```
$find -name "list.h"
./btrfs/list.h
```

内核链表与普通链表的区别：

1. 普通链表当中数据域和指针域，没有做到区分。数据与指针形成了一个整体，而内核链表数据与指针是完全剥离的没有直接的关系。

2. 在普通链表当中所有节点的数据都是一样的类型，而内核链表中每一个节点的类型都可以是不同的。

3. 普通的链表每一个节点都只能出现在一个链表当中，因为它只有一对前驱/后继指针，而内核链表可以有多对前驱/后继指针。

纯粹的内核链表（没有数据）：

只是一个链表的逻辑，并没有任何的意义

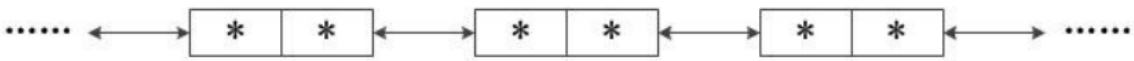


图 3-29 纯粹的链表逻辑

在数据的结构体中加入内核链表，就可以使得内核链表携带数据出现在链表当中去。

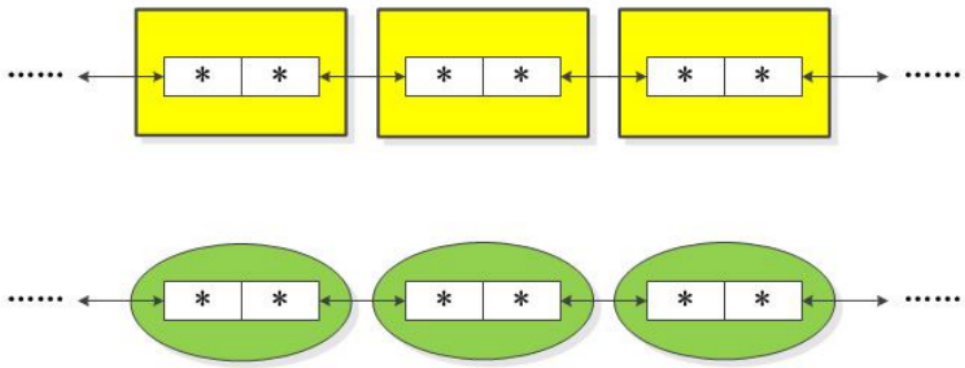


图 3-30 镶嵌了纯粹链表的节点

由于内核链表它的特性：数据与指针的完全剥离， 可以使得一个节点出现在多个链表当中

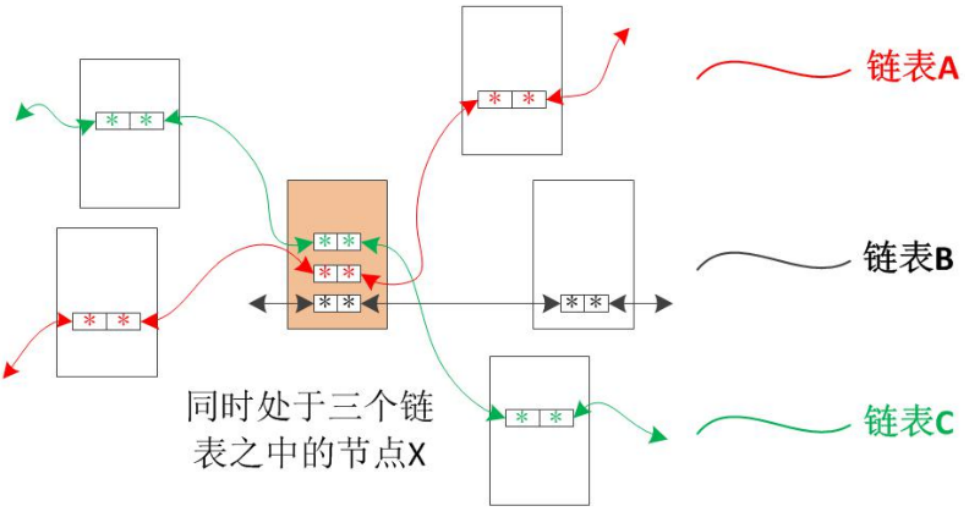


图 3-33 内核链表的优势

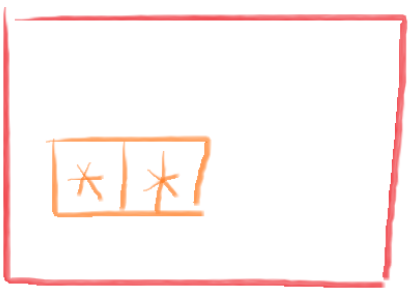
内核链表头文件：

```
$find -name "list.h"
./btrfs/list.h
```

大结构体与小结构体：

大结构体： 包含了数据以及指针（内核链表中的小的结构体）

小结构体： 内核链表中用来指向前一个节点和后一个节点的指针



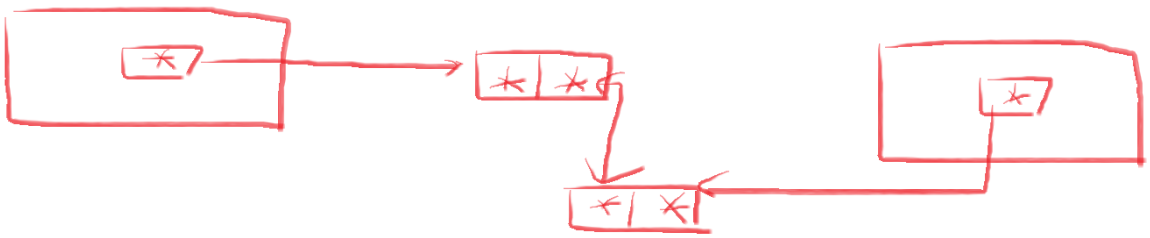
```
struct list_head {
    struct list_head *next, *prev;
};
```

```
// 节点设计
struct node
{
    Data_Type data ;
    struct list_head ptr ;
};
```

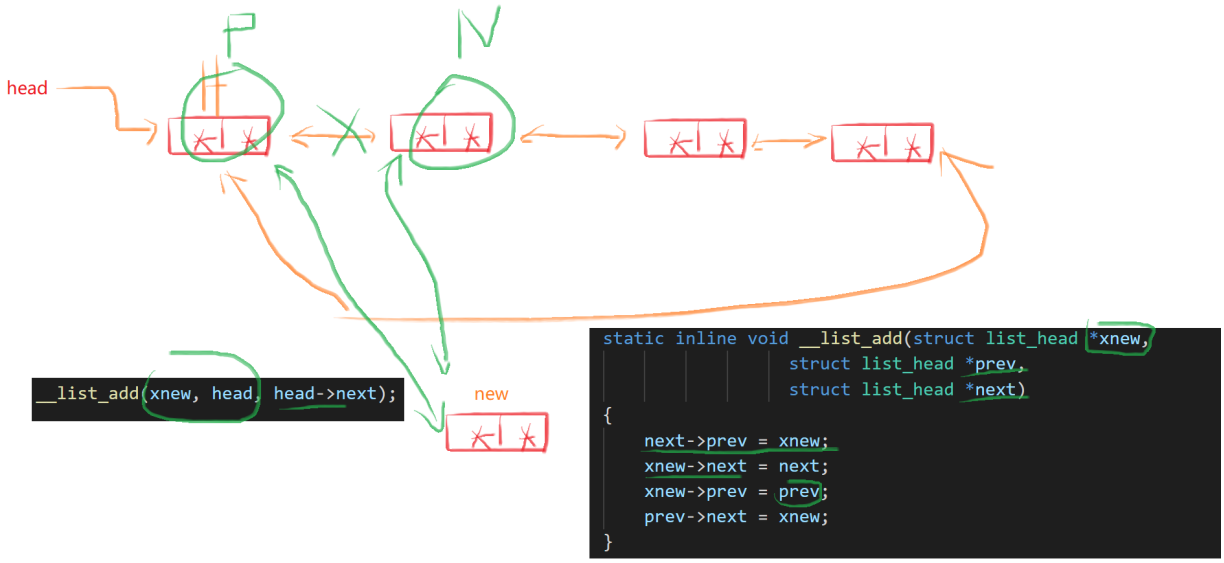
注意：

在大结构体当中小结构体必须是一个普通变量不可以使用指针。

错误示范



节点初始化：



内核链表中插入数据的思路：

把用户传进来的两个参数： 新节点 + 头节点

演变成很三个参数： 新节点 + 新节点的前驱节点 + 新节点的后继节点

```c
/**
 * 节点头插
 *
 * 使用内联函数list_add(struct list_head *xnew, struct list_head *head),
 * 将会自动调用__list_add，实现功能：将节点 xnew 插入到 head 后面；实现头插功能
 */
static inline void list_add(struct list_head *xnew, struct list_head *head)
{
    __list_add(xnew, head, head->next);
}
```

计算获得大结构体的入口地址：


```c
P_Node kernelListInit(Data_Type data )
{
    // 申请大结构体的内存空间
    P_Node new = calloc(1, sizeof(Node));

    //初始化数据
    new->data = data ;

    // 初始化小结构体（指针）
    INIT_LIST_HEAD( &new->ptr );

    return new ;
}
```

遍历显示：

```c

int display_list(P_Node head)
{
    if( head->ptr.next == &head->ptr )
    {
        printf("该表为空！！\n");
        return -1 ;
    }

    struct list_head * tmp = head->ptr.next ;
```

```
    P_Node node = NULL ;
    for ( ; tmp != &head->ptr ; tmp = tmp->next)
    {
        // 计算获得大结构体的入口地址
        node = list_entry(tmp , Node , ptr );
        printf("data:%d\n" , node->data);
    }

    return 0 ;
}
```