第 11 章 可调内核参数

遵循 Unix 的 BSD 4.4 版本所倡导的风格，Linux 提供 **sysctl** 系统调用以便在系统运行过程中对它所拥有的某些特性进行检查和重新配置，它并不需要你编辑内核的源代码、重新编译，然后重启机器。这是对早期 Unix 版本的一个十分重要的改进，在早期版本里调整系统经常是令人头痛的琐碎事务。Linux 把可以被检查和重新配置的系统特性有机地组织成了几个种类：常规内核参数、虚拟内存参数、网络参数，等等。

同样的特性也可以从一个不同的接口进行访问：/proc 文件系统。（因为它真正的是系统的一个透视区（window）而不只是真实文件的一个容器，所以/proc 是一个“ 伪的文件系统 ”，不过那是一个蹩脚的词汇，而且无论如何这个区别在此并不重要。）每种可调内核参数在/proc/sys 下都表现为一个子目录，而每个单独的可调系统参数由某个子目录下的一个文件来代表。这些子目录可能又包含一级子目录，它们仍然含有更多的代表可调系统参数的文件和子目录，等等，但是这种嵌套级数从来都不会很深。

/proc/sys 绕过了通常的 **sysctl** 接口：一个可调内核参数的值可以简单的通过读取相应的文件来得到，通过写入该文件可以设置它的值。普通 Unix 文件系统的许可被应用于这些文件，以便对能够对它们进行读写的用户进行控制。大多数文件对所有用户是可读的但是只对 root（根用户）可写，不过也有例外：比如，/proc/sys/vm 下的文件（虚拟内存参数）只能被 root 来读写。如果不使用/proc/sys，检查和调整系统将需要编写程序并使用必须的参数调用 **sysctl**——虽然不是任务艰巨的劳动，可是也比不上使用/proc/sys 来得方便。

struct ctl_table

18274：这是本章涉及的代码中所使用的一个主要数据结构。**struct ctl_table**s 通常是由数组聚合起来的，每个这样的数组对应于/proc/sys 下某处一个单独目录里的条目。（依我之见，称它为 **struct ctl_table_entry** 可能更好。）**root_table**（30328 行）以及在它之后的数组通过 **struct ctl_table** 的 **child** 指针连结节点而形成了一个数组树（**child** 将在下边的列表中介绍）。注意所有这些都是 **ctl_table** 的数组，它只是为 **struct ctl_table** 进行 **typedef**；18184 行完成这项工作。

图 11.1 示意出了数组树间的关系。这幅图显示了由 **root_table** 形成的树的一小部分以及它所指向的树。

struct ctl_table 具有如下成员：

- **ctl_name**——是唯一标识表项的一个整数——在它所在的数组中是唯一的；这个数字在不同的数组中是可以重用的。数组的任何一项都已经存在这样一个唯一的数字了——就是它的数组下标——可是这个数字不能被用于该目的，因为我们想要维护不同内核发布版本中的二进制兼容性。与某内核版本里一个数组项相关联的可调内核参数可能不会出现在将来的内核版本里，所以假如参数是被它们的数组下标定义的，对数组里废弃项目位置的重新使用将使还没有在新内核版本下编译过的程序变得混乱。随着时间的推移，为了向后兼容而带上的只浪费空间但没有作用的元素项将会使数组变得乱七八糟。相反的，这种方法只会“ 浪费 ”整数，而整数资源却无疑是非常丰富的。（另一方面，查找也会更慢，因为一个简单的数组下标还不足以满足这种方法。）

  要注意的是这与有系统调用的情形相当类似：每个系统调用都与一个在系统调

用表里唯一标识它位置的数字相关联。但是在这种情况里使用了一个不同的解决办法，可能由于速度在此并不重要的缘故。

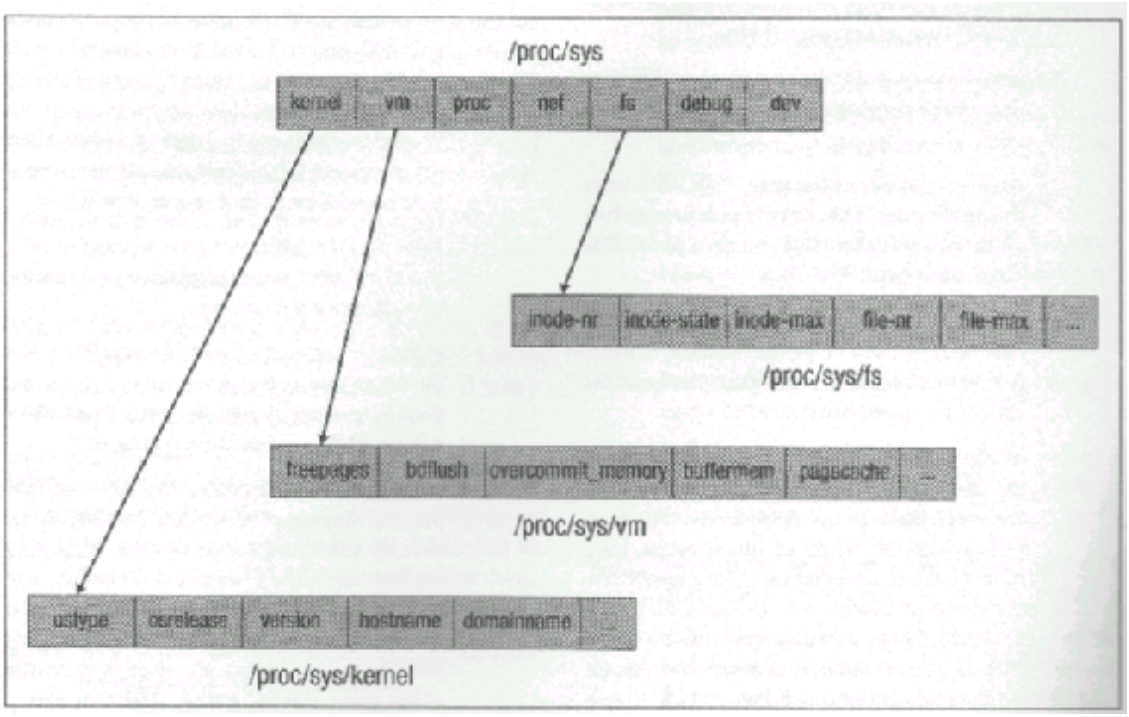

图 11.1 **ctl_table** 树的一部分

struct ctl_table 具有如下成员：

- **ctl_name**——是唯一标识表项的一个整数——在它所在的数组中是唯一的；这个数字在不同的数组中是可以重用的。数组的任何一项都已经存在这样一个唯一的数字了——就是它的数组下标——可是这个数字不能被用于该目的，因为我们想要维护不同内核发布版本中的二进制兼容性。与某内核版本里一个数组项相关联的可调内核参数可能不会出现在将来的内核版本里，所以假如参数是被它们的数组下标定义的，对数组里废弃项目位置的重新使用将使还没有在新内核版本下编译过的程序变得混乱。随着时间的推移，为了向后兼容而带上的只浪费空间但没有作用的元素项将会使数组变得乱七八糟。相反的，这种方法只会“浪费”整数，而整数资源却无疑是非常丰富的。（另一方面，查找也会更慢，因为一个简单的数组下标还不足以满足这种方法。）

  要注意的是这与有系统调用的情形相当类似：每个系统调用都与一个在系统调用表里唯一标识它位置的数字相关联。但是在这种情况里使用了一个不同的解决办法，可能由于速度在此并不重要的缘故。
- **procname**——是用于/proc/sys 下的相应项的一个可供我们阅读的简短文件名。
- **data**——一个指向与此表项关联的数据的指针。它通常指向一个 **int** 或者一个 **char**（当然，指向 **char** 的指针是字符串）。
- **maxlen**——可以读取或者写入 **data** 的最大字节数。如果 **data** 指向一个单精度型的 **int**，举例来说，**maxlen** 就应该是 **sizeof（int）**。
- **mode**——Unix 类型的文件许可位，它对应于这一项的/proc 文件（或目录）。对此的解释需要少量文件系统的内容。就像其它 Unix 的实现一样，Linux 使用三

个三元组，其中每一位都记录一个文件许可（在 **ls -l** 命令产生的列表里它们表现为 r、w，和 x 的三组字母）——参见图 11.2。它们占据 **mode** 的低端 9 位。文件系统把文件的 **mode** 里剩余的位留作它用，比如用来跟踪是否文件是常规文件（第 16 位，当它如此时）、目录（第 15 位）、setuid 或 setgid 执行程序（第 12 和 11 位），等等。不过就本章的目的来说，那些其它位都不是我们所关心的内容。

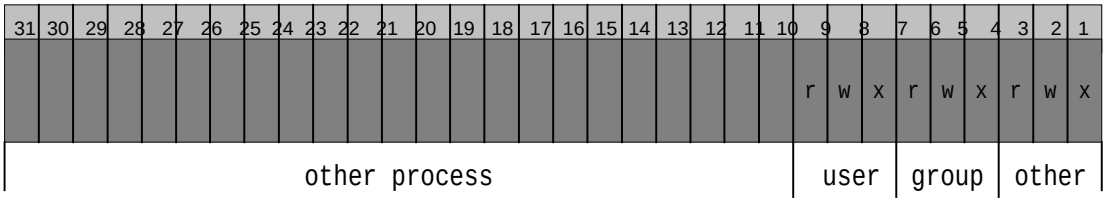

图 11.2 文件的 **mode** 位

这种方式的结果是，读者将经常见到八进制的常数 004、002，和 001 与 **mode** 一起使用——它们分别是在移位 **mode** 后可能得到的适当的三位组中检测读（r）、写（w），和执行（x）位。这种移位和检查工作基本上是在 30544 行的 **test_perm** 里完成的。

注意如果一个表项的 **maxlen** 是 0，那么不管它的 **mode** 是什么，从最终效果上看它都是既不可读也不可写的。

- **child**——如果这是一个目录类型的条目，那么它就是指向子表（child table）的一个指针。在这样的情况下，因为没有数据与此条目相关联，**data** 将是 **NULL**，而 **maxlen** 则将是 0。
- **proc_handler**——指针，指向对 **data** 成员实际进行读取和写入操作的一个函数；它在通过/proc 文件系统读写数据时被使用。以这种方法，任何类型的数据都可以通过 **data** 来进行指向，而且 **proc_handler** 函数会正确的处理对它的工作。**Proc_handler** 通常指向 **proc_dostring** 函数（30820 行）或 **proc_dointvec** 函数（30881 行）；这两个以及其它被普遍适用的函数将在本章后面被讨论。（当然，任何具有适当原型（prototype）的函数都可以使用。）对于目录类型的条目，**proc_handler** 是 **NULL**。
- **strategy**——指针，指向对 **data** 成员实际进行读取和写入操作的另一个函数；它使用在通过 **sysctl** 系统调用进行读写的时候。它通常是 **sysctl_string**（31121 行），不过也可以是 **stringctl_intvec**（31163 行）；这两个函数在本章后面进行讨论。出于种种原因，大多数可调内核参数是通过/proc 接口而不是 **sysctl** 系统调用进行调整的，所以这个指针是 **NULL** 会比非空更为常见。
- **de**——指向 **struct proc_dir_entry** 的一个指针，它在/proc 文件系统代码中使用以追踪文件系统里的文件或目录。如果它非空，**struct ctl_table** 就在/proc 下的某处注册过了。
- **extra** 和 **extra2**——指向在处理这个表元素时所需的任何补充数据。它们当前只用于指定某些整数参数的最小和最大值。

/proc/sys 支持

不是所有实现用于可调内核参数/proc/sys 接口的代码都包括在这本书中——的确，大部分代码并没有包括在内，因为它们基本上属于/proc 文件系统本身。尽管如此，只要你不关心/proc 剩下的部分是如何工作的，就不难理解在 kernel/sysctl.c 里的代码，它们与/proc 文件系统一起工作用来使/proc 下的可调内核参数是可见的。

register_proc_table

30689：**register_proc_table** 函数在/proc/sys 下注册一个 **ctl_table**。注意这里并不要求所提供的表是根一级的节点（即 **ctl_table** 没有双亲）——它本应该是，不过这取决于调用者是否能够进行保证。

这个表被直接建立在 **root** 之下，它应该对应于/proc/sys 或者其下的一个子目录。（在初次调用时，**root** 总是指向 **proc_sys_root** 的，但是在递归调用时它的值改变了。）

30696：开始在 **table** 数组的所有元素中进行循环；在当前元素的 **clt_name** 成员为 0 时循环结束，表示这是数组的末尾。

30698：如果 **ctl_table** 的 **procname** 元素是 **NULL**，那么即使同一数组的其它元素都可以为用户所见，它也不可以在/proc/sys 下被用户所见。这样的数组元素会被跳过。

30701：如果表项有 **procname**，表明它应该在/proc/sys 下被注册，那么它一定还有一个 **proc_handler**（如果是一个叶子，或文件类型的节点）或者一个 **child**（如果是一个目录类型的节点）。如果它同时缺少这两者，那么系统将显示一条警告，而后循环继续进行。

30711：若表项有一个 **proc_handler**，它被标记成常规文件。

30713：否则，正如可从第 30701 行推断的那样，它一定有一个非空的 **child**，这样该条目将被看作是一个目录。注意并没有禁止 **ctl_table** 同时拥有非空 **proc_handler** 和 **child** 这两者——在这种情形下，所有代码将对其一视同仁。

30715：用给定的名字搜索一个存在的子目录，如果找到就让 **de** 指向它，如果没找到则 **de** 为 **NULL**。为什么对文件不做类似的检查比较难于理解——这可能是我没有领会的文件系统的某个细节问题，答案无疑就在那里。

30723：如果指定的子目录已经不存在了，或者假如 **table** 对应于一个文件而不是一个目录，新的文件或者目录就会通过调用 **create_proc_entry**（未包含在本书中）来创建。

30728 :如果表项是一个叶子节点，**register_proc_table** 会告诉文件系统代码使用由 **proc_sys_inode_operations**（30295 行）定义的文件操作。**proc_sys_inode_operations** 只定义了两个操作，读和写（不是搜索、内存映射，或者其它）。这些操作是用 **proc_readsys** 和 **proc_writesys** 函数（30802 和 30808 行）来执行的，在本章的后面章节中将对它们进行介绍。

30731：到了这一行，**de** 就不可能是 **NULL** 了——它或者已经非空或者在第 30723 行被初始化了。

30733：如果增加的条目是目录类型，**register_proc_table** 会被递归调用来增加这一项的所有子孙。这是内核里不多见的一次递归调用。

unregister_proc_table

30739：**unregister_proc_table** 函数删除 **ctl_table** 数组树和/proc 文件系统之间的关联。**ctl_table** 里的条目以及它们下面所有的“子目录”里的条目也将会从/proc/sys 消失。

30743：同第 30396 行一样，这一行开始在给定的表项数组上进行循环。

30744：与/proc/sys 下任意条目都不关联的表项具有一个为 **NULL** 的 **de** 成员；显然这些表项可被忽略。

30748：如果/proc 文件系统认为这是一个目录，但表项是一个叶子（非目录），这两个结构就是不一致的。**unregister_proc_table** 就会显示一条警告并继续循环，而不会移去这一项。

30752：目录被逐层的进行释放——内核中另一次并不多见的递归过程。

30756：在递归调用结束之后，**unregister_proc_table** 检查是否所有子目录和文件都被逐层删除了——如果不是，当前元素就不能被安全的移去，接着要继续循环。

30762：这里就是为什么子目录（以及其中的文件）可能还没有被移去的原因：它们可能当前还正被使用着。如果这个元素正在被使用，循环将继续，这样该元素就不会被移走。

30765：节点通过 **proc_unregister**（本书不进行介绍）从文件系统里被删除，接着用于追踪该节点而分配的内存被释放。

do_rw_proc

30771：**do_rw_proc** 实现 **proc_readsys**（30802 行）和 **proc_writesys**（30806 行）函数的核心部分，这两个函数被/proc 文件系统代码用于对 **ctl_table** 执行读取和写入操作。

30782：确保一个表与/proc/sys 下的这一条目相关联。

30785：注意这一行的第一个测试与第 30782 行的第二个测试是相重复的，这是因为 **table** 是从 **de->data** 初始而来。

30788：确保调用进程有适当的读或写权限。

30795：调用该表项的 **proc_handler** 来完成真正的读操作或写操作。（要注意第 30785 行证实了 **proc_ handler** 成员是非空的。）如前所述，**proc_handler** 成员通常是 **proc_dostring** 或 **proc_ dointvec**（30820 行和 30792 行），在随后的几段中我们将对它们进行讨论。

30799：**do_rw_proc** 返回实际读取或写入的字节数。注意到本地变量 **res** 完全是多余的；它可以被参数 **count** 所替代。

proc_dostring

30820：**proc_dostring** 是供文件系统代码调用以对 C 语言字符串型的内核参数进行读取或写入操作的函数。

注意 **write** 标志表示调用者正在写表元素，不过这主要是涉及从输入缓冲区里进行读取——因此，用来写入的代码是受读控制的。类似的，如果 **write** 未被设置，调用者正从该表项读取，这里主要涉及的是写入给定的缓冲区。

这个函数在第 31085 行还实现了一个存根程序（stub）；这个存根程序在/proc 文件系统被编译出内核时使用。大多数其它函数中的类似存根程序将在这个存根程序之后被介绍。

30835：从输入缓冲区内读取字符直到一个表示结束的 ASCII NUL（0）字节或者发现新的一行，再或者到达了被允许从该输入缓冲区内读出数据的最大值（被 **lenp** 所指定）为止。（为了不引起混淆，牢记 **NULL** 是一个 C 指针常量，而 NUL——只有一个 L——是 ASCII 用于字符数字 0 的术语。）

30842：如果从缓冲区读出的字符数超出了可在表项里存储的限度，该数目会被降低。在循环之前就限制最大输入长度（**lenp**）可能会更高效，因为不管怎样从 **buffer** 里读取

大于 **table->maxlen** 字节的数据是无意义的。实际上，循环可能读出，假设是 1024 字节，然后降低计数到 64，因为表项里只能存储这么多。

30844：该字符串从输入缓冲区里被读出，然后以 NUL 结束。

30847：内核为每个进程所拥有的每个文件维护一个“当前位置”变量；这就是 **struct file** 的 **f_pos** 成员。它是 **tell** 系统调用返回的值并由 **seek** 系统调用进行设置。因此，文件的当前位置是由写入的字节数所推进的。

proc_doutsstring

30871：在获得 **uts_sem** 信号量后（29975 行），**proc_doutsstring** 仅是调用 **proc_dostring**。这个函数被 **kern_table**（30341 行）里的一些条目用来设置 **system_utsname** 结构体的不同部分（20094 行）。

do_proc_dointvec

30881：**proc_dointvec**（30972 行）把它的工作委托给了该函数。**do_proc_dointvec** 读或写一个被 **table** 的 **data** 成员所指向的 **int** 类型数组。要读写的 **int** 类型数目通过 **lenp** 传递；它通常是 1，所以本函数通常只被用于读写单独一个 **int**。

用于 **int** 的值是被 **buffer** 指定的。这些 **int** 是不会被以一个未经加工的 **int** 数组传递的；相反的，它们以 ASCII 文本给出，而这正是用户写入相关/proc 文件的。

30898：在所有要读写的 **int** 中循环。**left** 追踪调用者想要读写 **int** 的剩余数目，而 **vleft** 追踪 **table->data** 里剩余的有效元素数目。在这二者中任何一个到达 0，或它从半途退出时，该循环结束。

注意如果从循环中去掉第 30899 行的 **if** 语句，可以使整个循环的效率稍微提高一些，尽管这样做的结果较难维护。取代的代码如下：

P556—1

这种方式使得并不在循环内改变的 **write** 的值将只需被检查一次，而不必在每次循环重复检查。

30900：向前搜索一个不是空格的字符，它是输入（缓冲区）里下一个数字的开头。

30913：从用户空间把一大块数据复制到本地缓冲区 **buf**，然后以 NUL 结束 **buf**。现在 **buf** 里包含了所有输入缓冲里剩余的 ASCII 文本——或者是它所能容纳的那些文本。

这种方法看起来不很有效率，原因在于它可能读取的超出了它所需要的。然而，因为 **buf** 的容量仅为 20（**TMPBUFLEN**，30885 行），它就不可能读取比它所需多出许多的数据。这里的思想可能是读入稍多一些数据要比检查每个字节以确定是否应该停止读操作所付出的代价要少些。

计划使 **buf** 足够大来包括任何 64 位整数的 ASCII 表示，以便这个函数不仅可以支持 32 位平台还可以支持 64 位平台。的确，它只能满足最大的正 64 位整数，它有 19 个数位（使终结的 NUL 字节是第 20 个字节）。可是要记住这些是有符号的整数，所以最小的 64 位有符号整数，即-9,223,372,032,854,775,808 也应是合法输入。这个数字无法被正确的读取。但是幸运的是，补救方法工作量不大而且也非常明显。

随后读者就能够看到当这个输入出现时代码将如何对其进行处理。

30919：处理打头的减号（-），如果发现一个减号就跳过它并设置一个标志。

30923：确保从 **buffer** 读取的文本（可能是打头的减号之后的部分）至少是以一个数字开始的，这样它才能顺利的转换为一个整数。若没有这次检查，就不可能分辨出第 30925

行调用 **simple_strtoul** 返回的 0 是因为输入就是“ 0 ”还是因为函数无法转换任何文本。

30925：把文本转换为一个整数，用 **conv** 参数换算结果。这个换算步骤对于 **proc_dointvec_jiffies** 这样的函数（31077 行）比较有用，它用乘以常数 **HZ** 的简单手段把它的输入从秒转换为一段时间值（jiffies）。然而一般情况里，这个比例因子是 1——即没有换算。

30927：如果还要从缓冲区读取更多的文本，而且下一个要读的字符不是分割参数的（argument-separating）空格，那么整个参数（argument）就无法装进 **buf**。这样的输入是无效的，所以循环提早结束。（一种可以导致函数处于这种状态的方式就是前边所描述的，输入表示的是最小的有符号 64 位整数。）不过，没有错误代码会被返回，因此调用者可能会错误地认为一切正常。当然这也不完全正确：一个错误代码将在第 31070 行被返回，不过这仅当无效参数是在第一次循环重复中被检测到的时候；如果它在后续的循环里被检测到，错误就不会被注意到。

30929：参数被成功的读取。如果有前导的减号，那么现在就对它进行考虑，其它的本地变量被调整转移到下一个参数上，然后这个参数通过指针 **i** 被存储在表项中。

30936：调用者从表项里读取值——由于无需对 ASCII 文本进行语法分析，这就是一种更为简单的情形。输出是由 tab（制表符）分隔的，所以在除了第一次之外的任何一次循环里都把一个 tab 写入临时缓冲区里（在最后一个参数之后也不用写，只需在参数之间即可）。

30938：接着，当前的整数被 **conv** 因子按比例缩减并打印到临时缓冲区里。这段代码同样会受读者前边已经见到的问题的损害：临时缓冲区 **buf** 的大小可能不足以容纳打印到它里边的全部整数值。在这种情况下，实际问题还会因缓冲区的第一个位置可以是一个 tab 制表符而被恶化。这会使 **buf** 的可用部分减少一个字符，进一步还会降低可被正确处理的输入范围。

在这里过大或过小的整数所造成的结果要比在写入情形里严重的多。在那种情形中，代码只要抛弃一些本应接受的输入即可。而在这儿，**sprintf** 会越过 **buf** 的末尾继续写下去。

然而令人惊讶的是，这正是实际工作中可能发生的。在一次典型的执行过程中将有可能发生如下执行过程：从总体上来说，超过 **buf** 的末尾之后还要写入两个额外的字节（一个是因为它可以写入比预期更长的数字，另一个是 tab 制表符）。在栈里 **p** 通常是紧跟在 **buf** 之后的，所以超出 **buf** 末尾写入的部分将会覆盖 **p**。可是由于 **p** 没有先被重新初始化时它是不会再被使用的，因此暂时覆盖它的值并没有危害。

这是一个看似有趣的事故，但是仅仅通过使 **buf** 稍微大一些就能够成为一个更好的解决方式，这样便于代码为正确的而不是错误的前提（reason）而工作。依照原样，对于 gcc 的代码生成器进行完全合法的很小的修改就能够揭示出潜在的缺陷。

30939：把当前 **int** 的文本型表示复制进输出缓冲区里——或者和它所能容纳的相等的文本——接着更新本地变量使其转移到表项的下一个数组元素。

30949：如果调用者刚才在读取，输出就被新的一行结束。**if** 条件语句也保证循环不会在其第一遍执行而且还有空间来写入新行时就结束。注意输出缓冲区不是用 ASCII NUL 字节（读者可能会这样猜测）来结束的，因为它无需如此：调用者能够利用 **lenp** 被写入新值来减少返回字符串的长度。

30954：如果调用者正向表项里写入数值，则略过从输入缓冲区读取的最后参数之后所有的空格。

30967：更新文件的当前位置和 **lenp**，然后返回 0 表示成功。

proc_dointvec_minmax

30978：**proc_dointvec_minmax** 函数类似于 **do_proc_dointvec**，区别是这个函数还把表项的 **extra1** 和 **extra2** 成员作为可以写入该表项的限制值数组来处理。**extra1** 里的值是最小限度，而 **extra2** 里的值则是最大限度。另一点区别是 **proc_dointvec_minmax** 不使用 **conv** 参数。

因为这两个函数颇为相似，所以这一段里只介绍其不同之处。

31033：最大的区别在于：当写入时，超过被 **min** 和 **max**（在 **extra1** 和 **extra2** 数组上循环得到）所定义的范围之外的值将悄无声息的被略过。这段代码的目的明显是要使 **min** 和 **max** 伴随着 **val** 一起继续。当一个数值从输入缓冲里被读取时，它应该被下一个 **min** 和 **max** 来检查，然后才能决定被接受或被忽略。可是，这并非是实际所发生的那样。

假设从 **buffer** 而来的当前值已经进行了语法分析并存入里 **val**，它小于最小值；为了更具体一些，再假设已是第三遍循环，以便 **min** 和 **max** 分别指向对应数组中的第三个元素。然后 **val** 将用 **min** 来检查并发现它超出了范围（太小），接着循环还要继续。可是 **min** 会作为检查的副作用被更新，而 **max** 则没有。现在，**min** 指向它对应数组的第四个元素了，可是 **max** 仍然指向它的数组的第三个元素。这两者不再同步，而且它们还将保存这种状态，这样在下一个从 **buffer** 里读取的值被检验时采用的就是错误的界限。下列代码是最简单的一种修补程序：

P558—1

正如读者将要在本章后边看到的，现在的 Linux 源代码永远不会暴露出这个缺陷。（未来发行的版本情况将有所不同，尽管还未曾明确写出。）

sysctl 系统调用

用于可调内核参数的另一个接口是 **sysctl** 系统调用，以及相关函数。我不很喜欢这个接口。为什么不呢？对于大部分实际工作目的来说，使用 **sysctl**——不过这种方法比修改源代码的旧方法来调整内核能够获得更大的性能提高——只会比访问/proc 文件更为笨拙。通过 **sysctl** 来进行读写需要 C 程序（或相似的东西），而/proc 却很容易通过外壳（shell）命令（或等价的通过命令解释程序脚本）来进行访问。

另一方面，如果你正在 C 环境下工作，调用 **sysctl** 就比打开文件、读取并/或写入，以及再关闭它要方便的多，所以 **sysctl** 在今后也有它的用武之地。与此同时，还是让我们来看一看它的实现吧。

do_sysctl

30471：**do_sysctl** 实现 **sys_sysctl**（30504 行），即 **sysctl** 系统调用的主要内容。注意 **sys_sysctl** 还在第 31275 行出现过——那个版本只是在 **sysctl** 系统调用被编译出内核时所使用的一个简单的存根程序（stub）函数。

如果 **oldval** 非空就用 **oldval** 返回内核参数原有的值，而它的新值在 **newval** 非空时从 **newval** 来进行设置。**oldlenp** 和 **newlen** 分别标识出有多少字节应被写入 **oldval** 和从 **newval** 读出，这是在相应的指针不是 **NULL** 的时候；当指针为 **NULL** 的时候，它们将被忽略。

要注意这里的不对称性：函数对旧值的长度使用指针，而对新的长度不使用指针。这是因为旧的长度既是输入参数也是输出参数；它的输入值是可以通过 **oldval** 返回

的最大字节数，而它的输出值是实际返回的字节数。与之相反，新的长度只是一个输入参数。

30482：如果调用者需要旧的内核参数值，从 **oldlenp** 来对 **old_len** 进行设置。

30490：开始遍历表树的循环列表。（参见本章随后对 **register_sysctl_table** 的讨论。）

30493：使用 **parse_table**（30560 行，在下一段里讨论）来定位可调内核参数，然后读和/或写它的值。

30495：如果 **parse_table** 分配了所有环境信息，它就被释放。很难准确地说出这个环境信息表示着什么。它不被本书所讨论的任何代码使用——据我所知，它目前甚至没有被内核里的任何代码所使用。

30497：**ENOTDIR** 错误表示没有在这一棵表树中找到指定的内核参数——它可能在另一棵还没有查找过的表树中。否则，**error** 将为某个其它的错误代码，或者是代表成功的 0；无论如何，函数应该返回了。

30499：用 **DLIST_NEXT** 宏（本书对此不做介绍）来增加循环控制变量的值（loop iterator）。

30501：返回 **ENOTDIR** 错误，报告出指定的内核参数在任何一个表里都未找到。

parse_table

30560：**parse_table** 用于在表树里查找一个条目，其方法类同于在一个目录树里解析出一个完全合格的文件名的方法。其思想如下：沿着一个 **int** 数组（数组 **name**）进行查找，并在一个 **ctl_table** 数组里搜索每个 **int**。当找到一个匹配项时，它对应的子孙表就被递归查阅（如果匹配项是目录类型的条目），或者该条目被读和/或写（如果它是文件类型的条目）。

30566：多少有些令人惊讶的是，这一行就开始了对整型数组 **name** 内所有元素的循环。习惯上的方法原本是把从这一行到第 30597 行所有代码用一个 **for** 循环包括起来，它的开始是这样的：

```
for ( ; nlen ; ++name , --nlen , table = table -> child )
```

（这个循环还需要删除第 30567 和 30568 行代码，并用一个语句来替代从 30587 直到 30590 行的代码。）推测起来，可能是实际使用的版本可以生成更好的目标代码吧。

30570：开始循环所有的表项，查找与当前 **name** 匹配的一项；当表已被遍历结束（**table->ctl_name** 为 0 了）或者指定的表项已被找到并处理时本循环结束。

30572：把 **name** 数组的当前项读入 **n** 里，以便它可以与当前表项的 **ctl_name** 进行检查。因为 **name** 在内层循环中没有变化，这个读取操作可以放在循环外边（也就是移至 30569 行）以提高一点速度。

30574：核查是否当前 **ctl_table** 的名字与被找到的名字相匹配，或是否它有特殊的“通配符（wildcard）”值，即 **CTL_ANY**（17761 行）。后者的使用目的还不清楚，因为现在并没有内核源代码的任何一处使用过 **CTL_ANY**。它可能用于将来的方案中——我也不认为它是过去版本的一个遗留物，因为 **CTL_ANY** 在 2.0 内核里也没有被用到，而且整个 **sysctl** 接口也只向后兼容到 2.0 以前的开发树版本。

30576：如果这个表元素有一个孩子，它就是一个“目录”。

30577：遵循标准 Unix 行为，检查目录的 x（可执行）位来判断是否当前进程可被允许对它进行访问。注意到这与文件系统所实现的工作非常类似，虽然这并不是（/proc）文件系统接口。这样可以使这两种接口在施用于可调内核参数时能够得到一致的结果——如果一个用户有通过一种接口来修改某个内核参数的权限而通过另一种却没有该权限，那么将是非常不可思议的。

30579：如果表项有一个策略（strategy）函数，它可能需要覆盖允许该进程进入目录的授权。

这个策略函数将被访问，如果它返回一个非零值，整个查找就被中止。

30587：进入目录。本行有效的继续外层循环，并转移到该名字的下一部分。

30592：这个表节点是一个叶子节点，因此内核参数就被找到了。注意这并不打扰对 **name** 数组是否已到其最后元素的检查（也就是现在 **nlen** 是否为 1），虽然可以证明如果不是那样就会有某类型错误产生。不管哪一种情况，**do_sysctl_strategy**（30603 行）都要负责对当前表元素进行读和/或写操作。

30598：**name** 数组非空，可是它的元素在叶子节点被找到之前均已用完。**parse_table** 就返回 **ENOTDIR** 错误，来表示查找指定节点失败。顺便提及一点，注意前一行里的分号是多余的。

do_sysctl_strategy

30603：**do_sysctl_strategy** 在单独一个 **ctl_table** 里读和/或写数据。计划使用该表元素里的 **strategy** 成员，如果存在的话，来完成读/写工作。如果表元素没有它自己的 **strategy** 例程，某些通用的读/写代码将被替代使用。不过读者将要看到，它并不完全按照计划工作。

30610：如果 **oldval** 非空，调用者将读取旧值，这样 r 位就会在 **op** 里被设置。类似的，如果 **newval** 非空则 w 位被设置。接着，第 30614 行核查许可，如果当前进程缺少所需的授权就返回 **EPERM** 错误。

30617：如果表项有它自己的 **strategy** 例程，这个例程就要处理读/写请求。如果它返回负数——一个错误——这个错误就被传送给调用者。如果返回的是正数，0（成功）就会被传送给调用者。如果是 0，**strategy** 例程就拒绝由它自己来处理请求，取而代之的将是缺省行为。（读者可以设想只返回 0 的 **strategy** 例程，如果它完成一些其它诸如收集被调用次数的统计数据这样的工作，它仍然是有用处的。）

30630：这里是通用读取代码开始的地方。注意 **get_user**（13254 行）的返回值不被检查。（类似的缺陷发生在第 9537 和 31186 行。）

30632：确保不会有多于与该表项的 **maxlen** 成员所指定的数值相等的数据被返回。

30634：通过 **oldval** 从表里复制所要求的数据，再将真正被写的数据总量存储在 **oldlenp** 中。

30642：类似于 **oldlenp**，要确保写入表项的数据不能多于它的 **maxlen** 成员所允许的值。注意如果 **copy_from_user** 在中途的第 30644 行检测到一个错误，**tabel->data** 可能会在仅仅被部分更新的情况下就结束。

30648：返回 0 表示成功。以下三种情况都可以达到这一点：

- 调用者对这个表项既不读也不写。
- 调用者尝试读和/或写这个表项，而且所有步骤都被成功执行。
- 表项没有关联的数据，或者因为它的 **maxlen** 是 0，所以它是只读的。

三种情形中的第一种有点儿奇怪，而最后一种则更令人奇怪。第一种情况有些奇特是因为调用 **sysctl** 却要求它对指定的表项既不读也不写，这并没有多少意义，所以可以正当的把它当作一个错误来处理。尽管如此，它要与其它系统调用的内核实现保持基本一致，那就是把一个无操作请求不看作是一个错误。比如说，在第 8 章中介绍的 **sys_brk**（33155 行）在由调用者指定的新 **brk** 值与旧值相同时并不产生一个错误信号。

第三种情况要比第一种奇怪一些，因为它可能真的反映着一个错误。例如，调用代码尝试写入一个 **maxlen** 是 0 的参数，而且由于系统调用返回成功值而认为该尝试已被完成。看起来事情好像不是这样，因为不管怎样为 0 的 **maxlen** 都会使该条目失效，不过还真的存在一个 **maxlen** 为 0 的表项——参见第 30380 行。最终，一切都归结为

sysctl 是怎样在文档中描述的，但是 man 帮助程序中却对此没有任何记载。我仍然认为 **do_sysctl_strategy** 在这种情况下应该返回一个 **EPERM** 错误。

register_sysctl_table

30651：把一个新的根已经被给出的 **ctl_table** 树插入到其它树所形成的循环链表里。

30655：分配一个 **struct ctl_table_header** 用来管理新树的信息。

30659：把新的首部（跟踪 **ctl_table**s 数组形成的新树）插入到首部组成的链表里。

30666：调用 **register_proc_table**（30689 行，本章前边讨论过）把新的表树注册在/proc/sys 目录下。如果没有内核在没有/proc 文件系统支持的情况下进行编译时，则这一行将被编译到内核以外。

30688：新分配的首部被返回给调用程序，以便调用程序能够在以后通过把该首部转递给 **unregister_sysctl_table**（30672 行）来删除相应的树。

unregister_sysctl_table

30672：如前所述，这个简单函数只是把一个 **ctl_table** 的树从内核里这样的树所组成的循环链表里删除。如果内核是在支持/proc 的情况下编译的，它也用于从/proc 文件系统里删除相应的数据。

回顾一下第 30490 和 30500 行，读者不难发现 **root_table_header**（30256 行）——对应于 **root_table** 的列表节点——是在遍历树的循环链表时被用作头和尾节点的。读者现在能够明白实际上在 **unregister_sysctl_table** 函数里没有什么可以避免 **root_table_header** 被从表头列表里删除——它只是还没有这样做而已。

sysctl_string

31121：**sysctl_string** 是 **ctl_table** 的策略例程之一。回忆一下，策略例程可以从第 30618 行（在 **do_sysctl_strategy** 里）被调用来有选择的覆盖一个表项的缺省读/写代码。（策略例程也可以从第 30580 行被调用，不过该例程却从不会被调用。）

31127：如果该表没有相关数据，或者如果可访问部分的长度是 0，则返回 **ENOTDIR** 错误。这与 **do_sysctl_strategy** 的做法是不一致的，在同样的情况里它返回的是成功。

31138：当前字符串的值被复制到用户空间，然后结果以 NUL 来结束（这意味着比由 **lenp** 指定值多一个字节的数据可能会被复制——依据文档记录，这可能是一个缺陷）。因为当前值已经是 NUL 结束的，这四行代码可以被简化为两行：

```
if ( copy_to_user ( oldval , table -> data , len + 1 ) )
  return –EFAULT ;
```

这种改变的正确性部分上依赖于当写入 **table->data** 时代码剩余部分所遵循的三个特征：

- 代码剩余部分不能把多于 **table->maxlen** 的 **char** 数据复制进 **table->data** 里。（这也使得第 31136 行的测试变得没有必要。即使还需要该测试，那也只用检查>，而不用检查>=了。）
- 然后 **table->data** 以 NUL 来结束，如果必要就复写最后一个拷贝进来的字节，以便包括 NUL 在内的总长度不大于 **table->maxlen**。
- **table->maxlen** 永不发生变化。

因为所有三个特征都有效，所以在第 31138 行 **len** 将总是严格小于 **table->maxlen**，而且结束 NUL 字节一定会在 **table->data[len+1]**或之前的位置出现。

31146：与前边的情况类似，从用户空间中复制新值，而且结果以 NUL 来结束。不过在这种情形下，不从用户空间复制 NUL 字节是一种正确的做法，因为把它从用户空间复制进来要比仅仅在 data 的适当字节安排一个 NUL 效率低。而且以这种方式，即使输入不是 NUL 结束的，**table->data** 也要如此。当然，从 **newval** 读出的字符串可能已经是 NUL 结束的，在那种情况里第 31154 行的赋值就是多余的。这还是另一种情况，直接完成工作比检查需要是否执行它还要快。

31156：返回 0 表示成功。相反，返回的值应该为正数，以便 30618 行代码认为结果是成功的。而又相反，调用代码认为 **sysctl_string** 想让缺省处理发生，然后它就继续从用户空间再次复制多余的数据。

sysctl_intvec

31163：**sysctl_intvec** 是在 kernel/sysctl.c 里定义的另一个策略例程。它确保假如调用程序正在写入表项，所有被写的 **int** 都应位于某个最小和最大值之间。（顺便提及一下，**sysctl_intvec** 在这个文件里只被使用了一次——在第 30414 行——尽管它被广泛的用于本书所没有包括的内核的其它代码之中。）

31170：如果新的欲写数据总量不符合一个 **int** 大小的边界，它就无效，所以尝试被抛弃。

31173：假如表项没有指定一组最大或最小值，输入的值就永远不可能超出范围，这样调用程序里的普通写代码（**do_sysctl_strategy**，30603 行）就足够好了。因此在这种情况里 **sysctl_intvec** 返回 0。

31184：进行循环以确保所有来自输入数组的值都位于适当范围之内。

31186：这行代码不检查 **get_user** 的返回值——没有迫切的需要去这样做。如果当不能读取一个输入内存位置时，**sysctl_intvec** 返回 0（成功），那么当它试图读取整个数组时 **do_sysctl_strategy** 就会注意到这个问题。作为另一选择，假如 **get_user** 无法读取内存位置，无用信息（garbage）可能在 **value** 里结束并且数值可能会不正确的被抛弃。在此情况里，调用程序将得到一个 **EINVAL** 错误而不是 **EFAULT** 错误，这只是一个小缺陷（bug）。

31187：注意这一行不会被折磨第 31033 行相似代码的缺陷所困扰，该行中在最小值和最大值之上进行的并行循环会产生不同步的情况。

这一行代码能够避免位于 31033 行的缺陷被暴露出来。正如实际中所进行的，**sysctl_intvec** 和 **proc_dointvec_minmax** 都总是与同一个 **ctl_table** 条目相关联的。因此，在调用处理例程（handler routine）**proc_dointvec_minmax** 之前，任何超出允许范围之外的数值将会被策略例程 **sysctl_intvec** 截获。所以，我们知道——在给定内核里所有的 **ctl_tables** 最新定义的情况下——**proc_dointvec_minmax** 将永远不会遇到超出界限的数值，而那里是唯一可以触发该缺陷的数值种类。某个调用程序或许可以注册一个使用 **proc_dointvec_minmax** 但没有策略例程的 **ctl_table**，但是尽管这样，在 **proc_dointvec_ minmax** 里的这个缺陷迟早会造成一定损害。

31193：返回 0 表示成功。这里不像在第 31156 行那样是一个错误，因为 **sysctl_intvec** 并不向 **table->data** 里写入。从用户空间读出的值只是被读进一个临时变量里作范围检查，然后就被删除；**do_sysctl_strategy** 将完成那项工作，并只向 **table->data** 里进行写入。