第十一章 内核中的数据类型

在我们继续更高级的主题之前，我们需要停下来简单地讨论一下可移植性问题。现代版本的 Linux 内核具有高度的可移植性，可以在许多不同的架构上运行。鉴于 Linux 的多平台性质，用于严肃使用的驱动程序也应该是可移植的。

但是，内核代码的一个核心问题是能够访问已知长度的数据项（例如，文件系统数据结构或设备板上的寄存器），并利用不同处理器的能力（32位和64位架构，可能还有16位）。

内核开发人员在将 x86 代码移植到新架构时遇到的几个问题与数据类型不正确有关。坚持严格的数据类型并使用 -Wall -Wstrict-prototypes 标志进行编译可以防止大多数错误。

内核数据使用的数据类型分为三个主要类别：标准 C 类型如 int，明确大小的类型如 u32，以及用于特定内核对象的类型，如 pid_t。我们将看到三种类型类别应该何时以及如何使用。本章的最后几节讨论了你在将驱动程序代码从 x86 移植到其他平台时可能遇到的一些典型问题，并介绍了最近的内核头文件导出的链表的通用支持。

如果你遵循我们提供的指南，你的驱动程序应该能够在你无法测试的平台上编译和运行。

11.1. 标准 C 类型的使用

尽管大多数程序员习惯于自由地使用像 int 和 long 这样的标准类型，但编写设备驱动程序需要一些小心，以避免类型冲突和难以发现的错误。

问题在于，当你需要“一个2字节的填充”或“代表4字节字符串的东西”时，你不能使用标准类型，因为在所有架构上，普通的 C 数据类型的大小并不相同。为了显示各种 C 类型的数据大小，datasize 程序已经包含在 O'Reilly 的 FTP 站点上提供的示例文件中，位于 misc-progs 目录中。这是在 i386 系统上运行程序的一个示例运行（最后四种类型将在下一节中介绍）：

```
morgana% misc-progs/datasize

arch   Size:  char  short  int  long   ptr long-
long  u8 u16 u32 u64

i386            1     2     4     4     4     8        1
 2   4   8
```

该程序可以用来显示长整数和指针在64位平台上具有不同的大小，如在不同的 Linux 计算机上运行程序所示：

```
arch   Size:  char  short  int  long   ptr long-
long  u8 u16 u32 u64

i386            1     2     4     4     4     8        1
 2   4   8

alpha           1     2     4     8     8     8        1
 2   4   8

armv4l          1     2     4     4     4     8        1
 2   4   8

ia64            1     2     4     8     8     8        1
 2   4   8

m68k            1     2     4     4     4     8        1
 2   4   8

mips            1     2     4     4     4     8        1
 2   4   8

ppc             1     2     4     4     4     8        1
 2   4   8

sparc           1     2     4     4     4     8        1
 2   4   8

sparc64         1     2     4     4     4     8        1
 2   4   8

x86_64          1     2     4     8     8     8        1
 2   4   8
```

有趣的是，SPARC 64 架构以32位用户空间运行，所以那里的指针宽度为32位，尽管在内核空间中它们的宽度为64位。这可以通过加载 kdatasize 模块（在示例文件的 misc-modules 目录中可用）来验证。该模块在加载时使用 printk 报告大小信息，并返回一个错误（所以不需要卸载它）：

```
kernel: arch   Size:  char short int long  ptr long-
long u8 u16 u32 u64

kernel: sparc64         1    2    4    8    8     8
1   2   4   8
```

尽管你在混合使用不同的数据类型时必须小心，但有时候有充分的理由这样做。这样的情况之一是内存地址，这对内核来说是特殊的。尽管从概念上讲，地址是指针，但内存管理通常更好地通过使用无符号整数类型来完成；内核将物理内存视为一个巨大的数组，内存地址只是数组的一个索引。此外，指针很容易被解引用；当直接处理内存地址时，你几乎永远不想以这种方式解引用它们。使用整数类型可以防止这种解引用，从而避免错误。因此，内核中的通用内存地址通常是 unsigned long，利用了指针和长整数总是相同大小的事实，至少在所有目前由 Linux 支持的平台上都是这样。

值得一提的是，C99 标准为可以保存指针值的整数变量定义了 intptr_t 和 uintptr_t 类型。然而，这些类型在 2.6 内核中几乎未被使用。

- 总的来说，为了避免类型冲突和难以发现的错误，编写设备驱动程序时需要特别注意数据类型的选择。在处理内存地址时，通常更倾向于使用无符号整数类型，而不是指针，这是因为内核将物理内存视为一个巨大的数组，内存地址只是数组的一个索引。此外，使用整数类型可以防止误解引用，从而避免错误。因此，内核中的通用内存地址通常是 unsigned long，利用了指针和长整数总是相同大小的事实。

11.2. 安排一个明确大小给数据项

有时内核代码需要特定大小的数据项，可能是为了匹配预定义的二进制结构，与用户空间通信，或者通过插入“填充”字段在结构内对齐数据（但请参阅“数据对齐”部分以获取有关对齐问题的信息）。

内核提供了以下数据类型，以便在你需要知道数据大小时使用。所有类型都在 <asm/types.h> 中声明，而 <asm/types.h> 又被 <linux/types.h> 包含：

```c
u8;   /* unsigned byte (8 bits) */

u16;  /* unsigned word (16 bits) */

u32;  /* unsigned 32-bit value */

u64;  /* unsigned 64-bit value */
```

相应的有符号类型存在，但很少需要；如果你需要它们，只需在名称中将 u 替换为 s。

如果用户空间程序需要使用这些类型，它可以在名称前加上双下划线：`__u8` 和其他类型是独立于 `**KERNEL**` 定义的。例如，如果驱动程序需要通过 ioctl 与在用户空间运行的程序交换二进制结构，那么头文件应该将结构中的 32 位字段声明为 `__u32`。

重要的是要记住，这些类型是 Linux 特有的，使用它们会阻碍将软件移植到其他 Unix 版本。具有最新编译器的系统支持 C99 标准类型，如 uint8_t 和 uint32_t；如果可移植性是一个问题，那么可以使用这些类型来代替 Linux 特有的类型。

你可能还注意到，有时内核使用传统类型，如 unsigned int，用于其维度与架构无关的项。这通常是为了向后兼容。当在版本 1.1.67 中引入 u32 和朋友们时，开发人员不能将现有的数据结构更改为新的类型，因为当结构字段和被赋值的值之间存在类型不匹配时，编译器会发出警告。† Linus 没有期望他为自己使用而编写的操作系统（OS）会变成多平台；因此，旧的结构有时会松散地进行类型化。

11.3. 接口特定的类型

内核中常用的一些数据类型有自己的 typedef 语句，从而防止任何可移植性问题。例如，进程标识符（pid）通常是 pid_t 而不是 int。使用 pid_t 可以掩盖实际数据类型可能存在的任何差异。我们使用表达式 interface-specific 来指代由库定义的类型，以便为特定数据结构提供接口。

请注意，近年来，定义的新的 interface-specific 类型相对较少。typedef 语句在许多内核开发人员中已经不受欢迎，他们更愿意在代码中直接使用真实的类型信息，而不是隐藏在用户定义的类型后面。然而，许多较旧的 interface-specific 类型仍然存在于内核中，而且它们不会很快消失。

即使没有定义 interface-specific 类型，始终使用与内核其余部分一致的适当数据类型也很重要。例如，jiffy 计数始终是 unsigned long，无论其实际大小如何，因此在处理 jiffies 时应始终使用 unsigned long 类型。在本节中，我们将重点讨论 `_t` 类型的使用。

许多 `_t` 类型在 <linux/types.h> 中定义，但列表很少有用。当你需要特定类型时，你会在需要调用的函数的原型或你使用的数据结构中找到它。

每当你的驱动程序使用需要这种“自定义”类型的函数并且你没有遵循约定时，编译器会发出警告；如果你使用 -Wall 编译器标志并且小心地消除所有警告，你可以确信你的代码是可移植的。

`_t` 数据项的主要问题是，当你需要打印它们时，选择正确的 printk 或 printf 格式并不总是容易的，你在一个架构上解决的警告在另一个架构上重新出现。例如，你如何打印一个 size_t，它在一些平台上是 unsigned long，在一些其他平台上是 unsigned int?

每当你需要打印一些 interface-specific 数据时，最好的方法是将值转换为最大可能的类型（通常是 long 或 unsigned long），然后通过相应的格式打印它。这种调整不会产生错误或警告，因为格式与类型匹配，你也不会丢失数据位，因为转换要么是空操作，要么是将项扩展到更大的数据类型。

实际上，我们正在讨论的数据项通常不是用来打印的，所以这个问题只适用于调试消息。最常见的是，代码只需要存储和比较 interface-specific 类型，除了将它们作为参数传递给库或内核函数。

尽管 `_t` 类型在大多数情况下是正确的解决方案，但有时可能不存在正确的类型。这种情况发生在一些尚未清理的旧接口上。我们在内核头文件中发现的一个模糊点是 I/O 函数的数据类型，这是松散定义的（参见第9章的“平台依赖性”部分）。松散的类型主要是出于历史原因，但在编写代码时可能会产生问题。例如，通过交换像 outb 这样的函数的参数，可能会遇到麻烦；如果有一个 port_t 类型，编译器会找到这种类型的错误。

11.4. 其他移植性问题

在编写驱动程序以实现跨 Linux 平台的可移植性时，除了数据类型外，还有一些其他的软件问题需要注意。

一个通用的规则是对显式常量值持怀疑态度。通常，代码已经使用预处理器宏进行了参数化。本节列出了最重要的可移植性问题。每当你遇到其他已经参数化的值时，你可以在头文件和官方内核分发的设备驱动程序中找到提示。

1. **字节顺序（Byte Order）**：不同的硬件平台可能有不同的字节顺序，即大端和小端。在处理硬件或网络数据时，需要注意这一点。Linux 提供了一组宏来处理字节顺序问题，如 `htons`，`htonl`，`ntohs` 和 `ntohl`。
2. **内存对齐（Memory Alignment）**：某些硬件平台对内存对齐有严格的要求。尽管编译器通常会处理这个问题，但在某些情况下，你可能需要手动处理。
3. **硬件访问**：不同的硬件平台可能有不同的方式来访问硬件。例如，某些平台可能需要使用特殊的函数来进行 I/O 访问。Linux 提供了一组函数和宏来处理这些问题，如 `inb`，`outb`，`readb`，`writeb` 等。
4. **定时和延迟**：不同的硬件平台可能有不同的时钟频率，因此在编写需要精确延迟的代码时，需要注意这一点。Linux 提供了一组函数来处理这些问题，如 `udelay`，`mdelay` 等。
5. **原子操作**：在多处理器系统中，需要确保并发操作的原子性。Linux 提供了一组函数来处理这些问题，如 `atomic_add`，`atomic_sub` 等。
6. **中断处理**：不同的硬件平台可能有不同的方式来处理中断。在编写中断处理程序时，需要注意这一点。

总的来说，为了编写可移植的驱动程序，你需要了解并遵循 Linux 的硬件抽象层（HAL）。这样，你的驱动程序就可以在不同的硬件平台上运行，而无需进行任何修改。

11.4.1. 时间间隔

在处理时间间隔时，不要假设每秒有1000个 jiffies。尽管这对于 i386 架构目前是正确的，但并非每个 Linux 平台都以这个速度运行。即使对于 x86，如果你调整 HZ 值（就像有些人做的那样），这个假设也可能是错误的，而且没人知道未来的内核会发生什么。每当你使用 jiffies 计算时间间隔时，都要使用 HZ（每秒定时器中断的次数）来缩放你的时间。例如，要检查半秒的超时，将经过的时间与 HZ/2 进行比较。更一般地说，对应于 msec 毫秒的 jiffies 数量总是 msec*HZ/1000。

11.4.2. 页大小

在处理内存问题时，记住一个内存页是 PAGE_SIZE 字节，而不是 4 KB。假设页面大小为 4 KB 并硬编码该值是 PC 程序员的常见错误，实际上，支持的平台显示的页面大小从 4 KB 到 64 KB 不等，有时它们在同一平台的不同实现之间也会有所不同。相关的宏是 PAGE_SIZE 和 PAGE_SHIFT。后者包含了将地址移位以获取其页码的位数。当前的数字对于 4 KB 及更大的页面是 12 或更大。这些宏在 <asm/page.h> 中定义；如果用户空间程序需要这些信息，它们可以使用 getpagesize 库函数。

让我们看一个非平凡的情况。如果驱动程序需要 16 KB 的临时数据，它不应该为 get_free_pages 指定 order 为 2。你需要一个可移植的解决方案。幸运的是，内核开

发人员已经编写了这样的解决方案，称为 get_order：

```c
#include <asm/page.h>

int order = get_order(16*1024);

buf = get_free_pages(GFP_KERNEL, order);
```

记住，传递给 get_order 的参数必须是 2 的幂。

11.4.3. 字节序

在处理字节顺序时，要小心不要做出假设。虽然 PC 以低字节优先（小端优先，因此是小端序）存储多字节值，但一些高级平台的工作方式却恰恰相反（大端序）。只要可能，你的代码应该写成不关心它操作的数据的字节顺序。然而，有时驱动程序需要从单个字节构建一个整数，或者反过来，或者它必须与期望特定顺序的设备通信。包含文件 <asm/byteorder.h> 根据处理器的字节顺序定义了 `__BIG_ENDIAN` 或 `__LITTLE_ENDIAN`。在处理字节顺序问题时，你可以编写一堆 `#ifdef __LITTLE_ENDIAN` 条件语句，但有更好的方法。Linux 内核定义了一组宏，用于处理处理器的字节顺序和你需要以特定字节顺序存储或加载的数据之间的转换。例如：

```c
u32 cpu_to_le32 (u32);

u32 le32_to_cpu (u32);
```

这两个宏将一个值从 CPU 使用的任何值转换为无符号的、小端的、32位的量，然后再转换回来。无论你的 CPU 是大端序还是小端序，无论它是不是 32 位处理器，它们都能工作。在不需要做任何工作的情况下，它们会原样返回它们的参数。使用这些宏可以很容易地编写可移植的代码，而不必使用大量的条件编译结构。

有几十个类似的例程；你可以在 <linux/byteorder/big_endian.h> 和 <linux/byteorder/little_endian.h> 中看到完整的列表。过一段时间后，这种模式并不难理解。be64_to_cpu 将一个无符号的、大端的、64位的值转换为内部 CPU 表示。le16_to_cpus，相反，处理有符号的、小端的、16位的量。在处理指针时，你也可以使

用像 cpu_to_le32p 这样的函数，它接受一个指向要转换的值的指针，而不是值本身。请参阅包含文件以获取其余内容。

11.4.4. 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的数据——例如，如何读取存储在非4字节倍数的地址上的4字节值。i386 用户经常访问未对齐的数据项，但并非所有的架构都允许这样做。许多现代架构每次程序尝试未对齐的数据传输时都会产生一个异常；数据传输由异常处理器处理，带来了很大的性能损失。如果你需要访问未对齐的数据，你应该使用以下的宏：

```c
#include <asm/unaligned.h>

get_unaligned(ptr);

put_unaligned(val, ptr);
```

这些宏是无类型的，适用于每一个数据项，无论它是一、二、四或八字节长。它们在任何内核版本中都有定义。

另一个与对齐相关的问题是跨平台的数据结构的可移植性。同一数据结构（如在 C 语言源文件中定义的）在不同平台上的编译可能会有所不同。编译器会根据不同平台的约定来对齐结构字段。

为了编写可以在架构之间移动的数据项的数据结构，你应该始终强制数据项的自然对齐，除了标准化特定的字节顺序。自然对齐意味着将数据项存储在它们大小的倍数的地址上（例如，8字节的项放在8的倍数的地址上）。为了在防止编译器以不可预测的方式排列字段的同时强制自然对齐，你应该使用填充字段来避免在数据结构中留下空洞。

为了展示编译器如何强制对齐，数据对齐程序在示例代码的 misc-progs 目录中分发，一个等效的 kdataalign 模块是 misc-modules 的一部分。这是该程序在几个平台上的输出，以及该模块在 SPARC64 上的输出。

```
arch Align: char short int long ptr long-long u8 u16 u32
u64
i386        1    2     4   4    4   4          1  2   4   4
i686        1    2     4   4    4   4          1  2   4   4
alpha       1    2     4   8    8   8          1  2   4   8
armv4l      1    2     4   4    4   4          1  2   4   4
ia64        1    2     4   8    8   8          1  2   4   8
mips        1    2     4   4    4   8          1  2   4   8
ppc         1    2     4   4    4   8          1  2   4   8
sparc       1    2     4   4    4   8          1  2   4   8
sparc64     1    2     4   4    4   8          1  2   4   8
x86_64      1    2     4   8    8   8          1  2   4   8

kernel: arch Align: char short int long ptr long-long u8
u16 u32 u64
kernel: sparc64     1    2     4   8    8   8          1  2
4   8
```

值得注意的是，并非所有平台都将64位值对齐到64位边界，因此你需要填充字段来强制对齐并确保可移植性。

最后，要注意编译器可能会静默地将填充插入到结构中，以确保每个字段都对齐，以便在目标处理器上获得良好的性能。如果你正在定义一个结构，该结构旨在匹配设备期望的结构，那么这种自动填充可能会阻碍你的尝试。解决这个问题的方法是告诉编译器，结构必须是“打包的”，不添加填充。例如，内核头文件 <linux/edd.h> 定义了几个用于与 x86 BIOS 接口的数据结构，并包含以下定义：

```c
struct {
        u16 id;
        u64 lun;
        u16 reserved1;
        u32 reserved2;
} __attribute__ ((packed)) scsi;
```

如果没有 **attribute** ((packed))，lun 字段前面会有两个填充字节，或者如果我们在64位平台上编译结构，前面会有六个填充字节。

11.4.5. 指针和错误值

许多内核内部函数返回一个指针值给调用者。这些函数中的许多也可能失败。在大多数情况下，通过返回一个 NULL 指针值来表示失败。这种技术是有效的，但它无法传达问题的确切性质。一些接口确实需要返回一个实际的错误代码，以便调用者可以根据实际出错的情况做出正确的决定。

一些内核接口通过在指针值中编码错误代码来返回这些信息。这样的函数必须谨慎使用，因为它们的返回值不能简单地与 NULL 进行比较。为了帮助创建和使用这种接口，提供了一小组函数（在 <linux/err.h> 中）。

返回指针类型的函数可以用以下方式返回错误值：

```c
void *ERR_PTR(long error);
```

其中 error 是通常的负错误代码。调用者可以使用 IS_ERR 来测试返回的指针是否是一个错误代码：

```c
long IS_ERR(const void *ptr);
```

如果你需要实际的错误代码，可以用以下方式提取：

```c
long PTR_ERR(const void *ptr);
```

你应该只在 IS_ERR 返回真值的情况下使用 PTR_ERR；任何其他值都是一个有效的指针。

11.5. 链表

操作系统内核，像许多其他程序一样，经常需要维护数据结构的列表。Linux 内核有时会同时承载几个链表实现。为了减少重复的代码，内核开发人员创建了一个标准的循环双向链表实现；鼓励需要操作列表的其他人使用这个设施。

在使用链表接口时，你应该始终记住，列表函数不执行锁定。如果有可能你的驱动程序会尝试对同一列表进行并发操作，那么实现锁定方案是你的责任。其他选择（如损坏的列表结构、数据丢失、内核恐慌）往往难以诊断。

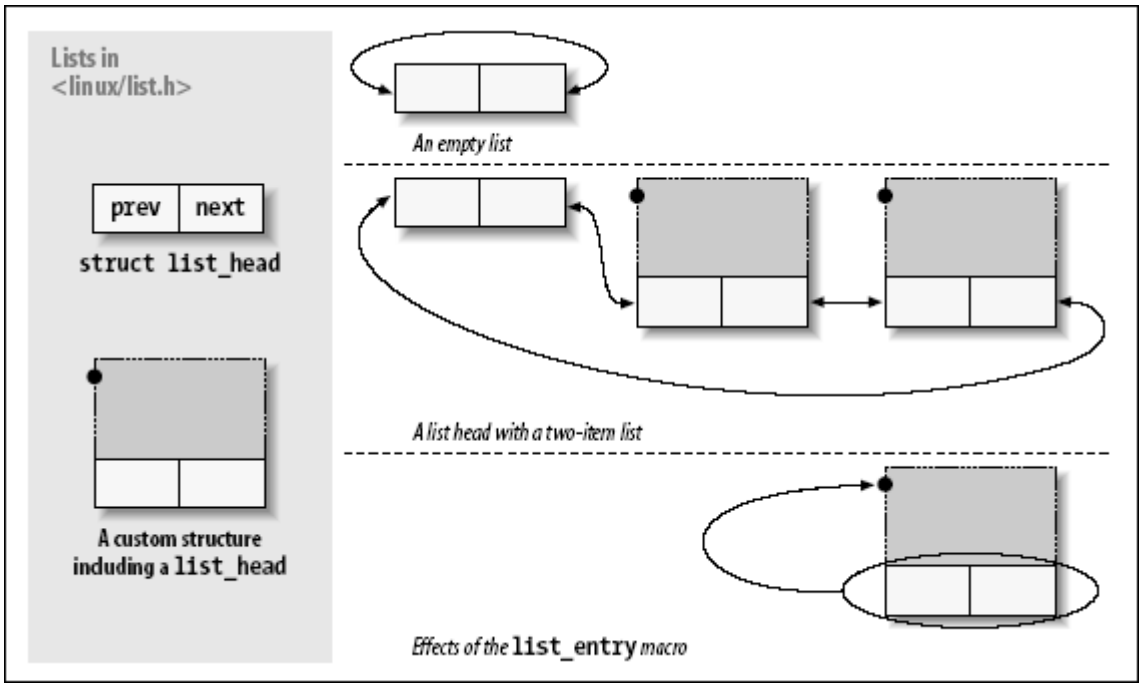
要使用列表机制，你的驱动程序必须包含文件 <linux/list.h>。这个文件定义了一个简单的 list_head 类型的结构：

```c
struct list_head {
    struct list_head *next, *prev;
};
```

在实际代码中使用的链表几乎总是由某种类型的结构组成，每一个结构描述列表中的一个条目。要在你的代码中使用 Linux 列表设施，你只需要在构成列表的结构中嵌入一个 list_head。如果你的驱动程序维护一个待办事项列表，比如，它的声明可能看起来像这样：

```c
struct todo_struct {
    struct list_head list;
    int priority; /* driver specific */
    /* ... add other driver-specific fields */
};
```

列表的头通常是一个独立的 list_head 结构。图 11-1 显示了如何使用简单的 struct list_head 来维护数据结构的列表。



在使用之前，必须使用 INIT_LIST_HEAD 宏初始化列表头。可以用以下方式声明和初始化一个“待办事项”列表头：

```c
struct list_head todo_list;
INIT_LIST_HEAD(&todo_list);
```

或者，列表可以在编译时初始化：

```c
LIST_HEAD(todo_list);
```

在 <linux/list.h> 中定义了几个与列表一起工作的函数：

```c
list_add(struct list_head *new, struct list_head *head);
```

将新条目立即添加到列表头后面——通常在列表的开始处。因此，它可以用来构建栈。但是，请注意，头不必是列表的名义头；如果你传递一个恰好在列表中间某处的 list_head 结构，新条目会立即在它后面。由于 Linux 列表是循环的，列表的头通常与其他条目没有区别。

```c
list_add_tail(struct list_head *new, struct list_head
*head);
```

在给定的列表头之前添加一个新条目——换句话说，是在列表的末尾。因此，list_add_tail 可以用来构建先进先出的队列。

```c
list_del(struct list_head *entry);

list_del_init(struct list_head *entry);
```

给定的条目从列表中删除。如果条目可能会被重新插入到另一个列表中，你应该使用 list_del_init，它会重新初始化链表指针。

```c
list_move(struct list_head *entry, struct list_head
*head);

list_move_tail(struct list_head *entry, struct list_head
*head);
```

给定的条目从当前列表中删除，并添加到 head 的开始。要将条目放在新列表的末尾，使用 list_move_tail。

```c
list_empty(struct list_head *head);
```

如果给定的列表为空，则返回非零值。

```c
list_splice(struct list_head *list, struct list_head
*head);
```

通过在 head 之后立即插入 list 来连接两个列表。

list_head 结构适合实现类似结构的列表，但调用程序通常更关心构成整个列表的较大结构。提供了一个宏，list_entry，它将 list_head 结构指针映射回包含它的结构的指针。它的调用方式如下：

```c
list_entry(struct list_head *ptr, type_of_struct,
field_name);
```

其中 ptr 是正在使用的 struct list_head 的指针，type_of_struct 是包含 ptr 的结构的类型，field_name 是结构中列表字段的名称。在我们之前的 todo_struct 结构中，列表字段简单地称为 list。因此，我们可以用如下行将列表条目转换为其包含的结构：

```c
struct todo_struct *todo_ptr =

    list_entry(listptr, struct todo_struct, list);
```

list_entry 宏需要一些时间来适应，但使用起来并不难。

遍历链表很简单：只需要跟随 prev 和 next 指针。例如，假设我们想要保持 todo_struct 项的列表按优先级降序排序。添加新条目的函数可能看起来像这样：

```c
void todo_add_entry(struct todo_struct *new)

{

    struct list_head *ptr;

    struct todo_struct *entry;

    for (ptr = todo_list.next; ptr != &todo_list; ptr =
ptr->next) {

        entry = list_entry(ptr, struct todo_struct,
list);

        if (entry->priority < new->priority) {

            list_add_tail(&new->list, ptr);

            return;

        }

    }

    list_add_tail(&new->list, &todo_list);

}
```

然而，一般来说，最好使用一组预定义的宏来创建遍历列表的循环。例如，前面的循环可以编码为：

```c
void todo_add_entry(struct todo_struct *new)
{
    struct list_head *ptr;
    struct todo_struct *entry;
    list_for_each(ptr, &todo_list) {
        entry = list_entry(ptr, struct todo_struct,
list);
        if (entry->priority < new->priority) {
            list_add_tail(&new->list, ptr);
            return;
        }
    }
    list_add_tail(&new->list, &todo_list);
}
```

这里，`list_for_each` 是一个宏，用于遍历链表。它的第一个参数是用于遍历的指针，第二个参数是链表的头。在每次迭代中，它都会更新 ptr 以指向下一个元素。

使用提供的宏有助于避免简单的编程错误；这些宏的开发者也已经付出了一些努力，以确保它们的性能良好。存在一些变体：

```c
list_for_each(struct list_head *cursor, struct list_head
*list)
```

此宏创建一个 for 循环，每次执行时，cursor 都指向列表中的连续条目。在遍历列表时要小心不要改变列表。

```c
list_for_each_prev(struct list_head *cursor, struct
list_head *list)
```

此版本通过列表进行反向迭代。

```c
list_for_each_safe(struct list_head *cursor, struct
list_head *next, struct list_head *list)
```

如果你的循环可能会删除列表中的条目，使用此版本。它只是在循环开始时将列表中的下一个条目存储在 next 中，所以如果 cursor 指向的条目被删除，它不会混淆。

```c
list_for_each_entry(type *cursor, struct list_head *list,
member)

list_for_each_entry_safe(type *cursor, type *next, struct
list_head *list, member)
```

这些宏简化了处理包含给定类型结构的列表的过程。这里，cursor 是指向包含结构类型的指针，member 是包含结构中的 list_head 结构的名称。使用这些宏，无需在循环内部调用 list_entry。

11.6. 快速参考

下列符号在本章中介绍了:

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

保证是 8-位, 16-位, 32-位 和64-位 无符号整型值的类型. 对等的有符号类型也存在. 在用户空间, 你可用 **u8,** u16, 等等来引用这些类型.

```
#include <asm/page.h>
PAGE_SIZE
PAGE_SHIFT
```

给当前体系定义每页的字节数, 以及页偏移的位数(对于 4 KB 页是 12, 8 KB 是 13)的符号.

```
#include <asm/byteorder.h>
__LITTLE_ENDIAN
__BIG_ENDIAN
```

这 2 个符号只有一个定义, 依赖体系.

```
#include <asm/byteorder.h>
u32 __cpu_to_le32 (u32);
u32 __le32_to_cpu (u32);
```

在已知字节序和处理器字节序之间转换的函数. 有超过 60 个这样的函数: 在 include/linux/byteorder/ 中的各种文件有完整的列表和它们以何种方式定义.

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

一些体系需要使用这些宏保护不对齐的数据存取. 这些宏定义扩展成通常的指针解引用, 为那些允许你存取不对齐数据的体系.

```
#include <linux/err.h>
void *ERR_PTR(long error);
long PTR_ERR(const void *ptr);
long IS_ERR(const void *ptr);
```

允许错误码由返回指针值的函数返回.

```
#include <linux/list.h>
list_add(struct list_head *new, struct list_head *head);
list_add_tail(struct list_head *new, struct list_head
*head);
list_del(struct list_head *entry);
list_del_init(struct list_head *entry);
list_empty(struct list_head *head);
list_entry(entry, type, member);
list_move(struct list_head *entry, struct list_head
*head);
list_move_tail(struct list_head *entry, struct list_head
*head);
list_splice(struct list_head *list, struct list_head
*head);
```

操作环形, 双向链表的函数.

```
list_for_each(struct list_head *cursor, struct list_head
*list)
list_for_each_prev(struct list_head *cursor, struct
list_head *list)
list_for_each_safe(struct list_head *cursor, struct
list_head *next, struct list_head *list)
list_for_each_entry(type *cursor, struct list_head *list,
member)
list_for_each_entry_safe(type *cursor, type *next struct
list_head *list, member)
```

方便的宏定义, 用在遍历链表上.