list\_head用法

**一、list\_head介绍**

在内核中，所有的链表都依靠list\_head实现。list\_head结构体声明如下：

#include <linux/list.h>

struct list\_head {

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

};

从上面定义可以看出，list\_head节点只包含两个成员，next和prev。所以是个双向链表的节点。

list\_head没有数据成员，它的用法是把自己包含在宿主结构体中，通过container\_of可以找到宿主结构体的地址。

**二、list\_head的用法：**

当我们需要一个链表的时候，可以通过在结构体中加入list\_head成员，然后依靠内核<linux/list.h>中已经实现的宏和函数来将结构体串成链表操作。

**1.节点初始化**

一个新的list\_head节点，需要初始化，初始化策略是让新节点的next和prev指针都指向自己。有如下三种初始化方法：

方法1：使用INIT\_LIST\_HEAD(struct list\_head \*list)函数

static inline void INIT\_LIST\_HEAD(struct list\_head \*list)

{

list->next = list;

list->prev = list;

}

方法2：使用LIST\_HEAD\_INIT(name)宏

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

方法3：使用LIST\_HEAD(name)宏

#define LIST\_HEAD(name) \

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

区别：

方法1是个内联函数，输入节点的指针，对节点初始化。

方法2是宏，输入节点本身，对节点初始化。

方法3也是宏，也是输入节点本书，对节点初始化，与方法2的区别是不需要提前创建节点，而是由此宏自己创建。

**2.将list\_head插入链表：**

有两个插入函数：list\_add和list\_add\_tail,为了了解这两个函数，先看下面的函数,源码如下：

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;

}

从上面源码可知，\_\_list\_add(new,prev,next)是将new插入到了prev之前，next之后，插入后顺序是prev-new-next.

再看下面代码可知,list\_add是将new插入到head之后，即从链表头插入。list\_add\_tail是将new插入到head之前，因为head是双向链表，所以插入head之前，等同于从表尾插入。

static inline void list\_add(struct list\_head \*new, struct list\_head \*head)

{

\_\_list\_add(new, head, head->next);

}

static inline void list\_add\_tail(struct list\_head \*new, struct list\_head \*head)

{

\_\_list\_add(new, head->prev, head);

}

**3.删除节点**

删除节点就调用list\_del(struct list\_head \*)，因为list\_head节点中存储由前后节点的指针，所以此函数只需要一个参数就可以。

list\_head常用操作就上面这些，遇到其他相关操作，可参考<linux/list.h>。

**三、list\_head在应用程序中的应用**

只需要参考<linux/list.h>，将list\_head的声明、初始化、add和del代码移入自己的应用程序中，就可以创建自己的双向链表。