1、常规链表的缺陷

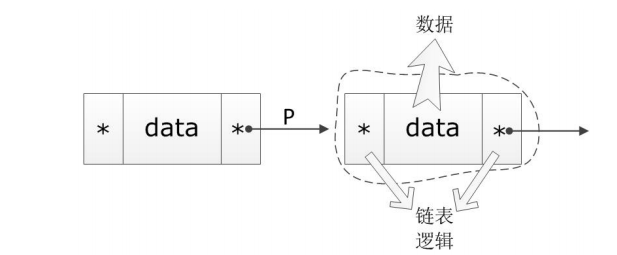
不仅包含了表达链表逻辑的指针，更包含了某一种具体的数据， 而要命的是这些指针指向了整个节点，这就导致了无法将链表逻辑跟具体数据分开，因此链 表也就“被逼”变成了了一种特殊的链表。

2、内核链表的原理

（1）将传统链表中的“链”抽象出来，取出节点中的具体数据，只保留逻辑的双向指针。使之成为一条只包含前后指针的“纯 粹“的双循环链表

（2）将链表“寄宿”于具体的数据节点之后，使他贯穿这些节点，可以通过一定的方式得到这个指针域的数据节点

传统



内核链表：



在Linux内核源码中，list.h保存在内核目录中的 inlucde/linux/list.h

3、查看内核链表：

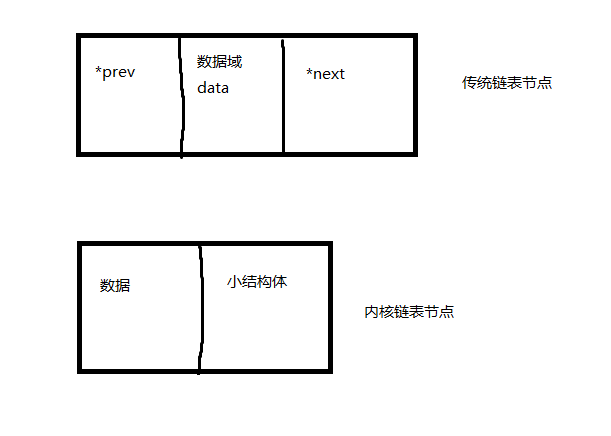
小结构体声明：

//指针域部分，小结构体，用它来构建“纯粹链表”

struct list\_head {

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

};

3.1、初始化链表

//节点结构体变量初始化

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

#define LIST\_HEAD(name) \

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

//节点结构体指针初始化

#define INIT\_LIST\_HEAD(ptr) do { \

(ptr)->next = (ptr); (ptr)->prev = (ptr); \

} while (0)

3.2、插入节点

//将new节点插入到prev和next节点之间

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;

}

//将new节点插入到head节点后面

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

{

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

}

//将new节点插入到head的前面

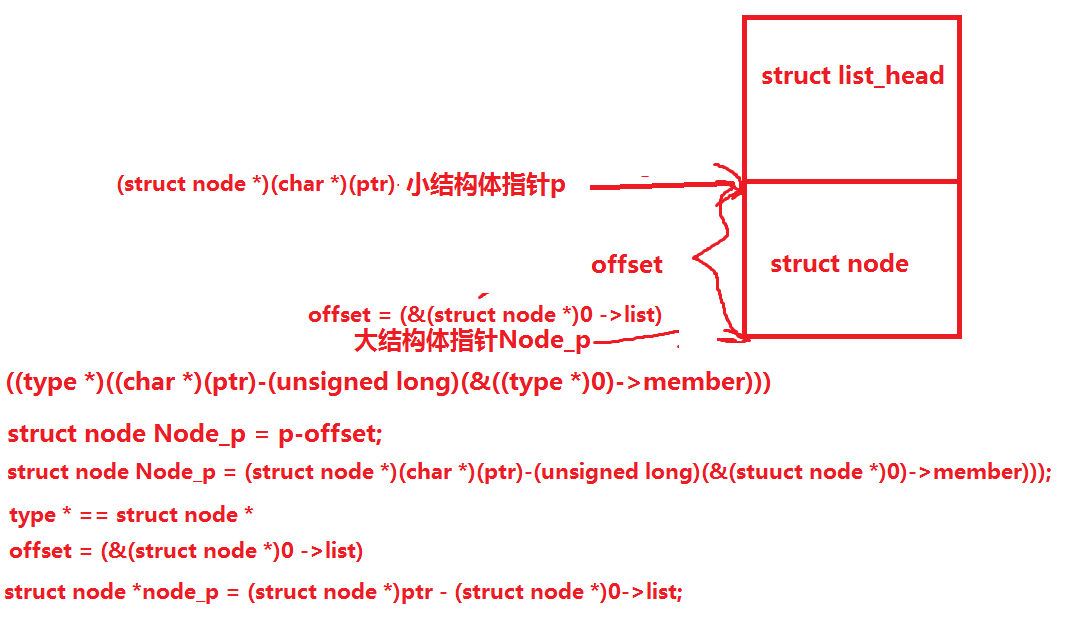
static inline void list\_add\_tail(struct list\_head \*new, struct list\_head \*head)

{

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

}

遍历节点（获取节点）



3.3、获取节点数据

//小结构体指针ptr，大结构体类型type，小结构体在大结构体内部的成员名list

//小结构体指针的地址，减去跟大结构体地址之间的偏移量

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

((type \*)((char \*)(ptr)-(unsigned long)(&((type \*)0)->member)))

3.4、遍历链表

//遍历链表的for循环，每一次循环体内得到的就是一个小结构体指针

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

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

pos = pos->next)

//安全的遍历，防止在循环体中有对节点的删除操作，这样会导致下一次进入循环体出现段错误

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

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

pos = n, n = pos->next)

//遍历过程中，得到大结构体指针:

//pos是遍历过程中的大结构体指针变量，

//head是小结构体链表头指针

//member小结构体在大结构体里面的成员名

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

for (pos = list\_entry((head)->next, typeof(\*pos), member); \

&pos->member != (head); \

pos = list\_entry(pos->member.next, typeof(\*pos), member))

//pos是遍历过程中的大结构体指针变量，

//n是遍历过程中的大结构体指针变量，

//head是小结构体链表头指针

//member小结构体在大结构体里面的成员名

#define list\_for\_each\_entry\_safe(pos, n, head, member) \

for (pos = list\_entry((head)->next, typeof(\*pos), member), \

n = list\_entry(pos->member.next, typeof(\*pos), member); \

&pos->member != (head); \

pos = n, n = list\_entry(n->member.next, typeof(\*n), member))

//从后往前遍历

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

for (pos = (head)->prev; pos != (head); \

pos = pos->prev)

3.5、取出节点

//取出prev和next节点之间的结点

static inline void \_\_list\_del(struct list\_head \*prev, struct list\_head \*next)

{

next->prev = prev;

prev->next = next;

}

//取出entry结点：让entry独立出来,前后指针域指向0地址

static inline void list\_del(struct list\_head \*entry)

{

\_\_list\_del(entry->prev, entry->next);

entry->next = (void \*) 0;

entry->prev = (void \*) 0;

}

//取出entry结点：让entry独立出来,前后指针域指向自己的地址

static inline void list\_del\_init(struct list\_head \*entry)

{

\_\_list\_del(entry->prev, entry->next);

INIT\_LIST\_HEAD(entry);

}

3.6、移动节点

//将list移动到head后面

static inline void list\_move(struct list\_head \*list,

struct list\_head \*head)

{

\_\_list\_del(list->prev, list->next);

list\_add(list, head);

}

//将list移动到head前面

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

struct list\_head \*head)

{

\_\_list\_del(list->prev, list->next);

list\_add\_tail(list, head);

}

3.7、合并链表

//将list链表与head链表进行拼接

//拼接完的链表头是head

//比如：list: 1 2 3 4 5 6 7

// head: 11 22 33 44

// 结果:head : 1 2 3 4 5 6 7 11 22 33 44

static inline void \_\_list\_splice(struct list\_head \*list,

struct list\_head \*head)

{

struct list\_head \*first = list->next;

struct list\_head \*last = list->prev;

struct list\_head \*at = head->next;

first->prev = head;

head->next = first;

last->next = at;

at->prev = last;

}

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

{

if (!list\_empty(list))

\_\_list\_splice(list, head);

}

3.8、空链表判断

//空链表判断

static inline int list\_empty(struct list\_head \*head)

{

return head->next == head;

}

小节：

1. 小结构体类型

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

1. 初始化小结构体链表（ptr是小结构体指针）

INIT\_LIST\_HEAD(ptr)

1. 插入节点

list\_add(struct list\_head \*new, struct list\_head \*head)

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

（4）取出节点

list\_del(struct list\_head \*entry)

（5）移动节点

list\_move(struct list\_head \*list, struct list\_head \*head)

list\_move\_tail(struct list\_head \*list, struct list\_head \*head)

（6）空链表判断

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

（7）两个链表的拼接

list\_splice(struct list\_head \*list, struct list\_head \*head)

list\_splice\_init(struct list\_head \*list, struct list\_head \*head)

（8）通过小结构体指针得到大结构体指针，数据访问需要

小结构体指针ptr，大结构体类型type，member:小结构体在大结构体内部的成员名list

list\_entry(ptr, type, member)

(9)遍历：实际上就是for循环的带参宏

list\_for\_each(pos, head)

list\_for\_each\_prev(pos, head)

list\_for\_each\_safe(pos, n, head)

list\_for\_each\_entry(pos, head, member)

list\_for\_each\_entry\_safe(pos, n, head, member)

练习1：设计一个内核链表，数据域只有一个正整数

1. 输入大于0的整数，将元素添加到链表里面，如果有相同的数，就不添加
2. 输入小于0的整数，将元素从链表里面删除，如果没有，则打印提示
3. 输入0，显示当前链表中的所有元素

练习2：

用内核链表创建一个数据集合，数据包含数据：1 2 3 4 5 6 7 8 9

要求将其重新排列成：9 7 5 3 1 2 4 6 8（奇数降序，偶数升序）并显示出来