第2章 代码初识

本章首先从较高层次介绍 Linux 内核源程序的概况，这些都是大家关心的一些基本特点。随后将简要介绍一些实际代码。最后以如何编译内核来检验个人所进行的修改的讨论来作为本章的收尾。

Linux 内核源程序的部分特点

在过去的一段时期，Linux 内核同时使用 C 语言和汇编语言实现的。这两种语言需要一定的平衡：C 语言编写的代码移植性较好、易于维护，而汇编语言编写的程序则速度较快。一般只有在速度是关键因素或者一些因平台相关特性而产生的特殊要求（例如直接和内存管理硬件进行通讯）时才使用汇编语言。

正如同实际中所做的，即使内核并未使用 C++的对象特性，部分内核也可以在 g++（GNU 的 C++编译器）下进行编译。同其它面向对象的编程语言相比较，相对而言 C++的开销是较低的，但是对于内核开发人员来说，这已经足够甚至太多了。

内核开发人员不断发展编程风格，形成了 Linux 代码独有的特色。本节将讨论其中的一些问题。

gcc 特性的使用

Linux 内核被设计为必须使用 GNU 的 C 编译器 gcc 来编译，而不是任何一种 C 编译器都可以使用。内核代码有时要使用 gcc 特性，伴随着本书的进程，我们将陆续介绍其中的一部分。

一些 gcc 特有代码只是简单地使用 gcc 语言扩展，例如允许在 C（不只是 C++）中使用 **inline** 关键字指示内联函数。也就是说，代码中被调用的函数在每次函数调用时都会被扩充，因而就可以节约实际函数调用的开销。

更为普遍的情况是代码的编写方式比较复杂。因为对于某些类型的输入，gcc 能够产生比其它输入效率更高的执行代码。从理论上讲，编译器可以优化具有相同功能的两种对等的方法，并且得到相同的结果。因此，代码的编写方式是无关紧要的。但在实际上，用一些方法编写所产生的代码要比用其它方法编写所产生的代码的执行速度快得多。内核开发人员清楚如何才能产生更高效的执行代码的方法，而且这种知识也不断在他们编写的代码中反映出来。

例如，考虑内核中经常使用的 **goto** 语句——为了提高速度，内核中经常大量使用这种一般要避免使用的语句。在本书中所包含的不到 40,000 行代码中，一共有 500 多条 **goto** 语句，大约是每 80 行一个。除汇编文件外，精确的统计数字是接近每 72 行一个 **goto** 语句。公平的说，这是选择偏向的结果：比例如此高的原因之一是本书中涉及的是内核源程序的核心，在这里速度比其它因素都需要优先考虑。整个内核的比例大概是每 260 行一个 **goto** 语句。然而，这仍然是我不再使用 Basic 进行编程以来见过的使用 **goto** 频率最高的地方。

代码必需受特定编译器限制的特性不仅与普通应用程序的开发有很大不同，而且也不同于大多数内核的开发。大多数的开发人员使用 C 语言编写代码来保持较高的可移植性，即使在编写操作系统时也是如此。这样做的优点是显而易见的，最为重要的一点是一旦出现更

好的编译器，程序员们可以随时进行更换。

内核对于gcc特性的完全依赖使得内核向新的编译器上的移植工作更加困难。最近Linus对这一问题在有关内核的邮件列表上表明了自己的观点。“ 记住，编译器只是一个工具。”这是对依赖于 gcc 特性的一个很好的基本思想的表述：编译器只是为了完成工作。如果通过遵守标准还不能达到工作要求，那么就不是工作要求有问题，而是对于标准的依赖有问题。

在大多数情况下，这种观点是不能够被人所接受的。通常情况下，为了保证和程序语言标准的一致，开发人员可能需要牺牲某些特性、速度或者其它相关因素。其它的选择可能会为后期开发造成很大的麻烦。

但是，在这种特定的情况下，Linus 是正确的。Linux 内核是一个特例，因为其执行速度要比向其它编译器的可移植性远为重要。如果设计目标是编写一个可移植性好而不要求快速运行的内核，或者是编写一个任何人都可以使用自己喜欢的编译器进行编译的内核，那么结论就可能会有所不同了；而这些恰好不是 Linux 的设计目标。实际上，gcc 几乎可以为所有能够运行 Linux 的 CPU 生成代码，因此，对于 gcc 的依赖并不是可移植性的严重障碍。

在第 3 章中我们将对内核设计目标进行详细说明。

内核代码习惯用语

内核代码中使用了一些显著的习惯用语，本节将介绍常用的几个。当你通读源程序代码时，真正重要的问题是并不在这些习惯用语本身，而是这种类型的习惯用语的确存在，而且是不断被使用和发展的。如果你需要编写内核代码，你应该注意到内核中所使用的习惯用语，并把这些习惯用语应用到你的代码中。当通读本书（或者代码）时，注意你还能找到多少习惯用语。

为了讨论这些习惯用语，我们首先需要对它们进行命名。为了便于讨论，笔者创造了这些名字。而在实际中，大家不一定非要参考这些用语，它们只是对内核工作方式的描述而已。

一个普通的习惯用语笔者称之为“ 资源获取 ”（resource acquisition idiom）。在这个用语中，一个函数必须实现一系列资源的获取，包括内存、锁等等（这些资源的类型未必相同）。只有成功地获取当前所需要的资源之后，才能处理后面的资源请求。最后，该函数还必须释放所有已经获取的资源，而不必对没有获取的资源进行考虑。

我采用“ 错误变量 ”这一用语（error variable idiom）来辅助说明资源获取用语，它使用一个临时变量来记录函数的期望返回值。当然，相当多的函数都能实现这个功能。但是错误变量的不同点在于它通常是用来处理由于速度的因素而变得非常复杂的流程控制中的问题。错误变量有两个典型的值，0（表示成功）和负数（表示有错）。

这两个用语结合使用，我们就可以十分自然地得到符合模式的代码如下：

```
Int f (void)
{
   int err;
   resource *r1, *r2;
   err = -ERR1      /*assume failure*/
   r1=acquire_ resource();
   if (!r1)             /*not aquired*/
        goto out          /*returns    -ERR1*/

   Got resource r1,try for r2.*/
```

```
    err = - ERR2;
    r2 = acquire_ resource2();
    if (!r2)            /*not aquired*/
        goto out1           /*returns –ERR2*/

    /*have both r1 and r2.*/
    err = 0;

    /* … use r1 and r2 … */

out2:
    release_resource(r2)

out2:
    release_resource(r2)

out:
        return err;
}
```

（注意变量 **err** 是使用错误变量的一个明确实例，同样，诸如 **out** 之类的标号则指明了资源获取用语的使用。）

如果执行到标号 **out2**，则都已经获取了 **r1** 和 **r2** 资源，而且也都需要进行释放。如果执行到标号 **out1**（不管是顺序执行还是使用 **goto** 语句进行跳转到），则 **r2** 资源是无效的（也可能刚被释放），但是 **r1** 资源却是有效的，而且必需在此将其释放。同理，如果标号 **out** 能被执行，则 **r1** 和 **r2** 资源都无效，**err** 所返回的是错误或成功标志。

在这个简单的例子中，对于 **err** 的一些赋值是没有必要的。在实践中，实际代码必须遵守这种模式。这样做的原因主要在于同一行中可能包含有多种测试，而这些测试应该返回相同的错误代码，因此对错误变量统一赋值要比多次赋值更为简单。虽然在这个例子中对于这种属性的必要性并不非常迫切，但是我还是倾向于保留这种特点。有关的实际应用可以参考 **sys_shmctl**（第 21654 行），在第 9 章中还将详细介绍这个例子。

减少#if 和#ifdef 的使用

现在的 Linux 内核已经移植到不同的平台上，但是我们还必须解决移植过程中所出现的问题。大部分支持各种不同平台的代码由于包含许多预处理代码现都已变得非常不规范，例如：

```
#if defined(SOLARIS)
/* … do things the solaris way … */
#elif defined(HPUX)
/* … do things the HP-UX way … */
#elif defined(LINUX)
/* … do things the right way … */
```

```
#endif
```

这个例子试图实现操作系统的可移植性，虽然 Linux 关注的焦点很明显是实现代码在各种 CPU 上的可移植性，但是二者的基本原理是一致的。对于这类问题来说，预处理器是一种错误的解决方式。这些杂乱的问题使得代码晦涩难懂。更为糟糕的是，增加对新平台的支持有可能要求重新遍历这些杂乱分布的低质量代码段（实际上你很难能找到这类代码段的全部）。

与现有方式不同的是，Linux 一般通过简单函数（或者是宏）调用来抽象出不同平台间的差异。内核的移植可以通过实现适合于相应平台的函数（或宏）来实现。这样不仅使代码的主体简单易懂，而且在移植的过程中还可以比较容易地自动检测出你没有注意到的内容：如引用未声明函数时会出现链接错误。有时用预处理器来支持不同的体系结构，但这种方式并不常用，而相对于代码风格的变化就更是微不足道了。

顺便说一下，我们可以注意到这种解决方法和使用用户对象（或者 C 语言中充满函数指针的 **struct** 结构）来代替离散的 **switch** 语句处理不同类型的方法十分相似。在某些层次上，这些问题和解决方法是统一的。

可移植性的问题并不仅限于平台和 CPU 的移植，编译器也是一个重要的问题。此处为了简化，假设 Linux 只使用 gcc 来编译。由于 Linux 只使用同一个编译器，所以就没有必要使用**#if** 块（或者**#ifdef** 块）来选择不同的编译器。

内核代码主要使用**#ifdef** 来区分需要编译或不需要编译的部分，从而对不同的结构提供支持。例如，代码经常测试 **SMP** 宏是否定义过，从而决定是否支持 SMP 机。

代码样例

上一节仅仅是一些讨论，了解 Linux 代码风格最好的方法就是实际研究一下它的部分代码。即使你不完全理解本节所讨论代码的细节也无关紧要，毕竟本节的主要目的不是理解代码，一些读者可以只对本节进行浏览。本节的主要目的是让读者对 Linux 代码进行初步了解，对今后的工作提供必要基础。而讨论将涉及部分广泛使用到的内核代码。

printk

printk（25836 行）是内核内部消息日志记录函数。在出现诸如内核检测到其数据结构出现不一致的事件时，内核会使用 **printk** 把相关信息打印到系统控制台上。对于 **printk** 的调用一般分为如下几类：

- 紧急事件（emergency）——例如，**panic** 函数（25563 行）多次使用了 **printk**。当内核检测到发生不可恢复的内部错误时就会调用 **panic** 函数，然后尽其所能的安全关闭计算机。这个函数中调用 **printk** 以提示用户系统将要关闭。
- 调试——从 3816 行开始的**#ifdef** 块使用 **printk** 来打印 SMP 逻辑单元（box）中每一个处理器的相关配置信息，但是此过程只有在使用 **SMP_DEBUG** 标志编译代码的情况下才能够被执行。
- 普通信息——例如，当机器启动时，内核必需估计系统速度以确保设备驱动程序能够忙等待（busy-waiting）一个精确的极短周期。计算这种估计值的函数名为 **calibrate_delay**（19654 行），它既在 19661 行使用 **printk** 声明马上开始计算，又在 19693 行报告计算结果。另外，在第 4 章将详细的介绍 **calibrate_delay** 函数。

如果你已经浏览过这些参照代码，你可能已经注意到 **printk** 和 **printf** 的参数十分类似：

一个格式化字符串，后跟零个或者多个参数加入字符串中。格式化字符串可能是以一组“ <N> ”开始，这里的 N 是从 0 到 7 的数字，包括 0 和 7 在内。数字区分了消息的日志等级（log level），只有当日志等级高于当前控制台定义的日志等级（**console_loglevel**，25650 行）时，才会打印消息。root 可以通过适当减小控制台的日志等级来过滤不是很紧急的消息。如果内核在格式化字符串中检测不到日志等级序列，那么就会一直打印消息。（实际上，日志等级序列并不一定要在格式化字符串中出现，可以在格式化文本中查找到它的代码。）

从 14946 行开始的**#define** 块说明了这些特殊序列，这些定义可以帮助调用者正确区分对 **printk** 的调用。简单的说，我称日志等级 0 到 4 为“ 紧急事件 ”，从等级 5 到等级 6 为“ 普通信息 ”，等级 7 自然就是我所说的“ 调试 ”。（这种分类方法并不意味着其它更好的分类方法没有用处，而只是目前我们还不关心它而已。）

在上面讨论的基础上，我们研究一下代码本身。

printk

25836：参数 **fmt** 是 **printf** 类型的格式化字符串。如果你对“ … ”部分的内容不熟悉，那就需要参阅一本好的 C 语言参考书（在其索引中查找“ 变参函数，variadic function ”）。另外，在安装的 GNU/Linux 中的 **stdarg** 帮助里也包含了一个有关变参函数的简明描述，在这儿只需要敲入“ man stdarg ”就可以看到。

简单的说，“ … ”部分提示编译器 **fmt** 后面可能紧跟着数量不定的任何类型的参数。由于这些参数在编译的时候还没有类型和名字，内核使用由三个宏 **va_start**，**va_arg** 和 **va_end** 组成的特殊组以及一个特殊类型——**va_list** 对它们进行处理。

25842：**msg_level** 记录了当前消息的日志等级。它是静态的，这看起来可能会有些奇怪——为什么下一次对 **printk** 的调用需要记录日志等级呢？问题的答案是只有打印出新行（\n）或者赋给一个新的日志等级序列以后，当前消息才会结束。这样通过在包含消息结束的新行里调用 **printk**，就保证了在多个短期冲突的情况下，调用者只打印唯一一个长消息。

25845：在 SMP 逻辑单元中，内核可能试图从不同的 CPU 向控制台同时打印信息。（有时在单处理机（UP）逻辑单元中也会发生同样问题，但由于中断还未被覆盖掉，所以问题也并不十分明显。）如果不进行任何协同的话，结果就将处于完全无法让人了解的杂乱无章的状态，每个消息的各个部分都和其它消息的各个部分混杂交织在一起。相反，内核使用旋转锁（spin-lock）来控制对控制台的访问。旋转锁将在第 10 章对它进行深入的介绍。

如果你对 **flags** 在传送给 **spin_lock_irqsave** 之前为什么不对它初始化感到疑惑，请不要担心：**spin_lock_irqsave**（对于不同的版本请分别参看 12614 行，12637 行，12716 行，和 12837 行）是一个宏，而不是一个函数。该宏实际上是将值写入 **flags** 中，而不是从 **flags** 中读出值。（在 25895 行中，存储在 **flags** 中的信息被 **spin_lock_irqsave** 回读，请参看 12616 行，12639 行，12728 行和 12841 行）

25846：初始化变量 **args**，该变量代表 **printk** 参数中的“ … ”部分。

25848：调用内核自身的 **vsprintf**（为节省空间而省略）实现。该函数的功能与标准 **vsprintf** 函数非常相似，向 **buf** 中写入格式化文本（25634 行）并返回写入字符串的长度（长度不包括最后一位终止字符 0 字节）。很快，你将可以看到为什么这种机制会忽略 **buf** 的前三个字符。

（正如 25847 行的注释中所述）我们应该注意到在这里并没有采取严格的措施来保证缓冲器不会过载。这里系统假定 1024 个字符长度的 **buf** 已经足够使用（参阅 25634 行）。如果内核在这里能够使用 **vsnprintf** 函数的话，情况就会好许多。然而，**vsnprintf** 还有另外一个参数限制了它能够写入缓冲器的字符长度。

25849：计算 **buf** 中最近使用的元素，调用 **va_end** 终止对“…”参数的处理。

25851：开始格式化消息的循环。其中存在一个内部循环能够处理更多内容（这一点随后就能看到），因此，每次内循环开始，都开始一个新的打印行。由于通常情况下 **printk** 只用于打印单行，所以在每次调用中这种循环通常只执行一次。

25853：如果预先不知道消息的日志等级，**printk** 会检查当前行是否以日志等级序列开头。

25860：如果不是，**buf** 中开始未使用的三个字符就能够起作用了。（第一次以后的每次循环，都会覆盖部分消息文本，但是这样并不会引起问题，因为这里的文本只是前面行中的一部分，它们已经被打印过，而且以后也不再需要了。）这样，就可以将日志等级插入 **buf** 中。

25866：此处有如下属性：**p** 指向日志等级序列（消息文本紧随其后），**msg** 指向消息文本——请注意 25852 行和 25865 行中对 **msg** 的赋值。

由于已知 **p** 用来指示日志等级序列的开头——该日志等级序列可能是由函数自身所创建的——日志等级可以从 **p** 中抽出并存到 **msg_level** 中。

25868：没有检测到新行，清空 **line_feed** 标志。

25869：这是前面谈到过的内循环，循环将运行到本行结束（也就是检测到新行标志）或者缓冲器的末尾为止。

25870：除了将消息打印到控制台之外，**printk** 还能够记录最近打印的长度为 **LOG_BUF_LEN** 的字符组。（**LOG_BUF_LEN** 为 16K，请参看 25632 行。）如果在控制台打开之前，内核就已经调用 **printk**，则显然不能在控制台上正确打印消息，但是这些消息将被尽可能的存储到 **log_buf** 中（25656 行）。当控制台打开以后，缓存在 **log_buf** 中的数据就可以转储并在控制台上打印出来，请参看 25988 行。

log_buf 是一个循环缓冲器，**log_start** 和 **log_size** 变量（25657 行和 25646 行）分别记录当前缓冲器的开始位置和长度。本行中的按位与（AND）操作实际上是快速求模（%）运算，它的正确性依赖于 **LOG_BUF_LEN** 的值是 2 的幂。

25872：保存变量跟踪记录循环日志的值。显然，日志大小会不断增长，直至达到 **LOG_BUF_LEN** 的值为止。此后，**log_size** 将保持不变，而插入新字符将导致 **log_start** 的增长。

25878：请注意 **logged_chars**（25658 行）记录从机器启动之后 **printk** 写入的所有字符的长度，它在每次循环中都会被更新，而不是在循环结束后才改变一次。基于同样的道理，**log_start** 和 **log_size** 的处理方式也是一样。这实际上是一种优化的时机，但是我们将在结束对函数的介绍之后再对它详细讨论。

25879：消息被分为若干行，这当然要使用新行标志符来进行分割。一旦内核检测到新行标志符，就写入一个完整行，从而内循环的执行也可以提前终止。

25884：在这里我们先不考虑内部循环是否会提前退出，从 **msg** 到 **p** 的字符序列是专门提供给控制台使用的。（这种字符序列我称之为行，但是不要忘了，这里的行可能并不意味着新行终止，因为 **buf** 也许还没有终止。）如果该行的日志等级高于系统控制台定义的日志等级，而且当前又有控制台可供打印，那么就能够正确打印该行。（记住，**printk** 可能在所有控制台打开之前就已经被调用过了。）

如果在该信息块中没有发现日志等级序列，并且在前面的 **printk** 调用中也没有对 **msg_level** 赋值，那么本行中的 **msg_level** 就是-1。由于 **console_leglevel** 总不小于 1（除非 root 通过 sysctl 接口锁定），于是总是可以打印这些行。

25886：本行应该能够被打印。**printk** 通过遍历打开的控制台驱动链表告知每一个控制台驱动去打印当前行。（因为虽然设备驱动在本书的讨论范围之外，但是控制台驱动代码则并不包含在内。）

25888：请注意这里消息文本的开头使用的是 **msg** 而不是 **p**，这样就在没有日志等级序列的情况下写入消息了。然而，日志等级序列已经被存储到 **log_buf** 缓冲器中了。这样就可以使后来访问 **log_buf** 以获取信息日志等级的代码（请参看 25998 行）能够正确执行，不会再产生显示混乱信息序列的现象。

25892：如果内层 **for** 循环发现一新行，那么 **buf** 中的剩余字符（如果有的话）将被认为是新的消息，因此 **msg_level** 会被重置。但是无论怎样，外层循环都会持续到 **buf** 清空为止。

25895：释放在 25845 行获取的控制台锁（console lock）。

25896：唤醒等待被写入控制台日志的所有进程。注意即使没有文本被实际写入任何控制台，这个过程也仍然会发生。这样处理是正确的，因为无论是否要往控制台中写入文本，等待进程实际上都是在等待从 **log_buf** 中读出信息。在 25748 行，进程被转入休眠状态以等待 **log_buf** 的活动。在休眠、唤醒和等待队列中所使用的机制将在下一节中进行讨论。

25897：返回日志中写入的字符长度。

如果对于每个字符的处理工作都能减少一点，那么从 25869 行开始的 **for** 循环就能执行得更快一点。当循环存在时，我们可以通过只在循环退出时将 **logged_chars** 更新一次来稍微提高运行速度。然而我们还可以通过其它努力来提高速度。由于我们可以预知消息的长度，因此 **log_size** 和 **log_start** 可以到最后再增长。让我们来实验一下这样能否提高速度，下面是一段经过理想优化的代码：

```
do {
static int wrapped = 0;
const int x = wrapped
   ? log_start
   : log_size;
const int lim = LOG_BUF_LEN – x;
int n = buf_end – p;
if ( n>= lim)
   n = lim;

memcpy(log_buf + x, p, n);
p += n;

if(log_size < LOG_BUF_LEN)
   log_size += n;
else {
   wrapped = 1;
   log_start += n;
   ;og_start &= LOG_BUF_LEN – 1;
}
} while (p < buf_end);
```

请注意循环通常只需要执行一次，只有在 **log_buf** 末尾写入信息需要折行时才会多次执行。因而 **log_size** 和 **log_buf** 只需要更新一次（或者当写入需要换行时是两次）。

这时速度的确提高了，但是有两个原因使我们并不能这样做。首先，内核可能有自己特有的 **memcpy** 函数，我们必须确保对 **memcpy** 的调用不会再次进入对 printk 的调用。（有一

部分内核移植版定义了自己特有的速度较快的 **memcpy** 函数版本，因此所有的移植都要在这一点上保持一致。）如果 **memecpy** 调用 **printk** 来报告失败，那么就有可能触发无限循环。

然而在这一点上也并不是真的无药可救。使用这种解决方案的最大问题在于该内核循环的形式中也要留意新行标志符，因此使用 **memcpy** 将整个消息拷贝到 **log_buf** 中是不正确的：如果此处存在新行，我们将无法对其进行处理。

我们可以试验一个一箭双雕的办法。下面这种替代的尝试虽然可能比前面那种初步解决方法速度要慢，但是它保持了内核版本的语意：

```
/* in declaration section:*/
int n;
char *start;
static char *log = log_buf;
/*……*/

for (start = p;p < buf_end;p++) {
   *log++ = *p;
   if    (log >= (log_buf + LOG_BUF_LEN))
      log = log_buf ;    /* warp*/

   if (*p == '/n') {
      line_feed = 1;
      break;
   }
}

/* p - start is number of chars copied. */
n = p – start;
logged_chars += n ;
/*
*exercise for the reader:
*also use n to update log size and log_ start.
*(it's not as simple as may look.)
*/
```

（请注意 gcc 的优化器十分灵敏，它足以能检测到循环内部的表达式 **log_buf+LOG_BUF_LEN** 并没有改变，因此在上面的循环中试图手工加速计算是没有任何效果的。）

不幸的是，这种方法并不能比现在内核版本在速度上快许多，而且那样会使得代码晦涩难懂（如果你编写过更新 **log_size** 和 **log_start** 的代码，你就能清楚地了解这一点）。你可以自己决定这种折衷是否值得。然而无论怎样，我们学到了一些东西，这是通常的成果：不管成功与否，改进内核代码都可以加深你对内核工作原理的理解。

等待队列

前一节我们曾简要的提到进程（也就是正在运行的程序）可以转入休眠状态以等待某个特定事件，当该事件发生时这些进程能够被再次唤醒。内核实现这一功能的技术要点是把等

待队列（wait queue）和每一个事件联系起来。需要等待事件的进程在转入休眠状态后插入到队列中。当事件发生之后，内核遍历相应队列，唤醒休眠的任务让它投入运行状态。任务负责将自己从等待队列中清除。

等待队列的功能强大得令人吃惊，它们被广泛应用于整个内核中。更重要的是，实现等待队列的代码量并不大。

wait_queue 结构

18662：简单的数据结构就是等待队列节点，它包含两个元素：

- **tast**——指向 **struct task_struct** 结构的指针，它代表一个进程。从 16325 行开始的 **struct task_struct** 结构将在第 7 章中进行介绍。
- **next**——指向队列中下一节点的指针。因而，等待队列实际上是一个单链表。

通常，我们用指向等待队列队首的指针来表示等待队列。作为一个例子，请参看 **printk** 使用的等待队列 **log_wait**（25647 行）。

wait_event

16840：通过使用这个宏，内核代码能够使当前执行的进程在等待队列 **wq** 中等待直至给定 **condition**（可能是任何的表达式）得到满足。

16842：如果条件已经为真，当前进程显然也就无需等待了。

16844：否则，进程必须等待给定条件转变为真。这可以通过调用 **__wait_event** 来实现（16824 行），我们将在下一节介绍它。由于 **__wait_event** 已经同 **wait_event** 分离，已知条件为假的部分内核代码可以直接调用 **__wait_queue**，而不用通过宏来进行冗余的（特别是在这些情况下）测试，实际上也没有代码会真正这样处理。更为重要的是，如果条件已经为真，**wait_event** 会跳过将进程插入等待队列的代码。

注意 wait_event 的主体是用一个比较特殊的结构封闭起来的：

```
do {
   /* … */
} while (0)
```

使我惊奇的是，这个小技巧并没有得到应有的重视。这里的主要思路是使被封闭的代码能够像一个单句一样使用。考虑下面这个宏，该宏的目的是如果 **p** 是一个非空指针，则调用 **free**：

```
#define FREE1(p)    if    (p)    free (p)
```

除非你在如下所述的情况下使用 **FREE1**，否则所有调用都是正确有效的：

```
if    (expression)
      FREE1(p)
else
      printf(“expression was false.\n”) ;
```

FREE1 经扩展以后，**else** 就和错误的 **if**——**FREE1** 的 **if**——联系在一起。

我曾经发现有些程序员通过如下途径解决这种问题：

```
#define FREE2(p)    if (p) { free(p); }
#define FREE3(p)    { if (p) { free(p); } }
```

这两种方法都不尽人意，程序员在调用宏以后自然而然使用的分号会把扩展信息弄乱。以 **FREE2** 为例，在宏展开之后，为了使编译器能更准确的识别，我们还需要进行一定的缩进调节，最终代码如下所示：

```
if (expression)
   if (p) { free(p);}
else
   printf(“expression was false./n”);
```

这样就会引起语法错误——**else** 和任何一个 **if** 都不匹配。**FREE3** 从本质上讲也存在同样的问题。而且在研究问题产生原因的同时，你也能够明白为什么宏体里是否包含 **if** 是无关紧要的。不管宏体内部内容如何，只要你使用一组括号来指定宏体，你就会碰到相同的问题。

这里是我们能够引入 **do/while(0)**技巧的地方。现在我们可以编写 **FREE4**，它能够克服前面所出现的所有问题。

```
#define FREE4(P)    \
do {                    \
      if (p)            \
         free(p);       \
      while (0)
```

将 **FREE4** 和其它宏一样插入相同代码之后，宏展开后其代码如下所示（为清晰起见，我们再次调整了缩进格式）：

```
if (expression)
   do {
      if (p)
         free(p);
   } while (0);      /* “;” following macro.*/
```

这段代码当然可以正确执行。编译器能够优化这个伪循环，舍弃循环控制，因此执行代码并没有速度的损失，我们也从而得到了能够实现理想功能的宏。

虽然这是一个可以接受的解决方案，但是我们不能不提到的是编写函数要比编写宏好得多。不过如果你不能提供函数调用所需的开销，那么就需要使用内联函数。这种情况虽然在内核中经常出现，但是在其它地方就要少得多。（无可否认，当使用 C++，gcc 或者任何实现了将要出现的修正版 ISO 标准 C 的编译器时，这种方案只是一种选择，就是最后为 C 增加内联函数。）

__wait_event

16842：**__wait_event** 使当前进程在等待队列 **wq** 中等待直至 **condition** 为真。

16829：通过调用 **add_wait_queue**（16791 行），局部变量**__wait** 可以被链接到队列上。注意**__wait** 是在堆栈中而不是在内核堆中分配空间，这是内核中常用的一种技巧。在宏运行结束之前，**__wait** 就已经被从等待队列中移走了，因此等待队列中指向它的指针总是有效的。

16830：重复分配 CPU 给另一个进程直至条件满足，这一点将在下面几节中讨论。

16831：进程被置为 **TASK_UNINTERRUPTIBLE** 状态（16190 行）。这意味着进程处于休眠状态，不应被唤醒，即使是信号量也不能打断该进程的休眠。信号量在第 6 章中介绍，而进程状态则在第 7 章中介绍。

16832：如果条件已经满足，则可以退出循环。

请注意如果在第一次循环时条件就已经满足，那么前面一行的赋值就浪费了（因为在循环结束之后进程状态会立刻被再次赋值）。**__wait_event** 假定宏开始执行时条件还没有得到满足。而且，这种对进程状态变量 **state** 的延迟赋值也并没有什么害处。在某些特殊情况下，这种方法还十分有益。例如当**__wait_event** 开始执行时条件为假，但是在执行到 16832 行时就为真了。这种变化只有在为有关进程状态的代码计算 **condition** 变量值时才会出现问题。但是在代码中这种情况我一处也没有发现。

16834：调用 **schedule**（26686 行，在第 7 章中讨论）将 CPU 转移给另一个进程。直到进程再次获得 CPU 时，对 **schedule** 的调用才会返回。这种情况只有当等待队列中的进程被唤醒时才会发生。

16836：进程已经退出了，因此条件必定已经得到了满足。进程重置 **TASK_RUNNING** 的状态（16188 行），使其适合 CPU 运行。

16837：通过调用 **remove_wait_queue**（16814 行）将进程从等待队列中移去。

wait_event_interruptible 和**__wait_event_interruptible**（分别参见 16868 行和 16847）基本上与 **wait_event** 和**__wait_event** 相同，但不同的是它们允许休眠的进程可以被信号量中断。如前所述，信号量将在第 6 章中介绍。

请注意 **wait_event** 是被如下结构所包含的。

```
({
  /* … */
})
```

和 **do/while(0)**技巧一样，这样可以使被封闭起来的代码能够像一个单元一样运行。这样的封闭代码就是一个独立的表达式，而不是一个独立的语句。也就是说，它可以求值以供其它更复杂的表达式使用。发生这种情况的原因主要在于一些不可移植的 gcc 特有代码的存在。通过使用这类技巧，一个程序块中的最后一个表达式的值将定义为整个程序块的最终值。当在表达式中使用 **wait_event_interruptible** 时，执行宏体后赋**__ret** 的值为宏体的值（参看 16873 行）。对于有 Lisp 背景知识的程序员来说，这是个很常见的概念。但是如果你仅仅了解一点 C 和其它一些相关的过程性程序设计语言，那么你可能就会觉得比较奇怪。

__wake_up

26829：该函数用来唤醒等待队列中正在休眠的进程。它由 **wake_up** 和 **wake_up_interruptible** 调用（请分别参看 16612 行和 16614 行）。这些宏提供 **mode** 参数，只有状态满足 **mode** 所包含的状态之一的进程才可能被唤醒。

26833：正如将在第 10 章中详细讨论的那样，锁（lock）是用来限制对资源的访问，这在 SMP 逻辑单元中尤其重要，因为在这种情况下当一个 CPU 在修改某数据结构时，另一个 CPU 可能正在从该数据结构中读取数据，或者也有可能两个 CPU 同时对同一个数据结构进行修改，等等。在这种情况下，受保护的资源显然是等待队列。非常有趣的是所有的等待队列都使用同一个锁来保护。虽然这种方法要比为每一个等待队列定义一个新锁简单得多，但是这就意味着 SMP 逻辑单元可能经常会发现自己正在等待一个实际上并不必须的锁。

26838：本段代码遍历非空队列，为队列中正确状态的每一个进程调用 **wake_up_process**（26356 行）。如前所述，进程（队列节点）在此可能并没有从队列中移走。这在很大程度上是由于即使队列中的进程正在被唤醒，它仍然可能希望继续存在于等待队列中，这一点正如我们在**__wait_event** 中发现的问题一样。

内核模块（Kernel Modules）

整个内核并不需要同时装入内存。应该确认，为保证系统能够正常运行，一些特定的内核必须总是驻留在内存中，例如，进程调度代码就必须常驻内存。但是内核其它部分，例如大部分的设备驱动就应该仅在内核需要的时候才装载，而在其它情况下则无需占用内存。

举例来说，只有在内核真正和 CD-ROM 通讯时才需要使用完成内核与 CD-ROM 通讯的设备驱动程序，因此内核可以被设置为在和设备通讯之前才装载相应代码。内核完成和设备的通讯之后可以将这部分代码丢弃。也就是说，一旦代码不再需要，就可以从内存中移走。系统运行过程中可以增减的这部分内核称为内核模块。

内核模块的优点是可以简化内核自身的开发。假设你购买了一个新的高速 CD-ROM 驱动器，但是现有的 CD-ROM 驱动程序并不支持该设备。你自然就希望增加对这种高速模式的支持以提高系统光驱设备的性能。如果作为内核模块来编译驱动程序，你的工作将会方便得多：编译驱动程序，加载到内核，测试，卸载驱动程序，修改驱动程序，再次加载驱动程序到内核，测试，如此周而复始。如果你的驱动程序是直接编辑在内核中的，那么你就必须重新编译整个内核并且在每次修改驱动程序之后重新启动机器。这样慢得很多。

自然，你也必须留意内核模块。对于指明其它内核模块在磁盘上的驻留位置的那些模块，一定不能从内存中卸载，否则，内核将只能通过访问磁盘来装载处理磁盘访问的内核模块，这是不可能实现的。这也是我们要选择把部分内核作为模块编译还是直接编译进内核使其常驻内存的又一个原因。你知道自己系统的设置方式，因而也就可以自己选择正确使用的方式。（如果为了确保安全，你可以简单的忽略内核模块系统的优点，而把所有的内容都编译到内核里面。）

内核模块会带来一些速度上的损失，这是因为一些必需的代码现在并不在 RAM 中，必需要从磁盘读入。但是整个系统的性能通常会有所提高，这主要是因为通过丢弃暂时不使用的模块可以释放出额外的 RAM 供应用程序使用。如果这部分内存被内核所占用，应用程序将只能更加频繁地进行磁盘交换（swap），而这种磁盘交换会显著的降低应用程序的性能。（磁盘交换将在第 8 章中讨论。）

内核模块还会带来因复杂度的增加所造成的开销，这是因为在系统运行的过程中移进移出部分内核需要额外的代码。然而，正如你将在本节中看到的，复杂度的开销是可以管理的。通过使用外部程序来代理一些必需的工作还可以更进一步降低复杂度的开销。（更为确切的说法是，这样做不是减少了复杂度的开销，而是把复杂度的开销重新分配了一下。）这是对内核模块原理的一个小小的扩展：即使是内核的支持模块对于内核来说也只是外部的，部分可用的，只有在需要的时候才被装入内存。

通常用于这种目的程序称为 modprobe。有关的 modprobe 代码超出了本书的范围，但是在 Linux 的每个发行版本中都有包含有它。本节的剩余部分将讨论同 modprobe 协同工作以装载内核模块的内核代码。

request_module

24432：作为函数说明之前的注释，**request_module** 是一个函数。内核的其它模块在需要装载其它内核模块的时候，都必须调用这个函数。就像内核处理其它工作一样，这种调用也是为当前运行的进程进行的。从进程的角度来看，这种调用的请求通常是隐含的——正在执行进程其它请求的内核可能会发现必须调入一个模块才能够完成该请求。例如，请参看 10070 行，这里是一些将在第 7 章中讨论的代码。

24446：以内核中的一个独立进程的形式执行 **exec_modprobe** 函数（24384 行，马上就会讨论到）。这并不能只通过函数的简单调用实现，因为 **exec_modprobe** 要继续调用 **exec**

来执行一个程序。因此，对函数 **exec_modprobe** 的简单调用将永远不会有返回。这和使用 **fork** 以准备 **exec** 调用十分类似，你可以认为 **kernel_thread** 对内核来说就是较低版本的 **fork**，虽然两者有很大不同。**fork** 是从指定函数开始执行新的进程，而不是从调用者的当前位置开始运行。正如 **fork** 一样，**kernel_thread** 返回的值是新进程的进程号。

24448：和 **fork** 一样，从 **kernel_thread** 返回的负值表示内部错误。

24455：正如函数中论述的一样，大部分的信号量将因当前进程而被暂时阻塞。

24462：等待 **exec_modprobe** 执行完毕，同时指出所需要的模块是已经成功装入内存还是装载失败了。

24465：结束运行，恢复信号量。如果 **exec_modpro** 返回错误代码，则打印错误消息。

exec_modprobe

24384：**exec_modprobe** 运行为内核增加内核模块的程序。这里的模块名是一个 **void***的指针，而不是 **char***的指针。原因简单说来就是 **kernel_thread** 产生的函数通常都使用 **void***指针参数。

24386：设置 modprobe 的参数列表和环境。**Modprobe_path**（24363 行）用来定位 modprobe 程序的位置。它可以通过内核的 sysctl 特性来修改，这一点将在第 11 章中介绍（请参看 30388 行）。这意味着 root 可以动态选择不同于/sbin/modprobe 的程序来运行，以适应当 modprobe 被安装到其它地方或者使用修改过的 modprobe 替换掉了原有的 modprobe 之类的情况。

24400：（正如代码中描述的一样）出于安全性考虑，丢弃所有挂起的信号量和信号量句柄（handlers）。这里最重要的部分是对 **flush_signal_handlers** 的调用（28041 行），它使用内核默认的信号量句柄代替所有用户定义的信号量句柄。如果在此时有信号量被传送到内核，它将获得默认响应——通常是忽略信号量或杀死进程。但是不管怎样都不会引起安全风险。由于该函数从触发它的进程中分离出来（如前所述），所以不管原始进程在此处是否改变其原来分配的信号量句柄都不会产生任何影响。

24405：关闭调用进程打开的所有文件。最重要的是，这意味着 modprobe 程序不再从调用进程中继承标准输入输出和标准错误。这很有可能会引起安全漏洞。（这可能在替代 modprobe 的程序中引起的问题，但是 modprobe 本身实际上并不关心这个差异。）

24413：modprobe 程序作为 root 运行，它拥有 root 所拥有的所有权限。和整个内核中其它地方一样，请注意 root 使用用户 ID 号 0 的假定在这里已经被写入程序。用户 ID 号和权能系统（capability system）（在接下来的几行中会用到）将在第 7 章中介绍。

24421：试图执行 modprobe 程序。如果尝试失败，内核将使用 **printk** 打印错误消息并返回错误代码。这里是可能产生 **printk** 的缓冲器过载的地点之一。**module_name** 的长度并没有明确限制，就我们对该调用的看法而言，它可能长达一百万个字符。为防止 **printk** 缓冲器过载，你必需遍历所有对于该函数的调用（实际上是对 **request_module** 的调用）以保证每个调用者使用足够短的不会为 printk 造成麻烦的模块名。

24427：当 **execve** 成功执行时，它不会返回任何结果，因此本处是不可能执行到的。但是编译器却并不知道这一点，因此此处使用了 **return** 语句以保证 gcc 不出错。

对于内核的进一步讨论将超出本章的既定范围，因此在这个问题上我们到此为止。然而本书中也包括了其它必需的内核代码。在读完第 4 章和第 5 章之后，也许你会希望再次仔细研读一下这部分内容。有关这个问题的两个文件是 include/linux/module.h（从 15529 行开始）和/kernel/module.c（从 24476 行开始）。和 **sys_create_module**（24586 行），**sys_init_module**（24637 行），**sys_delete_module**（24860 行）和 **sys_query_module**（25148 行）四个函数需要特别注意一样，**struct module**（15581 行）也要特别引起注意。这些函数实现了 modprobe

以及 insmod，lsmod 和 rmmod 所使用的系统调用以完成模块的装载、定位和卸载。

内核触发直接回调内核程序的现象看起来很令人奇怪。但是，实际上进行的工作不止于此。例如，modprobe 必须实际访问磁盘以搜寻要装载的模块。而且更为重要的一点是这种方法赋予 root 对内核模块系统更多的控制能力。这主要是因为 root 也可以运行 modprobe 以及相关程序。因此，root 既可以手工装载、查询、卸载模块，也可以由内核自动完成。

配置与编译内核

你可能仅仅研读、欣赏而并不修改 Linux 内核源代码。但是，更普遍的情况是，用户有强烈的愿望去改进内核代码并完成相应的测试，这样我们就需要知道如何重建内核。本节就是要告诉你如何实现这一点，而最终则归结于如何把你所做的修改发行给别人，以使得每个人都能从你的工作中受益。

配置内核

编译内核的第一步就是配置内核，这是增加或者减少对内核特性的支持以及修改内核的一些特性发挥作用的方式的必要步骤。例如，你可以要求内核为自己的声卡指定一个不同的 DMA 通道。如果内核配置和你的需要相同，那么你可以直接跳过本节，否则请继续阅读以下内容。

为了完成内核的配置，请先切换到 root 用户，然后转入如下内核源程序目录：

```
cd /usr/src/linux
```

接着敲入如下命令组：

```
make config
make menuconfig
make xconfig
```

这三条命令都可以让你来配置内核，但它们发挥作用的方式各不相同：

- **make config**——三种方法中最简单也是最枯燥的一种。但是最基本的一点是，它可以适应任何情况。这种方法通过为每一个内核支持的特性向用户提问的方式来决定在内核中需要包含哪些特性。对于大多数问题，你只要回答 y（yes，把该特性编译进内核中），m（作为模块编译）或者 n（no，根本不对该特性提供支持）。在决定之前用户应该考虑清楚，因为这个过程是不可逆的。如果你在该过程中犯了错误，就只能按 Ctrl+C 退出。你也可以敲入?以获取帮助。图 2.1 显示了这种方法正在 X 终端上运行的情况。

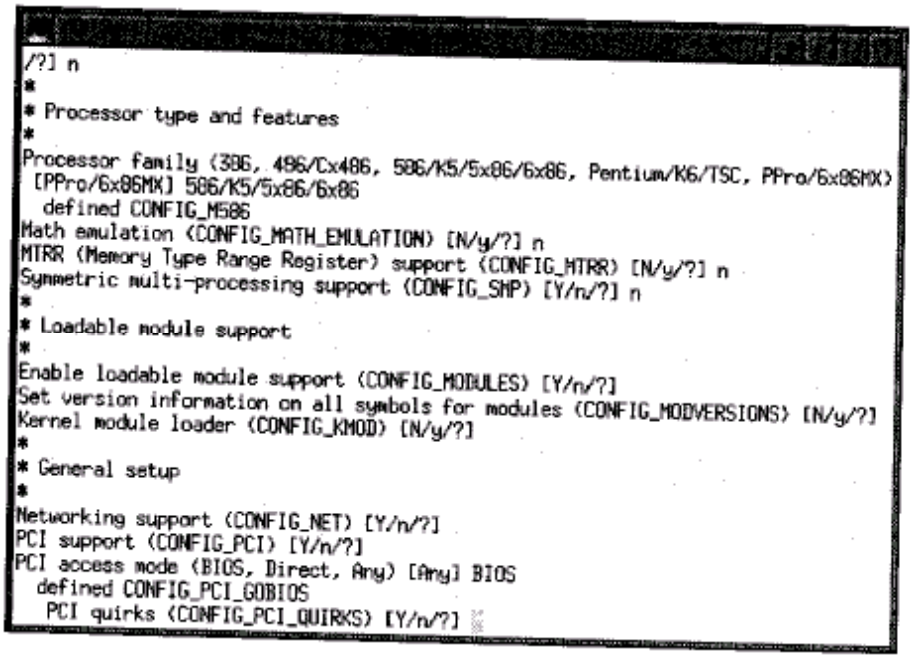

图 2.1 运行中的 **make config**

幸运的是，这种方法还有一些智能。例如，如果你对 SCSI 支持回答 no，那么系统就不会再询问你有关 SCSI 的细节问题了。而且你可以只按回车键以接受缺省的选择，也就是当前的设置（因此，如果当前内核将对于 SCSI 的支持编译进了内核，在这个问题上按回车键就意味着继续把对 SCSI 的支持编译进内核中）。即使是这样，大部分用户还是宁愿使用另外的两种方法。

- **make menuconfig**——一种基于终端的配置机制，用户拥有通过移动光标来进行浏览等功能。图 2.2 显示了在 X 终端上运行的 **make menuconfig**。虽然在控制台上显示的是彩色，但是在终端上的显示仍然相当单调。使用 menuconfig 必须要有相应的 ncurses 类库。

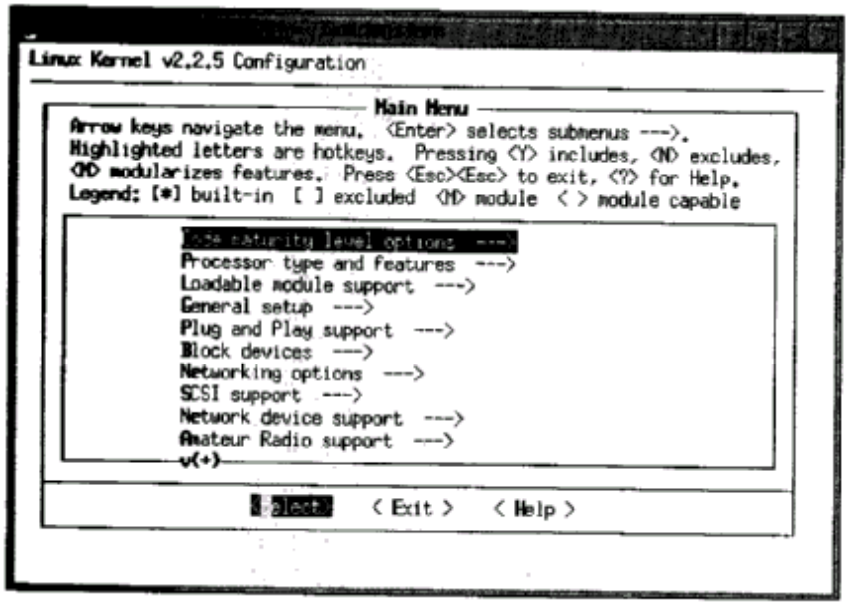

图 2.2 运行中的 **make menuconfig**

- **make xconfig**——这是我最喜欢的一种配置方式。只有你能够在 X server 上用 root 用户身份运行 X 应用程序时，这种配置方式才可以使用（有些偏执的用户就不愿意使用这种方式）。你还必须拥有 Tcl 窗口系统（Tcl windowing system），这实际上还意味着你必须拥有 Tcl，Tk 以及一个正在运行的 X 安装程序。作为补偿，用户获得的是更漂亮的，基于 X 系统的以及和 menuconfig 功能相同的配置方法。图 2.3 显示这种方法运行过程中打开“ 可装载模块支持（Loadable module support）”子窗口的情况。


t_5.JPG

图 2.3 运行中的 **make xconfig**

如上所述，这三种方法都实现了相同的功能：它们都生成在构建内核时使用的.config 文件。而唯一的区别是在于创建这个文件时的难易程度不同。

构建内核

构建内核要做的工作要比配置内核所做的工作少得多。虽然有几种方式都能实现这一功能，但是选择哪一种依赖于你希望怎样对系统进行设置。长期以来，我已经形成了如下的习惯。虽然这种习惯比我所必须要做的略微多一些，但是它包含了所有基本的问题。首先，如果你还不在内核源程序目录中，请先再次转入这一目录：

```
cd  /usr/src/linux
```

现在，切换到 root 用户，使用下面显示的命令生成内核。现在在 shell 中敲入下面的命令，注意 make 命令因为空间关系分成了两行，但实际上这在 shell 输入时是一个只有一行的命令：

```
make dep clean zlilo boot
     modules modules_install
```

当给出了如上多个目标时，除非前面所有的目标都成功了，否则 make 能够知道没有必要继续尝试下面的目标。因此，如果 make 能够运行结束，成功退出，那么这就意味着所有的目标都正确构建了。现在你可以重新启动机器以运行新的内核。

备份的重要性

当修改（fooling）内核时，你必须准备一个能够启动的备用内核。实现该目的的一种方式是通过配置 Linux 加载程序（LILO）以允许用户选择启动的内核映象，其中之一是从没有修改过的内核的备份（我总是这样做的）。

如果你比较有耐心，那么你就可以使用 zdisk 目标而不使用 zlilo 目标；它可以把能够启动的内核映象写入软盘中。这样你就可以通过在启动时插入软盘的方式启动你的测试内核；如果没有插入软盘，则启动正常的内核。

但是请注意：内核模块并没有被装载到软盘中，它们实际上是装在硬盘中的（除非你愿意承担更多的麻烦）。因此，如果你弄乱了内核模块，即使是 zdisk 目标也救不了你。实际上，上面提到的这两种方法都存在这个问题。虽然有比较好的解决方法可用，但是最简单的方法（也就是我所使用的方法）是把备份内核作为严格独立的内核来编译，而不使用可装载模块的支持。通过这种方法，即使我弄乱了内核而不得不使用备份启动系统，那么不管问题是实验性内核不正确还是内核模块的原因都无关紧要。不管怎样，在备份的内核中已经有我

需要的所有东西了。

由于用户所作的修改可能导致系统的崩溃，如损坏磁盘上的数据等等，并不仅仅只是打乱设备驱动程序或文件系统，在测试新内核之前，备份系统的最新数据也是一个英明的决策。（虽然设备驱动程序的开发不是本书的主题，但是必需指出的是，设备驱动程序的缺陷可能会引起系统的物理损坏。例如显示器是不能备份的，而且因价格昂贵而不易替换。）作为一个潜在的内核黑客，你的最佳投资（当然是读过本书以后）是一个磁带驱动器和充足的磁带。

发行你的改进

下面是有关发行你所做修改的一些基本规则：

- 检查最新发行版本，确保你所处理的不是已经解决了的问题
- 遵守 Linux 内核代码编写的风格。简要的说就是 8 字符缩进以及 K&R 括号风格（**if**，**else**，**for**，**while**，**switch** 或者 **do** 后面同一行中紧跟着开括号）。在内核源程序目录下面的文档编写和代码风格文件给出了完整的规则，不过我们已经介绍了其中的关键部分。注意本书中包含的源程序代码为节省空间而进行了大量的重新编辑，在该过程中我可能打破了其中的一些规则。
- 独立发行相对无关的修改。这样，只想使用你所做的某部分修改的人就可以十分方便地获得想要的东西，而不用一次检验所有的修改内容。
- 让使用你所做修改的用户清楚他们可以从你的修改中获取什么。同样地，你也应该给出这些问题的可信度。你是 15 分钟之前才匆匆完成你的修改，甚至还没有时间对它们进行编译，还是已经在你和你的朋友的系统中从去年 3 月开始就长期稳定的运行过这个修改？

假设现在你已经准备好发行自己的修改版本了，那么要做的第一步是建立一个说明你所做的修改的文件。你可以使用 diff 程序自动创建这个文件。结果或者被称为 diffs，也或者在 Linux 中更普遍的被称为补丁（patch）。

发布的过程十分简单。假设原来没有修改过的源程序代码在 linux-2.2.5 目录下，而你修改过的源程序代码在 linux-my 目录下，那么只要进行如下的简单工作就可以了（只有在链接不存在的情况下才需要执行 ln）：

```
ln –s linux-my linux
make –C linux-2.2.5 distclean
make –C linux distclean
diff –urN linux-2.2.5 linux >my.patch
```

现在，输出文件 my.patch 包含了其它用户应用这个修改程序时所必须的一切内容。（警告：如上所述，两个源程序间的所有差别都会包含在这个补丁文件中。Diff 不能区分修改部分之间的关系，所以就把它们都罗列了出来。）如果补丁文件相对较小，你可以使用邮件直接发往内核邮件列表。如果补丁很大，那么就需要通过 FTP 或者 Web 站点发布。这时发给邮件列表的信件中就只需要包含一个 URL。

Linux 内核邮件列表的常见问题解答（FAQ）文件位于 **http://www.ececs.uc.edu/~rreilova/linux/lkmlfaq.html**。该 FAQ 中包含了邮件列表的订阅，邮件发布以及阅读邮件列表的注意事项等等。

顺便提一下，如果你想随时了解内核更新开发的进程，我向你强烈推荐下面这个具有很高价值的内核交流站点 Kernel Traffic：**http://www.kt.opensrc.org**。