关于container_of和list_for_each_entry 及其相关函数的分析

Linux 代码看的比较多了，经常会遇到 container_of 和 list_for_each_entry，特别是 list_for_each_entry 比较多，因为 Linux 经常用到链表，虽然知道这些函数的大概意思，但一旦出现一个类似的函数比如 list_for_each_entry_safe 就又会感到头大，所以下定决心分析总结一下这些函数的用法，以后再看到这些面孔的时候也会轻松很多，读 Linux 代码的时候不会那么吃力。

我们知道 list_for_each_entry 会用到 list_entry，而 list_entry 用到 container_of，所以首先讲讲 container_of。

在讲 container_of 之前我们不得不提到 offsetof，因为在 container_of 中会使用到它，所以我们看下来，把 list_for_each_entry 函数的用法理顺我们对整个 Linux 中经常用到的一些函数就会比较清楚了。

1. **offsetof**

   #define offsetof(TYPE, MEMBER) ((size_t)&((TYPE *)0)->MEMBER)

   理解 offsetof 的关键在于&((TYPE *)0)->MEMBER，几乎可以说只要理解了这一部分，后面的几个函数都能够解决，那么我们看看这一部分究竟完成了怎样的工作。根据优先级的顺序，最里面的小括号优先级最高，TYPE *将整型常量 0 强制转换为 TYPE 型的指针，且这个指针指向的地址为 0，也就是将地址 0 开始的一块存储空间映射为 TYPE 型的对象，接下来再对结构体中 MEMBER 成员进行取址，而整个 TYPE 结构体的首地址是 0，这里获得的地址就是 MEMBER 成员在 TYPE 中的相对偏移量。再将这个偏移量强制转换成 size_t 型数据（无符号整型）。

   所以整个 offsetof 的功能就是获取 MEMBER 成员在 TYPE 型数据中的偏移量。接下来我们可以讲一下 container_of 了。

2. **container_of**

   /**
   * container_of - cast a member of a structure out to the containing structure
   * @ptr: the pointer to the member.
   * @type: the type of the container struct this is embedded in.

```
* @member:      the name of the member within the struct.
*
*/
#define container_of(ptr, type, member) ({                      \
     const typeof( ((type *)0)->member ) *__mptr = (ptr);     \
     (type *)( (char *)__mptr -
offsetof(type,member) );})
```

首先可以看出 container_of 被预定义成一个函数，函数的第一句话，通过((type *)0)->member 定义一个 MEMBER 型的指针__mptr，这个指针指向 ptr，所以第一句话获取到了我们要求的结构体，它的成员 member 的地址，接下来我们用这个地址减去成员 member 在结构体中的相对偏移量，就可以获取到所求结构体的地址， (char *)__mptr - offsetof(type,member)就实现了这个过程，最后再把这个地址强制转换成 type 型指针，就获取到了所求结构体指针，define 预定义返回最后一句话的值，将所求结构体指针返
回。

所以整个 container_of 的功能就是通过指向结构体成员 member 的指针 ptr 获取指向整个结构体的指针。container_of 清楚了，那 list_entry 就更是一目了然了。

3. **list_entry**

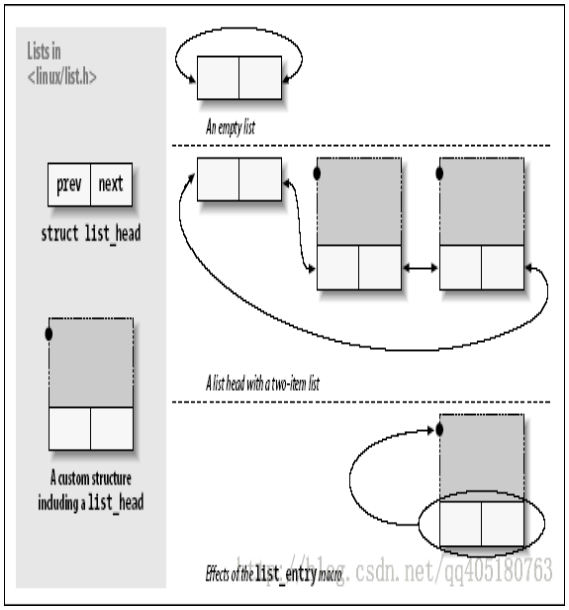
```
                              /**
* list_entry - get the struct for this entry
* @ptr:      the &struct list_head pointer.
* @type:      the type of the struct this is embedded in.
* @member:      the name of the list_struct within the struct.
*/
#define list_entry(ptr, type, member) \
      container_of\(ptr,type,member)
```

list_entry 的功能等同于 container_of。接下来分析我们最终想要知道的 list_for_each_entry 的实现过
程。

4. **list_for_each_entry**

```
                                                                    /**
* list_for_each_entry      -      iterate over list of given type
* @pos:      the type * to use as a loop cursor.
* @head:      the head for your list.
* @member:      the name of the list_struct within the struct.
*/
#define list_for_each_entry(pos, head, member)                    \
     for (pos = list_entry((head)->next, typeof(*pos), member);      \
          &pos->member != (head);       \
          pos = list_entry(pos->member.next, typeof(*pos), member))
```

在理解了 list_entry 的基础上分析 list_for_each_entry 本来是一件比较轻松的事情，但在这里还是要强调一下双向链表及链表头的概念，否则对 list_for_each_entry 的理解还是一知半解。建立一个双向链表通常有一个独立的用于管理链表的链表头，链表头一般是不含有实体数据的，必须用 INIT_LIST_HEAD（）进行初始化，表头建立以后，就可以将带有数据结构的实体链表成员加入到链表中，链表头和链表的关系如图所示：



链表头和链表的关系清楚了，我们才能完全理解 list_for_each_entry。list_for_each_entry 被预定义成一个 for 循环语句，for 循环的第一句话获取(head)->next 指向的 member 成员的数据结构指针，也就是将 pos 初始化为除链表头之外的第一个实体链表成员，for 的第三句话通过 pos->member.next 指针遍历整个实体链表，当 pos->member.next 再次指向我们的链表头的时候跳出 for 循环。整个过程没有对链表头进行遍历（不需要被遍历），所以使用 list_for_each_entry 遍

历链表必须从链表头开
始。

因此可以看出，list_for_each_entry 的功能就是遍历以 head 为链表头的实体链表，对实体链表中的数据结构进行处理。

5. **list_for_each_entry_safe**

```
/**
* list_for_each_entry_safe - iterate over list of given type safe against removal of list entry
* @pos:     the type * to use as a loop cursor.
* @n:          another type * to use as temporary storage
* @head:     the head for your list.
* @member:     the name of the list_struct within the struct.
*/
#define list_for_each_entry_safe(pos, n, head, member)              \
    for (pos = list_entry((head)->next, typeof(*pos), member),     \
        n = list_entry(pos->member.next, typeof(*pos), member);     \
         &pos->member != (head);                    \
         pos = n, n = list_entry(n->member.next, typeof(*n), member))
```

相比于 list_for_each_entry，list_for_each_entry_safe 用指针 n 对链表的下一个数据结构进行了临时存储，所以如果在遍历链表的时候可能要删除链表中的当前项，用 list_for_each_entry_safe 可以安全的删除，而不会影响接下来的遍历过程（用 n 指针可以继续完成接下来的遍历， 而 list_for_each_entry 则无法继续遍历）。