Linux内核部件分析

连通世界的list

[日期：2011-10-06] 来源：Linux社区 作者：qb_2008 [字体：大 中 小]

在linux内核中，有一种通用的双向循环链表，构成了各种队列的基础。链表的结构定义和相关函数均在include/linux/list.h中，下面就来全面的介绍这一链表的各种API。

```
1. struct list_head {
2.     struct list_head *next, *prev;
3. };
```

这是链表的元素结构。因为是循环链表，表头和表中节点都是这一结构。有prev和next两个指针，分别指向链表中前一节点和后一节点。

```
1. #define LIST_HEAD_INIT(name) { &(name), &(name) }
2.
3. #define LIST_HEAD(name) \
4.     struct list_head name = LIST_HEAD_INIT(name)
5.
6. static inline void INIT_LIST_HEAD(struct list_head *list)
7. {
8.     list->next = list;
9.     list->prev = list;
10. }
```

在初始化的时候，链表头的prev和next都是指向自身的。

```
1. static inline void __list_add(struct list_head *new,
2.                               struct list_head *prev,
3.                               struct list_head *next)
4. {
5.     next->prev = new;
6.     new->next = next;
7.     new->prev = prev;
8.     prev->next = new;
9. }
10.
11. static inline void list_add(struct list_head *new, struct list_head *head)
12. {
13.     __list_add(new, head, head->next);
14. }
15.
16. static inline void list_add_tail(struct list_head *new, struct list_head *head)
17. {
18.     __list_add(new, head->prev, head);
19. }
```

双向循环链表的实现，很少有例外情况，基本都可以用公共的方式来处理。这里无论是加第一个节点，还是其它的节点，使用的方法都一样。

另外，链表API实现时大致都是分为两层：一层外部的，如list_add、list_add_tail，用来消除一些例外情况，调用内部实现；一层是内部的，函数名前会加双下划线，如__list_add，往往是几个操作公共的部分，或者排除例外后的实现。

```
1. static inline void __list_del(struct list_head * prev, struct list_head * next)
2. {
3.     next->prev = prev;
4.     prev->next = next;
5. }
6.
7. static inline void list_del(struct list_head *entry)
8. {
9.     __list_del(entry->prev, entry->next);
10.     entry->next = LIST_POISON1;
11.     entry->prev = LIST_POISON2;
```

```
}

static inline void list_del_init(struct list_head *entry)
{
    __list_del(entry->prev, entry->next);
    INIT_LIST_HEAD(entry);
}
```

list_del是链表中节点的删除。之所以在调用__list_del后又把被删除元素的next、prev指向特殊的LIST_POSITION1和LIST_POSITION2，是为了调试未定义的指针。

list_del_init则是删除节点后，随即把节点中指针再次初始化，这种删除方式更为实用。

```
static inline void list_replace(struct list_head *old,
                struct list_head *new)
{
    new->next = old->next;
    new->next->prev = new;
    new->prev = old->prev;
    new->prev->next = new;
}

static inline void list_replace_init(struct list_head *old,
                    struct list_head *new)
{
    list_replace(old, new);
    INIT_LIST_HEAD(old);
}
```

list_replace是将链表中一个节点old，替换为另一个节点new。从实现来看，即使old所在地链表只有old一个节点，new也可以成功替换，这就是双向循环链表可怕的通用之处。

list_replace_init将被替换的old随即又初始化。

```
static inline void list_move(struct list_head *list, struct list_head *head)
{
    __list_del(list->prev, list->next);
    list_add(list, head);
}

static inline void list_move_tail(struct list_head *list,
                  struct list_head *head)
{
    __list_del(list->prev, list->next);
    list_add_tail(list, head);
}
```

list_move的作用是把list节点从原链表中去除，并加入新的链表head中。

list_move_tail只在加入新链表时与list_move有所不同，list_move是加到head之后的链表头部，而list_move_tail是加到head之前的链表尾部。

```
static inline int list_is_last(const struct list_head *list,
                const struct list_head *head)
{
    return list->next == head;
}
```

list_is_last 判断list是否处于head链表的尾部。

```
static inline int list_empty(const struct list_head *head)
{
    return head->next == head;
}

static inline int list_empty_careful(const struct list_head *head)
{
    struct list_head *next = head->next;
    return (next == head) && (next == head->prev);
}
```

list_empty 判断head链表是否为空，为空的意思就是只有一个链表头head。

list_empty_careful 同样是判断head链表是否为空，只是检查更为严格。

```
static inline int list_is_singular(const struct list_head *head)
```

```
2. {
3.     return !list_empty(head) && (head->next == head->prev);
4. }
```

list_is_singular 判断head中是否只有一个节点，即除链表头head外只有一个节点。

```
1. static inline void __list_cut_position(struct list_head *list,
2.         struct list_head *head, struct list_head *entry)
3. {
4.     struct list_head *new_first = entry->next;
5.     list->next = head->next;
6.     list->next->prev = list;
7.     list->prev = entry;
8.     entry->next = list;
9.     head->next = new_first;
10.    new_first->prev = head;
11. }
12.
13. static inline void list_cut_position(struct list_head *list,
14.         struct list_head *head, struct list_head *entry)
15. {
16.     if (list_empty(head))
17.         return;
18.     if (list_is_singular(head) &&
19.         (head->next != entry && head != entry))
20.         return;
21.     if (entry == head)
22.         INIT_LIST_HEAD(list);
23.     else
24.         __list_cut_position(list, head, entry);
25. }
```

list_cut_position 用于把head链表分为两个部分。从head->next一直到entry被从head链表中删除，加入新的链表list。新链表list应该是空的，或者原来的节点都可以被忽略掉。可以看到，list_cut_position中排除了一些意外情况，保证调用__list_cut_position时至少有一个元素会被加入新链表。

```
1. static inline void __list_splice(const struct list_head *list,
2.                  struct list_head *prev,
3.                  struct list_head *next)
4. {
5.     struct list_head *first = list->next;
6.     struct list_head *last = list->prev;
7.
8.     first->prev = prev;
9.     prev->next = first;
10.
11.    last->next = next;
12.    next->prev = last;
13. }
14.
15. static inline void list_splice(const struct list_head *list,
16.                  struct list_head *head)
17. {
18.     if (!list_empty(list))
19.         __list_splice(list, head, head->next);
20. }
21.
22. static inline void list_splice_tail(struct list_head *list,
23.                  struct list_head *head)
24. {
25.     if (!list_empty(list))
26.         __list_splice(list, head->prev, head);
27. }
```

list_splice的功能和list_cut_position正相反，它合并两个链表。list_splice把list链表中的节点加入head链表中。在实际操作之前，要先判断list链表是否为空。它保证调用__list_splice时list链表中至少有一个节点可以被合并到head链表中。

list_splice_tail只是在合并链表时插入的位置不同。list_splice是把原来list链表中的节点全加到head链表的头部，而list_splice_tail则是把原来list链表中的节点全加到head链表的尾部。

```
1. static inline void list_splice_init(struct list_head *list,
2.                  struct list_head *head)
3. {
```

```
4.    if (!list_empty(list)) {
5.        __list_splice(list, head, head->next);
6.        INIT_LIST_HEAD(list);
7.    }
8. }
9.
10. static inline void list_splice_tail_init(struct list_head *list,
11.                     struct list_head *head)
12. {
13.    if (!list_empty(list)) {
14.        __list_splice(list, head->prev, head);
15.        INIT_LIST_HEAD(list);
16.    }
17. }
```

list_splice_init 除了完成list_splice的功能，还把变空了的list链表头重新初始化。

list_splice_tail_init 除了完成list_splice_tail的功能，还吧变空了得list链表头重新初始化。

list操作的API大致如以上所列，包括链表节点添加与删除、节点从一个链表转移到另一个链表、链表中一个节点被替换为另一个节点、链表的合并与拆分、查看链表当前是否为空或者只有一个节点。接下来，是操作链表遍历时的一些宏，我们也简单介绍一下。

```
1. #define list_entry(ptr, type, member) \
2.     container_of(ptr, type, member)
```

list_entry主要用于从list节点查找其内嵌在的结构。比如定义一个结构struct A{ struct list_head list; }; 如果知道结构中链表的地址ptrList，就可以从ptrList进而获取整个结构的地址(即整个结构的指针) struct A *ptrA = list_entry(ptrList, struct A, list);

这种地址翻译的技巧是linux的拿手好戏，container_of随处可见，只是链表节点多被封装在更复杂的结构中，使用专门的list_entry定义也是为了使用方便。

```
1. #define list_first_entry(ptr, type, member) \
2.     list_entry((ptr)->next, type, member)
```

list_first_entry是将ptr看完一个链表的链表头，取出其中第一个节点对应的结构地址。使用list_first_entry是应保证链表中至少有一个节点。

```
1. #define list_for_each(pos, head) \
2.     for (pos = (head)->next; prefetch(pos->next), pos != (head); \
3.             pos = pos->next)
```

list_for_each循环遍历链表中的每个节点，从链表头部的第一个节点，一直到链表尾部。中间的prefetch是为了利用平台特性加速链表遍历，在某些平台下定义为空，可以忽略。

```
1. #define __list_for_each(pos, head) \
2.     for (pos = (head)->next; pos != (head); pos = pos->next)
```

__list_for_each与list_for_each没什么不同，只是少了prefetch的内容，实现上更为简单易懂。

```
1. #define list_for_each_prev(pos, head) \
2.     for (pos = (head)->prev; prefetch(pos->prev), pos != (head); \
3.             pos = pos->prev)
```

list_for_each_prev与list_for_each的遍历顺序相反，从链表尾逆向遍历到链表头。

```
1. #define list_for_each_safe(pos, n, head) \
2.     for (pos = (head)->next, n = pos->next; pos != (head); \
3.         pos = n, n = pos->next)
```

list_for_each_safe 也是链表顺序遍历，只是更加安全。即使在遍历过程中，当前节点从链表中删除，也不会影响链表的遍历。参数上需要加一个暂存的链表节点指针n。

```
1. #define list_for_each_prev_safe(pos, n, head) \
2.     for (pos = (head)->prev, n = pos->prev; \
3.          prefetch(pos->prev), pos != (head); \
4.          pos = n, n = pos->prev)
```

list_for_each_prev_safe 与list_for_each_prev同样是链表逆序遍历，只是加了链表节点删除保护。

```
1. #define list_for_each_entry(pos, head, member)              \
2.     for (pos = list_entry((head)->next, typeof(*pos), member);   \
3.          prefetch(pos->member.next), &pos->member != (head);  \
4.          pos = list_entry(pos->member.next, typeof(*pos), member))
```

list_for_each_entry不是遍历链表节点，而是遍历链表节点所嵌套进的结构。这个实现上较为复杂，但可以等价于list_for_each加上list_entry的组合。

```
1. #define list_for_each_entry_reverse(pos, head, member)          \
2.     for (pos = list_entry((head)->prev, typeof(*pos), member);   \
3.          prefetch(pos->member.prev), &pos->member != (head);  \
```

```
4.          pos = list_entry(pos->member.prev, typeof(*pos), member))
```

list_for_each_entry_reverse 是逆序遍历链表节点所嵌套进的结构，等价于list_for_each_prev加上list_etnry的组合。

```
1. #define list_for_each_entry_continue(pos, head, member)         \
2.     for (pos = list_entry(pos->member.next, typeof(*pos), member);   \
3.          prefetch(pos->member.next), &pos->member != (head);  \
4.          pos = list_entry(pos->member.next, typeof(*pos), member))
```

list_for_each_entry_continue也是遍历链表上的节点嵌套的结构。只是并非从链表头开始，而是从结构指针的下一个结构开始，一直到链表尾部。

```
1. #define list_for_each_entry_continue_reverse(pos, head, member)     \
2.     for (pos = list_entry(pos->member.prev, typeof(*pos), member);   \
3.          prefetch(pos->member.prev), &pos->member != (head);  \
4.          pos = list_entry(pos->member.prev, typeof(*pos), member))
```

list_for_each_entry_continue_reverse 是逆序遍历链表上的节点嵌套的结构。只是并非从链表尾开始，而是从结构指针的前一个结构开始，一直到链表头部。

```
1. #define list_for_each_entry_from(pos, head, member)             \
2.     for (; prefetch(pos->member.next), &pos->member != (head);    \
3.          pos = list_entry(pos->member.next, typeof(*pos), member))
```

list_for_each_entry_from 是从当前结构指针pos开始，顺序遍历链表上的结构指针。

```
1. #define list_for_each_entry_safe(pos, n, head, member)          \
2.     for (pos = list_entry((head)->next, typeof(*pos), member),   \
3.          n = list_entry(pos->member.next, typeof(*pos), member);  \
4.          &pos->member != (head);                    \
5.          pos = n, n = list_entry(n->member.next, typeof(*n), member))
```

list_for_each_entry_safe 也是顺序遍历链表上节点嵌套的结构。只是加了删除节点的保护。

```
1. #define list_for_each_entry_safe_continue(pos, n, head, member)         \
2.     for (pos = list_entry(pos->member.next, typeof(*pos), member),      \
3.          n = list_entry(pos->member.next, typeof(*pos), member);     \
4.          &pos->member != (head);                    \
5.          pos = n, n = list_entry(n->member.next, typeof(*n), member))
```

list_for_each_entry_safe_continue 是从pos的下一个结构指针开始，顺序遍历链表上的结构指针，同时加了节点删除保护。

```
1. #define list_for_each_entry_safe_from(pos, n, head, member)             \
2.     for (n = list_entry(pos->member.next, typeof(*pos), member);     \
3.          &pos->member != (head);                    \
4.          pos = n, n = list_entry(n->member.next, typeof(*n), member))
```

list_for_each_entry_safe_from 是从pos开始，顺序遍历链表上的结构指针，同时加了节点删除保护。

```
1. #define list_for_each_entry_safe_reverse(pos, n, head, member)      \
2.     for (pos = list_entry((head)->prev, typeof(*pos), member),   \
3.          n = list_entry(pos->member.prev, typeof(*pos), member);  \
4.          &pos->member != (head);                    \
5.          pos = n, n = list_entry(n->member.prev, typeof(*n), member))
```

list_for_each_entry_safe_reverse 是从pos的前一个结构指针开始，逆序遍历链表上的结构指针，同时加了节点删除保护。

至此为止，我们介绍了linux中双向循环链表的结构、所有的操作函数和遍历宏定义。相信以后在linux代码中遇到链表的使用，不会再陌生。

1 2 3 4 5 6 7 8 9 下一页 2