不受系统运行任务数的影响。

新的调度器为每个处理器维护 2 个优先级数组：有效数组和过期数组。有效数组内任务队列的进程都还有可以运行的时间片；过期数组内任务队列的进程都已经没有时间片可以执行。当一个进程的时间片用光时，就把它从有效数组移到过期数组，并且时间片也已经重新计算好了。当需要重新调度这些任务的时候，只要在有效数组和过期数组之间切换就好了。这种交换是 O(1) 算法的核心。它根本不需要从头到尾重新计算所有任务的时间片，调度器的效率更高。

O(1) 调度器具有以下优点。

- SMP 效率高。如果有工作需要完成，那么所有处理器都会工作。
- 没有进程需要长时间地等待处理器；也没有进程会无端地占用大量的 CPU 时间。
- SMP 进程只映射到一个 CPU 而且不会在 CPU 之间跳跃。
- 不重要的任务可以设置低优先级，重要的任务设置高优先级。
- 负载平衡功能。调度器会降低那些超出处理器负载能力的进程的优先级。
- 交互性能提高。即使在高负载的情况下，也不会再发生长时间不响应鼠标点击或者键盘输入的情况。

(2) 内核抢占

Linux 2.6 采纳了内核抢占的补丁，大大减小了用户交互、多媒体等应用程序的调度延迟。这一特性对实时系统和嵌入式系统来说特别有用。这项工作是由 Robert Love 完成的。

在 Linux 2.4 以前的内核版本中，内核空间运行的任务（包括通过系统调用进入内核空间的用户任务）不允许被抢占，直到它们自己主动释放 CPU。

在 Linux 2.6 内核中，内核是可抢占的。一个内核任务可以被抢占，为的是让重要的用户应用程序可以继续运行。这样做可以极大地增强系统的用户交互性，用户将会觉得鼠标点击和击键的事件得到了更快速的响应。

当然，不是所有的内核代码段都可以被抢占。可以锁定内核代码的关键部分，不允许抢占。这样可以确保每个 CPU 的数据结构和状态始终受到保护。

(3) 新的线程模型

Linux 2.6 内核重写了线程框架。它也是由 Ingo Molnar 完成的。它基于一个 1:1 的线程模型，能够支持 NPTL (Native Posix Threading Library) 线程库。NPTL 是一个改进的 Linux 线程库，它是由 Molnar 和 Ulrich Drepper 合作开发的。

对于 2.4 内核的 Linux 线程库，存在一些不足。例如：总是需要一个管理线程，来负责创建和删除子线程，负责接收和分发信号等。如果系统中使用大量的线程，这种 Linux 线程库就存在严重的效率问题。

NPTL 线程库解决了传统的 Linux 线程库存在的问题，对系统有很大性能提升。实际上，RedHat 已经将它向后移植到了 Linux 2.4 内核，从 RedHat 9.0 版本就开始包含对它的支持。新的线程框架的改进包含 Linux 线程空间中的许多新的概念，包括线程组、线程各自的本地存储区、POSIX 风格的信号以及其他改进。

(4) 文件系统

相对于 Linux 2.4，Linux 2.6 对文件系统的支持在很多方面都有大的改进。关键的变化包括对扩展属性 (extended attributes) 以及 POSIX 标准的访问控制 (access controls) 的支持。

EXT2/EXT3 文件系统作为多数 Linux 系统缺省安装的文件系统，是在 2.6 中改进最大的一个。最主要的变化是对扩展属性的支持，也即给指定的文件在文件系统中嵌入一些元数据 (metadata)。新的扩展属性子系统的第一个用途就是实现 POSIX 访问控制链表。POSIX 访问控制是标准 UNIX 权限控制的超集，支持更细粒度的访问控制。EXT3 还有其他一些细微变化。

Linux 对文件系统层还进行了大量的改进以兼容其他操作系统。Linux 2.6 对 NTFS 文件系统的支持也进行了重写；同时也支持 IBM 的 JFS(journaling file system)和 SGI 的 XFS。

此外，Linux 文件系统中还有很多零散的变化。

(5) 声音

Linux 2.6 内核还添加了新的声音系统：ALSA(Advanced Linux Sound Architecture)。老的声音系统 OSS (Open Sound System)存在一些系统结构的缺陷。新的声音体系结构支持 USB 音频和 MIDI 设备，全双工重放等。

(6) 总线

Linux 2.6 的 IDE/ATA、SCSI 等存储总线也都被更新。最主要的是重写了 IDE 子系统，解决了许多可扩展性问题以及其他限制。其次是可以象微软的 Windows 操作系统那样检测介质的变动，以更好地兼容那些并不完全遵照标准规范的设备。

Linux 2.6 还大大提升了对 PCI 总线的支持，增强或者扩展了 USB、蓝牙 (Bluetooth)、红外 (IrDA) 等外围设备总线。所有的总线设备类型 (硬件、无线和存储) 都集成到了 Linux 新的设备模型子系统中。

(7) 电源管理

Linux 2.6 支持高级电源配置管理界面 (ACPI, Advanced Configuration and Power Interface)，最早 Linux 2.4 中有些支持。ACPI 不同于 APM (高级电源管理)，拥有这种接口的系统在改变电源状态时需要分别通知每一个兼容的设备。新的内核系统允许子系统跟踪需要进行电源状态转换的设备。

(8) 网络

Linux 是一种网络性能优越的操作系统，已经可以支持世界上大多数主流网络协议，包括 TCP/IP (IPv4/IPv6)、AppleTalk、IPX 等。

在网络硬件驱动方面，利用了 Linux 的设备模型底层的改进和许多设备驱动程序的升级。例如，Linux 2.6 提供一个独立的 MII (媒体独立接口，或是 IEEE 802.3u) 子系统，它被许多网络设备驱动程序使用。新的子系统替换了原先系统中各自运行的多个实例，消除了原先系统中多个驱动程序使用重复代码、采用类似的方法处理设备的 MII 支持的情况。

在网络安全方面，Linux 2.6 的一个重要改进是提供了对 IPsec 协议的支持。IPsec 是在网络协议层为 IPv4 和 IPv6 提供加密支持的一组协议。由于安全是在协议层提供的，对应用层是透明的。它与 SSL 协议及其他 tunneling/security 协议很相似，但是位于一个低得多的层面。当前内核支持的加密算法包括 SHA (“安全散列算法”)、DES (“数据加密标准”) 等。

在协议方面，Linux 2.6 还加强了对多播网络的支持。网络多播使得由一点发出的数据包可以被多台计算机接收 (传统的点对点网络每次只能有两方通信)。

还有其他一些改进。例如：IPv6 已经成熟；VLAN 的支持也已经成熟等。

(9) 用户界面层

Linux 2.6 中一个主要的内部改动是人机接口层的大量重写。人机接口层是一个 Linux 系统中用户体验的中心，包括视频输出、鼠标、键盘等。内核的新版本中，这一层的重写以及模块化工作超出了以前的任何一个版本。

Linux 2.6 对显示器输出处理的支持也有不少改进，但大部分只在配置使用内核内部的帧缓冲控制台子系统时才有用。

人机界面层还加入了对近乎所有可接入设备的支持，从触摸屏到盲人用的设备，到各种各样的鼠标。

(10) 统一的设备模型

Linux 2.6 内核最值得关注的变化是创建了一个统一的设备模型。这个设备模型通过维持大量的数据结构囊括了几乎所有的设备结构和系统。这样做的好处是，可以改进设备的电源管理和简化设备相关的任务管理。

这种设备模型可以跟踪获取以下信息。

- 系统中存在的设备，其所连接的总线。
- 特定情形下设备的电源状态。
- 系统清楚设备的驱动程序，并清楚哪些设备受其控制。
- 系统的总线结构，哪个设备连接在哪个总线上，以及哪些总线互连（例如，USB 和 PCI 总线的互连）。
- 设备在系统中的类别描述（类别包括磁盘，分区等）。

Linux 2.6 内核引入了 sysfs 文件系统，提供了系统的设备模型的用户空间描述。通常 sysfs 文件系统挂载在/sys 目录下。

7.2 配置编译内核源码

在广大爱好者的支持下，Linux 内核版本不断更新。新的内核修订了旧内核的 bug，并增加了许多新的特性。如果用户想要使用这些新特性，或想根据自己的系统度身定制一个更高效、更稳定的内核，就需要重新编译内核。

通常，新的内核会支持更多的硬件，具备更好的进程管理能力，运行速度更快、更稳定，并且一般会修复老版本中发现的许多漏洞等，经常性地选择升级更新的系统内核是 Linux 使用者的必要操作内容。

为了正确、合理地设置内核编译配置选项，从而只编译系统需要的功能的代码，一般主要有下面 4 个考虑。

- (1) 尺寸小。自己定制内核可以使代码尺寸减小，运行将会更快。
- (2) 节省内存。由于内核部分代码永远占用物理内存，定制内核可以使系统拥有更多的可用物理内存。
- (3) 减少漏洞。不需要的功能编译进入内核可能会增加被系统攻击者利用的机会。
- (4) 动态加载模块。根据需要动态地加载或者卸载模块，可以节省系统内存。但是，将某种功能编译为模块方式会比编译到内核内的方式速度要慢一些。

7.2.1 内核源码结构

由于内核版本是不断升级更新的，最好下载使用最新版本的内核源代码。但是，有时候也需要比较分析老版本的内核。

浏览 <http://kernel.org> 站点，可以查看 Linux 官方发布的内核版本，从而确定需要的内核版本。然后通过 HTTP 或者 FTP 下载相应的源码包。

Linux 的下载工具，例如：gftp、kget、wget 等。其中，wget 下载工具就很好用，它可

以支持 FTP 和 HTTP，还支持断点续传，不过是命令行的方式。下面都以 wget 为例来下载源码包。

下例就是下载内核源码包和电子签名文件到当前目录。由于源码包一般都在 30MB 以上，可以使用断点续传的下载方式，加上选项“-c”。下载命令如下。

```
$ wget -c http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.14.tar.bz2
$ wget http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.14.tar.bz2.sign
```

下载完成以后，先验证一下电子签名。

```
$ gpg -verify linux-2.4.26.tar.bz2.sign
```

如果没有问题，就可以使用源码包了。可以把源码包解压到工作目录下。

```
$ cd ~/workspace
$ tar jxvf ~/linux-2.6.14.tar.bz2
```

新版本的内核分两种，一种是完整源码版本，另外一种 patch 文件，即补丁。完整的内核版本比较大，一般是 tar.gz 或者是 bz2 文件，二者分别是使用 gzip 或者 bzip2 进行压缩的文件，使用时需要解压缩。patch 文件则比较小，一般只有几十 K 到几百 K，但是 patch 文件是针对于特定的版本的，你需要找到自己对应的版本才能使用。

每一个补丁都反映了最近的 2 个正式版本之间的差别。也就是说，上一个版本的 Linux 内核源码，通过打补丁可以得到下一个版本。

另外，Linux 社区经常有开发版本、分支版本或者非官方修改，都是以补丁的形式发布的。

假设已经下载了 Linux-2.6.14 版本，kernel.org 又发布了 Linux-2.6.15 内核。这时下载补丁 patch-2.6.14.15，就可以升级到新的版本。下载命令如下。

```
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/patch-2.6.14.15.bz2
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/patch-2.6.14.15.bz2.sign
```

下载完成后，也要检查电子签名。

```
$ gpg -verify patch-2.6.14.15.bz2.sign
```

通常使用 *.bz2 文件，因为 *.bz2 的压缩比更高一些。这里我们不需要解压补丁文件，直接使用 bzip2 来读取文件信息就行了。

```
$ cd linux-2.6.14/
$ bzip2 .. /patch-2.6.14.15.bz2 | patch -p1
```

上面通过管道的方式，把补丁内容传递给 patch 命令，应用到内核源代码中去。然后，可以把 Linux-2.6.14 的目录名称改成 Linux-2.6.15，就得到新版本的 Linux 内核源码了。

那么补丁文件是什么呢？不妨分析一下补丁文件的内容。补丁文件是通过 diff 命令比较两个源码目录中文件的结果，把两个目录中所有文件的变化体现出来。下面是补丁文件中的

一段，说明了 Makefile 文件的一些修改。

```
diff --git a/Makefile b/Makefile
index 1fa7e53..497884d 100644
--- a/Makefile
+++ b/Makefile
@@ -1,8 +1,8 @@
VERSION = 2
PATCHLEVEL = 6
-SUBLEVEL = 14
+SUBLEVEL = 15
EXTRAVERSION =
-NAME=Affluent Albatross
+NAME=Sliding Snow Leopard
```

上面是 a 目录和 b 目录比较的结果，也就是从 a 目录到 b 目录的变化。“-”表示删除当前行，“+”表示添加当前行，这样可以实现代码的修改替换。上面的 SUBLEVEL 从 14 变成了 15。

patch 命令可以根据补丁文件内容修改指定目录下的文件。几种命令使用方式如下。

```
$ patch -p<n> < diff_file
$ cat diff_file | patch -p<n>
$ bzcat diff_file.bz2 | patch -p<n>
$ zcat diff_file.gz | patch -p<n>
```

其中，<n>代表按照 patch 文件的路径忽略的目录级数，每个“/”代表一级。例如：

p0 是完全按照补丁文件中的路径查找要修改的文件；

p1 则使用去掉第一级“/”得到相对路径，再基于当前目录，到相应的相对路径下查找要修改的文件。

接下来，就可以仔细阅读内核源代码。Linux 内核源代码非常庞大，随着版本的发展不断增加。它使用目录树结构，并且使用 Makefile 组织配置编译。

初次接触 Linux 内核，要仔细阅读顶层目录的 readme，它是 Linux 内核的概述和编译命令说明。readme 的说明更加针对 X86 等通用的平台，对于某些特殊的体系结构，可能有些特殊的地方。

顶层目录的 Makefile 是整个内核配置编译的核心文件，负责组织目录树中子目录的编译管理，还可以设置体系结构和版本号等。

内核源码的顶层有许多子目录，分别组织存放各种内核子系统或者文件。具体的目录说明见表 7.1。

表 7.1 Linux 内核源码顶层目录说明

arch/	体系结构相关的代码，例如：arch/i386，arch/arm，arch/ppc
-------	--

crypto	
drivers/	各种设备驱动程序，例如：drivers/char drivers/block ...
Documentation/	内核文档
fs/	文件系统，例如：fs/ext3/ fs/jffs2 ...
include/	内核头文件： include/asm 是体系结构相关的头文件，它是 include/asm-arm、include/asm-i386 等目录的链接。 include/linux 是 Linux 内核基本的头文件
init/	Linux 初始化，例如：main.c
ipc/	进程间通信的代码
kernel/	Linux 内核核心代码（这部分很小）
lib/	各种库子程序，例如：zlib， crc32
mm/	内存管理代码
net/	网络支持代码，主要是网络协议
sound	声音驱动的支持
scripts/	内部或者外部使用的脚本
usr/	用户的代码

7.2.2 内核配置系统

Linux 内核源代码支持二十多种体系结构的处理器，还有各种各样的驱动程序等选项。因此，在编译之前必须根据特定平台配置内核源代码。Linux 内核有上千个配置选项，配置相当复杂。所以，Linux 内核源代码组织了一个配置系统。

Linux 内核配置系统可以生成内核配置菜单，方便内核配置。配置系统主要包含 Makefile、Kconfig 和配置工具，可以生成配置界面。配置界面是通过工具来生成的，工具通过 Makefile 编译执行，选项则是通过各级目录的 Kconfig 文件定义。

Linux 内核配置命令有：make config、make menuconfig 和 make xconfig。分别是字符界面、ncurses 光标菜单和 X-window 图形窗口的配置界面。字符界面配置方式需要回答每一个选项提示，逐个回答内核上千个选项几乎是行不通的；图形窗口的配置界面很好，光标菜单也方便实用。例如执行 make xconfig，主菜单界面如图 7.1 所示。

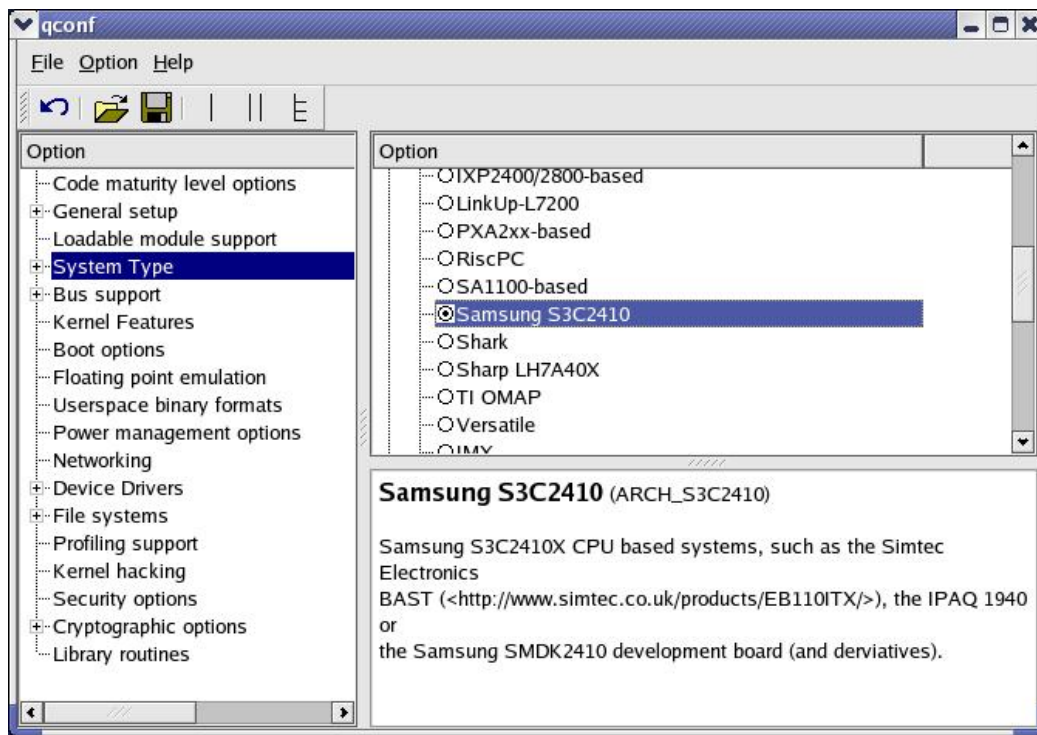


图 7.1 内核图形配置界面

那么这个配置界面到底是如何生成的呢？这里结合配置系统的 3 个部分分析一下。

1. Makefile

Linux 内核的配置编译都是由顶层目录的 Makefile 整体管理的。顶层目录的 Makefile 定义了配置和编译的规则。关于 Makefile 的具体使用方法可以参考第 3 章的内容，这里重点分析相关的变量和规则。

参考内核源码包中的 Documentation/kbuild/makefiles.txt，可以得到内核使用 Makefile 的详细说明。

在顶层的 Makefile 中，可以查找到如下几行定义的规则。

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

这就是生成内核配置界面的命令规则，它也定义了执行的目标和依赖的前提条件，还有要执行的命令。

这条规则定义的目标为 config %config，通配符 % 意味着可以包括 config、xconfig、gconfig、menuconfig 和 oldconfig 等。依赖的前提条件是 scripts_basic outputmakefile，这些在 Makefile 也是规则定义，主要用来编译生成配置工具。

那么这条规则执行的命令就是执行 scripts/kconfig/Makefile 指定的规则。相当于：

```
make -C scripts/kconfig/ config
```

或者

```
make -C scripts/kconfig/ %config
```

这两行命令是使用配置工具解析 arch/\$(ARCH)/Kconfig 文件，生成内核配置菜单。\$(ARCH)变量是 Linux 体系结构定义，对应 arch 目录下子目录的名称。Kconfig 包含了内核配置菜单的内容，那么 arch/\$(ARCH)/Kconfig 是配置主菜单的文件，调用管理其他各级 Kconfig。

根据配置工具的不同，内核也有不同的配置方式。有命令行方式，还有图形界面方式。表 7.2 是各种内核配置方式的说明。

表 7.2 内核配置方式说明

配 置 方 式	功 能
config	通过命令行程序更新当前配置
menuconfig	通过菜单程序更新当前配置
xconfig	通过 QT 图形界面更新当前配置
gconfig	通过 GTK 图形界面更新当前配置
oldconfig	通过已经提供的.config 文件更新当前配置
randconfig	对所有的选项随机配置
defconfig	对所有选项使用缺省配置
allmodconfig	对所有选项尽可能选择“m”
allyesconfig	对所有选项尽可能选择“y”
allnoconfig	对所有选项尽可能选择“n”的最小配置

这些内核配置方式是在 scripts/kconfig/Makefile 中通过规则定义的。从这个 Makefile 中，可以找到下面一些规则定义。如果把变量或者通配符带进去，就可以明白要执行的操作。这里的 ARCH 以 arm 为例来说明。

```
xconfig: $(obj)/qconf
    $< arch/$(ARCH)/Kconfig
```

执行命令：scripts/kconfig/qconf arch/arm/Kconfig

使用 QT 图形库，生成内核配置界面。arch/arm/Kconfig 是菜单的主配置文件，每种配置方式都需要。

```
gconfig: $(obj)/gconf
    $< arch/$(ARCH)/Kconfig
```

执行命令：scripts/kconfig/gconf arch/arm/Kconfig

使用 GTK 图形库，生成内核配置界面。

```
menuconfig: $(obj)/mconf
$(Q)$ (MAKE) $(build)=scripts/lxdialog
$< arch/$(ARCH)/Kconfig
```

执行命令: `scripts/kconfig/mconf arch/arm/Kconfig`
 使用 `lxdialog` 工具, 生成光标配置菜单。
 因为 `mconf` 调用 `lxdialog` 工具, 所以需要先编译 `scripts/lxdialog` 目录。

```
config: $(obj)/conf
$< arch/$(ARCH)/Kconfig
```

执行命令: `scripts/kconfig/conf arch/arm/Kconfig`
 完全命令行的内核配置方式。

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

执行命令: `scripts/kconfig/conf -o arch/arm/Kconfig`
 完全命令行的内核配置方式。使用“-o”选项, 直接读取已经存在的`.config` 文件, 要求确认内核新的配置项。

```
silentoldconfig: $(obj)/conf
$< -s arch/$(ARCH)/Kconfig
```

执行命令: `scripts/kconfig/conf -s arch/arm/Kconfig`
 完全命令行的内核配置方式。使用“-s”选项, 直接读取已经存在的`.config` 文件, 提示但不要求确认内核新的配置项。

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

执行命令: `scripts/kconfig/conf -D arch/arm/configs/_defconfig arch/arm/Kconfig`
 完全命令行的内核配置方式。读取缺省的配置文件 `arch/arm/configs/_defconfig`, 另存成`.config` 文件。

通过上述各种都可以完成配置内核的工作, 在顶层目录下生成`.config` 文件。这个`.config` 文件保存大量的内核配置项, `.config` 会自动转换成 `include/linux/autoconf.h` 头文件。在 `include/linux/config.h` 文件中, 将包含使用 `include/linux/autoconf.h` 头文件。

2. 配置工具

不同的内核配置方式, 分别通过不同的配置工具来完成。scripts 目录下提供了各种内核配置工具, 表 7.3 是这些工具的说明。

表 7.3 内核配置工具说明

配置工具	Makefile 相关目标	依赖的程序和软件
------	---------------	----------

conf	defconfig oldconfig ...	conf.c zconf.tab.c
mconf	menuconfig	mconf.c zconf.tab.c 调用 scripts/ldialog/ldialog
qconf	xconfig	qconf.c kconfig_load.c zconf.tab.c 基于 QT 软件包实现图形界面
gconf	gconfig	gconf.c kconfig_load.c zconf.tab.c 基于 GTK 软件包实现图形界面

其中 zconf.tab.c 程序实现了解析 Kconfig 文件和内核配置主要函数。zconf.tab.c 程序还直接包含了下列一些 C 程序，这样各种配置功能都包含在 zconf.tab.o 目标文件中了。

```
#include "lex.zconf.c"      //lex 语法解析器
#include "util.c"           //配置工具
#include "confdata.c"       //.config 等相关数据文件保存
#include "expr.c"           //表达式函数
#include "symbol.c"         //变量符号处理函数
#include "menu.c"           //菜单控制函数
```

理解这些工具的使用，可以更加方便地配置内核。至于这些工具的源代码实现，一般没有必要去详细分析。

3. Kconfig

Kconfig 文件是 Linux 2.6 内核引入的配置文件的源文件。内核源码中的 Documentation/kbuild/kconfig-language.txt 文档有详细说明。

前面已经提到了 arch/\$(ARCH)/Kconfig 文件，这是主 Kconfig 文件，跟体系结构有关系。主 Kconfig 文件调用其他目录的 Kconfig 文件，其他的 Kconfig 文件又调用各级子目录的配置文件的，成树状关系。

菜单按照树状结构组织，主菜单下有子菜单，子菜单还有子菜单或者配置选项。每个选项可以有依赖关系，这些依赖关系用于确定它是否显示。只有被依赖项父项已经选中，子项才会显示。

下面解释一下 Kconfig 的特点和语法。

(1) 菜单项

多数选项定义一个配置选项，其他选项起辅助组织作用。举例说明单个的配置选项的定义。

```
config MODVERSIONS
    bool "Set version information on all module symbols"
    depends MODULES
    help
        Usually, modules have to be recompiled whenever you switch to a new
        kernel. ...
```

每一行开头用关键字“config”，后面可以跟多行。后面的几行定义这个配置选项的属性。属性包括配置选项的类型、选择提示、依赖关系、帮助文档和缺省值。同名的选项可以重复定义多次，但是每次定义只有一个选择提示并且类型不冲突。

(2) 菜单属性

一个菜单选项可以有多种属性，不过这些属性也不是任意用的，受到语法的限制。

每个配置选项必须有类型定义。类型定义包括：bool、tristate、string、hex、int 共 5 种。其中有 2 种基本的类型：tristate 和 string，每种类型定义可以有一个选择提示。表 7.4 说明了菜单的各种属性。

表 7.4 内核菜单属性说明

属 性	语 法	说 明
选择提示	"prompt" <prompt> ["if" <expr>]	每个菜单选项最多有一条提示，可以显示在菜单上。某选择提示可选的依赖关系可以通过“if”语句添加
缺省值	"default" <expr> ["if" <expr>]	配置选项可以有几个缺省值。如果有多个缺省值可选，只使用第一个缺省值。某选项缺省值还可以在其他地方定义，并且被前面定义的缺省值覆盖。如果用户没有设置其他值，缺省值就是配置符号的唯一值。如果有选择提示出现，就可以显示缺省值并且可以配置修改。某缺省值可选的依赖关系可以通过“if”语句添加
依赖关系	"depends on"/"requires" <expr>	这个定义了菜单选项的依赖关系。如果定义多个依赖关系，那么要用“&&”符号连接。依赖关系对于本菜单项中其他所有选项有效（也可以用“if”语句）
反向依赖	"select" <symbol> ["if" <expr>]	普通的依赖关系是缩小符号的上限，反向依赖关系则是符号的下限。当前菜单符号的值用作符号可以设置的最小值。如果符号值被选择了多次，这个限制将被设成最大选择值。反向依赖只能用于布尔或者三态符号
数字范围	"range" <symbol> <symbol> ["if" <expr>]	这允许对 int 和 hex 类型符号的输入值限制在一定范围内。用户输入的值必须大于等于第一个符号值或者小于等于第二个符号值
帮助文档	"help" 或者 "---help---"	这可以定义帮助文档。帮助文档的结束是通过缩进层次判断的。当遇到一行缩进比帮助文档第一行小的时候，就认为帮助文档已经结束。“---help---”和“help”功能没有区别，主要给开发者提供的不同于“help”的帮助

(3) 菜单依赖关系

依赖关系定义了菜单选项的显示，也能减少三态符号的选择范围。表达式的三态逻辑比布尔逻辑多一个状态，用来表示模块状态。表 7.5 是菜单依赖关系的语法说明。

表 7.5 菜单依赖关系语法说明

表 达 式	结 果 说 明
<expr> ::= <symbol>	把符号转换成表达式，布尔和三态符号可以转换成对应的表达式值。其他类型符号的结果都是“n”
<symbol> '=' <symbol>	如果两个符号的值相等，返回“y”，否则返回“n”

续表

表 达 式	结 果 说 明
<symbol> '!=' <symbol>	如果两个符号的值相等，返回“n”，否则返回“y”
'(' <expr> ')'	返回表达式的值，括号内表达式优先计算
'!' <expr>	返回(2-/expr/)的计算结果
<expr> '&&' <expr>	返回 min(/expr/, /expr/)的计算结果
<expr> ' ' <expr>	返回 max(/expr/, /expr/)的计算结果

一个表达式的值是“n”、“m”或者“y”（或者对应数值的 0、1、2）。当表达式的值为“m”或者“y”时，菜单选项变为显示状态。

符号类型分为两种：常量和非常量符号。

非常量符号最常见，可以通过 config 语句来定义。非常量符号完全由数字符号或者下划线组成。

常量符号只是表达式的一部分。常量符号总是包含在引号范围内的。在引号中，可以使用其他字符，引号要通过“\”号转义。

(4) 菜单组织结构

菜单选项的树状结构有两种组织方式。

第一种是显式的声明为菜单。

```
menu "Network device support"
    depends NET
    config NETDEVICES
    ...
endmenu
```

“menu”与“endmenu”之间的部分成为“Network device support”的子菜单。所有子选项继承这菜单的依赖关系，例如，依赖关系“NET”就被添加到“NETDEVICES”配置选项的依赖关系列表中。

第二种是通过依赖关系确定菜单的结构。

如果一个菜单选项依赖于前一个选项，它就是一个子菜单。这要求前一个选项和子选项同步地显示或者不显示。

```
config MODULES
    bool "Enable loadable module support"

config MODVERSIONS
    bool "Set version information on all module symbols"
    depends MODULES

comment "module support disabled"
    depends !MODULES
```

MODVERSIONS 依赖于 MODULES，这样只有 MODULES 不是“n”的时候，才显示。反之，MODULES 是“n”的时候，总是显示注释“module support disabled”。

(5) Kconfig 语法

Kconfig 配置文件描述了一系列的菜单选项。每一行都用一个关键字开头 (help 文字例外)。菜单的关键字见表 7.6 所示。其中菜单开头的关键字有: config、menuconfig、choice/endchoice、comment、menu/endmenu。它们也可以结束一个菜单选项, 另外还有 if/endif、source 也可以结束菜单选项。

表 7.6 Kconfig 菜单关键字说明

关 键 字	语 法	说 明
config	"config" <symbol> <config options>	这可以定义一个配置符号<symbol>, 并且可以配置选项属性
menuconfig	"menuconfig" <symbol> <config options>	这类似于简单的配置选项, 但是它暗示: 所有的子选项应该作为独立的选项列表显示
choices	"choice" <choice options> <choice block> "endchoice"	这定义了一个选择组, 并且可以配置选项属性。每个选择项只能是布尔或者三态类型。布尔类型只允许选择单个配置选项, 三态类型可以允许把任意多个选项配置成“m”。如果一个硬件设备有多个驱动程序, 内核一次只能静态链接或者加载一个驱动, 但是所有的驱动程序都可以编译为模块。 选择项还可以接受另外一个选项“optional”, 可以把选择项设置成“n”, 并且不需要选择什么选项
comment	"comment" <prompt> <comment options>	这定义了一个注释, 在配制过程中显示在菜单上, 也可以回显到输出文件中。唯一可能的选项是依赖关系
Menu	"menu" <prompt> <menu options> <menu block> "endmenu"	这定义了一个菜单项, 在菜单组织结构中有些描述。唯一可能的选项是依赖关系
If	"if" <expr> <if block> "endif"	这定义了一个 if 语句块。依赖关系表达式<expr>附加给所有封装好的菜单选项
Source	"source" <prompt>	读取指定的配置文件。读取的文件也会解析生成菜单

7.2.3 Kbuild Makefile

Linux 内核源代码是通过 Makefile 组织编译的。

1. Makefile 的组织结构

Makefiles 包含 5 个部分, 见表 7.7 所示。

表 7.7 Makefiles 的 5 个部分

Makefile	顶层目录下的 Makefile
.config	内核配置文件
arch/\$(ARCH)/Makefile	对应体系结构的 Makefile

续表

Makefile	顶层目录下的 Makefile
scripts/Makefile.*	所有 kbuild Makefiles 的通用规则等定义
kbuild Makefiles	内核编译各级目录下的 Makefile，大约有 500 多个

顶层目录的 Makefile 读取.config 文件，根据.config 文件中的配置选项编译内核。这个.config 文件是内核配置过程生成的。

顶层目录的 makefile 负责编译 vmlinux（常驻内存的内核映像）和 module（任何模块文件）。它递归地遍历内核源码树中所有子目录，编译所有的目标文件。

编译访问的子目录列表依赖于内核配置。顶层目录的 Makefile 原原本本的包含了一个 arch Makefile（后面将使用这个英文名称），就是 arch/\$(ARCH)/Makefile。这个 arch Makefile 给顶层目录提供了体系结构相关的信息。

每个子目录都有一个 Kbuild Makefile（内核编译过程调用），这些 Makefile 执行从上层传递下来的命令。这些 Makefile 使用.config 文件中的信息，构建各种文件列表，由 Kbuild 编译静态链接的或者模块化的目标程序。

scripts/Makefile.*几个文件包含了 Kbuild Makefile 所有的定义和规则等，用于编译内核。

内核源码的大多数 Makefile 是 Kbuild Makefile，使用 Kbuild 组织结构。下面介绍一下 Kbuild Makefile 的语法。

Kbuild 大体上按照下列步骤执行编译过程。

- （1）内核配置，生成.config 文件。
- （2）保存内核版本信息到 include/linux/version.h。
- （3）创建链接符号 include/asm，链接 include/asm-\$(ARCH)源目录。
- （4）升级所有依赖的前提文件，在 arch/\$(ARCH)/Makefile 中指定附加依赖条件。
- （5）递归地遍历各级子目录并且编译所有的目标。

init-*、core*、drivers-*、net-*、libs-*的目录变量值在 arch/\$(ARCH)/Makefile 文件中有些扩展。

（6）链接所有的目标文件，生成顶层目录的 vmlinux。链接的第一个目标文件在 head-y 列表中，是在 arch/\$(ARCH)/Makefile 中定义的。

（7）最后，体系结构相关的部分作必须的后期处理，编译生成最终的引导映像。这可以包括编译引导记录；准备 initrd 映像等类似工作。

2. Makefile 语言

内核 Makefile 是配合 GNU make 使用的。除了 GNU make 的文档中的特点，内核的 Makefile 还有一些 GNU 扩展的功能。

GNU make 支持基本的链接表处理功能。内核 Makefile 使用新颖的编译列表格式，编译过程几乎可以不用 if 语句。

GNU make 有多种变量赋值操作符：“=”、“:=”、“?=”、“+=”。

第 1 种是“=”操作符，在“=”左侧是变量，右侧是变量的值，右侧变量的值可以定义在文件的任何一处，也就是说，右侧中的变量不一定非要是已定义好的值，其也可以使用后

面定义的值。

可以把变量的真实值推到后面来定义。但是这种形式也有不好的地方，那就是递归定义，这会让 `make` 陷入无限的变量展开过程中去，当然，`make` 是有能力检测这样的定义，并会报错的。还有就是如果在变量中使用函数，那么，这种方式会让 `make` 运行时非常慢，更糟糕的是，会使得 `wildcard` 和 `shell` 两个函数发生不可预知的错误。因为不会知道这两个函数会被调用多少次。

第 2 种是 “:=” 操作符，这种方法，前面的变量不能使用后面的变量，只能使用前面已定义好了的变量。如果是这样：

```
y := $(x) bar
x := foo
```

那么，`y` 的值是 “bar”，而不是 “foo bar”。

第 3 种是 “?=” 操作符，先看示例：

```
FOO ?= bar
```

其含义是：如果 `FOO` 没有被定义过，那么变量 `FOO` 的值就是 “bar”；如果 `FOO` 先前被定义过，那么这条语将什么也不做。

第 4 种是 “+=” 操作符，将右边的变量值附加给左边的变量。例如：

```
FOO = string1
FOO += string2
```

这时，`FOO` 的变量值为 “string1 string2”。

3. Kbuild 变量

顶层 Makefile 输出下列变量。

(1) `VERSION`, `PATCHLEVEL`, `SUBLEVEL`, `EXTRAVERSION` 定义了当前内核版本。

`$(VERSION)`、`$(PATCHLEVEL)`和`$(SUBLEVEL)`定义了基本的 3 个版本号，例如：2、6 和 14，都是数字，对应内核版本号。

`$(EXTRAVERSION)`为预先或者附加的补丁定义了更细的子版本号。通常是非数字的字符串或者空的，例如：-mm1。

(2) `KERNELRELEASE` 定义了内核发布的版本，一般是单个的字符串，例如：2.6.14。常用来作为版本显示。

(3) `ARCH` 定义了目标板体系结构，例如：i386、arm 或者 sparc。一些 kbuild Makefile 测试`$(ARCH)`来确定要编译哪一个文件。顶层 Makefile 缺省地把`$(ARCH)`设置成主机系统的体系结构。对于交叉编译，需要修改定义或者在命令行重载这个值。例如：

```
make ARCH=arm
```

(4) `INSTALL_PATH` 为 arch Makefile 定义了安装驻留内存的内核映像和 `System.map` 文件。使用这个体系结构安装目标板。

(5) `INSTALL_MOD_PATH` 和 `MODLIB`。

`$(INSTALL_MOD_PATH)` 在安装模块的时候作为 `$(MODLIB)` 的前缀。这个变量没有在 `Makefile` 中定义，但是可以通过命令行传递。

`$(MODLIB)` 指定模块安装的路径。顶层 `Makefile` 的 `$(MODLIB)` 缺省定义如下。

`$(INSTALL_MOD_PATH)/lib/modules/$(KERNELRELEASE)`

4. Kbuild Makefile 的定义

(1) 目标定义

目标定义 `kbuild` `Makefile` 的核心。它们定义了要编译的文件、特殊的编译选项和要递归地遍历的子目录。

最简单的 `kbuild` `makefile` 包含一行，例如：

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

这是告诉 `kbuild`，当前目录中要编译一个目标文件 `foo.o`，`foo.o` 应该从 `foo.c` 或者 `foo.S` 编译过来。

如果要把 `foo.o` 编译为模块，就使用变量 `obj-m`。因此，经常用下列方式：

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

`$(CONFIG_FOO)` 可以配置为 `y`（静态链接）或者 `m`（动态模块）。如果 `CONFIG_FOO` 即不是 `y`，也不是 `m`，那么这个文件就不被编译或者链接。

(2) 静态链接目标文件- `obj-y`

`kbuild` `Makefile` 指定了 `vmlinux` 的目标文件，就在 `$(obj-y)` 列表中。这些列表依赖于内核的配置。

`Kbuild` 编译所有的 `$(obj-y)` 文件，再用 `$(LD)` 命令把目标文件链接成一个 `built-in.o` 文件。然后 `built-in.o` 将被链接到顶层目录的 `vmlinux` 中去。

`$(obj-y)` 中的文件顺序很重要。因为列表中允许重复，第一个实例链接到 `built-in` 中以后，后面的实例将被忽略。另外，某些函数（`module_init()` / `__initcall`）会在启动过程中按照排列顺序调用。如果改变链接顺序，也可能改变设备的初始化顺序。例如：改变 `SCSI` 控制器的探测顺序，就会导致磁盘重复编号。

举例说明 `obj-y`。

```
#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
```

(3) 可加载模块目标文件- `obj-m`

`$(obj-m)` 用来指定要编译成可加载模块的目标文件。

一个模块可以由一个或者几个源文件编译生成。对于单个源文件的情况，`kbuild` `Makefile` 可以简单地把文件添加到 `$(obj-m)` 中即可。

例如：

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



注意

这里的\$(CONFIG_ISDN_PPP_BSDCOMP)配置为“m”。

如果内核模块由几个源文件编译生成，要指定要编译成一个模块。Kbuild 需要知道这个模块包含哪些目标文件，那么必须设置一个\$(<module_name>-objs)变量。

例如：

```
#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，模块的名字是 isdn.o。Kbuild 会编译\$(isdn-objs)列表中的目标文件，然后执行“\$(LD) -r”命令，把这些目标文件链接成 isdn.o。

Kbuild 可以通过-objs 和-y 后缀识别组成复合目标的目标文件。这允许 Makefile 通过 CONFIG_符号的值来确定一个目标文件是不是一个复合目标文件。

例如：

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

这个例子中，如果\$(CONFIG_EXT2_FS_XATTR)配置为“y”，xattr.o 就是复合目标 ext2.o 的一部分。



注意

当把这些目标文件编译链接到内核中去的时候，上面的语法仍然有效。如果配置 CONFIG_EXT2_FS=y，kbuild 会单独编译 ext2.o 文件，再链接到 built-in.o 文件中。

(4) 库目标文件 lib-y

用 obj-列出的目标文件可以用于模块或者指定目录的 built-in.o 文件，也可以列出要包含到一个库 lib.a 中的目标文件。用 lib-y 列出的目标文件可以组合到目录下的一个库中。在 obj-y 中列出并且在 lib-y 中列出的目标文件不会包含到这个库中，因为它们是内核可以访问的。为了一致性，在 lib-m 中列出的目标文件会包含到 lib.a 中。



注意

同一个 kbuild makefile 可以把文件列到 built-in 表中，同时还是一个库的列表的一部分。因此，同一个目录可以包含一个 built-in.o 和一个 lib.a 文件。

例如：

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


这会基于 checksum.o 和 delay.o 创建一个 lib.a 文件。为了让 kbuild 知道有一个 lib.a 要编译，这个目录应该添加到 libs-y 列表中。

lib-y 一般仅限于 lib/和 arch/*/lib/目录。

(5) 遍历子目录

一个 Makefile 只负责在自己的目录下编译目标文件。各子目录下的文件应该由各自的 Makefile 来管理。编译系统会自动在子目录中递归地调用 make。

这项工作也要用到 obj-y 和 obj-m。比如 ext2 在一个单独的目录中，在 fs 目录下的 Makefile 使用下面的配置方法。

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

如果 CONFIG_EXT2_FS 设成“y”或者“m”，对应的 obj-变量就会设置，并且 Kbuild 就会下到 ext2 目录中编译。Kbuild 只使用这个信息决定是否访问这个目录，编译的工作是子目录中的 Makefile 负责。

对于 CONFIG_选项既不是“y”也不是“m”的目录，使用 CONFIG_变量可以让 Kbuild 忽略掉。这是一个很好的办法。

(6) 编译标志

编译标志包括：EXTRA_CFLAGS、EXTRA_AFLAGS、EXTRA_LDFLAGS、EXTRA_ARFLAGS。

所有的 EXTRA_变量只适用于当前的 Kbuild makefile。EXTRA_变量适用 Kbuild makefile 中执行的所有的命令。

\$(EXTRA_CFLAGS)通过\$(CC)指定编译 C 文件的选项。

例如：

```
# drivers/sound/emul0k1/Makefile
EXTRA_CFLAGS += -I$(obj)
ifdef DEBUG
    EXTRA_CFLAGS += -DEMU10K1_DEBUG
endif
```

因为顶层目录 Makefile 的变量\$(CFLAGS)用于整个源码树的编译，所以这种变量定义是必须的。

在编译汇编语言源码的时候，\$(EXTRA_AFLAGS)是与每个目录的选项类似的字符串。

例如：

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

\$(EXTRA_LDFLAGS)和\$(EXTRA_ARFLAGS)分别是与每个目录\$(LD)和\$(AR)的选项类似的字符串。

例如：

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

CFLAGS_@, AFLAGS_@

CFLAGS_@和 AFLAGS_@仅适用于当前 kbuild makefile 的命令。

\$(CFLAGS_@)为\$(CC)指定每个文件的选项。@代表指定的文件名。

例如：

```
# 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
```

这些行指定了 aha152x.o、gdth.o 和 seagate.o 的编译标志。

\$(AFLAGS_@)对于汇编语言编译有类似的特点。

例如：

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

(7) 依赖跟踪

Kbuild 按照下列步骤跟踪依赖关系。

- 所有依赖的前提文件 (*.c 和 *.h 文件)
- 在依赖的前提文件中用到的 CONFIG_选项
- 编译目标文件用到的命令行

因此，如果修改\$(CC)的一个选项，所有相关的文件都会重新编译。

(8) 特殊的规则

当 Kbuild 结构不提供必须的支持的时候，要使用特殊规则。一个典型的例子是在编译过程中生成头文件。另外一个例子是体系结构相关的 Makefile 需要特殊的规则准备映像等。

特殊的规则可以按照普通的规则来写。Kbuild 不在 Makefile 所在的目录执行，因此所有特殊的规则应该为依赖的文件和目标文件提供相对路径。

定义特殊规则的时候，常用到两个变量：\$(src)和\$(obj)。

\$(src)是指向 Makefile 所在的目录的相对路径。当引用位于源码树的文件的时候，总是使用\$(src)。

\$(obj)是指向目标文件保存的相对路径。当引用生成文件的时候，总是使用\$(obj)。

例如：

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

这是一个特殊规则，遵守普通的 make 语法 `uo$(src)` 前缀。

5. 体系结构相关的 Makefile 定义

顶层的 Makefile 在开始遍历各级子目录之前，要设置环境变量和做准备工作。

顶层目录 Makefile 包含通用的部分，`arch/$(ARCH)/Makefile` 则包含了设置 Kbuild 指定的体系结构需要的东西。因此，`arch/$(ARCH)/Makefile` 设置一些变量并且定义一些目标规则。

(1) 通过变量设置编译体系结构相关代码

`LDFLAGS_vmlinux` 是用来指定 `vmlinux` 额外的编译标志，在链接最终的 `vmlinux` 时传递给链接器，通过 `LDFLAGS_@$` 调用。

例如：

```
#arch/arm/Makefile
LDFLAGS_vmlinux := -p --no-undefined -X
```

`CFLAGS` 是 `$(CC)` 编译选项标志。缺省值在顶层 Makefile 中定义。对于不同的体系结构，有附加选项。通常 `CFLAGS` 变量依赖于内核配置。

例如：

```
arch-$(CONFIG_CPU_32v4) := -D__LINUX_ARM_ARCH__=4 -march=armv4
CFLAGS += $(CFLAGS_ABI) $(arch-y) $(tune-y)
```

许多体系结构相关的 Makefile 通过目标板 C 编译器动态地探测支持选项。比如可以把相关选项中的配置选项扩展为 “y”。

`CFLAGS_KERNEL` 是 `$(CC)` 编译 `built-in` 的专用选项。它包含了用于编译驻留内存内核代码的额外 C 编译标志。

`CFLAGS_MODULE` 是 `$(CC)` 编译模块专用选项。它包含了用于编译可动态加载的内核模块的 C 编译选项。

(2) 添加 `archprepare` 规则的依赖条件

`archprepare` 规则用来列出编译依赖的前提条件，在开始进入各级子目录编译之前，先生成依赖的文件。例如：

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

这个例子中，在进入子目录编译之前，要先处理 `maketools` 文件。许多头文件的生成也使用 `archprepare` 规则。

(3) 列出要遍历的子目录

`arch` Makefile 配合顶层目录的 Makefile 定义如何编译 `vmlinux` 的变量。对于模块没有对应体系结构的定义，所以模块编译方法是和体系结构无关的。

编译列表包括：`head-y`，`init-y`，`core-y`，`libs-y`，`drivers-y`，`net-y`。

`$(head-y)` 列出链接到 `vmlinux` 的起始位置的目标文件。

`$(libs-y)` 列出 `lib.a` 的库文件所在的目录。

剩余列出的目录都是 `built-in.o` 文件所在的目录。

链接过程中，\$(init-y)列出的目标文件紧跟在\$(head-y)后面。然后是\$(core-y)、\$(libs-y)、\$(drivers-y)和\$(net-y)。

顶层目录 Makefile 的定义包含了所有普通目录，arch/\$(ARCH)/Makefile 只添加体系结构相关的目录。

例如：

```
#arch/arm/Makefile
core-y          += arch/arm/kernel/ arch/arm/mm/ arch/arm/common/
core-y          += $(MACHINE)
libs-y          += arch/arm/lib/
drivers-$(CONFIG_OPROFILE) += arch/arm/oprofile/
```

(4) 体系结构相关的映像

arch Makefile 还定义 vmlinux 文件编译的规则，并且压缩打包到自引导代码中，在相应的目录下生成 zImage。这里包括各种不同的安装命令。不同的体系结构没有标准的规则。

通常是在 arch/\$(ARCH)/boot 目录下做一些特殊处理。

Kbuild 不负责支持 arch/\$(ARCH)/boot 目录的编译。因此，arch/\$(ARCH)/Makefile 文件应该自己定义编译目标。

推荐的方法是包含 arch/\$(ARCH)/Makefile 中包含快捷方式，使用全路径调用子目录下的 Makefile。

例如：

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

“\$(Q)\$(MAKE) \$(build)=<dir>”是在一个子目录<dir>中调用 make 的推荐方法。

在这里，不同体系结构的相关目标定义没有一致的规则，可以通过“make help”命令列出相关的帮助。因此，还要定义帮助信息。

例如：

```
#arch/arm/Makefile
define archhelp
    echo '* zImage          - Compressed kernel image (arch/$(ARCH)/boot/zImage)'
    .....
endef
```

当不带参数执行 make 的时候，首先会编译遇到的第一个目标。顶层目录 Makefile 中的第一个目标是 all。不同体系结构应该定义缺省的引导映像，在“make help”中加注*号，而且加到目标 all 的前提条件中。

例如：

```
#arch/arm/Makefile
# Default target when executing plain make
ifeq ($(CONFIG_XIP_KERNEL),y)
all: xipImage
else
all: zImage
endif
```

当配置好了内核以后，执行 `make`。如果没有把内核配置成 XIP 方式，就调用 `zImage` 的规则。

(5) 编译非 `kbuild` 目标

除了使用 `obj-*` 列表指定编译的目标文件以外，还可以使用 `extra-y` 列表指定当前目录下要创建的附加目标。

对于以下两种情况需要 `extra-y` 列表。

- 使 `kbuild` 在命令行中检查文件修改变化。比如使用 `$(call if_changed,xxx)` 语句。
- 告诉 `kbuild` 要编译或者删除哪些文件。

例如：

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

在这个例子中，`extra-y` 用来列出应该编译的目标文件，但是不应该连接到 `built-in.o` 中。

(6) 编译自引导映像有用的命令

`kbuild` 提供了一些编译引导映像时很有用的宏。

- `if_changed`

`if_changed` 是下列命令的基本构成部分。

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

编译这个规则的时候，首先检查是否有文件需要更新或者命令行有没有改变。如果任何编译选项改变，会强制重新编译。任何使用 `if_changed` 的目标必须列在 `$(targets)` 中，否则命令行检查会出错，并且目标总是会编译。



注意

一个常见的错误是忘记 `FORCE` 前提条件。

另外一个问题是有无空格很重要。例如：下列语句的逗号后面有一个空格，这个空格会导致语法错误。

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

- ld

具有链接目标的功能。通常使用 LDFLAGS_@\$ 设置 ld 的选项。

- objcopy

复制转换二进制程序。使用在 arch/\$(ARCH)/Makefile 中的 OBJCOPYFLAGS 编译选项。

OBJCOPYFLAGS_@\$ 可以用来添加附加的编译选项。

- gzip

压缩目标。使用最大压缩方式。

例如：

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

(7) 定制 kbuild 命令

当 kbuild 带 KBUILD_VERBOSE=0 选项执行的时候，通常只会显示一个命令的简写。要在自定义的 kbuild 命令中使能这种功能，必须设置以下两个变量。

- quiet_cmd_<command> 代表要显示的命令

- cmd_<command> 代表要执行的命令

例如：

```
#arch/arm/boot/Makefile
quiet_cmd_uimage = UIMAGE $@
    cmd_uimage = $(CONFIG_SHELL) $(MKIMAGE) -A arm -O linux -T kernel \
        -C none -a $(ZRELADDR) -e $(ZRELADDR) \
        -n 'Linux-$(KERNELRELEASE)' -d $< $@
$(obj)/uImage: $(obj)/zImage FORCE
    $(call if_changed,uimage)
    @echo ' Image $@ is ready'
```

当带 “KBUILD_VERBOSE=0” 更新编译的时候，只显示下列一行，而不会把编译信息都显示出来。

```
UIMAGE arch/arm/boot/uImage
```

(8) 预处理链接脚本

当编译 vmlinux 映像的时候，将用到链接脚本 arch/\$(ARCH)/kernel/vmlinux.lds。这个脚本的预处理变体文件是相同目录下的 vmlinux.lds.S。

Kbuild 知道.lds 文件并且包含*.lds.S 到*.lds 的转换规则。

例如：

```
#arch/i386/kernel/Makefile
always := vmlinux.lds
```

\$(always)列表告诉 kbuild 编译目标 vmlinux.lds。


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

\$(CPPFLAGS_vmlinux.lds)列表告诉 kbuild 在编译 vmlinux.lds 的时候使用指定的选项。当编译*.lds 目标文件的时候，kbuild 使用以下变量。

- CPPFLAGS: 在顶层目录 Makefile 中定义。
- EXTRA_CPPFLAGS: 可以在 kbuild Makefile 中定义。
- CPPFLAGS_\$(@F): 目标板特定选项。

Kbuild 的*.lds 文件结构在多种体系结构的文件中使用。

(9) \$(CC)支持的函数

内核编译可能会使用不同版本的\$(CC)，不同版本支持独立的一套选项和特点。Kbuild 提供检查\$(CC)有效选项的基本支持。\$(CC)一般就是 gcc 编译器，但是可能会有其他替代。

- cc-option

cc-option 选项用于检查\$(CC)是否支持一个给定的选项或者第二个可选项。

例如：

```
#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-march=pentium-mmx，那么 cflags-y 会被赋给这个选项。否则，使用-march-i586 选项。如果没有后一个选项，在前一个选项不支持的情况下，cflags-y 不会赋什么值。

- cc-option-yn

cc-option-yn 用于检查 gcc 是否支持给定的选项。如果支持，返回“y”，否则返回“n”。

例如：

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

在上面的例子中，如果\$(CC)支持-m32 选项，\$(biarch)就设成“y”。当\$(biarch)等于“y”时，扩展变量\$(aflags-y)和\$(cflags-y)会赋值-a32 和-m32。

- cc-option-align

gcc 版本大于等于 3.0 时，使用选项的移位类型指定函数的对齐。用作选项前缀的\$(cc-option-align)会选择合适的前缀。

cc-option-align 的伪语言描述如下。

```
if gcc < 3.00
    cc-option-align = -malign
else if gcc >= 3.00
    cc-option-align = -falign
```

例如：

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

上面的例子，对于 `gcc>=3.00`，选项为 `-falign-functions=4`；对于 `gcc<3.00`，选项为 `-malign-functions=4`。

- `cc-version`

`cc-version` 返回\$(CC)编译器的版本号数字。格式为代表主从版本号的 2 个十进制数。例如：`gcc 3.41` 应该返回 `0341`。

当\$(CC)版本在特定区域会导致错误的时候，`cc-version` 是很有用的。例如：`-mregparm=3` 选项的支持在一些 `gcc` 版本中不完整。

例如：

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

上面的例子中，`-mregparm=3` 只能用于版本大于等于 3.0 的 `gcc`。

7.2.4 内核编译

1. 编译命令

`Makefile` 还提供了配置编译的选项或者规则。执行 `make help`，可以打印出详细的帮助信息。解释一下帮助信息列出的各种选项的含义，分别在每一行信息下面加以注释。

```
$make help
```

打印出下列帮助信息。

(1) 用于清理生成文件的目标 (Cleaning targets)

```
clean          - remove most generated files but keep the config
```

`clean` 目标可以清除大多数生成的文件，但是保留 `.config`。

```
mrproper       - remove all generated files + config + various backup files
```

`mrproper` 可以清除所有生成的文件，包括 `.config` 和各种备份文件。

(2) 内核配置的目标 (Configuration targets)

```
config         - Update current config utilising a line-oriented program
```

`config` 是命令行的内核配置方式。

```
menuconfig     - Update current config utilising a menu based program
```

menuconfig 是光标菜单内核配置方式。

```
xconfig          - Update current config utilising a QT based front-end
```

xconfig 是基于 QT 图形界面的内核配置方式。

```
gconfig          - Update current config utilising a GTK based front-end
```

gconfig 是基于 GTK 图形界面的内核配置方式

```
oldconfig        - Update current config utilising a provided .config as base
```

oldconfig 基于已有的.config 文件进行内核配置。

```
randconfig       - New config with random answer to all options
```

randconfig 是对所有的选项按照随机回答（Y/M/N）的方式生成新配置。

```
defconfig        - New config with default answer to all options
```

defconfig 是对所有的选项都按照缺省回答生成新配置。

```
allmodconfig     - New config selecting modules when possible
```

allmodconfig 是对所有选项尽可能配置模块的新配置。

```
allyesconfig     - New config where all options are accepted with yes
```

allyesconfig 是对所有选项都配置成“Yes”的最大配置。

```
allnoconfig      - New minimal config
```

allnoconfig 是对所有选项都配置成“No”的最小配置。

(3) 其他通用目标（Other generic targets）

```
all              - Build all targets marked with [*]
```

all 是编译所有标记星号的目标，也就是编译所有缺省目标。

```
* vmlinux        - Build the bare kernel
```

vmlinux 是编译最基本的内核映像，就是顶层的 vmlinux。

```
* modules        - Build all modules
```

modules 是编译所有的模块。

```
modules_install  - Install all modules
```

modules_install 是安装所有的模块。

```
dir/ - Build all files in dir and below
```

dir 是编译 dir 目录及其子目录的所有文件，当然 dir 代表具体的一个目录名。

```
dir/file.[ois] - Build specified target only
```

dir/file.[ois]是仅编译 dir 目录下指定的目标。

```
dir/file.ko - Build module including final link
```

dir/file.ko 是编译并且链接指定目录的模块。

```
rpm - Build a kernel as an RPM package
```

rpm 是以 RPM 包方式编译内核。

```
tags/TAGS - Generate tags file for editors
```

tags/TAGS 是为编辑器生成 tag 文件，方便编辑器识别关键词。

```
cscope - Generate cscope index
```

cscope 是生成 cscope 索引，方便代码浏览。

```
kernelrelease - Output the release version string
```

kernelrelease 是输出内核版本的字符串。

(4) 静态解析器 (Static analysers)

```
buildcheck - List dangling references to vmlinux discarded sections  
and init sections from non-init sections
```

buildcheck 是列出对 vmlinux 废弃段的虚引用和从非 init 段引用 init 段的虚引用。

```
checkstack - Generate a list of stack hogs
```

checkstack 是生成栈空间耗费者的列表。

```
namespacecheck - Name space analysis on compiled kernel
```

namespace 是对编译好的内核做命名域分析。

(5) 内核打包 (Kernel packaging)

```
rpm-pkg - Build the kernel as an RPM package
```

rpm-pkg 是以一个 RPM 包的方式编译内核。

```
binrpm-pkg - Build an rpm package containing the compiled kernel  
and modules
```

binrpm-pkg 是编译一个包含已经编译好的内核和模块的 rpm 包。

```
deb-pkg      - Build the kernel as an deb package
```

deb-pkg 是以一个 deb 包的方式编译内核。

```
tar-pkg      - Build the kernel as an uncompressed tarball
```

tar-pkg 是以一个不压缩的 tar 包方式编译内核。

```
targz-pkg    - Build the kernel as a gzip compressed tarball
```

targz-pkg 是以一个 gzip 压缩包的方式编译内核。

```
tarbz2-pkg   - Build the kernel as a bzip2 compressed tarball
```

tarbz2-pkg 是以一个 bzip2 压缩包的方式编译内核。

(6) 文档目标 (Documentation targets)

```
Linux kernel internal documentation in different formats:
xmldocs (XML DocBook), psdocs (Postscript), pdfdocs (PDF)
htmldocs (HTML), mandocs (man pages, use installmandocs to install)
```

Linux 内核内部支持各种形式的文档。

(7) 体系结构相关的目标 (ARM) (Architecture specific targets (arm))

```
* zImage     - Compressed kernel image (arch/arm/boot/zImage)
```

zImage 是编译生成压缩的内核映像 (arch/arm/boot/zImage)。

```
Image        - Uncompressed kernel image (arch/arm/boot/Image)
```

Image 是编译生成非压缩的内核映像 (arch/arm/boot/Image)。

```
* xipImage    - XIP kernel image, if configured (arch/arm/boot/xipImage)
```

xipImage 是编译生成 XIP 的内核映像 (arch/arm/boot/xipImage)，前提是内核配置成 XIP。

```
bootpImage    - Combined zImage and initial RAM disk
                (supply initrd image via make variable INITRD=<path>)
```

bootpImage 是编译包含 zImage 和 initrd 的映像 (可以通过 make 变量 INITRD=<path> 提供 initrd 映像)。

```
install       - Install uncompressed kernel
```

install 是安装非压缩的内核。

```
zinstall      - Install compressed kernel
```

```
Install using (your) ~/bin/installkernel or
(distribution) /sbin/installkernel or
install to $(INSTALL_PATH) and run lilo
```

zinstall 是安装压缩的内核。通过发行版的/bin/installkernel 工具安装，或者安装到 \$(INSTALL_PATH) 路径下，然后再执行 lilo。这些目标对于 X86 平台适用。

还有各种开发板的缺省内核配置文件，这些配置文件都保存在 arch/arm/configs 目录下。

```
assabet_defconfig      - Build for assabet
.....
smdk2410_defconfig     - Build for smdk2410
spitz_defconfig        - Build for spitz
versatile_defconfig    - Build for versatile
```

每种支持的目标板都会保存一个缺省的内核配置文件。

```
make V=0|1 [targets] 0 => quiet build (default), 1 => verbose build
```

V=0 表示不显示编译信息（缺省），V=1 表示显示编译信息。

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

O=dir 用来指定所有输出文件的目录，包括.config 文件，都将放到 dir 目录下。

```
make C=1 [targets] Check all c source with $CHECK (sparse)
```

C=1 表示检查所有\$CHECK 的 C 程序。

```
make C=2 [targets] Force check of all c source with $CHECK (sparse)
```

C=2 表示强制检查所有\$CHECK 的 C 程序。

```
Execute "make" or "make all" to build all targets marked with [*]
```

执行 make 或者 make all，将自动编译所有带星号标志的目标。

```
For further info see the ./README file
```

更多信息参看 README 文件。

其中，vmlinux modules zImage 和 xipImage 是 Makefile 缺省的目标。执行 make，缺省地就可以执行这些编译规则。但是，zImage 和 xipImage 是互斥的，因为两种内核映像格式不可能同时配置。

2. 编译链接内核映像

一般情况下，先编译链接生成顶层目录的 vmlinux，再把 vmlinux 精简压缩成 piggy.gz，然后加上自引导程序链接成 arch/\$(ARCH)/boot/zImage，这样就得到一个具备自启动能力的

Linux 内核映像。

除了 zImage 之外，还有其他一些映像格式，分别适用于不同的体系结构和引导程序。

由于 zImage 是最通用的，所以我们只分析一下 zImage 编译链接的过程。

(1) 编译链接 vmlinux

vmlinux 的规则是在顶层的 Makefile 中定义的。vmlinux 是由\$(vmlinux-init)和\$(vmlinux-main)列表中指定的目标文件链接而成的，大多数是来自顶层子目录下的 built-in.o 文件，其他都在 arch/\$(ARCH)Makefile 中指定。这些目标文件的链接顺序非常重要，\$(vmlinux-init)必须排在第一位。参考 Makefile 注释中的结构图。

vmlinux 的版本 (uname-v 可以显示) 不是在各级目录的编译阶段更新的，因为还不知道是否需要更新 vmlinux。除了在添加内核符号之前生成 kallsyms 信息的情况，直到链接 vmlinux 才更新 vmlinux 版本信息。还生成 System.map 文件，用来描述所有符号（全局变量、函数等）的地址。

```
#Makefile
#
# vmlinux
# ^
# |
# +--< $(vmlinux-init)
# |   +--< init/version.o + more
# |
# +--< $(vmlinux-main)
# |   +--< driver/built-in.o mm/built-in.o + more
# |
# +--< kallsyms.o (see description in CONFIG_KALLSYMS section)
#
vmlinux-init := $(head-y) $(init-y)
vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
vmlinux-all := $(vmlinux-init) $(vmlinux-main)
vmlinux-lds  := arch/$(ARCH)/kernel/vmlinux.lds

# Rule to link vmlinux - also used during CONFIG_KALLSYMS
# May be overridden by arch/$(ARCH)/Makefile
quiet_cmd_vmlinux__ := LD      $@
      cmd_vmlinux__ := $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux) -o $@ \
      -T $(vmlinux-lds) $(vmlinux-init) \
      --start-group $(vmlinux-main) --end-group \
      $(filter-out $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) FORCE,$^)
```

这里通过定制 Kbuild 命令来定义 vmlinux 的规则。cmd_vmlinux__ 命令就是具体链接生

成 vmlinux 的 Kbuild 命令。命令各部分的具体含义如下。

\$(LD)是链接工具，对于 ARM 平台，就是 arm-linux-ld;

\$(LDFLAGS)和\$(LDFLAGS_vmlinux)是链接选项列表;

“-o \$@" 选项指定了生成的文件，\$@在编译 vmlinux 目标的时候，代表 vmlinux 文件;

“-T \$(vmlinux-lds)” 选项指定链接脚本，arch/\$(ARCH)/kernel/vmlinux.lds 就是这个脚本，vmlinux-lds 是在 Makefile 中定义的。

“\$(vmlinux-init)” 是初始化部分目标文件列表，放在开头。

“--start-group \$(vmlinux-main) --end-group” 是内核主要的目标文件，链接的时候要检查函数等符号是否确定。

剩余的代码或者数据链接在最后。

(2) 生成 vmlinux.lds 链接脚本

vmlinux 的链接脚本是 arch/\$(ARCH)/kernel/vmlinux.lds。Kbuild 可以根据模板 vmlinux.lds.S 转换生成。在 scripts/Makefile.build 中定义了一条把.lds.S 转换成.lds 的规则。

```
#scripts/Makefile.build
# Linker scripts preprocessor (.lds.S -> .lds)
# -----
quiet_cmd_cpp_lds_S = LDS      $@
      cmd_cpp_lds_S = $(CPP) $(cpp_flags) -D__ASSEMBLY__ -o $@ $<
%.lds: %.lds.S FORCE
      $(call if_changed_dep, cpp_lds_S)
```

摘取 arch/arm/kernel/vmlinux.lds.S 的部分内容，解释一下。

```
/* arch/arm/kernel/vmlinux.lds.S */
/* 由于使用了一些宏，所以需要包含这几个宏定义的头文件 */
#include <asm-generic/vmlinux.lds.h>
#include <linux/config.h>
#include <asm/thread_info.h>

OUTPUT_ARCH(arm)          /* 指定目标板体系结构 */
ENTRY(stext)              /* 代码段入口 */
SECTIONS                  /* 代码段各部分 */
{
    . = TEXTADDR;         /* 代码段起始地址，大多数 Linux 内核是 0xC0008000 */
    .init : {              /* 内核初始化的代码和数据 */
        _stext = .;
        _sinittext = .;
        *(.init.text)
        _einittext = .;
    }
```

```
    __proc_info_begin = .;
    *(.proc.info.init)
    __proc_info_end = .;
    __arch_info_begin = .;
    *(.arch.info.init)
    __arch_info_end = .;
    .....
}

/DISCARD/ : {
    *(.exit.text)
    *(.exit.data)
    *(.exitcall.exit)
}

.text : {
    /* 真正的代码段部分 */
    __text = .;
    /* 代码和只读数据 */
    *(.text)
    SCHED_TEXT
    LOCK_TEXT
    *(.fixup)
    *(.gnu.warning)
    *(.rodata)
    *(.rodata.*)
    *(.glue_7)
    *(.glue_7t)
    *(.got)
    /* Global offset table */
}
RODATA
__etext = .;
/* 代码段和只读数据结束 */
.....

.data : AT(__data_loc) { /* 数据段起始 */
    __data_start = .; /* 内存中的地址 */
    .....
    /* 例外修正表（可能需要在运行时修正） */
    . = ALIGN(32);
    __start__ex_table = .;
    *(__ex_table)
```

```

__stop__ex_table = .;

/* 普通的数据段 */
*(.data)
CONSTRUCTORS
_edata = .;      /* 数据段结束 */
}

.bss : { /* 未初始化的全局变量 */
    __bss_start = .; /* BSS */
    *(.bss)
    *(COMMON)
    _end = .;
}

/* 调试信息和数据段 */
.stab 0 : { *(.stab) }
.stabstr 0 : { *(.stabstr) }
.stab.excl 0 : { *(.stab.excl) }
.stab.exclstr 0 : { *(.stab.exclstr) }
.stab.index 0 : { *(.stab.index) }
.stab.indexstr 0 : { *(.stab.indexstr) }
.comment 0 : { *(.comment) }
}

```

vmlinux 程序的链接组装，就是完全按照上面脚本的顺序。这样，内核代码和数据才能加载到相应的位置运行。

(3) 链接生成 zImage

zImage 的规则是在 arch/\$(ARCH)Makefile 中定义的，它总是与目标板体系结构有关。

```

#arch/arm/Makefile
zImage Image xipImage bootpImage uImage: vmlinux
    $(Q)$(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $(boot)/$@

```

zImage 的前提条件是 vmlinux，也就是说，只有顶层的 vmlinux 编译通过，才能生成 zImage。

编译命令是到 arch/\$(ARCH)/boot 目录下，调用 Makefile 的 zImage 规则，同时传递变量 MACHINE。其中\$@就是 zImage，这个规则又在 arch/arm/boot/Makefile 中定义。

```

#arch/arm/boot/Makefile
$(obj)/zImage: $(obj)/compressed/vmlinux FORCE

```

```
$(call if_changed,objcopy)
@echo ' Kernel: $@ is ready'
```

这条规则的前提条件是\$(obj)/compressed/vmlinux，那么这又要编译子目录 compressed。事实上，对于不同的体系结构，这部分代码有很大差异。对于 ARM 平台来说，\$(obj)/compressed/vmlinux 就是压缩的自引导映像，只不过没有精简。

在 arch/arm/boot/compressed/Makefile 文件中，定义了编译链接\$(obj)/compressed/vmlinux 的规则。

```
#arch/arm/boot/compressed/Makefile
$(obj)/vmlinux: $(obj)/vmlinux.lds $(obj)/$(HEAD) $(obj)/piggy.o \
    $(addprefix $(obj)/, $(OBJS)) FORCE
$(call if_changed,ld)
@:
```

前提条件中，\$(obj)/vmlinux.lds 是链接脚本，\$(obj)/\$(HEAD)是自引导的目标代码，\$(obj)/piggy.o 是顶层 vmlinux 的精简压缩代码。为了保证自引导代码组装在映像起始位置，还要使用链接脚本。

3. 编译内核模块

Linux 2.6 内核的模块采用新的加载器，它是由 Rusty Russel 开发的。它使用内核编译机制，生成一个*.ko（内核目标文件，kernel object）模块目标文件，而不是一个*.o 模块目标文件。

内核编译系统首先编译这些模块，然后链接上 vermagic.o。这样就在目标模块创建了一个特殊区域，用来记录编译器版本号、内核版本号、是否使用内核抢占等信息。

新的内核编译系统如何来编译并加载一个简单的模块的呢？举例一个简单的例子说明。我们写一个最简单的“hello”模块，只要实现模块初始化函数和退出函数就够了。这个模块程序叫作 hello.c。

```
#drivers/char/hello/hello.c
void init_module (void)
{
    printk( "Hello module!\n");
}
void cleanup_module (void);
{
    printk( "Bye module!\n");
}
```

相应的 Makefile 文件如下。

```
KERNEL_SRC = ~/linux-2.6.14
```

```
SUBDIR = $(KERNEL_SRC)/drivers/char/hello/
all: modules
obj-m := hello_mod.o
hello-objs := hello.o
EXTRA_FLAGS += -DDEBUG=1
modules:
$(MAKE) -C $(KERNEL_SRC) SUBDIR=$(SUBDIR) modules
```

makefile 文件使用内核编译机制来编译模块。编译好的模块将被命名为 `hello_mod.ko`，它是编译 `hello.c` 并且链接 `vermagic.o` 而得到的。`KERNEL_SRC` 指定内核源文件所在的目录，`SUBDIR` 指定放置模块的目录。`EXTRA_FLAGS` 指定了需要给出的编译标记。

新模块要用新的模块工具加载或卸载。原来 2.4 内核的工具不能再用来加载或卸载 2.6 内核模块。2.4 的内核模块可能会发生使用和卸载冲突的情况，这是由于模块使用计数是由模块代码自己来控制的。新的模块加载工具可以尽量避免这种情况发生。

Linux 2.6 内核模块不再需要对引用计数进行加或减操作，这些工作已经由模块代码外部处理。任何要使用模块的代码都必须调用 `try_module_get(&module)`，只有在调用成功以后才能访问那个模块。如果被调用的模块已经被卸载，那么这次调用会失败。访问完成时，要通过 `module_put()` 函数释放模块。

7.2.5 内核编译结果

相对于 Linux 2.4 内核，Linux 2.6 内核配置编译过程要简单一些，不再需要 `make dep`；`make zImage`；`make modules` 的命令。配置好内核之后，只要执行 `make` 就可以编译内核映像和模块。

内核的配置菜单选项内容也有了较大变化，我们在下一节中再详细讨论。

内核编译完成以后，将生成几个重要的文件。它们是 `vmlinux`、`vmlinuz` 和 `System.map`。

(1) vmlinux

`vmlinux` 是在内核源码顶层目录生成的内核映像。它是内核在虚拟空间运行时代码的真实反映。编译的过程就是按照特定顺序链接目标代码，生成 `vmlinux`。因为 Linux 内核运行在虚拟地址空间，所以名字附加“vm”（Virtual Memory）。`Vmlinux` 不具备引导的能力，需要借助其他 `bootloader` 引导启动。

(2) vmlinuz

`vmlinuz` 是可引导的、压缩的内核映像，也就是 `zImage`。它是 `vmlinux` 的压缩映像，是可执行的 Linux 内核映像。`vmlinuz` 的生成跟体系结构很有关系，不同体系结构的内核一般有不同的格式。大多数 `vmlinuz` 包含 2 部分：压缩的 `vmlinux` 和自引导程序。`Vmlinuz` 通过自引导程序初始化系统，并且解压启动 `vmlinux`。`Vmlinuz` 采用 `gzip` 压缩格式，都包含 `gzip` 的解压缩函数。

(3) System.map

`System.map` 是一个特定内核的内核符号表，它包含内核全局变量和函数的地址信息。

System.map 是内核编译生成文件之一。当 vmlinux 编译完成时，再通过\$(NM)命令解析 vmlinux 映像生成。可以直接通过 nm 命令来查看任何一个可执行文件的信息。

```
$ nm vmlinux > System.map
```

不过，内核源码还要对 nm 生成的信息加以过滤排序，才能得到 System.map。

```
$ nm vmlinux | grep -v '\(compiled\)\|\(\.o$\)\|\([aUw] \)\|\(\.\.ng$\)\|\(LASH[RL]DI\)' | sort > System.map
```

Linux 内核是一个很复杂的代码块，有许许多多的全局符号。它不使用符号名，而是通过变量或函数的地址来识别变量或函数名。比如：不使用 size_t BytesRead 这样的符号，而使用地址 c0343f20 引用这个变量。

内核主要是用 C 写的，编译成目标代码或者映像就可以直接使用地址了。如果我们需要知道符号的地址，或者需要知道地址对应的符号，就需要由符号表来完成。符号表是所有符号连同它们的地址的列表。

System.map 是在内核编译过程中生成的，每一个内核映像对应自己的 System.map。它是保存在文件系统上的文件。当编译一个新内核时，各个符号名的地址要发生变化，就应当用新的 System.map 来取代旧的 System.map。它可以提供给 klogd、lsof 和 ps 等程序使用。

Linux 内核还有另外一种符号表使用方式：/proc/ksyms。它是一个“proc”接口，是在内核映像引导时创建的/proc/ksyms 条目。用户空间的程序可以通过/proc/ksyms 接口可以读取内核符号表。这需要预先配置 CONFIG_ALLKSYMS 选项，内核映像将包含符号表。

7.3 内核配置选项

基于内核配置系统，可以对内核的上千个选项进行配置。那么，这些选项应该如何使用呢？下面以 ARM 平台为例，介绍常用的内核配置选项。

7.3.1 使用配置菜单

内核配置过程比较繁琐，但是配置是否适当与 Linux 系统运行直接相关，所以需要了解一下主要选项的设置。

配置内核可以选择不同的配置界面，图形界面或者光标界面。由于光标菜单运行时不依赖于 X11 图形软件环境，可以运行在字符终端上，所以光标菜单界面比较通用。图 7.2 所示就是执行 make menuconfig 出现的配置菜单。

在各级子菜单项种，选择相应的配置时，有 3 种选择，它们代表的含义分别如下。

Y—将该功能编译进内核。

N—不将该功能编译进内核。

M—将该功能编译成可以在需要时动态插入到内核中的模块。

如果使用的是 make xconfig，使用鼠标就可以选择对应的选项。如果使用的是 make menuconfig，则需要使用回车键进行选取。

在每一个选项前都有个括号，有的是中括号，有的是尖括号，还有的是圆括号。用空格键选择时可以发现，中括号里要么是空，要么是“*”，而尖括号里可以是空，“*”和“M”。这表示前者对应的项要么不要，要么编译到内核里；后者则多一样选择，可以编译成模块。而圆括号的内容是要你在所提供的几个选项中选择一项。

在编译内核的过程中，最麻烦的事情就是这步配置工作了。初次接触 Linux 内核的开发者往往弄不清楚该如何选取这些选项。实际上在配置时，大部分选项可以使用其缺省值，只有小部分需要根据用户不同的需要选择。选择的原则是将与内核其他部分关系较远且不经常使用的部分功能代码编译成为可加载模块，有利于减小内核的长度，减小内核消耗的内存，简化该功能相应的环境改变时对内核的影响；不需要的功能就不要选；与内核关系紧密而且经常使用的部分功能代码直接编译到内核中。

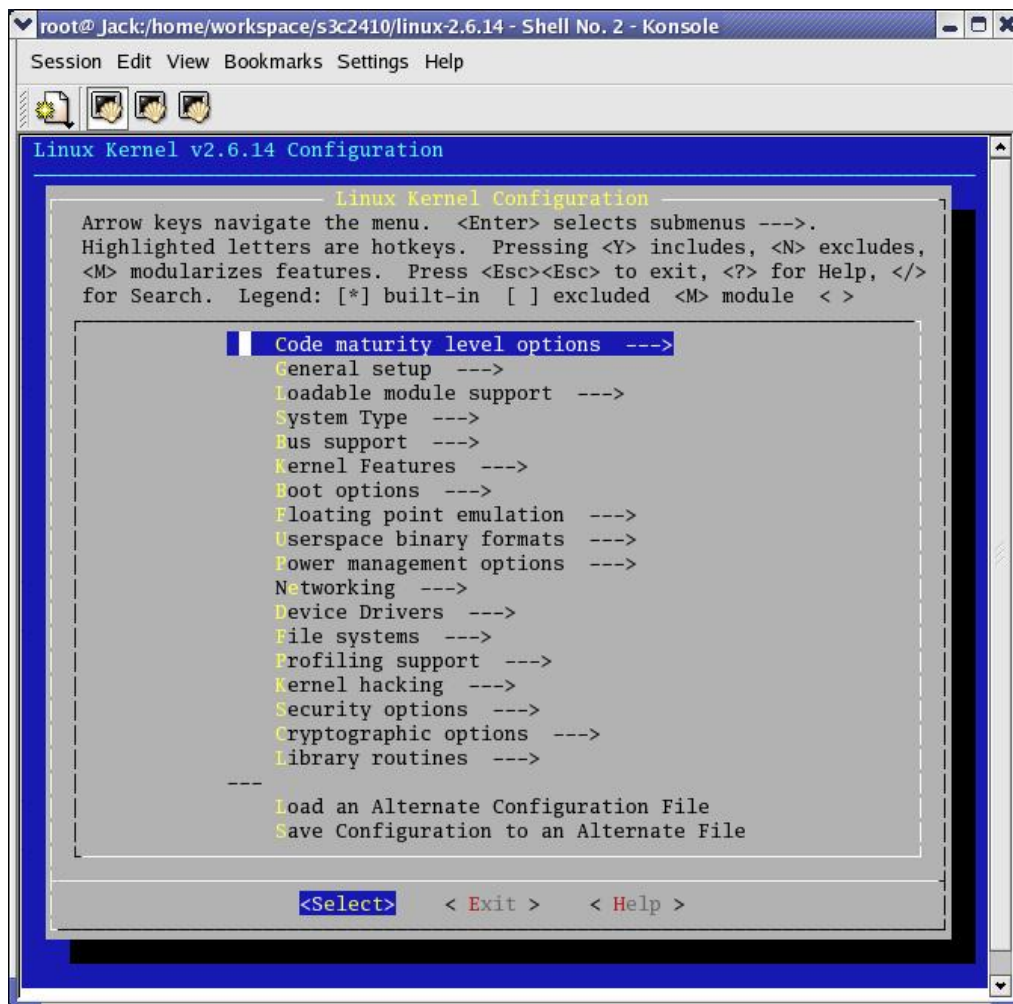


图 7.2 内核配置主菜单

7.3.2 基本配置选项

相对于 Linux 2.4 内核，2.6 内核的配置菜单有了很大变化，而且随着版本的发展还有些调整。下面以 Linux-2.6.14 内核版本为例，介绍主菜单选项和常用的配置选项的功能。

(1) “Code maturity level options” 菜单包含配置控制代码成熟度的一些选项

CONFIG_EXPERIMENTAL 选项可以包含一些处于开发状态或者不成熟的代码或者驱动程序。

(2) “General setup” 菜单包含通用的一些配置选项

CONFIG_LOCALVERSION 可以定义附加的内核版本号。

CONFIG_SWAP 可以支持内存页交换（swap）的功能。

CONFIG_EMBEDDED 支持嵌入式 Linux 标准内核配置。

CONFIG_KALLSYMS 支持加载调试信息或者符号解析功能。

(3) “Loadable module support” 菜单包含支持动态加载模块的一些配置选项

CONFIG_MODULES 是支持动态加载模块功能选项。

CONFIG_MODVERSIONS 是模块版本控制支持选项。

CONFIG_KMOD 选项可以支持内核自动加载模块功能。

(4) “System Type” 菜单包含系统平台列表及其相关的配置选项

对于不同的体系结构，显示不同的提示信息。ARM 体系结构显示 “ARM system type”。

CONFIG_ARCH_CLPS7500 是 Cirrus Logic PS7500FE 开发板的配置选项。

还有其他很多处理器和板子的配置选项，不一一说明。

(5) “Bus support” 菜单包含系统各种总线的配置选项

CONFIG_PCI 是 PCI 总线支持选项。

(6) “Kernel Features” 菜单包含内核特性相关选项

CONFIG_PREEMPT 选项支持内核抢占特性。

CONFIG_SMP 选项支持对称多处理器的平台。

(7) “Boot options” 菜单包含内核启动相关的选项

CONFIG_CMDLINE 选项可以定义缺省的内核命令行参数。

CONFIG_XIP_KERNEL 选项可以支持内核从 ROM 中运行的功能。

(8) “Floating point emulation” 菜单包含浮点数运算仿真功能

CONFIG_FPE_NWFPE 选项支持 “NWFPE” 数学运算仿真。

CONFIG_FPE_FASTFPE 选项支持 “FastFPE” 数学运算仿真。

(9) “Userspace binary formats” 菜单包含支持的应用程序格式

CONFIG_BINFMT_ELF 选项支持 ELF 格式可执行程序，这是 Linux 程序缺省的格式。

CONFIG_BINFMT_AOUT 选项支持 AOUT 格式可执行程序，现在已经少用。

(10) “Power management options” 菜单包含电源管理有关的选项

CONFIG_PM 支持电源管理功能。

CONFIG_APM 支持高级电源管理仿真功能。

(11) “Networking” 菜单包含网络协议支持选项

CONFIG_NET 选项支持网络功能。

CONFIG_PACKET 支持 socket 接口的功能。

CONFIG_INET 选项支持 TCP/IP 网络协议。

CONFIG_IPV6 选项支持 IPv6 协议的支持。

(12) “Device Drivers” 菜单包含各种设备驱动程序

这个菜单下面包含很多子菜单，几乎包含了所有的设备驱动程序。我们将在第 7.3.3 节具体描述。

(13) “File systems” 菜单包含各种文件系统的支持选项

CONFIG_EXT2_FS 选项支持 EXT2 文件系统。

CONFIG_EXT3_FS 选项支持 EXT3 文件系统。

CONFIG_JFS_FS 选项支持 JFS 文件系统。

CONFIG_INOTIFY 选项支持文件改变通知功能。

CONFIG_AUTOFS_FS 选项支持文件系统自动挂载功能。

“CD-ROM/DVD Filesystems” 子菜单包含 iso9660 等 CD ROM 文件系统类型选项。

“DOS/FAT/NT Filesystems” 子菜单包含 DOS/Windows 的一些文件系统类型选项。

“Pseudo filesystems” 子菜单包含 sysfs procfs 等驻留在内存中的伪文件系统选项。

“Miscellaneous filesystems” 子菜单包含 JFFS2 等其他类型的文件系统。

“Network File Systems” 子菜单包含 NFS 等网络相关的文件系统。

(14) “Profiling support” 菜单包含用于系统测试的工具选项

CONFIG_PROFILING 选项支持内核的代码测试功能。

CONFIG_OPROFILE 选项使能系统测试工具 Oprofile。

(15) “Kernel hacking” 菜单包含各种内核调试的选项

这些选项的功能将在第 9.1.1 节详细介绍。

(16) “Security options” 菜单包含安全性有关的选项

CONFIG_KEYS 选项支持密钥功能。

CONFIG_SECURITY 选项支持不同的密钥模型。

CONFIG_SECURITY_SELINUX 选项支持 NSA SELinux。

(17) “Cryptographic options” 菜单包含加密算法

CONFIG_CRYPTODEV 选项支持加密的 API。

还有各种加密算法的选项可以选择。

(18) “Library routines” 菜单包含几种压缩和校验库函数

CONFIG_CRC32 选项支持 CRC32 校验函数。

CONFIG_ZLIB_INFLATE 选项支持 zlib 压缩函数。

CONFIG_ZLIB_DEFLATE 选项支持 zlib 解压缩函数。

7.3.3 驱动程序配置选项

几乎所有 Linux 的设备驱动程序都在 “Device Drivers” 菜单下，它对设备驱动程序加以归类，放到子菜单下。下面解释常用的一些菜单项的内容。

(1) “Generic Driver Options” 菜单对应 drivers/base 目录的配置选项，包含 Linux 驱动程

序基本和通用的一些配置选项。

(2) “Memory Technology Devices (MTD)” 菜单对应 drivers/mtd 目录的配置选项，包含 MTD 设备驱动程序的配置选项。

(3) “Parallel port support” 菜单对应 drivers/parport 目录的配置选项，包含并口设备驱动程序。

(4) “Plug and Play support” 菜单对应 drivers/pnp 目录的配置选项，包含计算机外围设备的热拔插功能。

(5) “Block devices” 菜单对应 drivers/block 目录的配置选项，包含软驱、RAMDISK 等驱动程序。

(6) “ATA/ATAPI/MFM/RLL support” 菜单对应 drivers/ide 目录的配置选项，包含各类 ATA/ATAPI 接口设备驱动。

(7) “SCSI device support” 菜单对应 drivers/scsi 目录的配置选项，包含各类 SCSI 接口的设备驱动。

(8) “Network device support” 菜单对应 drivers/net 目录的配置选项，包含各类网络设备驱动程序。

(9) “Input device support” 菜单对应 drivers/input 目录的配置选项，包含 USB 键盘鼠标等输入设备通用接口驱动。

(10) “Character devices” 菜单对应 drivers/char 目录的配置选项，包含各种字符设备驱动程序。这个目录下的驱动程序很多。串口的配置选项也是从这个子菜单调用的，但是串口驱动所在的目录是 drivers/serial。

(11) “I²C support” 菜单对应 drivers/i2c 目录的配置选项，包含 I²C 总线的驱动。

(12) “Multimedia devices” 菜单对应 drivers/media 目录的配置选项，包含视频/音频接收和摄像头的驱动程序。

(13) “Graphics support” 菜单对应 drivers/video 目录的配置选项，包含 Framebuffer 驱动程序。

(14) “Sound” 菜单对应 sound 目录的配置选项，包含各种音频处理芯片 OSS 和 ALSA 驱动程序。

(15) “USB support” 菜单对应 drivers/usb 目录的配置选项，包含 USB Host 和 Device 的驱动程序。

(16) “MMC/SD Card support” 菜单对应 drivers/mmc 目录的配置选项，包含 MMC/SD 卡的驱动程序。

对于特定的目标板，可以根据外围设备选择对应的驱动程序选项，然后才能在 Linux 系统下使用相应的设备。

这里不准备讨论 Linux 设备驱动程序的话题。有关设备驱动程序的内容，可以阅读《Linux Device Drivers 3rd Edition》。