第4章

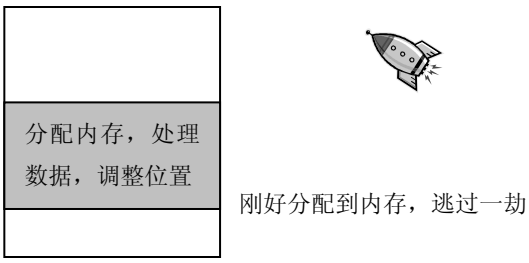
RT-Thread 内存管理器

在计算系统中，一般需要一定的存储空间来存放变量或中间数据，按照存储方式来分类可以分为两种，内部存储空间和外部存储空间。内部存储空间通常访问速度比较快，能够随机访问（按照变量地址）。这一章主要讨论内部存储空间的管理。

实时系统中由于它对时间要求的严格性，其中的内存分配往往要比通用操作系统苛刻得多：首先，分配内存的时间必须是确定性的。一般内存管理算法是搜索一个适当范围的空间去寻找适合长度的空闲内存块，然而这对于实时系统是不可接受的，因为实时系统必须要保证内存块的分配必须在确定的时间内完成，否则实时任务在对外部事响应也将变得时间不可确定，例如一个处理外部数据的例子：
当一个外部数据达到时（通过传感器，或者一些网络数据包），为了把它提交给上层的任务进行处理，它可能会先申请一块内存，把数据块的地址附加上，还可能有，数据长度，以及一些其他信息附加在一起（放在一个结构体中），然后提交给上层任务。

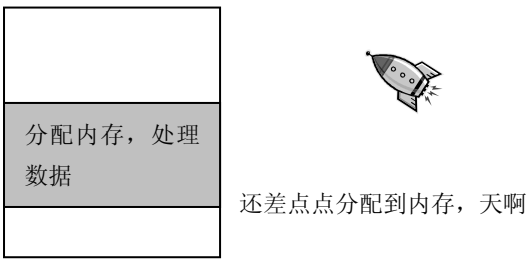
内存申请是当中的一个组成环节，如果因为使用的内存占用比较零乱，从而操作系统需要搜索一个不确定性长度的队列寻找适合的内存，那么申请时间将变得不可确定，进而对整个响应时间产生不可确定性。如果此时是一次导弹袭击，估计很可能会灰飞烟灭了！

第一次：



刚好分配到内存，逃过一劫

第 N 次：



还差点点分配到内存，天啊!!

其次，随着使用的内存分块被释放，整个内存区域会产生越来越多的碎片，从总体上来说，

系统中还有足够的空闲内存，但因为它们不在一起，不能组成一块连续的内存块，从而造成程序不能申请到内存。对于通用系统而言，这种不恰当的内存分配算法可以通过重新启动系统来解决(每个月或者数个月进行一次)，但是这个对于嵌入式系统来说是不可接受的，他们通常需要连续不断地运行数年。

最后，嵌入式系统的资源环境也不是都相同，有些系统中资源比较紧张，只有数百 KB 的内存可供分配，而有些系统则存在数 MB 的内存。

RT-Thread 操作系统在内存管理上，根据上层应用及系统资源的不同，有针对性的了数种内存分配算法：静态分区内存管理及动态内存管理。动态内存管理又划分为两种情况，一种是针对小内存块的分配管理，一种是针对大内存块的分配管理。

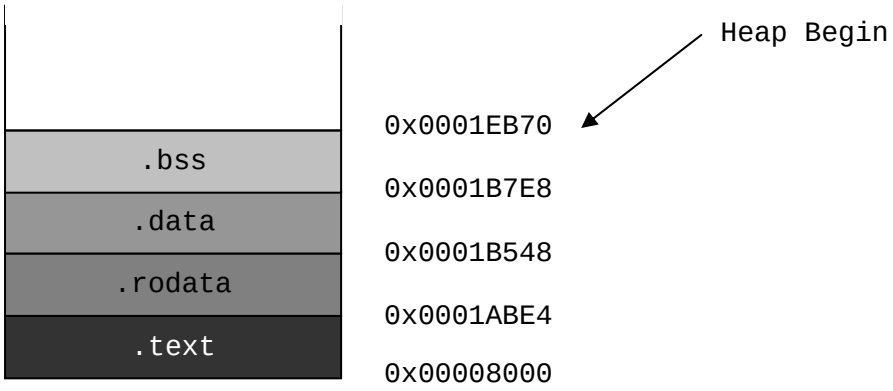
4.1 RT-Thread 的内核对象管理

在讨论 RT-Thread 内存管理器前，有必要详细描述下 RT-Thread 的内核对象系统。RT-Thread 的内核映像文件在编译时会形成如下图所示的结构（以 lumit4510 为例）：
其中主要包括了这么几段：

表 4-1　运行目标段

段	描　　述
.text	代码正文段
.data	数据段，用于放置带初始值的全局变量
.rodata	只读数据段，用于放置只读的全局变量（常量）
.bss	bss 段，用于放置未初始化的全局变量

图 4-1　lumit4510 运行时内存映像图



当系统运行时，这些段也会相应的映射到内存中。在 RT-Thread 系统初始化时，通常 bss 段会清零，而堆（Heap）则是除了以上这些段以外可用的内存空间（具体的地址空间在系统启动时由参数指定），系统运行时动态分配的内存块就在堆的空间中分配出来的，如程序 4-1：

程序 4-1

```
rt_uint8_t* msg;
msg = (rt_uint8_t*) rt_malloc (128);
rt_memset(msg, 0, 128);
```

`msg_ptr` 指向的内存空间就是处于堆空间中的。

而一些全局变量则是存放于.data 和.bss 段中，.data 存放的是具有初始值的全局变量（.rodata 可以看成一个特殊的 data 段，是只读的），如程序 4-2：

程序 4-2

```
#include <rtthread.h>

const static rt_uint32_t sensor_enable = 0x000000FE;
rt_uint32_t sensor_value;
rt_bool_t sensor_inited = RT_FALSE;

void sensor_init()
{
…
}
…
```

`sensor_value` 存放在`.bss` 段中，系统启动后会自动初始化成零。`sensor_inited` 变量则存放在`.data` 段中，而 `sensor_enable` 存放在`.rodata` 端中。

在 RT-Thread 内核对象中分为两类：静态内核对象和动态内核对象。
静态内核对象通常放在.data 或.bss 段中，在系统启动后在程序中初始化；动态内核对象则是从堆中创建的，而后手工做初始化。

内核对象 API 命名规则
`rt_xxx_init`
初始化系统静态内核对象，它只会对对象所在的内存块进行初始化，并添加到系统容器的链表中进行管理。

`rt_xxx_detach`
脱离内核对象容器管理，它会将对象从系统容器的链表中脱离出来，但不会去释放内存块。

`rt_xxx_create`
创建一个动态内核对象，它会先申请出一块空间，然后进行初始化并添加到系统容器的链表中进行管理。

`rt_xxx_delete`
它会将对象从系统容器的链表中删除出来，并是否所占用的内存块。

RT-Thread 中操作系统级的设施都是一种内核对象，例如线程，信号量，互斥量，定时器等。如程序 4-3 中所示的静态线程和动态线程的例子：

程序 4-3 静态内核对象与动态内核对象

```
static rt_uint8_t thread1_stack[512];
static struct rt_thread thread1;

/* 线程 1 入口 */
void thread1_entry(void* parameter)
{
    int i;

    while (1)
    {
        for (i = 0; i < 10; i ++)
        {
            rt_kprintf("%d\n", i);
            rt_thread_delay(100);
        }
    }
}
/* 线程 2 入口 */
void thread2_entry(void* parameter)
{
    int count = 0;
    while (1)
    {
        rt_kprintf("Thread2 count:%d\n", ++count);
        rt_thread_delay(50);
    }
}

/* 用户应用程序入口 */
int rt_application_init()
{
    rt_thread_t thread2_ptr;

    rt_thread_init(&thread1,
        "thread1",
        thread1_entry, RT_NULL,
        &thread1_stack[0], sizeof(thread1_stack),
        200, 10);

    thread2_ptr = rt_thread_create("thread2",
        thread2_entry, RT_NULL,
        512, 250, 25);

    rt_thread_startup(&thread1);
```

```
    rt_thread_startup(thread2_ptr);

    return 0;
}
```

例子中，thread1 是一个静态线程对象，而 thread2 是一个动态线程对象。thread1 对象的内存空间，包括 thread tcb：thread1，栈空间 thread1_stack 都是编译时刻决定的，因为都不存在初始值，都统一放在.bss 段中。thread2 运行中用到的空间都是动态分配的，包括 thread tcb（thread2_ptr 指向的内容）和栈空间。

4.1.1 内核对象管理工作模式

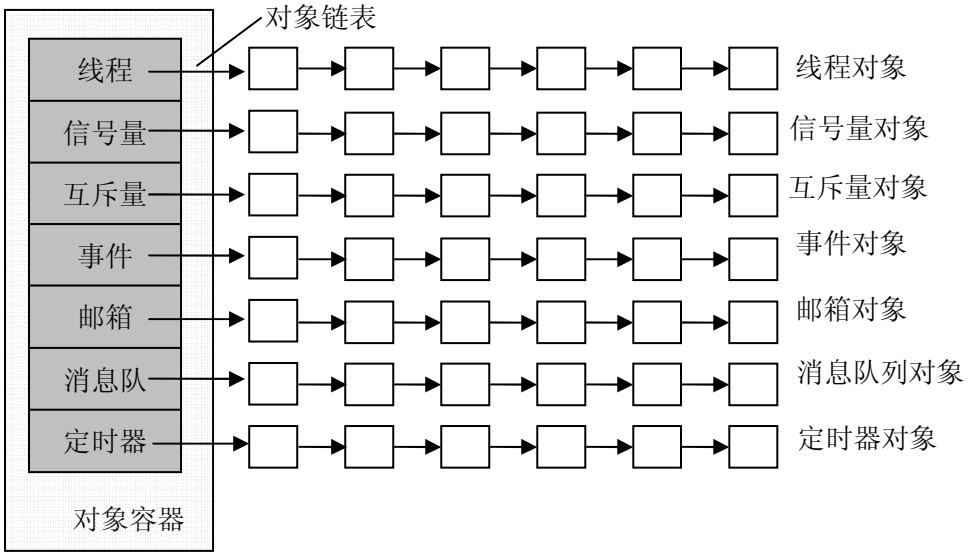


图 4-1 内核对象管理器结构图

系统采用内核对象管理系统来访问/管理所有内核对象。内核对象包含了内核中绝大部分设施，而这些内核对象可以是静态分配的静态对象，也可以是从系统内存堆中分配的动态对象。通过内核对象系统，RT-Thread 可以做到不依赖于具体的内存分配方式，伸缩性得到极大的扩展。

RT-Thread RTOS 使用 C 语言和汇编语言开发，本来不具备面向对象的特征。但这里，RT-Thread 借用了面向对象语言中对象的概念，定义了对象管理模块。

系统中线程同步和通信机制对象都是从 object 中继承而来，具备 object 的属性，并由系统的对象容器统一分配并管理。

内核对象管理器结构如图 4-1 所示，RT-Thread 内核对象包括：线程，信号量，互斥锁，事件，邮箱，消息队列和定时器，内存池。对象容器中包含了每类内核对象的信息，包括对象类型，大小等。对象容器给每类内核对象分配了一个链表，所有的内核对象都被链接到该链表上。

对于每一种具体内核对象和对象控制块，除了基本结构外，还有自己的扩展属性（私有属性），例如，对于线程控制块，在此基类对象基础上进行扩展，增加了线程状态、优先级

等属性。这些属性在基类对象的基本操作中不会用到，只有在与具体线程相关的函数才会使用。因此用面向对象的观点，我们可以认为每一种具体对象是抽象对象的派生，继承了基本对象的属性并在此基础上增加与自己相关的属性。图 4-2 显示了 RT-Thread 中各类内核对象的派生和继承关系。

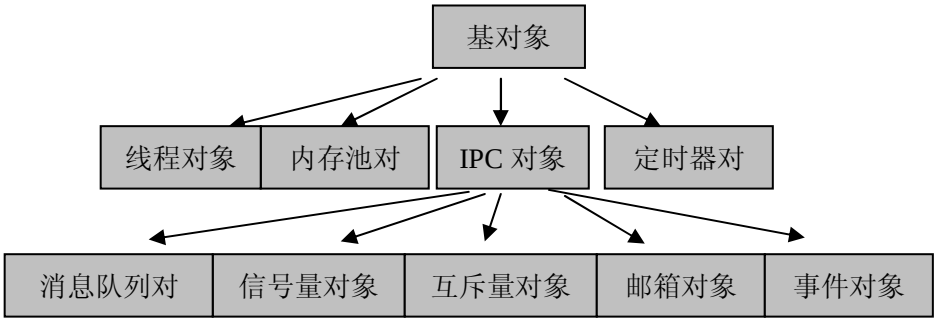


图 4-2 内核对象的继承和派生关系图

在对象管理模块中，定义了通用的数据结构，用来保存各种对象的共同属性，各种具体对象只需要在此基础上加上自己的某些特别的属性，就可以清楚的表示自己的特征。这种设计方法的优点：

1) 提高了系统的可重用性和扩展性，增加新的对象类别很容易，只需要继承通用对象的属性再加少量扩展即可。
2) 提供统一的对象操作方式，简化了各种具体对象的操作，提高了系统的可靠性。

4.1.2 对象控制块

```
struct rt_object
{
    /* name of kernel object         */
    char     name[RT_NAME_MAX];
    /* type of kernel object         */
    rt_uint8_t  type;
    /* flag of kernel object1        */
    rt_uint8_t  flag;
    /* list pointer of kernel object */
    rt_list_t list;
};
```

4.1.3 内核对象接口

4.1.3.1初始化系统对象

在初始化各种内核对象之前，首先需对对象管理系统进行初始化。在系统中，每类内核对象都有一个静态对象容器，一个静态对象容器放置一类内核对象，初始化对象管理系统的任务就是初始化这些对象容器，使之能够容纳各种内核对象，初始化系统对象使用以下接口：
void rt_system_object_init(void)
以下是对象容器的数据结构：

```
struct rt_object_information
{
    enum rt_object_class_type type;  /* object class type. */
    rt_list_t object_list;           /* object list. */
    rt_size_t object_size;           /* object size. */
};
```

一种类型的对象容器维护了一个对象链表 object_list，所有对于内核对象的分配，释放操作均在该链表上进行。

4.1.3.2初始化对象

使用对象前须先对其进行初始化。初始化对象使用以下接口：

```
void rt_object_init(struct rt_object* object, enum rt_object_class_type type, const char* name)
```

对象初始化，实际上就是把对象放入到其相应的对象容器中，即将对象插入到对象容器链表中。

4.1.3.3脱离对象

从内核对象管理器中脱离一个对象。脱离对象使用以下接口：

```
void rt_object_detach(rt_object_t object)
```

使用该接口后，静态内核对象将从内核对象管理器中脱离，但是对象占用的内存不会被释放。

4.1.3.4分配对象

在当前内核中，共定义了十类内核对象，这些内核对象被广泛的用于线程的管理，线程之间的同步，通信等。因此，系统随时需要新的对象来完成这些操作，分配新对象使用以下接口：

```
rt_object_t rt_object_allocate(enum rt_object_class_type type, const char* name)
```

使用以上接口，首先根据对象类型来获取对象信息，然后从内存堆中分配对象所需内存空间，然后对该对象进行必要的初始化，最后将其插入到它所在的对象容器链表中。

4.1.3.5删除对象

不再使用的对象应该立即被删除，以释放有限的系统资源。删除对象使用以下接口：

```
void rt_object_delete(rt_object_t object)
```

使用以上接口时，首先从对象容器中脱离对象，然后释放对象所占用的内存。

4.1.3.6查找对象

通过指定的对象类型和对象名查找对象，查找对象使用以下接口：

```
rt_object_t rt_object_find(enum rt_object_class_type type, const char* name)
```

使用以上接口时，在对象类型所对应的对象容器中遍历寻找指定对象，然后返回该对象，如果没有找到这样的对象，则返回空。

4.1.3.7辨别对象

判断指定对象是否是系统对象（静态内核对象）。辨别对象使用以下接口：

```
rt_err_t rt_object_is_systemobject(rt_object_t object)
```

通常采用 rt_object_init 方式挂接到内核对象管理器中的对象是系统对象。

4.2 静态分区内存管理

4.2.1 内存池工作模式

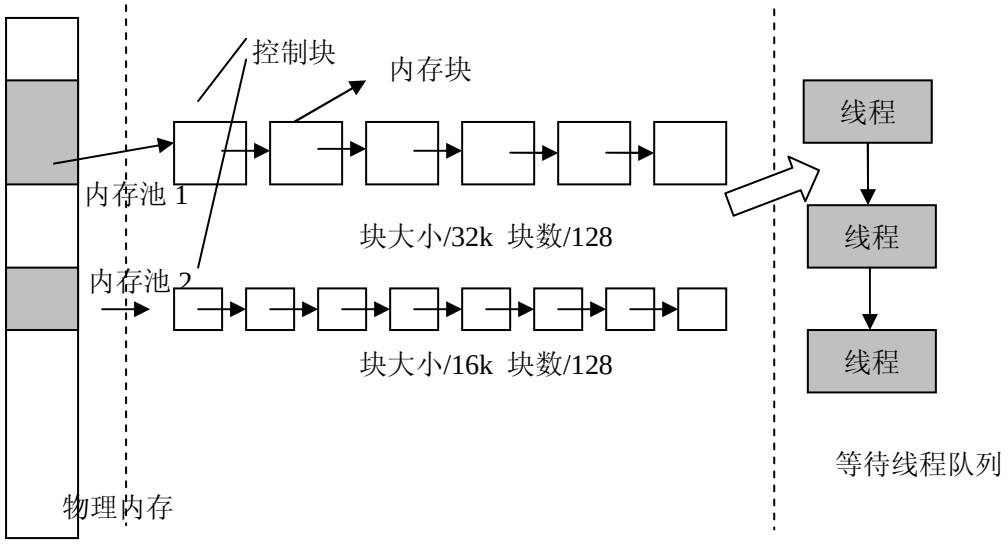


图 4-19 内存池管理结构示意图

内存池（Memory Pool）是一种用于分配大量大小相同的小对象的技术。它可以极大加快内存分配/释放过程。

内存池在创建时向系统申请一大块内存，然后分成同样大小的多个小内存块，形成链表连接起来（此链表也称为空闲链表）。每次分配的时候，从空闲链表中取出头上一块，提供给申请者。如图 4-19 所示，物理内存中可以有多个大小不同的内存池，一个内存池由多个空闲内存块组成，内核用它们来进行内存管理。当一个内存池对象被创建时，内存池对象就被分配了内存池控制块：内存池名，内存缓冲区，内存块大小，块数以及一个等待线程队列。

内核负责给内存池分配内存池对象控制块，它同时也接收用户线程传入的参数，像内存块大小以及块数，由这些来确定内存池对象所需内存大小，当获得了这些信息后，内核就可以从内存堆或者线程私有内存空间中为内存池分配内存。

4.2.2 内存池控制块

```
struct rt_mempool
{
    struct rt_object parent;

    void* start_address;          /* memory pool start */
    rt_size_t size;               /* size of memory pool */
```

```
    rt_size_t block_size;         /* size of memory blocks */
    rt_uint8_t* block_list;       /* memory blocks list */

    rt_size_t block_total_count;  /* numbers of memory block */
    rt_size_t block_free_count;   /* numbers of free memory block */

    rt_list_t suspend_thread;     /* threads pended on this pool */
    rt_size_t suspend_thread_count;  /* numbers of thread pended on
this pool */
};
```

4.2.3 内存池接口

4.2.3.1创建内存池

创建内存池操作将会创建一个内存池对象并且从堆上分配一个内存池。创建内存池是分配，释放内存块的基础，创建该内存池后，线程便可以从内存池中完成申请，释放操作，创建内存池使用如下接口，接口返回一个已创建的内存池对象。

```
rt_mp_t rt_mp_create(const char* name, rt_size_t block_count,
rt_size_t block_size);
```

使用该接口可以创建与需求相匹配的内存块大小和数目的内存池，前提是在系统资源允许的情况下。创建内存池时，需要给内存池指定一个名称。根据需要，内核从系统中申请一个内存池对象，然后从内存堆中分配一块由块数目和块大小计算得来的内存大小，接着初始化内存池对象结构，并将申请成功的内存缓冲区组织成可用于分配的空闲块链表。

4.2.3.2删除内存池

删除内存池将删除内存池对象并释放申请的内存。使用如下接口：

```
rt_err_t rt_mp_delete(rt_mp_t mp)
```

删除内存池时，必须首先唤醒等待在该内存池对象上的所有线程，然后再释放已从内存堆上分配的内存，然后删除内存池对象。

4.2.3.3初始化内存池

初始化内存池跟创建内存池类似，只是初始化邮箱用于静态内存管理模式，内存池控制块来源于用户线程在系统中申请的静态对象。还与创建内存池不同的是，此处内存池对象所使用的内存空间是由用户线程指定的一个缓冲区空间，用户把缓冲区的指针传递给内存池对象控制块，其余的初始化工作与创建内存池相同。接口如下：

```
rt_err_t rt_mp_init(struct rt_mempool* mp, const char* name, void
```

```
*start, rt_size_t size, rt_size_t block_size)
```

初始化内存池时，把需要进行初始化的内存池对象传递给内核，同时需要传递的还有内存池用到的内存空间，以及内存池管理的内存块数目和块大小，并且给内存池指定一个名称。这样，内核就可以对该内存池进行初始化，将内存池用到的内存空间组织成可用于分配的空闲块链表。

4.2.3.4脱离内存池

脱离内存池将使内存池对象被从内核对象管理器中删除。脱离内存池使用以下接口。

```
rt_err_t rt_mp_detach(struct rt_mempool* mp)
```

使用该接口后，内核先唤醒所有挂在该内存池对象上的线程，然后将该内存池对象从内核对象管理器中删除。

4.2.3.5分配内存块

从指定的内存池中分配一个内存块，使用如下接口：

```
void *rt_mp_alloc (rt_mp_t mp, rt_int32 time)
```

如果内存池中有可用的内存块，则从内存池的空闲块链表上取下一个内存块，减少空闲块数目并返回这个内存块，如果内存池中已经没有空闲内存块，则判断超时时间设置，若超时时间设置为零，则立刻返回空内存块，若等待大于零，则把当前线程挂起在该内存池对象上，直到内存池中有可用的自由内存块，或等待时间到达。

4.2.3.6释放内存块

任何内存块使用完后都必须被释放，否则会造成内存泄露，释放内存块使用如下接口：

```
void rt_mp_free  (void *block)
```

使用以上接口时，首先通过需要被释放的内存块指针计算出该内存块所在的内存池对象，然后增加内存池对象的可用内存块数目，并把该被释放的内存块加入空闲内存块链表上。接着判断该内存池对象上是否有挂起的线程，如果有，则唤醒挂起线程链表上的首线程。

4.3 动态内存管理

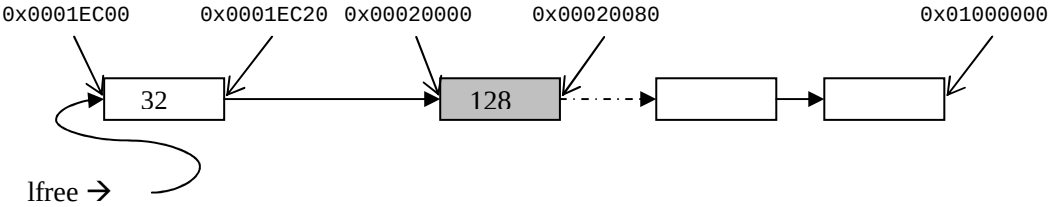
动态内存管理是一个真实的堆内存管理模块，可以根据用户的需求（在当前资源满足的情况下）分配任意大小的内存块。RT-Thread 系统中为了满足不同的需求，提供了两套动态内存管理算法，分别是小堆内存管理和 SLAB 内存管理。下堆内存管理模块主要针对系统资源比较少，一般小于 2M 内存空间的系统；而 SLAB 内存管理模块则主要是在系统资源比较丰富时，提供了一种近似的内存池管理算法。两种内存管理模块在系统运行时只能选择其中之

一（或者完全不使用动态堆内存管理器），两种动态内存管理模块 API 形式完全相同。

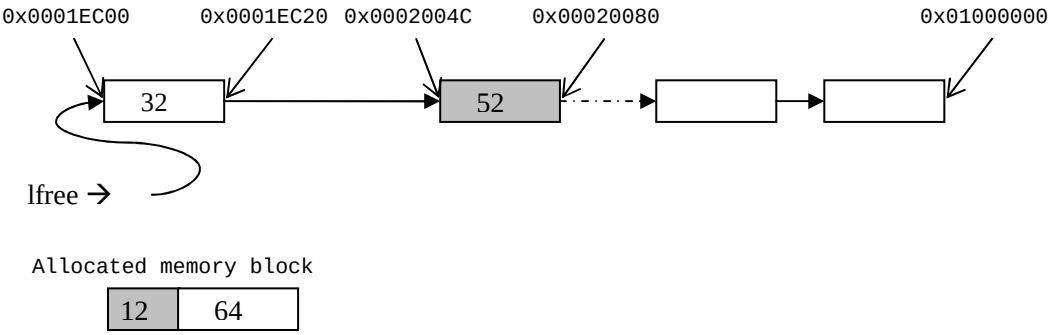
注：不要在中断服务例程中分配或释放动态内存块。

4.3.1 小内存管理模块

小内存管理算法是一个简单的内存分配算法，当有可用内存的时候，会从中分割出一块来作为分配的内存，而余下的则返回到动态内存堆中。如图 4-5 所示



当用户线程要分配一个 64 字节的内存块时，空闲链表指针 lfree 初始指向 0x0001EC00 内存块，但此内存块只有 32 字节并不能满足要求，它会继续寻找下一内存块，此内存块大小为 128 字节满足分配的要求。分配器将把此内存块进行拆分，余下的内存块（52 字节）继续留在 lfree 链表中。如图 4-8 所示



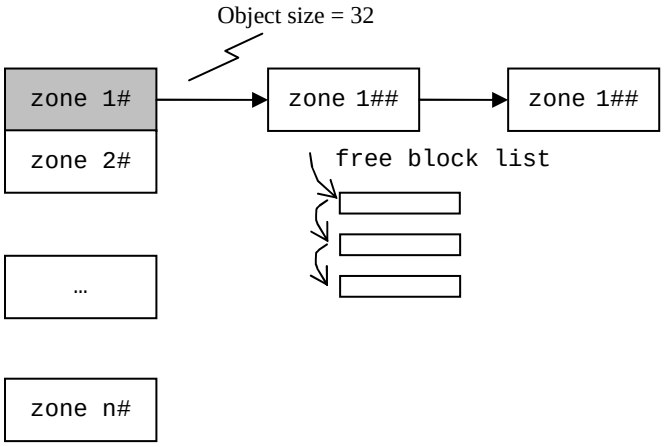
在分配的内存块前约 12 字节会存放内存分配器管理用的私有数据，用户线程不应访问修改它，这个头的大小会根据配置的对齐字节数稍微有些差别。

释放时则是相反的过程，但分配器会查看前后相邻的内存块是否空闲，如果空闲则合并成一个大的空闲内存块。

4.3.2 SLAB 内存管理模块

RT-Thread 实现的 SLAB 分配器是在 Matthew Dillon 在 DragonFly BSD 中实现的 SLAB 分配器基础上针对嵌入式系统优化过的内存分配算法。原始的 SLAB 算法是 Jeff Bonwick 为 Solaris 操作系统首次引入的一种高效内核内存分配算法。

RT-Thread 的 SLAB 分配器实现主要是去掉了其中的对象构造及析构过程，只保留了纯粹的缓冲型的内存池算法。SLAB 分配器会根据对象的类型（主要是大小）分成多个区（zone），也可以看成每类对象有一个内存池，如图所示：



一个 zone 的大小在 32k ～ 128k 字节之间，分配器会在堆初始化时根据堆的大小自动调整。系统中最多包括 72 种对象的 zone，最大能够分配 16k 的内存空间，如果超出了 16k 那么直接从页分配器中分配。每个 zone 上分配的内存块大小是固定的，能够分配相同大小内存块的 zone 会链接在一个链表中，而 72 种对象的 zone 链表则放在一个数组（zone_arry）中统一管理。

动态内存分配器主要的两种操作：
内存分配：假设分配一个 32 字节的内存，SLAB 内存分配器会先按照 32 字节的值，从 zone_array 链表表头数组中找到相应的 zone 链表。如果这个链表是空的，则向页分配器分配一个新的 zone，然后从 zone 中返回第一个空闲内存块。如果链表非空，则这个 zone 链表中的第一个 zone 节点必然有空闲块存在（否则它就不应该放在这个链表中），然后取相应的空闲块。如果分配完成后，导致一个 zone 中所有空闲内存块都使用完毕，那么分配器需要把这个 zone 节点从链表中删除。

内存释放：分配器需要找到内存块所在的 zone 节点，然后把内存块链接到 zone 的空闲内存块链表中。如果此时 zone 的空闲链表指示出 zone 的所有内存块都已经释放，即 zone 是完全空闲的 zone。当中 zone 链表中，全空闲 zone 达到一定数目后，会把这个全空闲的 zone 释放到页面分配器中去。

4.3.3 动态内存接口

4.3.3.1初始化系统堆空间

在使用堆内存时，必须要在系统初始化的时候进行堆内存的初始化，可以通过如下接口完成：

```
void rt_system_heap_init(void* begin_addr, void* end_addr);
```

入口参数分别为堆内存的其实地址和结束地址。

4.3.3.2分配内存块

从内存堆上分配用户线程指定大小的内存块，接口如下：

```
void* rt_malloc(rt_size_t nbytes);
```

用户线程需指定申请的内存空间大小，成功时返回分配的内存块地址，失败时返回RT_NULL。

4.3.3.3重分配内存块

在已分配内存块的基础上重新分配内存块的大小（增加或缩小），可以通过如下接口完成：

```
void *rt_realloc(void *rmem, rt_size_t newsize);
```

在进行重新分配内存块时，原来的内存块数据保持不变（缩小的情况下，后面的数据被自动截断）。

4.3.3.4分配多内存块

从内存堆中分配连续内存地址的多个内存块，可以通过如下接口完成：

```
void *rt_calloc(rt_size_t count, rt_size_t size);
```

返回的指针指向第一个内存块的地址，并且所有分配的内存块都被初始化成零。

4.3.3.5释放内存块

用户线程使用完从内存分配器中申请的内存后，必须及时释放，否则会造成内存泄漏，释放接口如下：

```
void rt_free (void *ptr);
```

用户线程需传递待释放的内存块指针，如果是空指针直接返回。

4.3.3.6设置分配钩子函数

在分配内存块过程中，用户可申请一个钩子函数，它会在内存分配完成后回调，接口如下：

```
void rt_malloc_sethook(void (*hook)(void *ptr, rt_size_t size));
```

回调时，会把分配到的内存块地址和大小做为入口参数传递进去。

4.3.3.7设置内存释放钩子函数

在释放内存时，用户可设置一个钩子函数，它会在调用内存释放完成前进行回调，接口如下：

```
void rt_free_sethook(void (*hook)(void *ptr));
```

回调时，释放的内存块地址会做为入口参数传递进去（此时内存块并没有被释放）。