- 我的首页
- 我的信息
- 留言板子
- 与我聊天
- 纯 C 语言
- Linux 研究
- Lua 学习
- TeX 研究
- 龙芯相关
- 工具使用
- 代码研究
- 想法片断
- 量子世界
- 所谓艺术
- 名人传记
- 列表文档
- 有趣网摘
- 转载文章
- 站点管理
- 标签列表

edit this panel

通用链表及其API

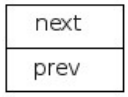
Linux 研究 » 通用链表及其API

内核通用链表

操作系统内核经常需要维护数据结构。内核中有标准的循环链表，双向链表的实现。

在内核源代码包的 `include/linux/list.h` 中，定义了一个 `list_head` 类型简单结构（实际上，这个文件中声明有一套完整的操作这个通用链表的接口函数）：

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

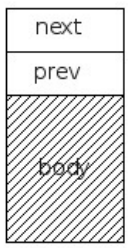


list_head 结构

这个结构相当于一个三向连接龙头，有两端分别连向前后的管道（同等结构体），另一端套在出水口上（附着结构体）。

我们使用 list_head 的一般方法是把它嵌入其它待连接的结构体中。下面来说明它的用法。

```
struct foo_rec {
    struct list_head listpin;
    int priority;
    ......              /* 其它成员 */
};
```



含 list_head 的自定义结构

由于内核实际上是封装了 list_head 的操作，所以我们不要自已去操作这个结构体，而应该用系统提供的接口。

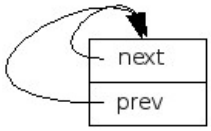
创建 list_head 结构（下面用到的两个宏都定义在 include/linux/list.h 中）
用

```
struct list_head listpin;
INIT_LIST_HEAD( &listpin );
```

或

```
LIST_HEAD( listpin );
```

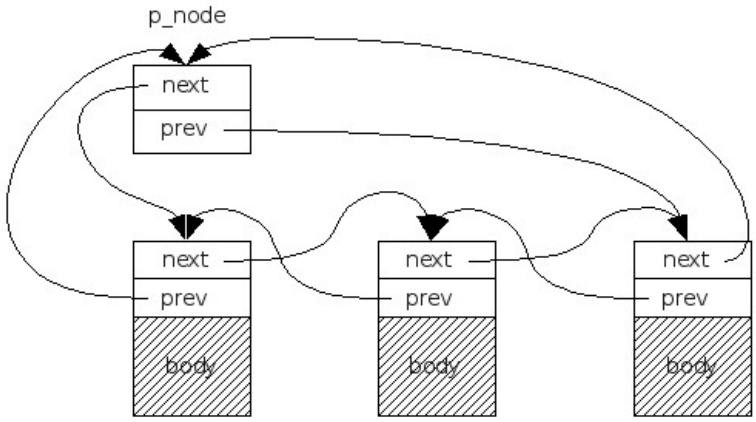
就初始化了一个链接头，初始化的结果如下：



刚初始化后的 list_head 结构体

在结构体的定义中我们已经把链接头与负载结构体合起来，下面我们做一些操作使其生成一串链表，并能方便地从生成的链表中访问负载的那些结构成员。

生成的链表的示意图如下：



链表形成环路

因为 list_head 的结构定义，我们很容易将其链成一串链表，INIT_LIST_HEAD 宏已经做了初步工作，接着只需做几次简单的赋值操作即可生成链表。但现在关键的问题在于生成的链表只是 list_head 结构体 作为节点的链表，我们常常需要的不是这个，而是要把自定义的包含 list_head 的结构体通过这个 list_head 串起来。下面提供了实现方法。

list_entry 宏用于从 listpin 连接头中抽取出指向 foo_rec 结构体的指针，过程非常巧妙。通过这种方法把整个链表连接起来了。
语法如下：

```
struct foo_rec *p_node =
    list_entry( ptr, struct foo_rec, listpin );

其中，
ptr 是一个指向 struct list_head 链接头的指针；
struct foo_rec 是用户自己定义的那个结构体；
listpin 就是我们定义的那个结构体中的那个 struct list_head 类型指针。
```

对 list_entry(ptr, type, member) 宏参数解释如下：
ptr 是一个 struct list_head 结构元素指针，指向自定义结构中的 list_head 链表头；
type 是用户定义的结构类型。其中包含 struct list_head 链接头；
member 是用户定义的结构中的 struct list_head 结构成员的名字。

list_entry 在定义时，实际上是另一个宏 container_of 宏的别名，在 include/linux/list.h 中定义如下：

```
/**
 * 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) \
    container_of(ptr, type, member)
```

而 container_of 宏在 include/linux/kernel.h 中定义

```
/**
 * container_of - cast a member of a structure out to the containing
 * @ptr:    the pointer to the member.
 * @type:    the type of the container struct this is embedded in.
 * @member:    the name of the member within the struct.
 *
 */
#define container_of(ptr, type, member) ({            \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
        (type *)( (char *)__mptr - offsetof(type,member) );})
```

对此宏详细解释如下：

此宏第一句，

```
const typeof( ((type *)0)->member ) *__mptr = (ptr);
```

定义了一个临时变量 __mptr ，它的类型为 type 中的 member 的类型，即是上例中的 foo_rec 中的 struct list_head 的类型，然后把指针 ptr 的值赋给 __mptr，这样的话，让 __mptr 也指向了（一般来说是另一个）节点中的 list_head 成员，上例中即 listpin。这里面，typeof 和 0 被用得出神入化，令人称奇！

第二句，

```
(type *)( (char *)__mptr - offsetof(type, member)
```

用 offsetof 宏求得 list_head 成员在 type 结构中的偏移，并用 __mptr 去减，这样就得到了那个节点的起始地址。这样做的原因是，list_head 结构在 type 中并不一定总是第一个成员（如果是第一个成员，就像我们上面的例子那样，那么 list_head 结构的起始地址就是 type 结构的起始地址），用这样的计算方式，就能够更加通用了。注意，这句话只是一个算术表达式，C语言里面，一个表达式会产生一个结果。要使用这个结果时，只要把这个表达式赋给一个变量就行了。于是，可以把 list_entry() 整体赋给一个表达式，相当于一个函数调用。

offsetof 宏在 include/linux/stddef.h 中定义

```
#define offsetof( TYPE, MEMBER )  ((size_t) &((TYPE *)0)->MEMBER)
```

注意，上面的 & 号取的是 TYPE 结构体中的 MEMBER 成员的地址，由于这里写了个 0，其相对点成了 0，于是就成了偏移地址了。（变量在编译时看来还不就是内存中的一个地址）只能惊叹，作者对C语言功力真是至高境界了。

通用链表操作的简单例子

下面举一个使用 list_entry 的简单例子

假设我们现在已经有了一串按如上方式建立起来的链表，并且有了一个外部的 list_head 类型的指针 p_node，它指向这个链表的一个节点（开始时往往为头节点），现在我们想在这个链表中插入一个元素，按其 priority 属性进行由大到小方式插入。程序如下：

```
void test_add_entry( struct foo_rec *p_new )
{
    struct list_head *ptr;
    struct foo_rec *p_entry;

    for ( ptr = p_node->next; ptr != &p_node; ptr = ptr->next ) {
        p_entry = list_entry( ptr, struct foo_rec, listpin );
        if ( p_entry->priority < p_new->priority ) {
            list_add_tail( &p_new->listpin, ptr );
            return;
        }
    }
    list_add_tail( &p_new->listpin, &p_node );
}
```