在 linux 内核中，有大量的数据结构需要用到双循环链表，例如进程、文件、模块、页面等。若采用双循环链表的传统实现方式，需要为这些数据结构维护各自的链表，并且为每个链表都要设计插入、删除等操作函数。因为用来维持链表的 next 和 prev 指针指向对应类型的对象，因此一种数据结构的链表操作函数不能用于操作其它数据结构的链表。

在 Linux 源代码树的 include/linux/list.h 文件中，采用了一种类型无关的双循环链表实现方式。其思想是将指针 prev 和 next 从具体的数据结构中提取出来构成一种通用的"双链表"数据结构 list_head。如果需要构造某类对象的特定链表，则在其结构（被称为宿主数据结构）中定义一个类型为 list_head 类型的成员，通过这个成员将这类对象连接起来，形成所需链表，并通过通用链表函数对其进行操作。其优点是只需编写通用链表函数，即可构造和操作不同对象的链表，而无需为每类对象的每种列表编写专用函数，实现了代码的重用。

list_head 结构

----------struct list_head{}及初始化宏---------

```
struct list_head

{

        struct list_head *next, *prev;

};
```

当用此类型定义一个独立的变量时，其为头结点。当其为某个结构体的一个成员时，其为普通结点。尽管形式一样，但表达的意义不同，是否应该定义为两个类型 list_head 和 list_node？？？无法分开，空链表时指向了自己

list_head 成员作为"连接件"，把宿主数据结构链接起来。如下图所示：

在 Linux 内核中的双循环链表实现方式下：

1. list_head 类型的变量作为一个成员嵌入到宿主数据结构内；

2. 可以将链表结构放在宿主结构内的任何地方；

3. 可以为链表结构取任何名字；

4. 宿主结构可以有多个链表结构；

5. 用 list_head 中的成员和相对应的处理函数来对链表进行遍历；

6. 如果想得到宿主结构的指针，使用 list_entry 可以算出来。

3、定义和初始化

--LIST_HEAD_INIT()--LIST_HEAD()--INIT_LIST_HEAD()------

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \

        struct list_head name = LIST_HEAD_INIT(name)
```

需要注意的是，Linux 的每个双循环链表都有一个链表头，链表头也是一个节点，只不过它不嵌入到宿主数据结构中，即不能利用链表头定位到对应的宿主结构，但可以由之获得虚拟的宿主结构指针。

LIST_HEAD()宏可以同时完成定义链表头，并初始化这个双循环链表为空。

静态定义一个 list_head 类型变量，该变量一定为头节点。name 为 struct list_head{}类型的一个变量，&(name)为该结构体变量的地址。用 name 结构体变量的始地址将该结构体变量进行初始化。

```
#define INIT_LIST_HEAD(ptr) do { \

        (ptr)->next = (ptr); (ptr)->prev = (ptr); \

} while (0)
```

动态初始化一个已经存在的 list_head 对象，ptr 为一个结构体的指针，这样可以初始化堆栈以及全局区定义的 list_head 对象。ptr 使用时候，当用括号，(ptr)，避免 ptr 为表达式时宏扩展带来的异常问题。此宏很少用于动态初始化内嵌的 list 对象，主要是链表合并或者删除后重新初始化头部。若是在堆中申请了这个链表头，调用 INIT_LIST_HEAD()宏初始化链表节点，将 next 和 prev 指针都指向其自身，我们就构造了一个空的双循环链表。

2.6 内核中内联函数版本如下：

```
static inline void INIT_LIST_HEAD(struct list_head *list)

{

        list->next = list;

        list->prev = list;

}
```

此时的参数有明确的类型信息 struct list_head，同时可以看出其为指针，list 无须象宏中那样（），即使参数为表达式，其也是求值后再作为参数传入的。内联函数有严格的参数类型检查，同时不会出现宏函数扩展带来的异常问题，但是运行效率和空间效率与宏函数一致。

4、通用链表操作接口

4.1 添加节点

在上面的设计下，所有链表（包括添加、删除、移动和拼接等）操作都是针对数据结构 list_head 进行的。提供给用户的的添加链表的操作有两种：表头添加和表尾添加。注意到，Linux 双循环链表中有一个链表头，表头添加是指添加到链表头之后，而表尾添加则是添加到链表头的 prev 所指链表节点之后。

----------__list_add()------list_add()------list_add_tail()-------

```
static inline void __list_add(struct list_head *new, struct list_head *prev,    struct list_head *next)

{

        next->prev = new;

        new->next = next;

        new->prev = prev;

        prev->next = new;

}
```

普通的在两个非空结点中插入一个结点，注意 new、prev、next 都不能是空值。

Prev 可以等于 next，此时在只含头节点的链表中插入新节点。

```
static inline void list_add(struct list_head *new, struct    list_head *head)

{

        __list_add(new, head, head->next);

}
```

在 head 和 head->next 两指针所指向的结点之间插入 new 所指向的结点。

即：在 head 指针后面插入 new 所指向的结点。Head 并非一定为头结点。

当现有链表只含有一个头节点时，上述__list_add(new, head, head->next)仍然成立。

```
static inline void list_add_tail(struct list_head *new, struct list_head *head)

{

        __list_add(new, head->prev, head);
```

}

在结点指针 head 所指向结点的前面插入 new 所指向的结点。当 head 指向头节点时，也相当于在尾结点后面增加一个 new 所指向的结点。

注意：

head->prev 不能为空，即若 head 为头结点，其 head->prev 当指向一个数值，一般为指向尾结点，构成循环链表。

上述三个函数实现了添加一个节点的任务，其中__list_add()为底层函数，“__”通常表示该函数是底层函数，供其他模块调用，此处实现了较好的代码复用，list_add 和 list_add_tail 虽然原型一样，但调用底层函数__list_add 时传递了不同的参数，从而实现了在 head 指向节点之前或之后添加新的对象。

4.2 删除节点

如果要从链表中删除某个链表节点，则可以调用 list_del 或 list_del_init。

需要注意的是，上述操作均仅仅是把节点从双循环链表中拿掉，用户需要自己负责释放该节点对应的数据结构所占用的空间，而这个空间本来就是用户分配的。

----------__list_del()---list_del()-----list_del_init()----------

```
static inline void __list_del(struct list_head * prev, struct list_head * next)

{

        next->prev = prev;

        prev->next = next;

}
```

在 prev 和 next 指针所指向的结点之间，两者互相所指。在后面会看到：prev 为待删除的结点的前面一个结点，next 为待删除的结点的后面一个结点。

```
static inline void list_del(struct list_head *entry)

{

        __list_del(entry->prev, entry->next);

        entry->next = LIST_POISON1;

        entry->prev = LIST_POISON2;
```

}

删除 entry 所指的结点，同时将 entry 所指向的结点指针域封死。

对 LIST_POISON1,LIST_POISON2 的解释说明：

Linux 内核中解释：These are non-NULL pointers that will result in page faults under normal circumstances, used to verify that nobody uses non-initialized list entries.

#define LIST_POISON1 ((void *) 0x00100100)

#define LIST_POISON2 ((void *) 0x00200200)

常规思想是：entry->next = NULL; entry->prev = NULL; 保证不可通过该节点进行访问。

----------list_del_init()----------

static inline void list_del_init(struct list_head *entry)

{

__list_del(entry->prev, entry->next);

INIT_LIST_HEAD(entry);

}

删除 entry 所指向的结点，同时调用 LIST_INIT_HEAD()把被删除节点为作为链表头构建一个新的空双循环链表。

4.3 移动节点

Linux 还提供了两个移动操作：list_move 和 list_move_tail。

----------list_move()--list_move_tail()----------

static inline void list_move(struct list_head *list, struct list_head *head)

{

__list_del(list->prev, list->next);

list_add(list, head);

}

将 list 结点前后两个结点互相指向彼此，删除 list 指针所指向的结点，再将此结点插入 head，和 head->next 两个指针所指向的结点之间。

即：将 list 所指向的结点移动到 head 所指向的结点的后面。

```
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);

}
```

删除了 list 所指向的结点，将其插入到 head 所指向的结点的前面，如果 head->prev 指向链表的尾结点的话，就是将 list 所指向的结点插入到链表的结尾。

4.4 链表判空

由 list-head 构成的双向循环链表中，通常有一个头节点，其不含有有效信息，初始化时 prev 和 next 都指向自身。判空操作是判断除了头节点外是否有其他节点。

```
----------------------list_empty（）-------------

static inline int list_empty(const struct list_head *head)

{

        return head->next == head;

}
```

测试链表是否为空，如果是只有一个结点，head，head->next，head->prev 都指向同一个结点，则这里会返回 1，表示空；但这个空不是没有任何结点，而是只有一个头结点，因为头节点只是纯粹的 list 节点，没有有效信息，故认为为空。

```
--------------------list_empty_careful（）---------

static inline int list_empty_careful(const struct list_head *head)

{

        struct list_head *next = head->next;

        return (next == head) && (next == head->prev);
```

}

分析：

1.只有一个头结点 head，这时 head 指向这个头结点，head->next,head->prev 指向 head，即：head==head->next==head->prev，这时候 list_empty_careful（）函数返回 1。

2.有两个结点，head 指向头结点，head->next,head->prev 均指向后面那个结点，即：head->next==head->prev,而 head!=head->next,head!=head->prev.所以函数将返回 0

3.有三个及三个以上的结点，这是一般的情况，自己容易分析了。

注意：这里 empty list 是指只有一个空的头结点，而不是毫无任何结点。并且该头结点必须其 head->next==head->prev==head

4.5 链表合并

Linux 还支持两个链表的拼接，提供给用户的具体函数是 list_splice 和 list_splice_init：

----------__list_splice()----------

```
static inline void __list_splice(struct list_head *list,          struct list_head *head)
{
        struct list_head *first = list->next;
        struct list_head *last    = list->prev;
        struct list_head *at       = head->next;
        first->prev = head;
        head->next = first;
        last->next = at;
        at->prev = last;
}
```

将一个非空链表插入到另外一个链表中。不作链表是否为空的检查，由调用者默认保证。因为每个链表只有一个头节点，将空链表插入到另外一个链表中是没有意义的。但被插入的链表可以是空的。

----------list_splice()----------

```
/**
* list_splice - join two lists
* @list:  被合并的链表的头节点.
* @head: the place to add it in the first list.
*/
static inline void list_splice(struct list_head *list, struct list_head *head)
{
        if (!list_empty(list))
                __list_splice(list, head);
}
```

这种情况会丢弃 list 所指向的头结点，这是特意设计的，因为两个链表有两个头结点，要去掉一个头结点。只要 list 非空链，head 无任何限制，该程序都可以实现链表合并。

```
----------list_splice_init()----------
/**
* list_splice_init - join two lists and reinitialise the emptied list.
* @list: the new list to add.
* @head: the place to add it in the first list.
*
* The list at @list is reinitialised
*/
static inline void list_splice_init(struct list_head *list,
                                    struct list_head *head)
{
```

```
        if (!list_empty(list)) 0
        {
                __list_splice(list, head);
                INIT_LIST_HEAD(list);
        }
}
```

将一个链表的有效信息合并到另外一个链表后，重新初始化空的链表头。

5、获取宿主对象指针

如果需要有某种数据结构的队列，就在这种数据结构定义内部放上一个 list_head 数据结构。例如，建立数据结构 foo 链表的方式是，在 foo 的定义中，嵌入了一个 list_head 成员 list。这里 foo 就是所指的"宿主"。

```
typedef struct foo {
    …
    struct list_head list;
    …
};
```

但是，如何通过 list_head 成员访问到宿主结构项呢？毕竟 list_head 不过是个连接件，而我们需要的是一个"特定"的数据结构链表。

先介绍几个基本宏：offsetof、typeof、containerof

-------\linux\stddef.h-----offsetof()----------

```
#define __compiler_offsetof(a,b)   __builtin_offsetof(a,b)
```

而__builtin_offsetof()宏就是在编译器中已经设计好了的函数，直接调用即可。

```
#undef offsetof   //取消先前的任何定义，可以保证下面的定义生效
#ifdef __compiler_offsetof
```

```
#define offsetof(TYPE,MEMBER)    __compiler_offsetof(TYPE,MEMBER)

#else

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

#endif
```

一共 4 步

1. ((TYPE *)0) 0 地址强制 "转换" 为 TYPE 结构的指针;

2. ((TYPE *)0)->MEMBER　　访问结构中的数据成员;

3.　&(((TYPE *)0)->MEMBER)取出数据成员的地址;

4.(size_t)(&(((TYPE*)0)->MEMBER))结果转换类型.巧妙之处在于将 0 转换成(TYPE*)，结构以内存空间首地址 0 作为起始地址，则成员地址自然为偏移地址；

举例说明：

```
#include<stdio.h>

typedef struct _test

{

        char i;

        int j;

        char k;

}Test;

int main()

{

        Test *p = 0;

        printf("%p\n", &(p->k));

}
```

这里使用的是一个利用编译器技术的小技巧（编译器自动算出成员的偏移量），即先求得结构成员变量在结构体中的相对于结构体的首地址的偏移地址，然后根据结构体的首地址为 0，从而得出该偏移地址就是该结构体变量在该结构体中的偏移，即：该结构体成员变量距离结构体首的距离。在 offsetof()中，这个 member 成员的地址实际上就是 type 数据结构中 member 成员相对于结构变量的偏移量。对于给定一个结构，offsetof(type,member)是一个常量，list_entry()正是利用这个不变的偏移量来求得链表数据项的变量地址。

----------typeof()----------

unsigned int i;

typeof(i) x;

x=100;

printf("x:%d\n",x);

typeof() 是 gcc 的扩展，和 sizeof() 类似。

在 container_of 宏中，它用来给 typeof() 提供参数，以获得 member 成员的数据类型；

----------container_of()----------

container_of() 来自\linux\kernel.h

内核中的注释：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) );})
```

自己分析：

1.(type *)0->member 为设计一个 type 类型的结构体，起始地址为 0，编译器将结构体的起始的地址加上此结构体成员变量的偏移得到此结构体成员变量的地址，由于结构体起始地址为 0，所以此结构体成员变量的偏移地址就等于其成员变量在结构体内距离结构体开始部分的偏移量。即： &(type *)0->member

就是取出其成员变量的偏移地址。而其等于其在结构体内的偏移量:即为：(size_t)(& ((type *)0)->member)经过 size_t 的强制类型转换后，其数值为结构体内的偏移量。该偏移量这里由 offsetof()求出。

2.typeof(((type *)0)->member)为取出 member 成员的变量类型。用其定义__mptr 指针.ptr 为指向该成员变量的指针。__mptr 为 member 数据类型的常量指针，其指向 ptr 所指向的变量处。

3.(char *)__mptr 转换为字节型指针。(char *)__mptr - offsetof(type,member))用来求出结构体起始地址（为 char *型指针），然后(type *)((char *)__mptr - offsetof(type,member))在(type *)作用下进行将字节型的结构体起始指针转换为 type *型的结构体起始指针。

这就是从结构体某成员变量指针来求出该结构体的首指针。指针类型从结构体某成员变量类型转换为该结构体类型。

介绍了上面的几种基本宏后，对 list_entry 的理解就容易了。

----------list_entry()----------

list_entry()宏，获取当前 list_head 链表节点所在的宿主结构项。第一个参数为当前 list_head 节点的指针，即指向宿主结构项的 list_head 成员。第二个参数是宿主数据结构的定义类型。第三个参数为宿主结构类型定义中 list_head 成员名。

```
#define list_entry(ptr, type, member) \

        container_of(ptr, type, member)
```

扩展替换即为：

```
#define list_entry(ptr, type, member) \

    ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

例如，我们要访问 foo 链表（链表头为 head）中首个元素，则如此调用：

```
list_entry(head->next, struct foo, list);
```

经过 C 预处理的文字替换，这一行的内容就成为：

```
((struct foo *)((char *)(head->next) - (unsigned long)(&((struct foo *)0)->list)))
```

获取宿主对象指针的原理如上图所示。我们考虑 list_head 类型成员 member 相对于宿主结构（类型为 type）起始地址的偏移量。对于所有该类型的宿主对象，这个偏移量是固定的。并且可以在假设宿主对象地址值为 0，通过返回 member 成员的地址获得，即等于(unsigned long)(&((type *)0)->member)。这样，将当前宿主对象的"连接件"地址（ptr）减去这个偏移量，得到宿主对象地址，再将它转换为宿主数据结构类型的指针。

需要重申的是，链表头没有被嵌入到宿主对象中，因此对链表头执行宿主对象指针获取操作是没有意义

的。

6、遍历

6.1 List-head 链表遍历

遍历是双循环链表的基本操作，为此 Linux 定义了一些宏。

list_for_each 对遍历链表中的所有 list_head 节点，不涉及到对宿主结构的处理。list_for_each 实际是一个 for 循环，利用传入的指向 list_head 结构的指针作为循环变量，从链表头开始（并跳过链表头），逐项向后移动指针，直至又回到链表头。

----------list_for_each()----------

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

head 为头节点，遍历过程中首先从(head)->next 开始，当 pos＝＝head 时退出，故 head 节点并没有访问，这和 list 结构设计有关，通常头节点就是纯粹的 list 结构，不含有其他有效信息，或者头节点含有其他信息，如内核 PCB 链表中的头节点为 idle 任务，但其不参予比较优先级，因此此时头节点只是作为双向链表遍历一遍的检测标志。

为提高遍历速度，还使用了预取。

-----asm-x86_64\processor.h---prefetch()---------

```
static inline void prefetch(void *x)
{
        asm volatile("prefetcht0 %0" :: "m" (*(unsigned long *)x));
}
```

将 x 指针作强制类型转换为 unsigned long *型，然后取出该内存操作数，送入高速缓存。

----------__list_for_each()----------

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

list_for_each()有 prefetch()用于复杂的表的遍历，而__list_for_each()无 prefetch()用于简单的表的遍历，此时表项比较少，无需缓存。

----------list_for_each_prev()----------

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

反向遍历节点

----------list_for_each_safe()----------

如果在遍历过程中，包含有删除或移动当前链接节点的操作，由于这些操作会修改遍历指针，这样会导致遍历的中断。这种情况下，必须使用 list_for_each_safe 宏，在操作之前将遍历指针缓存下来：

内核中解释的精华部分：

```
/*
* list_for_each_safe    -    iterate over a list safe against removal of list entry
   */
#define list_for_each_safe(pos, n, head) \
        for (pos = (head)->next, n = pos->next; pos != (head); \
                pos = n, n = pos->next)
```

在 for 循环中 n 暂存 pos 下一个节点的地址，避免因 pos 节点被释放而造成的断链。也就是说你可以遍历完当前节点后将其删除，同时可以接着访问下一个节点，遍历完毕后就只剩下一个头节点。这就叫 safe。十分精彩。典型用途是多个进程等待在同一个等待队列上，若事件发生时唤醒所有进程，则可以唤醒后将其依次从等待队列中删除。

6.2 遍历宿主对象

如果只提供对 list_head 结构的遍历操作是远远不够的，我们希望实现的是对宿主结构的遍历，即在遍历时直接获得当前链表节点所在的宿主结构项，而不是每次要同时调用 list_for_each 和 list_entry。对此，Linux 提供了 list_for_each_entry()宏，第一个参数为传入的遍历指针，指向宿主数据结构，第二个参数为链表头，为 list_head 结构，第三个参数为 list_head 结构在宿主结构中的成员名。

----------list_for_each_entry()----------

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

这是用于嵌套的结构体中的宏：

```
struct example_struct
{
        struct list_head list;
        int priority;
        ... //其他结构体成员
};
struct example_struct *node = list_entry(ptr,struct example_struct,list);
```

自己分析：对比 list_entry(ptr,type,member)可知有以下结果：

其中 list 相当于 member 成员，struct example_struct 相当于 type 成员，ptr 相当于 ptr 成员。而 list{}成员嵌套于 example_struct{}里面。ptr 指向 example_struct{}中的 list 成员变量的。在 list_entry（）作用下，将 ptr 指针回转指向 struct example_struct{}结构体的开始处。

pos 当指向外层结构体，比如指向 struct example_struct{}的结点，最开始时候，head 指向链表结构体 struct list_head{}的头结点，头节点不含有有效信息，(head)->next 则指向第一个外层结点的内嵌的链表结点 struct list_head{} list，由此得出的 pos 当指向第一个有效结点。member 即是指出该 list 为其内嵌的结点。

思路：用 pos 指向外层结构体的结点，用 head 指向内层嵌入的结构体的结点。用(head)->next,pos->member.next(即：ptr->list.next)来在内嵌的结构体结点链表中遍历。每遍历一个结点，就用 list_entry()将内嵌的 pos->member.next 指针回转为指向该结点外层结构体起始处的指针，并将指针进行指针类型转换为外层结构体型 pos。&pos->member! = (head)用 pos 外层指针引用 member 即：list 成员，与内层嵌入的链表之头结点比较来为循环结束条件。

当遍历到头节点时，此时并没有 pos 这样一个 type 类型数据指针，而是以 member 域强制扩展了一个 type 类型的 pos 指针，此时其 member 域的地址就是 head 指针所指向的头节点，遍历结束，头节点的信息没有被访问。

----------list_for_each_entry_reverse()-------

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

分析类似上面。

----------list_prepare_entry()----------

如果遍历不是从链表头开始，而是从已知的某个 pos 结点开始，则可以使用 list_for_each_entry_continue(pos,head,member)。但为了确保 pos 的初始值有效，Linux 专门提供了一个 list_prepare_entry(pos,head,member)宏，如果 pos 有值，则其不变；如果没有，则从链表头强制扩展一个虚 pos 指针。将它的返回值作为 list_for_each_entry_continue()的 pos 参数，就可以满足这一要求。

内核中的 list_prepare_entry()的代码：

```
#define list_prepare_entry(pos, head, member) \
        ((pos) ? : list_entry(head, typeof(*pos), member))
```

分析：

:前面是个空值，即：若 pos 不为空，则 pos 为其自身。等效于：

(pos)? （pos）: list_entry(head,typeof(*pos),member)

注意内核格式：:前后都加了空格。

----------list_for_each_entry_continue()--------

内核中的 list_for_each_entry_continue()的代码：

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

此时不是从头节点开始遍历的，但仍然是以头节点为结束点的，即没有遍历完整个链表。

要注意并不是从 pos 开始的，而是从其下一个节点开始的，因为第一个有效 pos 是从 pos->member.next

扩展得到的。

----------list_for_each_entry_safe()----------

它们要求调用者另外提供一个与 pos 同类型的指针 n，在 for 循环中暂存 pos 下一个节点的地址，避免因 pos 节点被释放而造成的断链。

内核中的注释与源代码：

```
/**
* 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 counter.
* @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))
```

分析类似上面。容易明白。

--------list_for_each_entry_safe_continue()-------

```
#define list_for_each_entry_safe_continue(pos, n, head, member)                 \
        for (pos = list_entry(pos->member.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))
```

7、如何使用 Linux 中的双循环链表

本文例子来自 http://isis.poly.edu/kulesh/stuff/src/klist/，只是对其中注释部分作了翻译。

```
#include <stdio.h>

#include <stdlib.h>

#include "list.h"

struct kool_list{

    int to;

    struct list_head list;

    int from;

};

int main(int argc, char **argv){

    struct kool_list *tmp;

    struct list_head *pos, *q;

    unsigned int i;

    struct kool_list mylist;

    INIT_LIST_HEAD(&mylist.list);

    /*  您也可以使用宏 LIST_HEAD(mylist)来声明并初始化这个链表  */

    /*向链表中添加元素*/

    for(i=5; i!=0; --i){

        tmp= (struct kool_list *)malloc(sizeof(struct kool_list));

        /*INIT_LIST_HEAD(&tmp->list);  调用这个函数将初始化一个动态分配的 list_head。也可以不调用它，因为在后面调用的 add_list()中将设置 next 和 prev 域。*/

        printf("enter to and from:");
```

```
        scanf("%d %d", &tmp->to, &tmp->from);

        /*将 tmp 添加到 mylist 链表中*/

        list_add(&(tmp->list), &(mylist.list));

        /*也可以使用 list_add_tail()将新元素添加到链表的尾部。*/

    }

    printf("\n");

    /*现在我们得到了数据结构 struct kool_list 的一个循环链表，我们将遍历这个链表，并打印其中的元素。*/

    /*list_for_each()定义了一个 for 循环宏，第一个参数用作 for 循环的计数器，换句话说，在整个循环过程中它指向了当前项的 list_head。第二个参数是指向链表的指针，在宏中保持不变。*/

    printf("traversing the list using list_for_each()\n");

    list_for_each(pos, &mylist.list){

        /*此刻：pos->next 指向了下一项的 list 变量，而 pos->prev 指向上一项的 list 变量。而每项都是 struct kool_list 类型。但是，我们需要访问的是这些项，而不是项中的 list 变量。因此需要调用 list_entry()宏。*/

        tmp= list_entry(pos, struct kool_list, list);

        /*给定指向 struct list_head 的指针，它所属的宿主数据结构的类型，以及它在宿主数据结构中的名称，list_entry 返回指向宿主数据结构的指针。例如，在上面一行， list_entry()返回指向 pos 所属 struct kool_list 项的指针。*/

        printf("to= %d from= %d\n", tmp->to, tmp->from);

    }

    printf("\n");

    /* 因为这是一个循环链表，我们也可以向前遍历。只需要将 list_for_each 替换为 list_for_each_prev。我们也可以使用 list_for_each_entry()遍历链表，在给定类型的项间进行循环。例如：*/

    printf("traversing the list using list_for_each_entry()\n");

    list_for_each_entry(tmp, &mylist.list, list)

    printf("to= %d from= %d\n", tmp->to, tmp->from);
```

```
    printf("\n");

    /*下面将释放这些项。因为我们调用 list_del()从链表中删除各项，因此需要使用 list_for_each()宏的"
安全"版本，即 list_for_each_safe()。务必注意，如果在循环中有删除项（或把项从一个链表移动到另一
个链表）的操作，必须使用这个宏。*/

    printf("deleting the list using list_for_each_safe()\n");

    list_for_each_safe(pos, q, &mylist.list){

        tmp= list_entry(pos, struct kool_list, list);

        printf("freeing item to= %d from= %d\n", tmp->to, tmp->from);

        list_del(pos);

        free(tmp);

    }

    return 0;

}
```