lwip 内存分配

LWIP内存分配

内存堆:其优点就是内存浪费小，比较简单，适合用于小内存的管理，其缺点就是如果频繁的动态分配和释放，可能会造成严重的内存碎片，如果在碎片情况严重的话，可能会导致内存分配不成功。

内存池:内存池的特点是预先开辟多组固定大小的内存块组织成链表，实现简单，分配和回收速度快，不会产生内存碎片，但是大小固定，并且需要预估算准确。

堆内存

文件名 mem.c

初始化

```
struct heap_mem
{
    rt_uint16_t magic;  /* 用于标记是否被改写或使用过 */
    rt_uint16_t used; /* 用于标记是该堆是否在被使用 */
    rt_size_t next, prev; /* 记录相邻内存块的地址 */
};
```

```
// rt_system_heap_init
void rt_system_heap_init(void *begin_addr, void *end_addr)
{
    struct heap_mem *mem;
    rt_uint32_t begin_align = RT_ALIGN((rt_uint32_t)begin_addr, RT_ALIGN_SIZE);
    rt_uint32_t end_align = RT_ALIGN_DOWN((rt_uint32_t)end_addr, RT_ALIGN_SIZE);

    RT_DEBUG_NOT_IN_INTERRUPT;

    /* alignment addr */
    if ((end_align > (2 * SIZEOF_STRUCT_MEM)) &&
        ((end_align - 2 * SIZEOF_STRUCT_MEM) >= begin_align))
    {
        /* calculate the aligned memory size */
        mem_size_aligned = end_align - begin_align - 2 * SIZEOF_STRUCT_MEM;
    }
    else
    {
        rt_kprintf("mem init, error begin address 0x%x, and end address 0x%x\n",
                   (rt_uint32_t)begin_addr, (rt_uint32_t)end_addr);

        return;
    }

    /* 初始化第一个内存块 */
    heap_ptr = (rt_uint8_t *)begin_align;
```

```
    RT_DEBUG_LOG(RT_DEBUG_MEM, ("mem init, heap begin address 0x%x, size %d\n",
                                (rt_uint32_t)heap_ptr, mem_size_aligned));

    /* initialize the start of the heap */
    mem        = (struct heap_mem *)heap_ptr;
    mem->magic = HEAP_MAGIC;
    mem->next  = mem_size_aligned + SIZEOF_STRUCT_MEM;
    mem->prev  = 0;
    mem->used  = 0;
#ifdef RT_USING_MEMTRACE
    rt_mem_setname(mem, "INIT");
#endif

    /* 初始化最后一个内存块 */
    heap_end        = (struct heap_mem *)&heap_ptr[mem->next];
    heap_end->magic = HEAP_MAGIC;
    heap_end->used  = 1;
    heap_end->next  = mem_size_aligned + SIZEOF_STRUCT_MEM;
    heap_end->prev  = mem_size_aligned + SIZEOF_STRUCT_MEM;
#ifdef RT_USING_MEMTRACE
    rt_mem_setname(heap_end, "INIT");
#endif

    rt_sem_init(&heap_sem, "heap", 1, RT_IPC_FLAG_FIFO);

    /* 总是指向没有被使用的第一个堆的指针, */
    lfree = (struct heap_mem *)heap_ptr;
}
```

分配

`rt_malloc(rt_size_t size);` 从lfree指向的内存块及之后找出没有被使用的内存块,并修改lfree(加锁)

释放

`rt_free(void *rmem);` 将需要释放的内存块地址放回堆内存中, 并更新lfree(加锁)

内存池

文件名 memp.c

初始化

```
//初始化内存池 memp_pools 参数, 其中对不同类型的协议(TCP,UDP等) 各初始化相对应类型的内存池
const struct memp_desc* const memp_pools[MEMP_MAX] = {
#define LWIP_MEMPOOL(name,num,size,desc) &memp_ ## name,
#include "lwip/priv/memp_std.h"
};

//内存池结构
```

```
struct memp_desc {
  /** Textual description */
  const char *desc;
  /** Element size */
  u16_t size;
  /** Number of elements */
  u16_t num;
  /** Base address */
  u8_t *base;
  /** 内存池指向开辟出来的内存块链表 */
  struct memp **tab;
};

//初始化函数 memp.c
memp_init(void);
memp_init_pool(const struct memp_desc *desc);
```

分配

```
// do_memp_malloc_pool 从池中拿出一个内存块
memp = *desc->tab;
*desc->tab = memp->next;
memp->next = NULL;
```

释放

```
// do_memp_free_pool 将内存块放回内存池中
memp->next = *desc->tab;
*desc->tab = memp;
```

---

pbuf数据包管理

```
typedef enum {
  /** pbuf数据存储在RAM中，主要用于TX、struct pbuf及其有效负载在一个连续内存中分配(那么第一个有效负载字节可以从struct pbuf计算)。pbuf_alloc()将PBUF_RAM分配为未链接的pbufs(尽管可能如此),未来版本的更改)。这应该用于所有输出数据包(TX)*/
  PBUF_RAM,
  /**pbuf数据存储在ROM中，即struct pbuf及其有效载荷位于完全不同的存储区域。因为它指向ROM，所以有效载荷不指向ROM在排队传输时必须被复制。 */
  PBUF_ROM,
  /** pbuf来自pbuf池。很像PBUF_ROM，但是有效负载可能会改变,所以它必须在排队传输之前被复制，取决于谁被引用 */
  PBUF_REF,
  /**  pbuf有效载荷指的是RAM。这个来内存池，应该使用RX。有效载荷可以是链式的(分散-聚集RX)，但像PBUF_RAM，structpbuf及其有效负载分配在一块连续内存中(so)第一个有效负载字节可以从struct pbuf计算)。不要使用这个TX，如果池变成空的，例如TCP排队，您无法接收TCP acks */
```

```
  PBUF_POOL
} pbuf_type;
```

分配

```
//pbuf_alloc(pbuf_layer layer, u16_t length, pbuf_type type)
//根据不同层级和数据包大小计算出偏移量，调用 内存池或堆内存 的内存块，然后将pbuf内存出开辟出固定大小的内存,并返回pbuf

/* make the payload pointer point 'offset' bytes into pbuf data memory */
p->payload = LWIP_MEM_ALIGN((void *)((u8_t *)p + (SIZEOF_STRUCT_PBUF + offset)));
```

数据包处理

```
/**文件 drv_eth.c  eth_dev_rx(rt_device_t dev)
 * 该方法用于接收网卡的数据包，申请内存块，和 payload 赋值过程
 */

new_p = pbuf_alloc(PBUF_RAW, PBUF_POOL_BUFSIZE, PBUF_POOL);
...
gmac_rx_buf_attach(net_dev->gmac_id, (uint8 *)new_p->payload);
```

释放

```
//pbuf_alloc 在申请内存的时候将memp 类型强转成pbuf
p = (struct pbuf *)memp_malloc(MEMP_PBUF_POOL);

//释放的时候 又强转了回去

//memp.c 方法 do_memp_free_pool
/* cast through void* to get rid of alignment warnings */
memp = (struct memp *)(void *)((u8_t*)mem - MEMP_SIZE);

//mem.c 方法 rt_free
/* Get the corresponding struct heap_mem ... */
mem = (struct heap_mem *)((rt_uint8_t *)rmem - SIZEOF_STRUCT_MEM);
```