第 10 章 合理使用数据类型

在进一步讨论更深的主题之前，我们需要先停一停，快速地回顾一下可移植问题。Linux1.2 版本和 2.0 版本之间的不同就在于额外的多平台能力；结果是，大多数源代码级的移植问题已经被排除了。这意味着一个规范的 Linux 驱动程序也应该是多平台的。

但是，与内核代码相关的一个核心问题是，能够同时存取各种长度已知的数据项(例如，文件系统数据类型或者设备卡上的寄存器)和利用不同处理器的能力(32 位和 64 位的体系结构，也有可能是 16 位的)。

当把 x86 的代码移植到新的体系结构上时，核心开发者遇到的好几个问题都和不正确的数据类型相关。坚持强数据类型以及编译时使用 *-Wall -Wstrict-prototypes* 选项能够防止大部分的臭虫。

内核使用的数据类型划分为三种主要类型：象 **int** 这样的标准 C 语言类型，象 **u32** 这样的确定数据大小的类型和象 **pid_t** 这样的接口特定类型。我们将看一下这三种类型在何时使用和如何使用。本章的最后一节将讨论把驱动器代码从 x86 移植到其它平台上可能碰到的其它一些典型问题。

如果你遵循我提供的这些准则，你的驱动程序甚至可能在那些你未能进行测试的平台上编译并运行。

使用标准 C 类型

大部分程序员习惯于自由的使用诸如 **int** 和 **long** 这样的标准类型，而编写设备驱动程序就必须细心地避免类型冲突和潜在的臭虫。

问题是，当你需要“2 个字节填充单位(filler)”或“表示 4 个字节字符串的某个东西”时，你不能使用标准类型，因为通常的 C 数据类型在不同的体系结构上所占空间大小并不相同。例如，长整数和指针类型在 Alpha 上和 x86 上所占空间大小就不一样，下面的屏幕快照表明了这一点：

```
morgana% ./datasize
system/machine: Linux i486
sizeof(char) =     1
sizeof(short) =    2
sizeof(int) =      4
sizeof(long) =     4
sizeof(longlong) = 8
sizeof(pointer) =  4
```

```
wolf% ./datasize
system/machine: Linux alpha
sizeof(char) =     1
sizeof(short) =    2
sizeof(int) =      4
sizeof(long) =     8
sizeof(longlong) = 8
sizeof(pointer) =  8

sandra% ./datasize
system/machine: Linux sparc
sizeof(char) =     1
sizeof(short) =    2
sizeof(int) =      4
sizeof(long) =     4
sizeof(longlong) = 8
sizeof(pointer) =  4
```

datasize 程序是一个可以从在 O'Reilly FTP 站点的 *misc-progs* 目录下获得的小程序。

在混合使用 int 和 long 类型时，你必须小心，有时有很好的理由这样做，一种情形就是内存地址，一涉及到内核，内存地址就变得很特殊。虽然概念上地址是指针，但是通过使用整数类型，可以更好地实现内存管理；内核把物理内存看做一个巨大的数组，内存地址就是这个数组的索引。而且，一个指针很容易被取地址(deference)，而使用整数表示内存地址可以防止它们被取地址，这正是人们所希望的(比使用指针更安全)。因而，内核中的地址属于 **unsigned long** 类型，这是利用了指针和长整数类型大小总是相同这一事实，至少在所有 Linux 当前支持的平台上是这样的。我们等着看看将来把 Linux 移植到不符合这一规则的平台上的时候，会发生些什么。

分配确定的空间大小给数据项

有时内核代码需要指定大小的数据项，或者用来匹配二进制结构*或者用来在结构中插入填充字段对齐数据。

为此目的，内核提供如下的数据类型，它们都在头文件**<asm/types.h>**中声明，这个文件又被头文件**<linux/types.h>**所包含：

```
u8;     /* 无符号字节(8 位) */
u16;    /* 无符号字(16 位) */
u32;    /* 无符号 32 位数值 */
u64;    /* 无符号 64 位数值 */
```

* 读分区表时，执行二进制文件时或者解码一个网络包时，就会发生这种情况。

这些数据类型只能被内核代码所访问(也即,在包含头文件<linux/types.h>之前必须先定义__KERNEL__)。相应的有符号类型也是存在的，但一般不用；如果你需要使用它们的话，只要把名字中的 u 替换为 s 就可以了。

如果用户空间的程序需要使用这些类型，可以在这些名字前面添加 2 个下划线：__u8 和其它类型是独立于__KERNEL__定义的。例如，如果一个驱动程序需要通过 ioctl 系统调用与一个运行在用户空间内的程序交换二进制结构的话，头文件必须将结构中的 32 位字段定义为__u32。

重要的是要记住这些类型特定于 Linux，使用它们就会防碍软件向其他 Unix 变体的移植。但是，有些情况下也需要明确说明数据大小，而标准头文件(在每个 Unix 系统上都能找到的)并未声明较合适的数据类型。

你也许注意到，有时内核也使用一般的数据类型，象unsigned int，用于那些大小与体系结构无关的项。这通常是为了向后兼容。当u32 及其相关类型在 1.1.67 版本引入时开发者没办法把存在的数据类型改成新类型，因为当结构字段和赋予的值之间类型不匹配时，编译器会发出警告[+]。Linus当初可没预料到为自己使用而编写的这个操作系统会发展成为多平台的；因此，一些旧的结构的数据类型定义上不是很严格。

接口特定的类型

内核中最常使用的数据类型有它们自己的 typedef 声明,这样就防止了任何移植上的问题。例如，进程号(pid)通常使用 pid_t，而不是 int。使用 pid_t 屏蔽了任何实际数据类型之间可能的差别。我使用“接口特定”这种表述来指代特定数据项的编程接口。

属于指定“标准”类型的其它数据项也可以认为是接口特定的。比如，一个 jiffy 计数总是属于 unsigned long 类型的，独立于它的实际大小－你喜欢那么频繁地使用 jiffy_t 类型么？这里我关注的是接口特定类型的第一类，那些以_t 结尾的类型。

_t 类型完整的列表在头文件<linux/types.h>中，但是该列表几乎没什么用。当需要一个特定类型时，你可以在你要调用的函数原型或者使用的数据结构中找到它。

只要你的驱动程序使用了需要这种“定制”类型的函数，又不遵循惯例的时候，编译器都会发出一个警告；如果你打开*-Wall* 编译开关并且细心地去除了所有警告，你就可以自信你的代码是可移植的了。

_t 数据项的主要问题是当你需要打印它们的时候，并不总是容易选择正确的 *printk* 或者 *printf* 格式，并且你在一种体系结构上排除了的警告，在另一种体系结构上可能又会出现。例如,当 size_t 在一些平台上是 unsigned long,而在另外一些平台上却是 unsigned int 时，你怎么打印它呢？

[+] 实际上，即使两种类型仅是同一对象的不同名字，例如PC上的**unsigned long**和**u32** 类型，编译器也会发出类型不匹配的信号。

任何时候，当你需要打印一些特定接口的数据的时候，最行之有效的方法就是，把它强制转换成最可能的类型(通常是 long 或 unsigned long 类型)，然后把它用相应的格式打印出来。这种做法不会产生错误或者警告，因为格式和类型相符，而且你也不会丢失数据位，因为强制类型转换要么是个空操作，要么是将该数据项向更大数据类型的扩展。

实际上，通常我们并不会去打印我们讨论的这些数据项，因此只有显示调试信息时才会碰到这些问题。更经常的，除了把接口特定的类型作为参数传递给库或内核函数以外，代码仅仅只会对它们进行些储存和比较。

虽然大多数情形下，_t 类型都是正确的解决方案，但有时候正确的类型也可能并不存在。这会发生在一些还没被抛弃的旧接口上。

在内核头文件中我发现一处疑点，为 I/O 函数声明数据类型时不是很严格(参见第 8 章*“硬件管理”*中的“平台相关性”一节)。这种不严格的类型定义主要是出于历史上的原因，但在编写代码时却会带来问题。就我而言，我经常在把参数交换给 *out* 函数时遇上麻烦；而如果定义了 **port_t**，编译器将会指出这些错误。

其它与移植有关的问题

除了数据类型定义问题之外，如果想让你编写的驱动程序能在不同的 Linux 平台间移植的话，还必须注意到其它一些软件上的问题:

时间间隔

在处理时间间隔时，不能假定每秒一定有 100 个 jiffy。虽然对当前的 Linux-x86 而言这是对的，但并不是所有 Linux 平台都是以 100**HZ** 运行。如果你改变了 **HZ** 的数值，那么即使对 x86，这种假设也是错误的，何况没人知道未来的内核会发生些什么变化。使用 jiffy 计算时间间隔的时候，应该把时间转换成以 **HZ** 为单位。例如，为了检测半秒钟的超时，可以把消逝的时间和 **HZ/2** 作比较。更常见的，与 **msec** 毫秒对应的 jiffy 的数目总是 **msec*HZ/1000**。许多的网络驱动程序在移植到 Alpha 上时都必须修正该细节；有些开始是为 PC 设计的驱动程序给超时明确定义了一个 jiffy 值，但是 Alpha 却有着不同的 **HZ** 数值。

页大小

使用内存时，要记住内存页的大小为 **PAGE_SIZE** 字节，而不是 4KB。假设页大小就是 4KB 并硬编码该数值是 PC 程序员常犯的错误－Alpha 页大小是这的两倍。相关的宏有 **PAGE_SIZE** 和 **PAGE_SHIFT**。后者包含要得到一个地址所在页的页号时需要对该地址右移的位数。对当前的 4KB 和 8KB 的页，这个数值通常是 12 或者 13。这些宏在头文件**<asm/page.h>**中定义。

让我们来看一种简单的情况。如果驱动程序需要 16KB 空间来存放临时数据，它不应当指定 *get_free_pages* 函数的参数 **order**(“2”的幂)。需要一种可移植的解决办法。此时，可以使用条件编译**#ifdef __alpha__**，但这只适用于已知的平台，而如果要支持别的平台，它就不能奏效了。我建议使用下面的代码：

```
buf = get_free_pages(GFP_KERNEL， 14 - PAGE_SHIFT， 0 /*dma*/)；
```

或者，更好一些的代码：

```
int order = (14 - PAGE_SHIFT >0) ? 14 - PAGE_SHIFT : 0；
buf = get_free_pages(GFP_KERNEL， order， 0 /*dma*/)；
```

两种解决办法都利用了 16KB 等于 **1<<14** 这一常识。两个数的商就是它们对数的差(的幂)，而 **14** 和 **PAGE_SHIFT** 都是幂。第二种解决办法就更好，因为它可以防止把一个负的 order 值传递给 *get_free_pages* 函数；**order** 值时在编译时就计算好的，没有运行时的额外开销，而且，上面给出的实现方法是不依赖于 **PAGE_SIZE** 来分配任何 2 的幂次大小的内存空间的安全方法。

字节序

要小心的是不要主观假设字节序。虽然 PC 是按低字节优先的方式存储多个字节(“小印地安，little endian”)，但是大多数更高级的平台是以另一种方式工作的(“大印地安，big endian”)。虽然好的程序不会依赖于字节序，但有时驱动程序需要创建占一个字节以上的整数，或者相反(一个字节以下)。此时，代码中就应该将头文件**<asm/byteorder.h>**包含进来，并且检测头文件中是否定义了**__BIG_ENDIAN** 或**__LITTLE_ENDIAN**。起始的下划线在 Linux-1.2 之后版本的头文件中却去掉了，在头文件**<asm/byteorder.h>**后再包含 *scull* 示例程序中的头文件 *sysdep.h* 就可以修正这个不兼容。

当字节序相关问题与网络传输有联系的时候，就应当使用下面各种函数来进行 16 位和 32 位数值的转换，这些函数也都是在头文件**<asm/byteorder.h>**中定义的：

```
unsigned long  ntohl(unsigned long)；
unsigned short ntohs(unsigned short)；
unsigned long  htonl(unsigned long)；
unsigned short htons(unsigned short)；
```

在网络程序员当中，这些函数是众所周知的。它们得名于“Network TO Host Long”(从网络到主机的 long 类型)或类似的短语。

2.1.10 版的内核增加了 cpu-to-little-endian 和 cpu-to-big-endian 两种转换，2.1.43 版的内核在这方面又加以扩充。新增的一些实用函数将在第 17 章“*近期发展*”中的“转换函数”一节中描述。

数据对齐

在编写可移植代码时最后一个值得考虑的问题是如何访问未对齐数据－例如，当一个 4 字节的数值被储存在不是 4 字节的整数倍的地址中时，如何将它读出来。PC 的用户常常访问未对齐的数据项，但并不是所有体系结构都允许这样做。举个例子，在 Alpha 上，每当程序试图传送未对齐数据时，都会产生一个异常。如果你需要访问未对齐数据，可以使用下面这些宏：

```
#include <asm/unaligned.h>
get_unaligned(ptr);
put_unaligned(val, ptr);
```

这些宏是与类型无关的。对各种数据项，不管它是 1 字节，2 字节，4 字节还是 8 字节，这些宏都有效。在 2.0 版以前的内核并不提供这些宏，但在 1.2 版的内核在头文件 *sysdep.h* 中对它们作了定义。

一个通用准则就对显式的常数值持怀疑态度。通常，使用预编译的宏来使代码参数化使代码更通用。虽然我不能在此列出所有参数化的值，但你可以在头文件中找到正确的提示。

不幸的是，有些地方问题还没有得到解决，例如对磁盘扇区数据的处理。出于历史的原因，Linux 只能处理**.5KB** 的磁盘扇区。所幸的是，现存所有设备都满足这个限制。目前正在逐渐地改进代码以支持不同的扇区大小。但要找到代码中所有在**.5KB** 的假设下进行硬编码的地方却非常困难。扇区大小的问题将在第 12 章“*加载块设备驱动程序*”中进一步描述。

快速参考

在本章引入了如下一些符号：

#include <linux/types.h>
typedef u8;
typedef u16;
typedef u32;
typedef u64;

这些类型保证是 **8-**、**16-**、**32-**和 **64-**位的无符号整数值。对应的有符号类型同样存在。在用户空间，你可以通过**__u8**，**__u16** 等来引用这些类型。

#include <asm/page.h>
PAGE_SIZE
PAGE_SHIFT

这些符号定义了当前体系结构下每页包含的字节数和页偏移量所占位数(12 对应 4KB 的页而 13 对应 8KB 的页)。

#include <asm/byteorder.h>
__LITTLE_ENDIAN
__BIG_ENDIAN

两个符号中只能定义其一，这依赖于体系结构。版本 1.3.18 和更老的版本中也声明了这些符号，但是没有打头的下划线(因而和一些网络部分的头文件会发生冲突)。

#include <asm/byteorder.h>
unsigned long ntohl(unsigned long);
unsigned short ntohs(unsigned short);
unsigned long htonl(unsigned long);

unsigned short htons(unsigned short)；

这些函数在网络字节序和主机字节序间转换 long 类型和 short 类型的数据。

#include <asm/unaligned.h>

get_unaligned(ptr)；

put_unaligned(val，ptr)；

一些体系结构需要使用这些宏来保护对未对齐数据的访问。在这些体系结构上，这些宏扩展为通常的对指针取地址的操作，以允许你访问未对齐的数据。