

# 深入浅出 linux 内核源代码之双向链表 list\_head

原创文章，转载请注明出处，谢谢！

作者：清林，博客名：飞空静渡

范金宝

email:[fjb2080@163.com](mailto:fjb2080@163.com)

blog:<http://blog.csdn.net/fjb2080>

前言：在 linux 源代码中有个头文件为 list.h。很多 linux 下的源代码都会使用这个头文件，它里面定义了一个结构，以及定义了和其相关的一组函数，这个结构是这样的：

```
struct list_head{
    struct list_head *next, *prev;
};
```

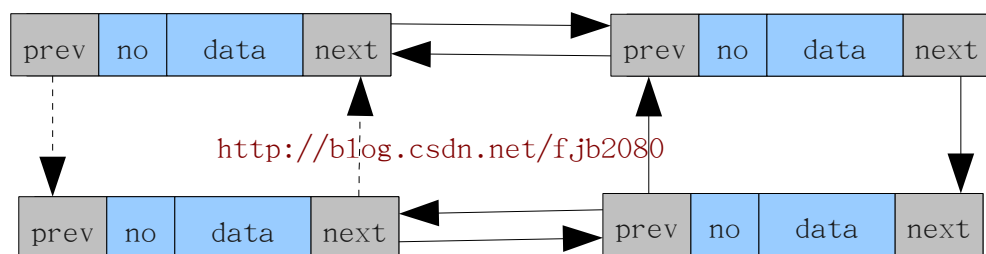
那么这个头文件又是有什么样的作用呢，这篇文章就是用来解释它的作用，虽然这是 linux 下的源代码，但对于学习 C 语言的人来说，这是算法和平台没有什么关系。

## 一、双向链表

学习计算机的人都会开一门课程《数据结构》，里面都会有讲解双向链表的内容。什么是双向链表，它看起来是这样的：

```
struct dlist
{
    int no;
    void* data;
    struct dlist *prev, *next;
};
```

它的图形结构图如下：



如果是双向循环链表，那么就加上虚线所示。

现在有几个结构体，它们是：  
表示人的：

```
struct person
{
```

```

        int    age;
        int    weight;
};

```

表示动物的：

```

struct animal
{
    int    age;
    int    weight;
};

```

如果有一组 filename 变量和 filedata 变量，把它们存起来，我们会怎么做，当然就用数组了，但我们想使用双向链表，让它们链接起来，那该怎么做，唯一可以做的就是给每个结构加如两个成员，如下：

表示人的：

```

struct person
{
    int    age;
    int    weight;
    struct person *next, *prev;
};

```

表示动物的：

```

struct animal
{
    int    age;
    int    weight;
    struct animal *next, *prev;
};

```

现在有一个人的一个链表的链头指针 person\_head（循环双向链表）和动物的链表的链头指针 aanimal\_head，我们要获得特定年龄和特定体重的人或动物（假设不考虑重叠），那么代码看起来可能是这样：

```

struct person * get_percent(int age, int weight)
{
    .....
    struct person *p;
    for(p = person_head->next; p != person_head; p=p->next)
    {
        if(p->age == age && p->weight == weight)
            return p;
    }
    .....
}

```

那同理，要获得一个特定年龄和重量的动物的函数 get\_animal(int age, int weight)的代码也是和上面的类似。

如果我们定义这样的两个函数，它们基本一样，会不会觉得有点冗余，如果是 C++ 就好了，但这里只说 C。

如果我们仔细观察一下这两个结构，我们会发现它们除了类型名字不一样外，其它的都一样。那么我们考虑用一个宏来实现，这个宏看起来可能是这样的。

```
#define get_one(list, age, weight, type, one) \
do \
{\
    type *p;\
    for(p = ((type*)list)->next; p != (type*)list; p=p->next)\
    if (p->age == age && p->weight == weight)\
    {\
        one = p;\
        break;\
    }\
}while(0)
```

那么我们获得一个年龄 50，体重 60 的人可以这样：

```
struct person *one = NULL;
get_one(person_head, 50, 60, struct person, one);
if(one)
{
    // get it
    .....
}
```

同样获得一个年龄 20，体重 130 的动物可以这样：

```
struct animal *one = NULL;
get_one(animal_head, 50, 60, struct animal, one);
if(one)
{
    // get it
    .....
}
```

我们再回过头来看这两个结构，它们的指向前和指向后的指针其实都差不多，那把它们综合起来吧，所以看起来如下面：

```
struct list_head{
    struct list_head *next, *prev;
};
```

表示人的：

```
struct person
{
```

```

        int    age;
        int    weight;
        struct list_head list;
};
表示动物的：
struct animal
{
    int    age;
    int    weight;
    struct list_head list;
};

```

现在这个两个结构看起来就更差不多一样了。现在为了方便，我们去掉那些暂时不用的数据，如下：

```

struct person
{
    struct list_head list;
};
表示动物的：
struct animal
{
    struct list_head list;
};

```

可能又会有人会问了，`struct list_head` 都不是 `struct person` 和 `struct animal` 类型，怎么做链表的指针呢？其实，无论是什么样的指针，它的大小都是一样的，32 位的系统中，指针的大小都是 32 位（即 4 个字节），只是不同类型的指针在解释的时候不一样而已，那么这个 `struct list_head` 又是怎么去做这些结构的链表指针呢，那么就请看下一节吧：）。

## 二、`struct list_head` 结构的操作

首先，让我们来看下和 `struct list_head` 有关的两个宏，它们定义在 `list.h` 文件中。

```

#define LIST_HEAD_INIT(name) { &(amp;name), &(name) }
#define LIST_HEAD(name) struct list_head name = LIST_HEAD_INIT(name)
#define INIT_LIST_HEAD(ptr) do { \
    (ptr)->next = (ptr); (ptr)->prev = (ptr); \
} while (0)

```

这两个宏是用了定义双向链表的头节点的，定义一个双向链表的头节点，我们可以这样：

```

struct list_head head;
LIST_HEAD_INIT(head);

```

又或者直接这样：

```

LIST_HEAD(head);

```

这样，我们就定义并初始化了一个头节点。

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

就是用 head 的地址初始化其两个成员 next 和 prev，使其都指向自己。

我们再看下和其相关的几个函数，这些函数都作为内联函数也都定义 list.h 中，这里要说明一下 linux 源码的一个风格，在下面的这些函数中以下划线开始的函数是给内部调用的函数，而以符开始的函数就是对外使用的函数，这些函数一般都是调用以下划线开始的函数，或是说是对下划线开始的函数的封装。

## 2.1 增加节点的函数

```
static inline void __list_add();  
static inline void list_add();  
static inline void list_add_tail();
```

其实看源代码是最好的讲解了，这里我再简单的讲一下。

```
/**  
 * __list_add - Insert a new entry between two known consecutive entries.  
 * @new:  
 * @prev:  
 * @next:  
 *  
 * This is only for internal list manipulation where we know the prev/next  
 * entries already!  
 */  
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;  
}
```

这个函数在 prev 和 next 间插入一个节点 new。

```
/**  
 * list_add - add a new entry  
 * @new:    new entry to be added  
 * @head:   list head to add it after  
 *  
 * Insert a new entry after the specified head.  
 * This is good for implementing stacks.  
 */
```

```
static __inline__ void list_add(struct list_head *new, struct list_head *head)
{
    __list_add(new, head, head->next);
}
```

这个函数在 head 节点后面插入 new 节点。

```
/**
 * list_add_tail - add a new entry
 * @new:    new entry to be added
 * @head:   list head to add it before
 *
 * Insert a new entry before the specified head.
 * This is useful for implementing queues.
 */
static __inline__ void list_add_tail(struct list_head *new, struct list_head *head)
{
    __list_add(new, head->prev, head);
}
```

这个函数和上面的那个函数相反，它在 head 节点的前面插入 new 节点。

## 2.2 从链表中删除节点的函数

```
/**
 * __list_del -
 * @prev:
 * @next:
 *
 * Delete a list entry by making the prev/next entries point to each other.
 *
 * This is only for internal list manipulation where we know the prev/next
 * entries already!
 */
static __inline__ void __list_del(struct list_head * prev,
                                  struct list_head * next)
{
    next->prev = prev;
    prev->next = next;
}

/**
 * list_del - deletes entry from list.
 * @entry:   the element to delete from the list.
 *

```

```
* Note: list_empty on entry does not return true after this, the entry is in
* an undefined state.
```

```
*/
```

```
static __inline__ void list_del(struct list_head *entry)
{
    __list_del(entry->prev, entry->next);
}
```

```
/**
```

```
* list_del_init - deletes entry from list and reinitialize it.
```

```
* @entry: the element to delete from the list.
```

```
*/
```

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

这里简单说一下，list\_del(struct list\_head \*entry)是从链表中删除entry节点。

list\_del\_init(struct list\_head \*entry)不但从链表中删除节点，还把这个节点的向前向后指针都指向自己，即初始化。

那么，我们怎么判断这个链表是不是空的呢！上面我说了，这里的双向链表都是有一个头节点，而我们上面看到，定义一个头节点时我们就初始化了，即它的prev和next指针都指向自己。所以这个函数是这样的。

```
/**
```

```
* list_empty - tests whether a list is empty
```

```
* @head: the list to test.
```

```
*/
```

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

讲了这几个函数后，这又到了关键了，下面讲解的一个宏的定义就是对第一节中，我们所要说的为什么在一个结构中加入struct list\_head变量就把这个结构变成了双向链表呢，这其中的关键就是怎么通过这个struct list\_head变量来获取整个结构的变量，下面这个宏就为你解开答案：

```
/**
```

```
* list_entry - get the struct for this entry
```


```
* @ptr: the &struct list_head pointer.
```

```
* @type: the type of the struct this is embedded in.
```

```
* @member: the name of the list_struct within the struct.
```

```
*/
```

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

$$((\text{type } *)((\text{char } *) (\text{ptr}) - (\text{unsigned long}) (\&((\text{type } *)0) \rightarrow \text{member})))$$

乍一看下，不知道这个宏在说什么，没关系，我举个例子来为你——解答：)

首先，我们还是用上面的结构：

```
struct person
{
    int    age;
    int    weight;
    struct list_head list;
};
```

我们一看到这样的结构就应该知道它定义了一个双向链表，下面来看下。  
我们有一个指针：

```
struct list_head *pos;
```

现在有这个指针，我们怎么去获得这个指针所在的结构的变量（即是 struct person 变量，其实是 struct person 指针）呢？看下面这样使用：

```
struct person *one = list_entry(pos, struct person, list);
```

不明白是吧，展开一下 list\_entry 结构如下：

```
((struct person *)((char *) (pos) - (unsigned long) (&((struct person *)0) -> list)))
```

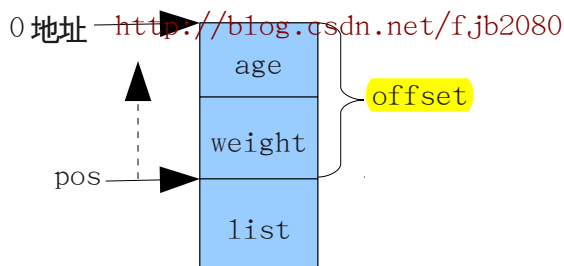
我慢慢的分解，首先分成两部分 (char \*) (pos) 减去 (unsigned long) (&((struct person \*)0) -> list) 然后转换成 (struct person \*) 类型的指针。

(char \*) (pos)：是将 pos 由 struct list\_head\* 转换成 char\*，这个好理解。

(unsigned long) (&((struct person \*)0) -> list)：先看最里面的 (struct person \*)0)，它是把 0 地址转换成 struct person 指针，然后 (struct person \*)0) -> list 就是指向 list 变量，之后是 &((struct person \*)0) -> list 是取这个变量的地址，最后是 (unsigned long) (&((struct person \*)0) -> list) 把这个变量的地址值变成一个整形数！

这么复杂啊，其实说白了，这个 (unsigned long) (&((struct person \*)0) -> list) 的意思就是取 list 变量在 struct person 结构中的偏移量。

用个图形来说 (unsigned long) (&((struct person \*)0) -> list)，如下：



而 (unsigned long) (&((struct person \*)0) -> list) 就是获取这个 offset 的值。

```
((char *) (pos) - (unsigned long) (&((struct person *)0) -> list))
```



就是将 pos 指针往前移动 offset 位置，即是本来 pos 是 struct list\_head 类型，它即是 list。即是把 pos 指针往 struct person 结构的头地址位置移动过去，如上图的 pos 和虚箭头。

当 pos 移到 struct person 结构头后就转换成 (struct person \*) 指针，这样就可以得到 struct person \* 变量了。

所以我们再回到前面的句子

```
struct person *one = list_entry(pos, struct person, list);
```

就是由 pos 得到 pos 所在的结构的指针，动物就可以这样：

```
struct animal *one = list_entry(pos, struct animal, list);
```

下面我们再来看下和 struct list\_head 相关的最后一个宏。

## 2.3 list\_head 的遍历的宏

```
/**
 * list_for_each - iterate over a list
 * @pos:    the &struct list_head to use as a loop counter.
 * @head:    the head for your list.
 */
#define list_for_each(pos, head) \
    for (pos = (head)->next; pos != (head); pos = pos->next)

/**
 * list_for_each_safe - iterate over a list safe against removal of list entry
 * @pos:    the &struct list_head to use as a loop counter.
 * @n:      another &struct list_head to use as temporary storage
 * @head:    the head for your list.
 */
#define list_for_each_safe(pos, n, head) \
    for (pos = (head)->next, n = pos->next; pos != (head); \
         pos = n, n = pos->next)
```

list\_for\_each(pos, head) 是遍历整个 head 链表中的每个元素，每个元素都用 pos 指向。

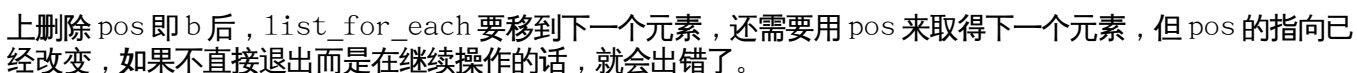
list\_for\_each\_safe(pos, n, head) 是用于删除链表 head 中的元素，不是上面有删除链表元素的函数了吗，为什么这里又要定义一个这样的宏呢。看下这个宏后面有个 safe 字，就是说用这个宏来删除是安全的，直接用前面的那些删除函数是不安全的。这个怎么说呢，我们看下下面这个图，有三个元素 a, b, c。



现在，我们要删除 b 元素，下面是删除的算法（先只用删除函数）：

```
struct list_head *pos;
```

上面的算法是不安全的，因为当我们删除 b 后，如下图这样：



```

struct list_head *pos, *n;
list_for_each_safe(pos, n, myhead)
{
    if (pos == b)
    {
        list_del_init(pos);
        //break;
    }
    . . .
}

```

说了那么多关于 `list_head` 的东西，下面应该总结一下，总结一下第一节想要解决的问题。

我用一个程序来说明在 `struct person` 中增加了 `struct list_head` 变量后怎么来操作这样的双向链表。

```
#include <stdio.h>
#include "list.h"

struct person
{
    int age;
```

```

    int weight;
    struct list_head list;
};

int main(int argc, char* argv[])
{
    struct person *tmp;
    struct list_head *pos, *n;
    int age_i, weight_j;

    // 定义并初始化一个链表头
    struct person person_head;
    INIT_LIST_HEAD(&person_head.list);

    for(age_i = 10, weight_j = 35; age_i < 40; age_i += 5, weight_j += 5)
    {
        tmp = (struct person*)malloc(sizeof(struct person));
        tmp->age = age_i;
        tmp->weight = weight_j;

        // 把这个节点链接到链表后面
        // 这里因为每次的节点都是加在 person_head 的后面，所以先加进来的节点就在链表里的最
        // 打印的时候看到的顺序就是先加进来的就在最后面打印
        list_add(&(tmp->list), &(person_head.list));

    }

    // 下面把这个链表中各个节点的值打印出来
    printf("\n");
    printf("===== print the list =====\n");
    list_for_each(pos, &person_head.list)
    {
        // 这里我们用 list_entry 来取得 pos 所在的结构体的指针
        tmp = list_entry(pos, struct person, list);
        printf("age:%d, weight: %d \n", tmp->age, tmp->weight);
    }
    printf("\n");

    // 下面删除一个节点中，age 为 20 的节点
    printf("===== print list after delete a node which age is 20
===== \n");
    list_for_each_safe(pos, n, &person_head.list)
    {

```

```

        tmp = list_entry(pos, struct person, list);
        if(tmp->age == 20)
        {
            list_del_init(pos);
            free(tmp);
        }

    }

    list_for_each(pos, &person_head.list)
    {
        tmp = list_entry(pos, struct person, list);
        printf("age:%d, weight: %d \n", tmp->age, tmp->weight);
    }

    // 释放资源
    list_for_each_safe(pos, n, &person_head.list)
    {
        tmp = list_entry(pos, struct person, list);
        list_del_init(pos);
        free(tmp);
    }

    return 0;
}

```

**编译：**

linux 下的可以：gcc -g -Wall main.c -o test

windows 下的可以建一个控制台工程，把 main.c 和 list.h 加到工程中编译。

**运行 test 后的输出如下：**

===== print the list =====

```

age:35, weight: 60
age:30, weight: 55
age:25, weight: 50
age:20, weight: 45
age:15, weight: 40
age:10, weight: 35

```

===== print list after delete a node which age is 20 =====

```

age:35, weight: 60
age:30, weight: 55
age:25, weight: 50

```

```
age:15, weight: 40
age:10, weight: 35
```

我们看到，这就是一个非常好和有效的双向链表，我们不需要为每一种结构去定义相关的函数，如遍历、增加和删除等函数，我们只需要简单的在结构中增加 `struct list_head` 的一个变量，我们的结构立马就变成了一个双向链表，而且，我们对链表的操作也不用自己写，直接调用已经定义好的函数和宏，一切就那么简单和有效。

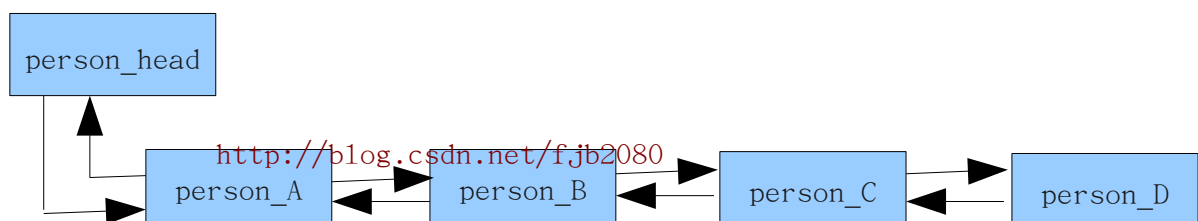
文章写到这里是不是应该结束了呢，没有，我还不想结束，还想在继续说。

#### 四、一个结构多个链表

在上面，我们看到人的结构是这样的：

```
struct person
{
    int age;
    int weight;
    struct list_head list;
};
```

它的链表图形看起来如下图所示：



但我们知道，一个人，他的熟悉还有很多，例如他有各种各样的衣服，各种不同的鞋子等。所以，我定义了两个这样的结构：

```
struct clothes
{
    int size; //衣服有各种大小
    Color color; //衣服有各种颜色，这里假设有一种 Color 的类型
};
```

```
struct shoot
{
    Kind kind; //鞋子有各种类型，秋、冬、运动、休闲等，同样假设已经定义过 Kind 这样的类型
    Color color; //鞋子也有各种颜色
};
```

那么这个人的定义可能就是这样的：

```
struct person
{
    int age;
```

```

    int weight;
    struct clothes    clo;
    struct shoot      sht;
};

```

在这里，我有意 clo 和 sht 这两个变量放在 list 后面，其实，代表链表的 list 在结构中的位置在哪里是没什么关系的，list\_entry 也一样可以将结构的指针找出来。

这里有一个问题是，一个人不止一件衣服，也不止一双鞋子，所以我们应该把他拥有的衣服和鞋子应该加上，那么怎么加呢？这里应该把衣服和鞋子的结构也变成链表，这不就解决了。

把结构改一下，变成了这样：

```

struct clothes
{
    struct list_head list;
    int    size;    //衣服有各种大小
    Color color; //衣服有各种颜色，这里假设有一种 Color 的类型
};

struct shoot
{
    struct list_head list;
    Kind  kind;    // 鞋子有各种类型，秋、冬、运动、休闲等，同样假设已经定义过 Kind 这样的类型
    Color color;   // 鞋子也有各种颜色
};

```

现在鞋子和衣服都是链表了，都可以把它们连接起来。那我们的结构是不是还应该这样定义：

```

struct person
{
    int age;
    int weight;
    struct clothes    clo;
    struct shoot      sht;
};

```

如果是，那么我们应该怎么定义这个头节点。在前面我们看到，定义一个 person\_head 的头节点是这样的：

```

// 定义并初始化一个链表头
struct person person_head;
INIT_LIST_HEAD(&person_head.list);

```

难道我们应该这样定义吗？

```

// 定义并初始化一个链表头
struct person person_head;
INIT_LIST_HEAD(&person_head.list);
INIT_LIST_HEAD(&person_head.col.list);
INIT_LIST_HEAD(&person_head.sht.list);

```

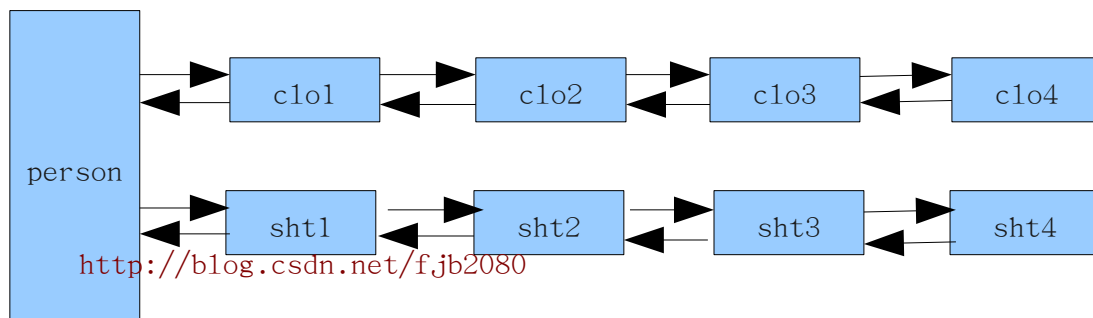
那么增加一件衣服进去呢，代码看起来是这样的：

```
struct clothes tmp =(struct clothes*)malloc(sizeof(struct clothes));
.....
list_add(&(tmp->list), &(person_head.clo.list));
```

这样会不会有点麻烦，其实，如果我们可以认真想一想，我们会发现，既然 struct peron 是一个含有 list\_head 的结构，它可以把它的类型节点链接在后面，那么 struct clothes 也是一个含有 list\_head 的结构，它们本质也没什么区别，应该也可以链接在它后面的。所以我们的 struct person 的结构应该变成这样：

```
struct person
{
    int age;
    int weight;
    struct list_head clo;
    struct list_head sht;
};
```

那么我们链接节点后的图形如下图所示：



由上面，我们可以知道，有了 struct list\_head 结构，我们可以为我们的结构体增加多个子节点链表。