Linux中采用环形双向链表
标记`head`节点后，方可进行遍历

链表
Linux内核链表的特殊实现：不是将数据结构塞入链表，而是将链表塞入数据结构
链表代码存于<linux/list.h>中
```c
struct list_head{
struct list_head *next;
struct list_head *prev;
}
```
利用`container_of`可以找到从链表指针开始找到父结构里的任何变量（因为结构的偏移地址在编译期就被ABI固定了）
```c
#define container_of(ptr, type, member) ({ \
const typeof( ((type*)0)->number ) * __mptr = (ptr); \
(type*)( (char*)__mptr - offsetof(type, member)); \
})
```
`list_entry`用于返回包含list_head的父结构体
```c
#define list_entry(ptr, typr, member) \
container_of(ptr, type, member)
```

假设有如下结构的链表
```c
struct idler{
unsigned long eat;
unsigned long sleep;
bool is_lazy;
struct list_head list;
}
```
则在初始化时可以使用如下方式，使其在编译期静态创建
```c
struct idler me = {
.eat = 7200,
.sleep = 36000,
.is_lazy = true,
.list = LIST_HEAD_INIT(me.list),
};
```
特殊的索引节点
```c
static LIST_HEAD(me);
```
链表操作
增加节点
在`head`节点之后插入一个new节点（可以实现栈）
```c
list_add(struct list_head *new, struct list_head *head)
```
在`head`节点之前插入一个new节点（可以实现队列）
```c
```

list_add_tail(struct list_head *new, struct list_head *head)
```

#### 删除节点
只是把该项从链中除去，但本身不释放该entry所占的空间
```
list_del(struct list_head *entry)
```

## 映射（HashMap）
Linux内核提供了一个简单的hash，目标是搞一个能够映射唯一表示UID到一个指针；
`struct idr`就是这个映射的结构，但是这个不是一个通用的结构
### 初始化
函数原型
```c
void idr_init(struct idr *idp);
```
分配
建立idr后就可以开始分配UID，过程分两步走
（1）告诉需要分配的UID，且允许在必要时调整后备树大小
成功返回1，失败返回0
```c
int idr_pre_get(struct idr *idp, gfp_t gfp_mask); //gfp是标识调整大小时的策略
```
（2）真正请求的UID
```c
int idr_get_new(struct idr *idp, void *ptr, int *id); //新的UID存于*id中，建立起了UID->ptr的映射关系
int idr_get_new_above(struct idr *idp, void *ptr, int starting_id, int*id); //可指定返回最小值，保证UID>=starting_id，可以保证分配期间具有唯一性
```
example
```c
// idr_get_new
int id;
do {
    if( !idr_pre_get(&idr_huh, GFP_KERNEL)
        return -ENOSPC;
    ret = idr_get_new(&idr_huh, ptr, &id);
} while (ret == -EAGAIN);
/*********************************************************/
// idr_get_new_above
int id;
do {
    if( !idr_pre_get(&idr_huh, GFP_KERNEL)
        return -ENOSPC;
    ret = idr_get_new_above(&idr_huh, ptr, next_id, &id);
} while (ret == -EAGAIN);
if (!ret)
   next_id = id + 1;
```

查找
成功则返回关联指针；失败返回NULL；
但是用`idr_get_new()`和`idr_get_new_above()`将空指针映射给UID，返回也是NULL，无法判断是否成功
```c
void *idr_find(struct idr *idp, int id); //在idp表中寻找UID=id的指针
```
```

```
一般不要把UID映射到空指针上去

### 删除
若删除成功，则关联指针一并删除；但是失败却没法提示错误
```c
void idr_remove(strcut idr *idp);
void idr_remove_all(strcut idr *idp); //强制删除所有UID
```
撤销
仅释放idr中未使用内存，但不释放已分配内存；所以要想全部删除，必须先调用
`idr_remove_all`后再调用如下函数
```c
void idr_destroy(strcut idr *idp);
```


红黑树
定义在lib/rbtree.c中，声明文件在<linux/rbtree.h>
创建一个红黑树，需要分配一个rb_root，并且初始化为RB_ROOT
```c
struct rb_root root = RB_ROOT;
```
树中其他节点由rb_node结构描述
```