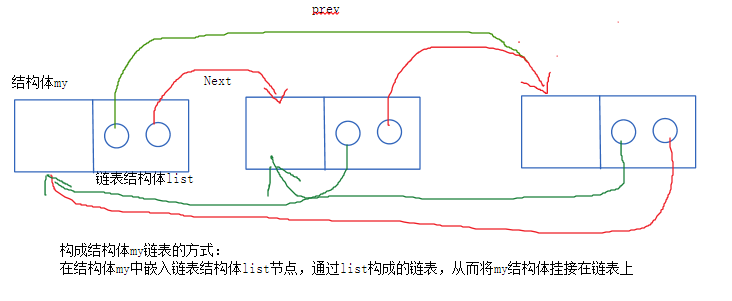
建立方法：通过将链表结构体嵌入到其他的结构体中，从而形成对于其他结构体的链表的建立。



Linux对双向循环链表的定义

    struct list\_head {

        struct list\_head \*next, \*prev;

    };

链表嵌入到其他结构体中，通过链表的方式管理新的数据结构

struct example {

        member a;

        struct list\_head list;

        member b;

    };

 1、链表的定义和初始化

   定义和初始化一个链表头结点的两种方式：

① LIST\_HEAD(mylist);  // 使用LIST\_HEAD宏定义并初始化一个链表

② struct list\_head mylist;  // 定义一个链表

     INIT\_LIST\_HEAD(&mylist); // 使用INIT\_LIST\_HEAD函数初始化链表

    其中方式①简单，LIST\_HEAD宏的展开：

#define LIST\_HEAD\_INIT(name) { &(name), &(name) }

#define LIST\_HEAD(name) /

         struct list\_head name = LIST\_HEAD\_INIT(name)

    很容易看出LIST\_HEAD(mylist);会被扩展为：

    struct list\_head mylist = { &(mylist), &(mylist) };

    list\_head结构只有两个成员：next和prev。从上面的代码可以看出，next和prev都被赋值为链表mylist的地址，也就是说，链表初始化后next和prev都是指向自己的。

    大多数情况下，list\_head是被嵌入到其他数据结构中的，比如上面的example结构里的list成员，那么如何对list成员进行初始化？通过调用INIT\_LIST\_HEAD函数：

    struct example test;

    INIT\_LIST\_HEAD(&test.list);   
    该函数简单地将list成员的prev和next指针指向自己。

    可以看出链表结点在初始化时，都将prev和next指向自己。注意：对链表的初始化非常重要，因为如果使用一个未被初始化的链表结点，很有可能会导致内核异常。

2、对链表常用的操作

    对链表常用的操作无非就是增加、删除、遍历等。

① 增加：list\_add和list\_add\_tail

     调用list\_add可以将一个新链表结点插入到一个已知结点的后面；

     调用list\_add\_tail可以将一个新链表结点插入到一个已知结点的前面；

具体实现：它们都以不同的参数调用了相同的函数\_\_list\_add：

    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;

    }

    该函数将new结点插入到prev结点和next之间；

② 删除：list\_del和list\_del\_init

     调用list\_del函数删除链表中的一个结点；

     调用list\_del\_init函数删除链表中的一个结点，并初始化被删除的结点（也就是使被删除的结点的prev和next都指向自己）；

    下面分析它们的具体实现，它们都调用了相同的函数\_\_list\_del：  
    static inline void \_\_list\_del(struct list\_head \* prev, struct list\_head \* next)  
    {  
         next->prev = prev;  
         prev->next = next;  
    }  
    该函数实际的作用是让prev结点和next结点互相指向；

 3、几个重要的宏

    内核提供了一组宏，以方便对链表进行管理

① list\_entry

获取该链表所在结构体的地址：通过调用container\_of宏

#define list\_entry(ptr, type, member)  container\_of(ptr, type, member) ......

所以： list\_entry(ptr, struct example, list); 来获取example对象的指针。

② list\_for\_each\_entry

    对链表的一个重要的操作就是对链表进行遍历，以达到某种应用目的，比如统计链表结点的个数等等。先来看看内核中对该宏的定义：

    #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))

    其中，pos是指向宿主结构的指针，在for循环中是一个迭代变量；head是要进行遍历的链表头指针；member是list\_head成员在宿主结构中的名字。