



内核方面的考虑

内核是所有 Linux 系统的中心软件组件。整个系统的能力完全受内核本身能力的限制。例如,倘若你使用的内核无法支持目标板上某个硬件组件,当在目标板上运行此内核时,该硬件组件将会变得毫无用处。

事实上,已经有许多书籍和在线文档在探讨内核的内部、程序设计、设置及其在用户系统上的使用,而且都很详细。因此本章不会涵盖这些问题。如果对这些问题有兴趣,不妨阅读《Running Linux》、《Linux Device Drivers》和《Understanding the Linux Kernel》等由 O'Reilly 出版的书。这些书分别涵盖了内核的设置与使用、内核的程序设计及内核的内部。你或许还会想看一下 LDP 的《Linux Kernel HOWTO》。

本章将专门讨论为了在嵌入式系统上使用 Linux 怎样准备内核。尤其是,我们将会探讨内核的选择、配置、编译及安装。每完成一个步骤我们就会越接近“让目标板系统获得可用内核和相关模块”这个目标。我们最后还会讨论嵌入式系统特有的内核操作的各个方面。

选择内核

尽管内核的主要供应网站只有 <http://www.kernel.org/>,但是从这个网站取得的内核版本不一定可以用在 Linux 支持的每个架构上。事实上,当你以这些版本为嵌入式 Linux 系统中最常见的架构建立内核时,有些架构甚至会失败,能正常执行的就更少了。主要是因为这些架构的 Linux 开发与主要的内核版本不同步。

要让目标板取得可用的内核,必须找到专门负责开发相应处理器架构的团队所提供的内核版本。因为每种架构都由不同的团队维护,所以你必须根据架构来选择供应内核的网站。表 5-1 列出了每种处理器架构最适合的内核供应网站以及下载方式。

表 5-1：每种处理器架构最适合的内核供应网站

处理器架构	最适合的内核供应网站	下载方式
x86	http://www.kernel.org/	ftp、http、sync
ARM	http://www.arm.linux.org.uk/developer/	ftp、rsync
PowerPC	http://penguinppc.org/	ftp、http、rsync、bitkeeper
MIPS	http://www.linux-mips.org/	cvs
SuperH	http://linuxsh.sourceforge.net/	cvs
M68k	http://www.linux-m68k.org/	ftp、http

正如所见，这些大部分是我在第三章提到每种架构时介绍的网站。有人说，要取得这些架构的内核，不一定要到这几个地方，别处也可以。有两种情况。首先，这些网站有的会有镜像网站，镜像网站提供的是相同的内容。其次，有许多人、公司和组织提供自己开发的内核版本。如果使用的是后者，千万要小心，这些内核可能不会得到社群的支持（注 1），当有问题发生时，就算有也只限于供应者提供的支持。

一旦找到了最适合的下载网站，将需要从该网站选出适用的内核版本。这是个困难的抉择，因为有些版本的某些功能有问题，但是在较旧的版本中这些功能又完全正常。找到这类信息的最佳办法就是，不断地跟维护架构内核的社群接触。这并不表示你要接触或写信给任何人，而是指订阅适当的邮件论坛并且不断地检查论坛上和该项移植的主要网站上的重要公告。

这些网站有的，例如 ARM 的网站，未必会发布完整的内核，而是发布正式内核的补丁。此时，如果要为架构取得可用内核，必须先到的主要供应网站下载内核，然后再用移植内核的网站提供的补丁对内核进行修补。

为了实现以 ARM 为基础的用户接口，我们会从 <http://www.kernel.org/> 下载一般的 2.4.18 内核，并从正式的 ARM Linux 网站 <http://www.arm.linux.org.uk/> 下载 2.4.18-rmk5 补丁。用 rmk5 修补过一般的 2.4.18 之后，我们便得到了 2.4.18-rmk5 内核，其中包含了以 ARM 为基础的系统所需要的功能。

注 1：社群之所以不支持，未必是因为没有源码可用（因为 Linux 的发行受到 GPL 授权条款的保护，所以不应该发生此事），最可能是因为供应者对内核机能所做的修改，只有供应者自己知道。也可能是供应者对内核所做的修改，被社群认为不够成熟或甚至不需要加入主内核树中。

内核版本编号的变化

其他内核供应网站发行的版本,通常会将内核编号系统加以变化,以区分其产品。ARM内核源码树的维护者Russell King,会为自己发行的内核加上-rmk的扩展名。以Russell的成果为基础的其他开发者,则会为他们所发行的内核加上自己的扩展名。另一个ARM Linux的开发者Nicolas Pitre会为自己的内核加上-np的扩展名,而handhelds.org的Familiar发行套件的开发者们,则会为他们的内核加上-hh的扩展名。因此,我在第一章所提到的2.4.20-rmk3-hh24这个版本,事实上是:Russell以Marcelo Tosatti的2.4.20版本为基础修改的内核,在第3次发行时,handhelds.org的开发者们以这个版本(-rmk3)为基础对内核进行了修改,并且发行了24次(-hh24)。

(尽管Linus Torvalds通常是Linux版本的维护者,但是Linus将2.4.x系列的维护责任交给了Marcelo,好让自己全力投入2.5.x系列的开发工作。)

通常情况下,选择已知可用的版本中越新的版本越好。因此,如果已知在目标板上可以使用2.4.17和2.4.18,则2.4.18应该是最优选的版本。然而,在某些情况下并非如此。例如,密切注意内核开发的人人都知道,应该避免使用版本2.4.10到2.4.15(含)的内核,因为它们是在许多修改正要被整合进内核的过程中产生的版本,因此偶尔会有不稳定的现象。再次提醒,要取得这类信息,必须持续地接触适当的邮件论坛及网站。

如果发现订阅移植版本的邮件论坛或主内核版本的邮件论坛太浪费时间,应该找时间至少一周探访一次移植版本的网站及阅读《Kernel Traffic》(<http://www.kerneltraffic.org/>)周报。《Kernel Traffic》摘录了上周主内核版本邮件论坛中最重要的讨论内容。

一旦为目标板找到了适当的内核版本之后,可以像第四章“内核头文件的设置”中那样,先将它下载到 $\{PRJROOT\}/kernel$ 目录并从中取出源码,如果有需要再为它更名。为内核目录更名,可以避免日后从所下载的另一个内核版本取出源码时,误将既有的内核目录覆盖掉。

不论你要选用哪个版本,务必为目标板多试几个不同的内核版本。除了可以参考网络上的建议和缺陷报告,多评估几个不同的版本将有助于你洞悉硬件与内核间的相互作用。

你或许还想尝试其他开发者提供的各种补丁。额外的内核功能被整合进主流内核之前通常是以独立的补丁存在的。例如,Robert Love对内核的抢占功能所做的修补,在被Linus整合进2.5开发版系列之前,被制作成独立的补丁。我们将会在第十一章探讨如何对内核进行修补。如果对修补的事情不熟悉的话,可参考《Running Linux》(O'Reilly)这本书。

内核配置

在你为目标板建立内核的过程中，配置属于最初的阶段。内核配置的方法很多，而且配置时有许多选项可以选择。

不管你使用哪种方法来设定配置或选择哪些配置选项，在你设好配置之后内核都将会产生 `.config` 文件以及建立过程其余步骤将会用到的一些符号链接和头文件。

以下的讨论将会局限于在嵌入式系统中配置内核有什么不同。如果对内核配置不熟悉的话，可查阅我前面所提到的各种参考资源。

配置选项

配置设定期间，想纳入内核的功能可以通过选项来选择。因目标板而定，所看到的选项可能不一样。然而，有些选项不管你选用哪种嵌入式架构都会存在。以下列出所有嵌入式 Linux 架构都看得到的主菜单选项：

- Code maturity level options
- Loadable module support
- General setup
- Memory technology devices
- Block devices
- Networking options
- ATA/IDE/MFM/RLL support
- SCSI support
- Network device support
- Input core support
- Character devices
- Filesystems
- Console drivers
- Sound
- Kernel hacking

我不准备深入探讨每个选项，因为内核配置菜单提供有辅助说明的功能，可以在设定配置的同时参考它。然而请注意，我们在第三章已经讨论过其中不少选项。

菜单中最重要的就是你用来为目标板选择处理器架构的选项。然而此选项的名称因架构而异。表 5-2 根据各种处理器架构列出了系统和处理器选项的名称以及正确的内核架构名称。在我们执行 *make* 命令的时候，需要将 ARCH 的值设成内核的 Makefile 中定义的正确架构名称。

表 5-2：根据处理器架构列出系统和处理器选项及内核架构的名称

处理器架构	系统和处理器选项	内核架构名称
x86	Processor type and features	i386
ARM	System type	arm
PPC	Platform support	ppc
MIPS	Machine selection/CPU selection	mips 或 mips64 ^a
SH	Processor type and features	sh
M68k	Platform-dependent support	m68k

a. 取决于 CPU。

有些选项只会在特定的架构中出现。表 5-3 列出了这些选项在内核配置菜单中的名称，并指出它们会出现在哪些架构上。

表 5-3：每种架构的硬件支持选项

选项	x86	ARM	PPC	MIPS	SH	M68k
Parallel port support	×	×		×		
IEEE 1394 support	×	×	×		×	
IrDA support	×	×	×	×		
USB support	×	×	×	×		
Bluetooth support	×	×	×			

有些架构具有专用的配置选项。下面列出的是 ARM 架构特有的选项：

- Acorn-specific block devices
- Synchronous serial interfaces
- Multimedia capabilities port drivers

下面则是 PPC 特有的选项：

- MPC8xx CPM options
- MPC8260 communication options

事实上，即使某个选项出现在架构的配置菜单中，并不代表目标板支持此功能。更确切地说，配置选项可能会允许你启用目标板未曾测试过的许多功能。例如，ARM 系统上没有 VGA 控制台。然而，内核的配置菜单却允许你启用 VGA 控制台的支持。因此，如果启用了这个选项，建立内核将会失败，或者造成所选用的功能甚至是整个内核毫无作用。要避免发生这类问题，确定目标板支持你启用的选项。大部分时候，就像 VGA 控制台那样，这是一般常识。当对某些选项不是很清楚的时候，可以访问相关计划的网站，例如第三章提到的网站将有助于你判断目标板是否支持此功能。

有些时候，即使某个选项并未在架构的配置菜单中显示，并不代表此功能无法在目标板上使用。表 5-3 列出了许多此类功能，例如 Bluetooth，它们大多数与架构无关，在任何架构上运行都应该没有问题。它们之所以未列在特定架构的配置菜单中，不是因为它们没有在那些架构测试过，就是因为那些移植成果或功能的维护者并未要求将此功能加入内核的因架构而异的 *config.in* 文件中（注 2）。再次提醒，如果想找出目标板可能支持哪些未列出的功能，第三章提到的参考资源是很好的起点。

设定配置的方法

内核支持四种设定配置的方法：

make config

通过命令接口，依次要求你设定每个选项。如果 *.config* 配置文件存在，它会根据该文件来设定（你设定的选项的）默认值。

make oldconfig

通过命令接口，但是会自动馈入既有的 *.config* 配置文件，并且只有在遇到先前没有设定过的选项时，才会要求你手动设定。然而，*make config* 却会要求你手动设定所有的选项，即使你之前曾设定过。

make menuconfig

显示以 *curses* 为基础的、终端式的配置菜单。如果 *.config* 文件存在，它会根据该文件来设定默认值，如同 *make config* 一样。

注 2： *config.in* 文件用来控制配置菜单应该显示哪些选项。

make xconfig

显示以 Tk 为基础的 X Window 配置菜单。如果 *.config* 文件存在，它会根据该文件来设定默认值，如同 *make config* 和 *make menuconfig* 一样。

以上四种方法都可以用来设定内核配置。它们都会在内核源码的根目录中产生 *.config* 文件。（此文件包含了你对选项所做的设定的全部细节。）

只有少数开发者会使用 *make config* 来设定内核配置。一般开发者通常会使用 *make menuconfig*。你也可以使用 *make xconfig*。然而不要忘了，在某些架构中，例如 PowerPC，使用 *make xconfig* 时，菜单可能有问题。

要检查内核配置菜单，只须在命令行上键入适当的命令以及恰当的参数。对这个以 ARM 为基础的用户接口模块，我们可以使用如下的命令行：

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

然后我们必须选择适合目标板使用的配置选项。许多功能和驱动程序可以以模块的形式存在，我们可以选择是否将它们建立在内核中，或是将它们建立成模块的形式。一旦设定好内核配置之后，我们可以使用 Esc 键或是选择 Exit 条目离开配置菜单。然后配置工具程序会问我们是否保存配置。我们可以选择 Yes 条目来保存内核的配置以及建立 *.config* 文件。这除了会建立 *.config* 文件，也会建立一些的头文件和符号连接。如果选择 No，则不会保存我们设定的配置，也不会改变既有的配置。

除了主要的配置选项，有些架构（例如 PPC 和 ARM）的配置设定，也可以使用为各种目标板实现时采用的架构而定制的配置。因此，它们提供给内核的默认值将会用来产生 *.config* 文件。举例来说，我会以如下的命令来为 TQM860L PowerPC 目标板设定内核配置：

```
$ make ARCH=ppc CROSS_COMPILE=powerpc-linux- TQM860L_config  
$ make ARCH=ppc CROSS_COMPILE=powerpc-linux- oldconfig
```

管理多种配置

我们通常都需要针对相同的内核源码测试不同配置。然而，如果改变内核的配置则会破坏先前的配置，这是因为所有配置文件都会被内核的配置工具程序（命令脚本）给覆盖掉。要将配置保存起来供日后使用，我们需要将内核的配置工具程序建立的 *.config* 文件保存起来。这些文件日后可用来恢复先前设定的内核配置。

备份及恢复配置的最简单方法就是使用内核自己的配置设定程序。*menuconfig* 和

`xconfig`这两个Makefile目标显示的菜单,可以让你保存和恢复配置。不论保存和恢复,都需要提供适当的文件名。

你还可以手动保存`.config`文件。方法是将内核配置命令脚本产生的配置文件(即`.config`文件)复制到其他地方供日后使用。要使用已保存的配置,必须将先前手动保存的`.config`文件复制回内核源码树的根目录,然后用`make`命令及`oldconfig`这个Makefile目标使用刚才复制的`.config`文件来设定内核的配置。如同`menuconfig`这个Makefile目标,`oldconfig`会产生若干头文件及符号链接。

不管你是手动或者是使用其他工具程序提供的菜单来复制文件,配置应该保存在凭直觉就可以找到的地方,并且使用有意义的命名机制来存放配置。以我们的项目工作空间为例,建议将所有的配置保存到`${PRJROOT}/kernel`目录,这样既可以独立于实际的内核源码之外,又可以表示它们是属于内核的数据。要区分每个配置文件,可以在其文件名前置相应内核版本的编号以及简单的描述或日期,或二者。最后,让`.config`变成扩展(副)文件名,这样一看便知道这是个内核配置文件。

以我们所使用的2.4.18版内核为例,我为它设定了一个不提供串行端口支持的配置。因此我将相应的配置文件称为`2.4.18-no-serial.config`。我还保存了一个到目前为止被认为是“最佳”的配置,并将它称为`2.4.18.config`。你可以随意采用任何你认为最直观的命名习惯,不过你可能要避免通称形式的名字,例如`2.4.18-test1.config`。

使用 EXTRAVERSION 变量

如果使用的是同一个内核版本的多个变体,会发现在你想要辨别每个实体时`EXTRAVERSION`变量非常有用。`EXTRAVERSION`变量的值会被附加在内核的版本编号之后,成为内核建立完成后的最终版本编号。例如,为2.4.18版的内核加入`rmk5`的补丁后,`EXTRAVERSION`变量的值会设成`-rmk5`,该内核建立完成后的最终版本编号为`2.4.18-rmk5`。

这个最终的版本编号还会当做一个目录的名称,你为内核建立的模块将会存放到此目录中。因此,当你为同一个内核版本建立两次模块时,如果`EXTRAVERSION`变量前后两次的值不同,则前后两次产生的模块将会存放不同的目录;如果前后两次都没有为`EXTRAVERSION`变量设置,则前后两次产生的模块将会存放相同的目录。

你还可以使用这个变量来区分基于同一个内核版本的各个变体。方法是编辑内核源码主目录中的Makefile文件,并将`EXTRAVERSION`变量设成你想要的值。`EXTRAVERSION`变量的值还有一个用法,就是你可以根据此值来为内容已改变的内核源码主目录改名。例如,倘若2.4.18版内核的`EXTRAVERSION`变量被设成`-motor-diff`,那么内核源码的

主目录应该被改名为 *2.4.18-motor-diff*。而你备份的 *.config* 文件应该使用 *EXTRAVERSION* 变量的值来命名。如果此内核的配置被设成不提供串行端口的支持, 则其配置文件应该被命名为 *2.4.18-motor-diff-no-serial.config*。

编译内核

内核的编译包括以下几个步骤: 建立内核源码的依存关系, 建立内核映像, 以及建立内核模块。以上每个步骤使用的 *make* 命令都不同, 以下我们会分节说明这几个步骤。当然, 也可以使用命令行来执行这几个步骤。

建立依存关系

内核源码树中大多数文件都会与一些头文件有依存关系。要想顺利建立内核, 内核源码树里各个 Makefile 必须知道这些依存关系。依存关系建立期间会在内核源码树中每个子目录里产生一个隐藏的 *.depend* 文件。此文件内含子目录里各文件所依存的头文件清单。如同其他靠 *make* 建立的软件, 自从上一次完成建立以来, 如果要重新建立内核, 只有在与头文件有依存关系的文件被改动后才需要经过重新编译的程序。

你可以从内核源码树的根目录, 以如下的命令来建立内核源码的依存关系:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- clean dep
```

这么做是因为前面在设定内核配置的时候, 我们曾设定 *ARCH* 和 *CROSS_COMPILE* 变量。正如第四章所说, 只有在实际编译源码的时候才需要设定 *CROSS_COMPILE* 变量。相对而言, 我们至少需要为自己发布的每道 *make* 命令设定 *ARCH* 变量, 因为我们正在进行内核的交叉编译。就算发布的是 *make clean* 或 *make distclean* 命令, 我们还是需要设定此变量。否则内核源码树里各个 Makefile 会假定, 目前对内核源码所进行的操作都跟主机的架构有关。

ARCH 变量用来指明要为哪种架构建立内核。内核源码树里各个 Makefile 会根据此变量来选择它准备使用的与架构相关的目录。当你为目标板编译内核的时候必须将此变量设成目标板的架构。

内核源码树里各个 Makefile 会根据 *CROSS_COMPILE* 来产生一些工具程序的名称, 这些工具程序将会用在建立内核的过程中。例如 C 编译器的名称就是 *CROSS_COMPILE* 变量的值与“*gcc*”串接而成的。以 ARM 目标板为例, C 编译器的最后名称为 *arm-linux-gcc*, 这就是我们根据第四章的指示建立的 C 编译器的实际名称。这也说明了, 为什么之前的命令行中结尾的连字号 (-) 如此重要的缘故。少了这个连字号, Makefile 所使用的编译器将会是 *arm-linuxgcc*, 这个编译器根本就不存在。

建立依存关系所需的时间相对较短。在我的PowerBook上,这个工作只需要2分钟的时间。这个阶段通常不会看到任何错误信息。如果看到了错误信息,表示内核可能遇到重大问题(译注1)。

建立内核

建立依存关系后,接着编译内核映像:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- zImage
```

zImage这个建立目标用来指示Makefile建立经gzip算法压缩过的内核映像(注3)。不过还有其他方法可用来建立内核映像。例如vmlinux这个建立目标可用来指示Makefile只建立未经压缩的映像。请注意,当你要求建立经过压缩的映像时,也会产生这个未经压缩的映像。

在x86上,还可以使用bzImage这个建立目标。bzImage是big zImage的简写,它与bzip2压缩工具程序无关。事实上,bzImage和zImage这两个建立目标都是使用gzip算法。差别在于zImage产生的经压缩的内核映像无法超过512 KB,而bzImage则不受这个限制。如果想要进一步了解zImage与bzImage的差异,可以检查内核源码树包含的*Documentation/i386/boot.txt*文档。

内核配置设定期间,如果选择了架构不支持的选项或一些有问题的内核选项,建立工作将会在此阶段失败。如果一切顺利,整个建立过程所花的时间应该会比建立依存关系多几分钟。以我的硬件配置来说,整个过程需要5分钟的时间。

验证交叉开发工具链

请注意,建立内核让我们能够实际测试前一章建立的交叉开发工具。如果前面建立的工具有能够成功编译出可用的内核,这代表所有其他软件的建立应该都没问题。当然,将需要下载内核源码,为目标板建立内核,以便验证其可用性,事实上,能够正确建立,已经具有正面意义了。

译注1: 例如找不到编译器。

注3: 尽管对我们在第三章深入探讨的所有架构来说,zImage是个有效的Makefile target(建立目标),但是对其他的Linux架构来说则不然。

建立模块

内核映像正确建立后，接着建立内核模块：

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

这个阶段的时间长短与“你选择不要链接成主内核映像的一部分而要建立成模块的内核选项的数量”有很大的关系。这个阶段的时间很少会长过建立内核映像的时间。如同内核映像一样，倘若配置不符合目标板的需要或是内核选项有问题，这个阶段也可能会失败。

内核映像和内核模块都正确建立之后，表示我们已经准备好可以为目标板进行安装工作了。有一件事很重要，在进行之前要特别注意，如果需要清理内核的源码，让它恢复到配置设定、依存关系建立或编译之前的初始状态，可以使用如下的命令：

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- distclean
```

但务必在执行此命令之前将内核的配置文件的备份起来，因为 *make distclean* 会清除前面这几个阶段产生的文件，包括 *.config* 文件、所有目标文件以及内核映像。

安装内核

最后，我们产生的内核映像以及它的模块必须复制到目标板上去。我将会在第六和第九章说明，如何复制内核映像以及它的模块。这里要讨论的是，如何管理多个内核映像以及相应模块的安装。目标板引导配置的安排以及它的根文件系统跟我们底下要讨论的技术有关。

管理多个内核映像

除了使用分开的目录来存放不同的内核版本，还会发现能够“使用多个内核映像来测试目标板”很有用。因为这些映像可能是由相同的源码建立的，所以我们需要将它们从内核源码树复制到不同的目录中。以我们的设置来说，这些映像会被存放到 *\${PRJROOT}/images* 目录。

对每个内核配置来说，我们将需要复制四个档案：压缩的内核映像、未压缩的内核映像、内核符号映射文件以及配置文件。最后三个文件可以在内核源码树的根目录中找到，它们的文件名分别是 *vmlinux*、*System.map* 和 *.config*。而未压缩的内核映像则可在 *arch/YOUR_ARCH/boot* 目录中找到（而 *YOUR_ARCH* 就是目标板架构的名称），其文件名可能是 *zImage* 或 *bzImage*，这取决于你前面使用的 Makefile 建立目标。对使用 ARM 的目标板来说，未压缩的内核映像就是 *arch/arm/boot/zImage*。

有些架构（例如 PPC）具有多个引导目录。在这种情况下，所要使用的内核映像，未必会放在 `arch/YOUR_ARCH/boot/zImage`。以之前提到的 TQM 板为例，压缩的内核映像应该会放在 `arch/ppc/images/vmlinux.gz`。根据目标板来检查 `arch/YOUR_ARCH/Makefile` 文件，可以看到各种引导映像的 Makefile 建立目标的完整细节。以我们的 PPC 范例来说，其产生的引导映像类型取决于编译内核时所针对的那个处理器型号。

为了区分这四个文件，我们使用与内核版本编号类似的命名机制。例如，倘若内核映像是由 `2.4.18-rmk5` 版的源码产生的，我们会以如下的方式复制这四个文件：

```
$ cp arch/arm/boot/zImage ${PRJROOT}/images/zImage-2.4.18-rmk5
$ cp vmlinux ${PRJROOT}/images/vmlinux-2.4.18-rmk5
$ cp System.map ${PRJROOT}/images/System.map-2.4.18-rmk5
$ cp .config ${PRJROOT}/images/2.4.18-rmk5.config
```

你还可以在文件名中纳入配置名称。因此在内核不提供串行支持的情况下，我们可以将这四个文件称为 `zImage-2.4.18-rmk5-no-serial`、`vmlinux-2.4.18-rmk5-no-serial`、`System.map-2.4.18-rmk5-no-serial` 和 `2.4.18-rmk5-no-serial.config`。

安装内核模块

内核源码的 Makefile 文件包含的 `modules_install` 建立目标用来安装内核模块。缺省情况下，内核模块会安装到 `/lib/modules` 目录。然而，因为我们使用的是交叉开发环境，所以我们必须指示 Makefile 将模块安装到另一个目录。

因为内核模块由相应的内核映像使用，所以我们会把模块安装到名称与内核映像类似的目录。以我们使用的 `2.4.18-rmk5` 内核为例，我们会把模块安装到 `/${PRJROOT}/images/modules-2.4.18-rmk5` 目录。此目录的内容稍后将会被复制到目标板的根文件系统，供目标板上相应的内核使用。我们会以如下的方式将模块安装到该目录：

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- \
> INSTALL_MOD_PATH=${PRJROOT}/images/modules-2.4.18-rmk5 \
> modules_install
```

`/lib/modules` 路径会前置 `INSTALL_MOD_PATH` 变量的值，因此模块会被安装到 `/${PRJROOT}/images/modules-2.4.18-rmk5/lib/modules` 目录。

一旦复制好模块之后，内核就会根据模块工具程序执行时的要求，试着建立模块依存关系。因为 `depmod` 这个用来建立模块依存关系的工具程序，并非用来处理交叉编译的模块，所以会执行失败。

要为模块建立模块依存关系必须用到BusyBox提供的模块依存关系建立程序。BusyBox在第六章将会有深入的讨论。现在我们可以从<http://www.busybox.net/>将BusyBox包的副本下载到`/${PRJROOT}/sysapps`目录，并在该处取出源码（注4）。接着我们可以从存放BusyBox源码的目录将`scripts/depmod.pl`命令脚本复制到`/${PREFIX}/bin`目录。

现在我们可以为目标板建立模块依存关系：

```
$ depmod.pl \  
> -k ./vmlinux -F ./System.map \  
> -b ${PRJROOT}/images/modules-2.4.18-rmk5/lib/modules > \  
> ${PRJROOT}/images/modules-2.4.18-rmk5/lib/modules/2.4.18-rmk5/modules.dep
```

其中，`-k`选项用来指定未压缩的内核映像，`-F`选项用来指定系统对映文件，`-b`选项用来指定内核需要建立依存关系的模块的基本目录。因此此工具的执行结果会送往标准输出，我们可以将它重定向到一个文件中，其文件名总是叫做`modules.dep`。

实地测试

当内核安装在目标板上准备执行的时候，让我们检查内核的运行状态。因为嵌入式系统的算法以及底下的源码跟一般系统没两样，所以内核在嵌入式系统上的运行几乎跟在工作站和服务器上完全一样。因此，对内核有更深入探讨的其他书籍和在线资料，例如O'Reilly出版的《Linux Device Drivers》和《Understanding the Linux Kernel》并未特别强调它们也适用于嵌入式Linux系统。

当内核失败时

Linux内核是一个非常稳定且成熟的软件。然而这并不表示Linux或Linux使用的硬件永远不会出故障。《Linux Device Drivers》一书便探讨oops信息和系统挂起之类的问题。除了需要在系统设置期间留意这些问题，还应该考虑到内核失败的最常见形式：内核恐慌。

当发生严重的错误而且被内核捕捉到时，内核将会停止所有的进程并且送出内核恐慌信息。发生内核恐慌的理由很多。其中最常见的是，忘了为内核指定它的根文件系统的位置。在这种情况下，内核将会正常引导，但是会在尝试安装根文件系统的时候发生内核恐慌。要从内核恐慌错误中恢复，唯一的办法就是让系统重新引导。因此内核可以接受用来指定“内核应该在内核恐慌之后几秒重新引导”的引导参数。例如，倘若你想让内核在内核恐慌之后1秒重新引导就应该使用`panic=1`这个内核引导参数。

注4：请下载BusyBox 0.60.5或之后的版本。

然而有时就算是重新引导可能还不够，这取决于我们的设置。例如我们的控制模块，重新引导甚至可能导致危险，因为它控制的化学或机械程序可能会失常。因此我们需要修改内核的 `panic` 函数，让它通知操作人员使用紧急程序手动控制系统。当然，系统对内核恐慌的实际反应，取决于系统的实际应用。

内核的 `panic` 函数（即 `panic()`）的程序代码就放在内核源码树里的 `kernel/panic.c` 文件中。我们首先注意到，缺省情况下，`panic` 函数的执行结果会输出到控制台（注 5）。因为系统可能连终端都没有，所以你可能会想要根据系统特殊的硬件需求来修改该函数。例如你可以将实际的错误信息字符串写入闪存中你特别为此用途保留的区段。在系统重新引导之后，就能够从该闪存区段取回文字信息，并根据此信息解决问题。

不论你是否对实际的文字信息感兴趣，都可以向内核注册自己的 `panic` 函数。该函数会在某个内核恐慌事件发生时，被内核的 `panic` 函数调用，可用来进行发出警报、通知紧急事件之类的事。

被内核自己的 `panic` 函数调用的其他 `panic` 函数，会被放在 `panic_notifier_list` 列表中。`notifier_chain_register` 函数可用来将一个条目（其他 `panic` 函数）加入此列表中。相反地，`notifier_chain_unregister` 函数可用来将一个条目从此列表中移除。

你自己的 `panic` 函数摆在哪儿不是很重要，但是务必在系统初始化期间完成函数的注册工作。以我们的例子来说，`mypanic.c` 文件会被放在内核源码树里的 `kernel/` 目录中，我们会依需要修改该目录的 `Makefile`。下面是供我们的控制模块使用的 `mypanic.c` 文件：

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/notifier.h>

static int my_panic_event(struct notifier_block *,
                          unsigned long,
                          void *);

static struct notifier_block my_panic_block = {
    notifier_call:    my_panic_event,
    next:            NULL,
    priority:        INT_MAX
};

int __init register_my_panic(void)
{
    printk("Registering buzzer notifier \n");

    notifier_chain_register(&panic_notifier_list,
                           &my_panic_block);
}
```

注 5： 所有系统信息都会送往控制台这个主要的终端。

```
        return 0;
    }

void ring_big_buzzer(void)
{
    ...
}

static int my_panic_event(struct notifier_block *this,
                        unsigned long event,
                        void *ptr)
{
    ring_big_buzzer();

    return NOTIFY_DONE;
}

module_init(register_my_panic);
```

`module_init(register_my_panic);`这个语句使得在内核初始化期间就可以调用 `register_my_panic` 函数，因此不必对内核的启动函数做任何修改。注册函数会在 `panic_notifier_list` 列表中加入 `my_panic_block` 条目。`notifier_block` 结构具有三个字段。第一个字段是所调用的函数，第二个字段会指向下一个 `notifier_block` 条目，第三个字段则是此条目的优先权。此例子中，我们想要让它具有最高的优先权，因此会使用 `INT_MAX` 这个值。

当发生内核恐慌时，`my_panic_event` 函数会被调用，成为所有 panic 函数的内核通告的一部分。接下来，`my_panic_event` 会调用 `ring_big_buzzer`，函数里的程序代码会启动响亮的警报器，让操作员注意到即将发生的问题。