# 内核基础‌

基础概念篇 1：链表

‌**引言**

找工作时常刷到链表指针操作的题，曾经思考在如今封装的函数库越来越多越来越便利的情况下，在做什么开发才会用到这些看似基础的链表指针操作呢？答案在Linux内核开发中会经常遇到（至少是常见到）。由于Linux内核没有使用任何C语言库，最初的内核开发者通过手动重新实现并封装了一些基础函数库/函数宏，以供后续开发人员服用，其中就包括链表的基本函数/函数宏。这些基础函数/函数宏在一些书籍中也被称作工具层，而应用这些工具进行开发的内核代码也被称作应用层。了解这些基础函数，是日后做任务调度、内存管理、设备驱动等开发的基础。

本章介绍对于链表Linux内核封装了哪些基础函数。使用list\_del删除节点后是否需要释放内存？是先删除还是先释放？list\_for\_each、list\_for\_each\_safe、list\_for\_each\_entry和list\_for\_each\_entry\_safe有什么区别？如果不能立即回答出这些基础问题，那么可以通过本章重温。

1. **链表基本函数**

Linux内核采用struct list\_head作为通用链表节点，其双向循环结构通过next和prev指针实现闭环管理。这种设计使链表操作与具体数据结构解耦，仅需在业务结构体中嵌入list\_head成员即可复用链表功能。例如进程调度模块的task\_struct即通过嵌套list\_head实现就绪队列。

* 1. **创建和初始化链表**

在Linux内核中，链表的创建和初始化可以通过以下两种方式进行：

### 1. 静态初始化

在‌**编译时‌**完成内存分配，通过宏（如LIST\_HEAD(name)）直接定义链表头并初始化，链表头位于全局数据段或栈内存中。

* ‌特点‌：无需运行时内存申请，链表头内存地址固定且连续。

‌**用法示例：**

static LIST\_HEAD(my\_static\_list); *// 定义并初始化静态链表头*

### ‌2. 动态初始化

在‌**运行时‌**通过函数INIT\_LIST\_HEAD(&list\_head)初始化，链表头需先动态分配内存（如kmalloc）。

* ‌特点‌：链表头内存位置不固定，需手动管理内存释放。

‌**用法示例**‌：

struct list\_head \*dynamic\_list = kmalloc(sizeof(struct list\_head), GFP\_KERNEL);

INIT\_LIST\_HEAD(dynamic\_list); *// 初始化动态分配的链表头*

在这里，INIT\_LIST\_HEAD宏将my\_dynamic\_list的next和prev指针都指向自身，表示初始化一个空链表。注意，在实际的内核代码中，通常不需要显式分配list\_head结构体的内存，因为它通常作为其他结构体的成员嵌入其中，并与其他结构体一起分配内存。上面的示例主要是为了说明动态初始化的过程。

* 1. **添加链表节点**

‌**list\_add/\_\_list\_add‌：**

* 功能：将一个新元素添加到链表的头部或尾部。
* 指针操作：\_\_list\_add函数是list\_add的内部实现，它调整新元素以及相邻元素的prev和next指针，确保链表结构的正确性。

**用法示例‌（\_\_list\_add简化版）：**

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;

}

* 1. **删除链表节点**

**list\_del‌：**

* 功能：从链表中删除一个元素。
* 指针操作：list\_del函数会调整被删除元素前后元素的prev和next指针，确保链表在删除元素后仍然保持正确结构。

**用法示例**‌：

struct my\_struct \*item = ...; // *假设这是你要删除的元素*

list\_del(&item->list); // *先从链表中删除*

kfree(item); // *然后释放内存*

**注意：**在调用list\_del之后，通常需要对被删除的元素进行kfree操作以释放内存，但具体是先kfree还是先list\_del取决于你的代码逻辑。一般来说，你应该先调用list\_del将元素从链表中移除，然后再调用kfree释放内存。这是因为如果先释放内存，那么链表中的指针将变成悬空指针，可能导致未定义行为。

* 1. **遍历链表节点**

**list\_for\_each/list\_for\_each\_safe/list\_for\_each\_entry/list\_for\_each\_entry\_safe:**

* 功能：遍历链表中的每个元素。
* 指针操作：通过移动一个临时指针来遍历链表中的每个节点。

### 1. list\_for\_each

list\_for\_each宏用于遍历一个由list\_head结构体组成的链表。它直接操作链表节点，不涉及链表节点所嵌入的结构体。

‌**用法示例**‌：

struct list\_head my\_list;

*// 假设my\_list已经被初始化并填充了数据*

struct list\_head \*pos;

list\_for\_each(pos, &my\_list) {

*// 在这里操作pos指向的list\_head节点*

}

### 2. list\_for\_each\_safe

list\_for\_each\_safe是list\_for\_each的一个安全版本，它提供了两个指针来遍历链表。当在遍历链表的同时需要删除节点时，应该使用list\_for\_each\_safe，以避免访问无效内存的问题。

‌**用法示例**‌：

struct list\_head my\_list;

*// 假设my\_list已经被初始化并填充了数据*

struct list\_head \*pos, \*tmp;

list\_for\_each\_safe(pos, tmp, &my\_list) {

if (some\_condition(pos)) {

list\_del(pos);

*// 执行其他操作，如释放内存等*

}

*// 在这里操作pos指向的list\_head节点（如果需要的话）*

}

在这个例子中，tmp指针用于在删除pos指向的节点后继续安全地遍历链表。

### 3. list\_for\_each\_entry

list\_for\_each\_entry宏用于遍历一个由list\_head结构体嵌入到其他结构体中的链表。它提供了直接访问嵌入list\_head的结构体的方式。

‌**用法示例**‌：

struct my\_struct {

struct list\_head list;

int data;

*// 其他成员...*

};

LIST\_HEAD(my\_list);

*// 假设my\_list已经被初始化并填充了my\_struct类型的数据*

struct my\_struct \*pos;

list\_for\_each\_entry(pos, &my\_list, list) {

*// 在这里操作pos指向的my\_struct结构体*

printk(KERN\_INFO "Data: %d\n", pos->data);

}

在这个例子中，list\_for\_each\_entry宏的第三个参数list是my\_struct结构体中嵌入的list\_head成员的名称。

### 4. list\_for\_each\_entry\_safe

list\_for\_each\_entry\_safe宏是list\_for\_each\_entry的一个安全版本。当在遍历链表（链表节点嵌入到其他结构体中）的同时需要删除节点时，应该使用list\_for\_each\_entry\_safe，以避免访问无效内存的问题。

‌**用法示例**‌：

struct my\_struct {

struct list\_head list;

int data;

*// 其他成员...*

};

LIST\_HEAD(my\_list);

*// 假设my\_list已经被初始化并填充了my\_struct类型的数据*

struct my\_struct \*pos, \*tmp;

list\_for\_each\_entry\_safe(pos, tmp, &my\_list, list) {

if (some\_condition(pos)) {

list\_del(&pos->list);

kfree(pos); *// 或者其他适当的内存释放操作*

}

*// 在这里操作pos指向的my\_struct结构体（如果需要的话）*

}

在这个例子中，tmp指针用于在删除pos指向的节点后继续安全地遍历链表。

**总结**

* list\_for\_each：用于遍历纯链表节点。
* list\_for\_each\_safe：list\_for\_each的安全版本，适用于在遍历过程中需要删除节点的场景。
* list\_for\_each\_entry：用于遍历嵌入list\_head的结构体链表，并直接访问这些结构体。
* list\_for\_each\_entry\_safe：list\_for\_each\_entry的安全版本，适用于在遍历过程中需要删除节点的场景。
  1. **其他常用函数**
* ‌list\_empty：检查链表是否为空。
* list\_is\_last：测试是否为链表的最后一个条目。
* list\_move/list\_move\_tail：从一个链表中删除并加入另一个链表的头部或尾部。
* list\_splice/list\_splice\_tail：将两个链表合并。

1. **用法示例**
   1. **代码示例**

下面给出一个代码示例，展示链表基本函数用法。示例定义一个简单的结构体my\_struct，它包含一个整数数据和一个链表头，并展示了如何初始化链表，如何添加、删除和遍历链表元素：

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/list.h>

#include <linux/slab.h>

struct my\_struct {

int data;

struct list\_head list;

};

LIST\_HEAD(my\_list); // 定义一个链表头

static int \_\_init my\_module\_init(void)

{

struct my\_struct \*item1, \*item2;

// 分配并初始化链表元素

item1 = kmalloc(sizeof(\*item1), GFP\_KERNEL);

if (!item1)

return -ENOMEM;

item1->data = 1;

INIT\_LIST\_HEAD(&item1->list); // 动态初始化链表元素头

item2 = kmalloc(sizeof(\*item2), GFP\_KERNEL);

if (!item2) {

kfree(item1);

return -ENOMEM;

}

item2->data = 2;

INIT\_LIST\_HEAD(&item2->list); // 动态初始化链表元素头

// 但实际上，当我们将元素添加到链表中时，不需要单独初始化每个元素的list头，

// 因为list\_add宏会处理这些细节。这里的INIT\_LIST\_HEAD调用是多余的，

// 仅用于展示如何初始化list\_head结构体。

// 将元素添加到链表中

list\_add(&item1->list, &my\_list);

list\_add(&item2->list, &my\_list);

// 遍历链表并打印元素数据

struct list\_head \*pos;

list\_for\_each(pos, &my\_list) {

struct my\_struct \*item = list\_entry(pos, struct my\_struct, list);

printk(KERN\_INFO "Data: %d\n", item->data);

}

// 从链表中删除元素并释放内存

list\_del(&item1->list);

kfree(item1);

list\_del(&item2->list);

kfree(item2);

return 0;

}

static void \_\_exit my\_module\_exit(void)

{

// 链表在模块退出时已经被清空，无需额外操作

}

module\_init(my\_module\_init);

module\_exit(my\_module\_exit);

* 1. **开发场景示例**

1. **设备驱动**：USB设备链通过list\_head维护热插拔设备列表。
2. **内存管理**：vm\_area\_struct用链表组织进程地址空间映射。
3. **进程调度**：CFS调度器使用红黑树+链表组合管理运行队列。