**list\_head**

**1. overview**

这两个macro在Kernel里也算蛮常出现的， 是用来将list\_head做初始化的，它的初始化就是将next和prev这两个指针设为跟结构的地址相同。

所以， 如果我们在程序里看到这样的程序， 它的意思就是宣告一个list\_head结构的变数hello，并将prev和next都设成hello的地址。

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

#define LIST\_HEAD(name) \

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

static inline void

INIT\_LIST\_HEAD(struct list\_head \*list)

{

list->next = list->prev = list;

}

因此， 如果要检查这个list是否是空的， 只要检查hello.next是否等于&hello就可以了。事实上， Linux也提供了一个叫list\_empty()的函式来检查list是否为空的。

static inline bool

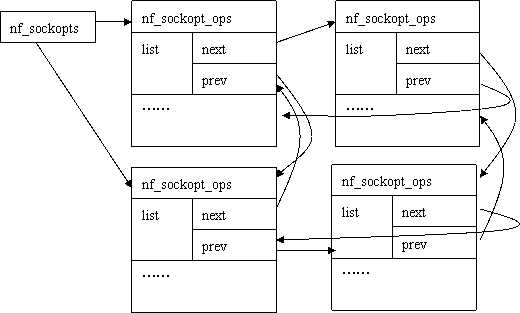
list\_empty(struct list\_head \*head)

{

return head->next == head;

}

在Linux内核链表中，需要用链表组织起来的数据通常会包含一个struct list\_head成员，例如在[include/linux/netfilter.h]中定义了一个nf\_sockopt\_ops结构来描述Netfilter为某一协议族准备的getsockopt/setsockopt接口，其中就有一个（struct list\_head list）成员，各个协议族的nf\_sockopt\_ops结构都通过这个list成员组织在一个链表中，表头是定义在[net/core/netfilter.c]中的nf\_sockopts(struct list\_head)，如下图

**2. 链表操作接口**

**2.1 声明和初始化**

实际上Linux只定义了链表节点，并没有专门定义链表头，那么一个链表结构是如何建立起来的呢？

当我们用LIST\_HEAD(nf\_sockopts)声明一个名为nf\_sockopts的链表头时，它的next、prev指针都初始化为指向自己，这样，我们就有了一个空链表，因为Linux用头指针的next是否指向自己来判断链表是否为空：

除了用LIST\_HEAD()宏在声明的时候初始化一个链表以外，Linux还提供了一个INIT\_LIST\_HEAD宏用于运行时初始化链表：

我们用INIT\_LIST\_HEAD(&nf\_sockopts)来使用它。

**2.2 插入/删除/合并**

对链表的插入操作有两种：在表头插入和在表尾插入。Linux为此提供了两个接口：

**a) 插入**

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

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

因为Linux链表是循环表，且表头的next、prev分别指向链表中的第一个和最末一个节点，所以，list\_add和list\_add\_tail的区别并不大，实际上，Linux分别用

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

和

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

来实现两个接口，可见，在表头插入是插入在head之后，而在表尾插入是插入在head->prev之后。

假设有一个新nf\_sockopt\_ops结构变量new\_sockopt需要添加到nf\_sockopts链表头，我们应当这样操作：

list\_add(&new\_sockopt.list, &nf\_sockopts);

从这里我们看出，nf\_sockopts链表中记录的并不是new\_sockopt的地址，而是其中的list元素的地址。如何通过链表访问到new\_sockopt呢？下面会有详细介绍。

**b) 删除**

当我们需要删除nf\_sockopts链表中添加的new\_sockopt项时，我们这么操作：

list\_del(&new\_sockopt.list);

被剔除下来的new\_sockopt.list，prev、next指针分别被设为LIST\_POSITION2和LIST\_POSITION1两个特殊值，这样设置是为了保证不在链表中的节点项不可访问--对LIST\_POSITION1和LIST\_POSITION2的访问都将引起页故障。与之相对应，list\_del\_init()函数将节点从链表中解下来之后，调用LIST\_INIT\_HEAD()将节点置为空链状态。

**c) 搬移**

Linux提供了将原本属于一个链表的节点移动到另一个链表的操作，并根据插入到新链表的位置分为两类：

static inline void list\_move(struct list\_head \*list, struct list\_head \*head);

static inline void list\_move\_tail(struct list\_head \*list, struct list\_head \*head);

**d) 合并**

除了针对节点的插入、删除操作，Linux链表还提供了整个链表的插入功能：

static inline void list\_splice(struct list\_head \*list, struct list\_head \*head);

**2.3遍历**

**a) 由链表节点到数据项变量**

我们知道，Linux链表中仅保存了数据项结构中list\_head成员变量的地址，那么我们如何通过这个list\_head成员访问到作为它的所有者的节点数据呢？Linux为此提供了一个list\_entry(ptr,type,member)宏，其中ptr是指向该数据中list\_head成员的指针，也就是存储在链表中的地址值，type是数据项的类型，member则是数据项类型定义中list\_head成员的变量名，例如，我们要访问nf\_sockopts链表中首个nf\_sockopt\_ops变量，则如此调用：

list\_entry(nf\_sockopts->next, struct nf\_sockopt\_ops, list);

这里"list"正是nf\_sockopt\_ops结构中定义的用于链表操作的节点成员变量名。

list\_entry的使用相当简单，相比之下，它的实现则有一些难懂：

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

kernel/include/linux/kernel.h

#define container\_of(ptr, type, member) ({ \

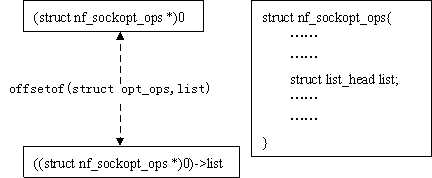
const typeof( ((type \*)0)->member ) \*\_\_mptr = (ptr); \

(type \*)( (char \*)\_\_mptr - offsetof(type,member) );})

kernel/include/linux/stddef.h

#define offsetof(TYPE, MEMBER) ((size\_t)&((TYPE \*)0)->MEMBER)

这里使用的是一个利用编译器技术的小技巧，即先求得结构成员在与结构中的偏移量，然后根据成员变量的地址反过来得出属主结构变量的地址。



container\_of()和offsetof()并不仅用于链表操作，这里最有趣的地方是((type \*)0)->member，它将0地址强制"转换"为type结构的指针，再访问到type结构中的member成员。在container\_of宏中，它用来给typeof()提供参数（typeof()是gcc的扩展，和sizeof()类似），以获得member成员的数据类型；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。

**b) 遍历宏**

在[net/core/netfilter.c]的nf\_register\_sockopt()函数中有这么一段话：

……

struct list\_head \*i;

……

list\_for\_each(i, &nf\_sockopts) {

struct nf\_sockopt\_ops \*ops = (struct nf\_sockopt\_ops \*)i;

……

}

……

函数首先定义一个(struct list\_head \*)指针变量i，然后调用list\_for\_each(i,&nf\_sockopts)进行遍历。在[include/linux/list.h]中，list\_for\_each()宏是这么定义的：

#define list\_for\_each(pos, head) \

for (pos = (head)->next, prefetch(pos->next); pos != (head); \

pos = pos->next, prefetch(pos->next))

它实际上是一个for循环，利用传入的pos作为循环变量，从表头head开始，逐项向后（next方向）移动pos，直至又回到head（prefetch()可以不考虑，用于预取以提高遍历速度）。

那么在nf\_register\_sockopt()中实际上就是遍历nf\_sockopts链表。为什么能直接将获得的list\_head成员变量地址当成struct nf\_sockopt\_ops数据项变量的地址呢？我们注意到在struct nf\_sockopt\_ops结构中，list是其中的第一项成员，因此，它的地址也就是结构变量的地址。

更规范的获得数据变量地址的用法应该是：

struct nf\_sockopt\_ops \*ops = list\_entry(i, struct nf\_sockopt\_ops, list);

大多数情况下，遍历链表的时候都需要获得链表节点数据项，也就是说list\_for\_each()和list\_entry()总是同时使用。对此Linux给出了一个list\_for\_each\_entry()宏：

与list\_for\_each()不同，这里的pos是数据项结构指针类型，而不是(struct list\_head \*)。nf\_register\_sockopt()函数可以利用这个宏而设计得更简单：

……

struct nf\_sockopt\_ops \*ops;

list\_for\_each\_entry(ops,&nf\_sockopts,list){

……

}

……