内核链表学习笔记

|  |  |  |  |
| --- | --- | --- | --- |
| Prepared by  编制 | 沈浩杰 | Date  日期 | yyyy-mm-dd  2022-12-12 |

修订记录

| 日期 | 版本号 | 修改描述 | 作者 |
| --- | --- | --- | --- |
| *yyyy-mm-dd* | *0.1* | *初稿发布* | *xx* |
|  |  |  |  |

目录

[1 offsetof介绍 5](#_Toc121760134)

[1.1 实例代码 5](#_Toc121760135)

[2 container\_of 6](#_Toc121760136)

[2.1 实例代码 7](#_Toc121760137)

[3 Linux内核中双向链表实现 8](#_Toc121760138)

[3.1 Linux中双向链表的使用思想 8](#_Toc121760139)

[3.2 内核抠出列表实现 9](#_Toc121760140)

[3.3 链表操作接口 10](#_Toc121760141)

[**3.3.1** 声明和初始化 10](#_Toc121760142)

[**3.3.2** 插入/删除/合并 11](#_Toc121760143)

[**3.3.3** 遍历 15](#_Toc121760144)

[4 安全性考虑 19](#_Toc121760145)

[4.1 list\_empty()判断 19](#_Toc121760146)

[4.2 遍历时节点删除 19](#_Toc121760147)

[4.3 测试代码 19](#_Toc121760148)

List of Tables

**未找到图形项目表。**

List of Figures

[Figure 3.1 双链表示意图 8](#_Toc121760149)

[Figure 3.2 nf\_sockopts链表示意图 10](#_Toc121760150)

[Figure 3.3链表合并list\_splice(&list1,&list2) 15](#_Toc121760151)

[Figure 3.4 offsetof()宏的原理 17](#_Toc121760152)

# offsetof介绍

在Linux内核中stddef.h文件中定义offsetof宏，其功能是获取某个结构体（Type）的变量成员（member）在结构体中的偏移字节数。

|  |
| --- |
| #define offsetof(TYPE, MEMBER) ((size\_t) &((TYPE \*)0)->MEMBER)    (01) ( (TYPE \*)0 ) 将零转型为TYPE类型指针，即TYPE类型的指针的地址是0。  (02) ((TYPE \*)0)->MEMBER 访问结构中的数据成员。  (03) &( ( (TYPE \*)0 )->MEMBER ) 取出数据成员的地址。由于TYPE的地址是0，这里获取到的地址就是相对MEMBER在TYPE中的偏移。  (04) (size\_t)(&(((TYPE\*)0)->MEMBER)) 结果转换类型。对于32位系统而言，size\_t是unsigned int类型；对于64位系统而言，size\_t是unsigned long类型。 |

## 实例代码

|  |
| --- |
| #include <stdio.h>    // 获得结构体(TYPE)的变量成员(MEMBER)在此结构体中的偏移量。  #define offsetof(TYPE, MEMBER) ((size\_t) &((TYPE \*)0)->MEMBER)    struct student  {  char gender;  int id;  int age;  char name[20];  };    void main()  {  int gender\_offset, id\_offset, age\_offset, name\_offset;    gender\_offset = offsetof(struct student, gender);  id\_offset = offsetof(struct student, id);  age\_offset = offsetof(struct student, age);  name\_offset = offsetof(struct student, name);    printf("gender\_offset = %d\n", gender\_offset);  printf("id\_offset = %d\n", id\_offset);  printf("age\_offset = %d\n", age\_offset);  printf("name\_offset = %d\n", name\_offset);  } |

简单说说"为什么id的偏移值是4，而不是1"。我的运行环境是linux系统，32位的x86架构。这就意味着cpu的数据总线宽度为32，每次能够读取4字节数据。gcc对代码进行处理的时候，是按照4字节对齐的。所以，即使gender是char(一个字节)类型，但是它仍然是4字节对齐的！

# container\_of

在Linux的kernel.h文件中定义宏container\_of，其功能是根据结构体（Type）变量中的域成员变量（member）的指针ptr来获取指向整个结构体变量的指针（即地址）。问题就相当简单了：已知'整体'和该整体中'某一个部分'，要根据该部分的地址，计算出整体的地址。

|  |
| --- |
| #define ccontainer\_of(ptr, type, member) ({\  const typeof( ((type \*)0)->member ) \*\_\_mptr = (ptr); \  (type \*)(char \*)\_\_mptr - offsetof(type, member) );})    (01) typeof( ( (type \*)0)->member ) 取出member成员的变量类型。  (02) const typeof( ((type \*)0)->member ) \*\_\_mptr = (ptr) 定义变量\_\_mptr指针，并将ptr赋值给\_\_mptr。经过这一步，\_\_mptr为member数据类型的常量指针，其指向ptr所指向的地址。  (04) (char \*)\_\_mptr 将\_\_mptr转换为字节型指针。  (05) offsetof(type,member)) 就是获取"member成员"在"结构体type"中的位置偏移。  (06) (char \*)\_\_mptr - offsetof(type,member)) 就是用来获取"结构体type"的指针的起始地址（为char \*型指针）。  (07) (type \*)( (char \*)\_\_mptr - offsetof(type,member) ) 就是将"char \*类型的结构体type的指针"转换为"type \*类型的结构体type的指针"。 |

## 实例代码

|  |
| --- |
| #include <stdio.h>  #include <string.h>    // 获得结构体(TYPE)的变量成员(MEMBER)在此结构体中的偏移量。  #define offsetof(TYPE, MEMBER) ((size\_t) &((TYPE \*)0)->MEMBER)    // 根据"结构体(type)变量"中的"域成员变量(member)的指针(ptr)"来获取指向整个结构体变量的指针  #define container\_of(ptr, type, member) ({ \  const typeof( ((type \*)0)->member ) \*\_\_mptr = (ptr); \  (type \*)( (char \*)\_\_mptr - offsetof(type,member) );})    struct student  {  char gender;  int id;  int age;  char name[20];  };    void main()  {  struct student stu;  struct student \*pstu;    stu.gender = '1';  stu.id = 9527;  stu.age = 24;  strcpy(stu.name, "zhouxingxing");    // 根据"id地址" 获取 "结构体的地址"。  pstu = container\_of(&stu.id, struct student, id);    // 根据获取到的结构体student的地址，访问其它成员  printf("gender= %c\n", pstu->gender);  printf("age= %d\n", pstu->age);  printf("name= %s\n", pstu->name);  } |

# Linux内核中双向链表实现

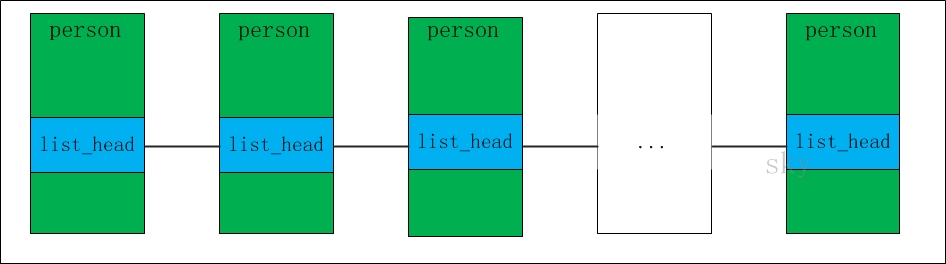
在内核中双向链表通过type.h和list.h两个文件实现，其中type.h给出列表头定义，list.h通过宏的方式给出列表的基本操作。

## Linux中双向链表的原理

它是将双向链表节点嵌套在其它的结构体中；在遍历链表的时候，根据双链表节点的指针获取"它所在结构体的指针"，从而再获取数据。

我举个例子来说明，可能比较容易理解。假设存在一个社区中有很多人，每个人都有姓名和年龄。通过双向链表将人进行关联的模型图如下，person代表人，它有name和age属性。为了通过双向链表对person进行链接，我们在person中添加了list\_head属性。通过list\_head，我们就将person关联起来了。

Figure 3.1 双链表示意图



|  |
| --- |
| struct person  {  int age;  char name[20];  struct list\_head list;  }; |

## 内核抠出列表实现

尽管这里使用2.6内核作为讲解的基础，但实际上2.4内核中的链表结构和2.6并没有什么区别。不同之处在于2.6扩充了两种链表数据结构：链表的读拷贝更新（rcu）和HASH链表（hlist）。

|  |
| --- |
| // 双向链表节点  // list\_head结构包含两个指向list\_head结构的指针prev和next，由此可见，内核的链表具备双  // 链表功能，实际上，通常它都组织成双循环链表。  struct list\_head {  struct list\_head \*next, \*prev;  }; |

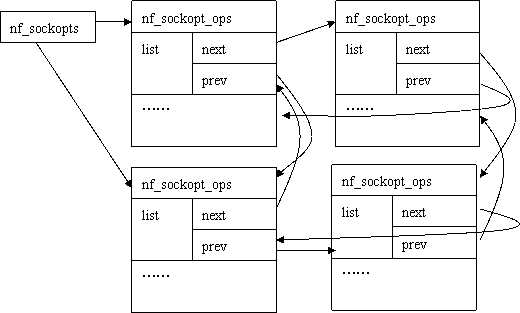
和第一节介绍的双链表结构模型不同，这里的list\_head没有数据域。在Linux内核链表中，不是在链表结构中包含数据，而是在数据结构中包含链表节点。在数据结构课本中，链表的经典定义方式通常是这样的（以单链表为例）：

|  |
| --- |
| struct list\_node {  struct list\_node \*next;  ElemType data;  }; |

因为ElemType的缘故，对每一种数据项类型都需要定义各自的链表结构。有经验的C++程序员应该知道，标准模板库中的<list>采用的是C++ Template，利用模板抽象出和数据项类型无关的链表操作接口。

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）。从下图中我们可以看到，这种通用的链表结构避免了为每个数据项类型定义自己的链表的麻烦。Linux的简捷实用、不求完美和标准的风格，在这里体现得相当充分。

Figure 3.2 nf\_sockopts链表示意图



## 链表操作接口

### 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？让我们来看看LIST\_HEAD()这个宏：

|  |
| --- |
| // 初始化节点：设置name节点的前继节点和后继节点都是指向name本身。  #define LIST\_HEAD\_INIT(name) { &(name), &(name) }    // 定义表头(节点)：新建双向链表表头name，并设置name的前继节点和后继节点都是指向name本身。  #define LIST\_HEAD(name) \  struct list\_head name = LIST\_HEAD\_INIT(name)  // 初始化节点：将list节点的前继节点和后继节点都是指向list本身。  static inline void INIT\_LIST\_HEAD(struct list\_head \*list)  {  list->next = list;  list->prev = list;  } |

|  |
| --- |
| #define LIST\_HEAD\_INIT(name) { &(name), &(name) }  #define LIST\_HEAD(name) struct list\_head name = LIST\_HEAD\_INIT(name) |

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：

|  |
| --- |
| static inline int list\_empty(const struct list\_head \*head)  {  return head->next == head;  } |

### 插入/删除/合并

#### 插入

对链表的插入操作有两种：在表头插入和在表尾插入。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

|  |
| --- |
| // 添加节点：将new插入到prev和next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;  }    // 添加new节点：将new添加到head之后，是new称为head的后继节点。  static inline void list\_add(struct list\_head \*new, struct list\_head \*head)  {  \_\_list\_add(new, head, head->next);  }    // 添加new节点：将new添加到head之前，即将new添加到双链表的末尾。  static inline void list\_add\_tail(struct list\_head \*new, struct list\_head \*head)  {  \_\_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。

#### 删除

|  |
| --- |
| // 从双链表中删除entry节点。  static inline void \_\_list\_del(struct list\_head \* prev, struct list\_head \* next)  {  next->prev = prev;  prev->next = next;  }    // 从双链表中删除entry节点。  static inline void list\_del(struct list\_head \*entry)  {  \_\_list\_del(entry->prev, entry->next);  }    // 从双链表中删除entry节点。  static inline void \_\_list\_del\_entry(struct list\_head \*entry)  {  \_\_list\_del(entry->prev, entry->next);  }    // 从双链表中删除entry节点，并将entry节点的前继节点和后继节点都指向entry本身。  static inline void list\_del\_init(struct list\_head \*entry)  {  \_\_list\_del\_entry(entry);  INIT\_LIST\_HEAD(entry);  } |

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。

#### 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); |

例如list\_move(&new\_sockopt.list,&nf\_sockopts)会把new\_sockopt从它所在的链表上删除，并将其再链入nf\_sockopts的表头。

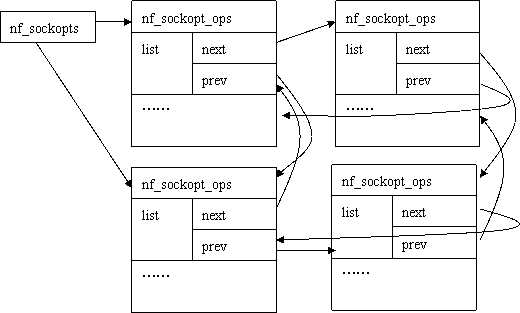
#### 合并

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：

|  |
| --- |
| // 用new节点取代old节点  static inline void list\_replace(struct list\_head \*old,  struct list\_head \*new)  {  new->next = old->next;  new->next->prev = new;  new->prev = old->prev;  new->prev->next = new;  } |

假设当前有两个链表，表头分别是list1和list2（都是struct list\_head变量），当调用list\_splice(&list1,&list2)时，只要list1非空，list1链表的内容将被挂接在list2链表上，位于list2和list2.next（原list2表的第一个节点）之间。新list2链表将以原list1表的第一个节点为首节点，而尾节点不变。如图（虚箭头为next指针）：

Figure 3.3链表合并list\_splice(&list1,&list2)



当list1被挂接到list2之后，作为原表头指针的list1的next、prev仍然指向原来的节点，为了避免引起混乱，Linux提供了一个list\_splice\_init()函数：

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

该函数在将list合并到head链表的基础上，调用INIT\_LIST\_HEAD(list)将list设置为空链。

### 遍历

|  |
| --- |
| // 遍历双向链表  #define list\_for\_each(pos, head) \  for (pos = (head)->next; pos != (head); pos = pos->next)    #define list\_for\_each\_safe(pos, n, head) \  for (pos = (head)->next, n = pos->next; pos != (head); \  pos = n, n = pos->next)    #define list\_entry(ptr, type, member) \  container\_of(ptr, type, member) |

遍历是链表最经常的操作之一，为了方便核心应用遍历链表，Linux链表将遍历操作抽象成几个宏。在介绍遍历宏之前，我们先看看如何从链表中访问到我们真正需要的数据项。

#### 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)  container\_of宏定义在[include/linux/kernel.h]中：  #define container\_of(ptr, type, member) ({ \  const typeof( ((type \*)0)->member ) \*\_\_mptr = (ptr); \  (type \*)( (char \*)\_\_mptr - offsetof(type,member) );})  offsetof宏定义在[include/linux/stddef.h]中：  #define offsetof(TYPE, MEMBER) ((size\_t) &((TYPE \*)0)->MEMBER) |

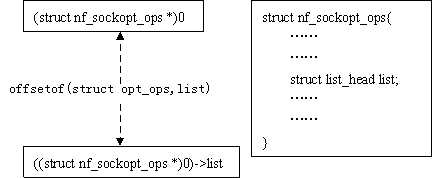
size\_t最终定义为unsigned int（i386）。

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。

如果这么说还不好理解的话，不妨看看下面这张图：

Figure 3.4 offsetof()宏的原理



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。

#### 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：

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

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){  ……  }  …… |

某些应用需要反向遍历链表，Linux提供了list\_for\_each\_prev()和list\_for\_each\_entry\_reverse()来完成这一操作，使用方法和上面介绍的list\_for\_each()、list\_for\_each\_entry()完全相同。

如果遍历不是从链表头开始，而是从已知的某个节点pos开始，则可以使用list\_for\_each\_entry\_continue(pos,head,member)。有时还会出现这种需求，即经过一系列计算后，如果pos有值，则从pos开始遍历，如果没有，则从链表头开始，为此，Linux专门提供了一个list\_prepare\_entry(pos,head,member)宏，将它的返回值作为list\_for\_each\_entry\_continue()的pos参数，就可以满足这一要求。

# 安全性考虑

在并发执行的环境下，链表操作通常都应该考虑同步安全性问题，为了方便，Linux将这一操作留给应用自己处理。Linux链表自己考虑的安全性主要有两个方面：

## list\_empty()判断

基本的list\_empty()仅以头指针的next是否指向自己来判断链表是否为空，Linux链表另行提供了一个list\_empty\_careful()宏，它同时判断头指针的next和prev，仅当两者都指向自己时才返回真。这主要是为了应付另一个cpu正在处理同一个链表而造成next、prev不一致的情况。但代码注释也承认，这一安全保障能力有限：除非其他cpu的链表操作只有list\_del\_init()，否则仍然不能保证安全，也就是说，还是需要加锁保护。

## 遍历时节点删除

前面介绍了用于链表遍历的几个宏，它们都是通过移动pos指针来达到遍历的目的。但如果遍历的操作中包含删除pos指针所指向的节点，pos指针的移动就会被中断，因为list\_del(pos)将把pos的next、prev置成LIST\_POSITION2和LIST\_POSITION1的特殊值。

当然，调用者完全可以自己缓存next指针使遍历操作能够连贯起来，但为了编程的一致性，Linux链表仍然提供了两个对应于基本遍历操作的"\_safe"接口：list\_for\_each\_safe(pos, n, head)、list\_for\_each\_entry\_safe(pos, n, head, member)，它们要求调用者另外提供一个与pos同类型的指针n，在for循环中暂存pos下一个节点的地址，避免因pos节点被释放而造成的断链。

## 测试代码

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include "list.h"    struct person  {  int age;  char name[20];  struct list\_head list;  };    void main(int argc, char\* argv[])  {  struct person \*pperson;  struct person person\_head;  struct list\_head \*pos, \*next;  int i;    // 初始化双链表的表头  INIT\_LIST\_HEAD(&person\_head.list);    // 添加节点  for (i=0; i<5; i++)  {  pperson = (struct person\*)malloc(sizeof(struct person));  pperson->age = (i+1)\*10;  sprintf(pperson->name, "%d", i+1);  // 将节点链接到链表的末尾  // 如果想把节点链接到链表的表头后面，则使用 list\_add  list\_add\_tail(&(pperson->list), &(person\_head.list));  }    // 遍历链表  printf("==== 1st iterator d-link ====\n");  list\_for\_each(pos, &person\_head.list)  {  pperson = list\_entry(pos, struct person, list);  printf("name:%-2s, age:%d\n", pperson->name, pperson->age);  }    // 删除节点age为20的节点  printf("==== delete node(age:20) ====\n");  list\_for\_each\_safe(pos, next, &person\_head.list)  {  pperson = list\_entry(pos, struct person, list);  if(pperson->age == 20)  {  list\_del\_init(pos);  free(pperson);  }  }    // 再次遍历链表  printf("==== 2nd iterator d-link ====\n");  list\_for\_each(pos, &person\_head.list)  {  pperson = list\_entry(pos, struct person, list);  printf("name:%-2s, age:%d\n", pperson->name, pperson->age);  }    // 释放资源  list\_for\_each\_safe(pos, next, &person\_head.list)  {  pperson = list\_entry(pos, struct person, list);  list\_del\_init(pos);  free(pperson);  }    } |