# lwip---动态内存管理

## 功能：

本功能模块是协议栈层面提供的动态内存管理（区别于编程语言、编译器、OS等层面的动态内存管理）。

协议栈的动态内存管理的目的是：为协议栈处理的每一个数据分组及处理时的需要的各种变量或数据结构分配内存，并在数据分组处理完成后回收内存。

例如：

1）建立TCP连接时，需要分配一个tcp\_pcb，连接断掉后，则要回收该tcp\_pcb占用的内存。

2）发送或接收一个数据分组时，需要分配一个pbuf结构，完成后回收该pbuf结构占用的内存。

动态内存管理包含两个模块：

1. 堆内存管理。

用于模拟C语言的动态内存管理函数库（mem(),free()）的功能。堆内存中存储的数据的数据类型信息只是对用户而言的，堆内存管理器只是看做字节串。用户可以按照需要 从堆内存中 申请允许范围内 的任意大小的内存块，在其中存储任意类型的数据，使用完后释放。

1. 池内存管理。

用于管理由协议栈各层在处理数据分组时需要的各种数据结构构成的 各种类型的内存池，内存池在编译时就已经分配好了空间（即分配在了静态存储区）。目的是为了在需要某种数据结构时，可以快速的获得，而不用从堆内存中在运行时申请。

## 子模块---堆内存管理（DMM）

### 功能：

用于模拟C语言的动态内存管理函数库（mem(),free()）的功能。作为动态存储区，供应用进程和协议栈内核进程共同使用。

堆内存中存储的数据的数据类型信息只是对用户而言的，堆内存管理器只是看做字节串。用户可以按照需要从堆内存中申请允许范围内的任意大小的内存块，在其中存储任意类型的数据，使用完后释放。

### 接口：(mem.h)

#### 用户配置项：（opt.h）

##### 一级配置：堆内存管理器实现策略配置

###### MEM\_LIBC\_MALLOC（决定使用C库DMM还是lwip内部DMM）

用于控制堆内存管理器的实现是直接使用C运行时库提供的DMM，还是使用lwip内部提供的DMM。1表示使用C运行时库的DMM，0表示使用lwip内部提供的DMM。默认配置是0。

##### 二级配置：lwip内部DMM实现配置

###### MEM\_USE\_POOLS(实现策略配置)

用于控制堆内存是使用基于固定内存块的池内存策略来实现，还是使用基于可变内存块的堆内存策略来实现。1表示使用前者，0表示使用后者。

opt.h中默认是0。

###### MEM\_SIZE(堆内存大小)

表示堆内存管理所管理的堆内存的字节大小。

如果堆内存管理器采用的是基于固定内存块的池内存实现策略，则堆内存由lwippools.h中自定义的各个内存池的构成，MEM\_SIZE必须等于所有内存池大小的总和；

如果采用的是基于可变内存开的堆内存实现策略，则MEM\_SIZE等于堆内存的大小。

opt.h中默认值等于1600个字节。

##### 基于可变内存块的堆内存策略内部配置

###### 堆内存空间分配

lwip内部的DMM在使用堆内存策略时，所管理的堆内存可以是用户自定义的一块内存，也可以是内核内部自行分配的一块内存。如果是用户自定义的一块内存，则要做以下两步：*（在哪个文件中？在cc.h还是直接在mem.c中？俩也许都可以）*

1、在内存中以全局变量的形式分配堆内存；

2、定义宏常量LWIP\_RAM\_HEAP\_POINTER，等于堆内存的实际起始地址

mem.c中的源码注释：

/\*\* If you want to relocate the heap to external memory, simply define

\* LWIP\_RAM\_HEAP\_POINTER as a void-pointer to that location.

\* If so, make sure the memory at that location is big enough (see below on

\* how that space is calculated). \*/

##### 基于固定内存块的池内存策略内部配置

#### 数据类型（mem.h）

##### mem\_size\_t

用于作为表示堆内存内的数据对象大小的数据类型。

功能与C标准库<stddef.h>中的size\_t类似。它实际是一个无符号整型数，但是具体是哪种大小的无符号数据类型，则与堆内存的实现策略以及堆内存的**实际分配的大小**（不是MEM\_SIZE，而是MEM\_SIZE+对齐用内存大小）有关。

#### 调试功能（宏）（mem.h）

##### MEM\_SIZE\_F（格式化输出说明符）

堆内存中mem\_size\_t类型数据格式化输出时的格式说明符。

#### 操作接口：(mem.h和mem.c)

##### ？？数据对齐操作：

##### mem\_init(初始化堆内存管理模块)

|  |  |
| --- | --- |
| 功能 | 供lwip\_init函数调用，在协议栈内核初始化时，初始化堆内存管理模块； |
| 函数原型 | mem\_init() |
| 输入 | 无； |
| 输出 | 无； |

##### mem\_malloc（分配一个内存块）

|  |  |
| --- | --- |
| 功能 | 从堆内存中分配一个指定大小的内存块。 |
| 函数原型 | void \*mem\_malloc(mem\_size\_t size) |
| 输入 | size: 申请的内存块的大小，mem\_size\_t类型； |
| 输出 | 分配的内存块的首地址或NULL； |

##### mem\_calloc（分配一个内存块，并清零）

|  |  |
| --- | --- |
| 功能 | 从堆内存中分配一个指定大小的内存块，并清零； |
| 函数原型 | void \*mem\_calloc(mem\_size\_t count, mem\_size\_t size) |
| 输入 | count：申请的内存块中的元素的个数；  size ：单个元素的字节大小； |
| 输出 | 分配的内存块的首地址或NULL； |

##### mem\_free（释放一个内存块）

|  |  |
| --- | --- |
| 功能 | 将已用内存块释放回堆内存中； |
| 函数原型 | void mem\_free(void \*mem) |
| 输入 | mem：已用内存块的首地址； |
| 输出 | 无； |

##### mem\_trim（调小一个内存块）

|  |  |
| --- | --- |
| 功能 | 缩小一个内存块的大小，多余的部分放回堆内存中； |
| 函数原型 | void \* mem\_trim(void \*mem, mem\_size\_t newsize) |
| 输入 | mem：内存块的首地址；  newsize：调小后的字节大小（必须小于等于原大小）； |
| 输出 | mem/NULL： 内存块的首地址或NULL；（for compatibility reasons: is always == mem, at the moment or NULL if newsize is > old size, in which case mem is NOT touched or freed!） |

### 实现：

#### 原理：

##### 两种策略及其配置

堆内存管理的实现有两种策略：

①直接用C运行时库中的malloc()等函数构成的DMM；

②使用LWIP核内部自带的DMM。

这两种当中只能使用其中的一种，通过配置opt.h中配置项**MEM\_LIBC\_MALLOC**决定使用其中的哪一种。MEM\_LIBC\_MALLOC=1使用C库提供的DMM，否则使用lwip内部提供的DMM。当使用其中某一种时，另一种相关的代码就不会编译，从而实现了内核的可裁剪。

#### 与策略无关的公共部分

##### ？？？对外依赖

##### ???接口---操作接口：数据对齐操作实现

#### 实现策略1：C库DMM（MEM\_LIBC\_MALLOC=1）

##### 原理：

堆内存管理器直接使用C库的malloc，free等函数构成的DMM。

##### 依赖：

###### 工具链：C函数库<stddef.h>

size\_t数据类型

需求：

用于当堆内存使用C库的DMM实现策略时，定义堆内存模块的数据类型mem\_size\_t。构建开发工具链时，C函数库需要根据具体的平台环境定义<steddef.h>库文件中的size\_t类型。

###### 平台环境移植文件：arch.h或cc.h

基本数据类型

需求：

移植时，在cc.h中自定义基本数据类型。详细内容参考对移植文件cc.h的说明。

调试功能的格式化输出说明符

SZT\_F（size\_t类型的格式化输出说明符，堆内存模块仅提出需求的名字）

需求：

用于当堆内存使用C库的DMM实现策略时，定义堆内存的mem\_size\_t类型数据的格式化输出说明符“MEM\_SIZE\_F”。

因为size\_t类型与平台环境有关，对应的在lwip中使用的格式输出说明符也就依赖于外部，但是堆内存模块的调试代码又要使用，所以由本模块提出规定名字为“SZT\_F”并使用，具体实现则在cc.h中定义或者使用arch.h中的默认定义。

##### 接口实现：(mem.h)

###### 数据类型：

mem\_size\_t

**原理：**

因为堆内存管理使用C库提供的DMM来实现，所以mem\_size\_t类型就与C库的DMM实现有关，C库中提供的就是size\_t。所以mem\_size\_t tepedef 为size\_t。

**C代码：**

#include <stddef.h> /\* for size\_t \*/

typedef size\_t mem\_size\_t;

###### 调试功能宏：

MEM\_SIZE\_F（格式化输出说明符）

设计详解：

因为size\_t类型与平台环境有关，对应的在lwip中使用的格式输出说明符也就依赖于外部，但是堆内存模块的调试代码又要使用，所以由本模块提出规定名字为“SZT\_F”并使用，具体实现则在cc.h中定义或者使用arch.h中的默认定义。

C代码：

#define MEM\_SIZE\_F SZT\_F

###### 操作接口：

**实现方法：**

在使用C库的DMM时，因为堆内存的操作接口函数实际上执行的是C库中的相应动态内存管理函数。定义每个函数，里面只调用相应的C库函数的方法，由于函数调用效率太低，且内核编译后的代码加大，所以lwip采用的方法是：

**直接使用宏替换成C库中的相应函数 。**

特别是：

1）mem\_init：在使用C库DMM时，lwip不用实现，所以将该函数宏替换成空。lwip\_init的代码中就不包含调用mem\_init的代码。

2）mem\_malloc,mem\_calloc,mem\_free,mem\_trim：

如果移植时，某一个移植文件中或构建的C库中在C库中的动态内存函数基础上封装了mem\_free(),mem\_malloc(),mem\_calloc(),mem\_trim()函数，则mem.h中的定义的这些函数的宏函数可以overriden。【即在定义这些函数的宏时，使用#ifndef..#endif条件预编译指令】

3）mem\_trim：使用C库DMM时，因为C库中并没有对应的shrink memory without moving it的函数，另外mem\_trim又要求有返回值且仍是输入参数中的内存块的首地址，所以本宏函数实现直接宏替换成输入参数中的内存块首地址。

**代码：**

**参考src\include\lwip\mem.h。**

#### 实现策略2：lwip内部DMM（MEM\_LIBC\_MALLOC=0）

##### 原理：

如果使用lwip内部的DMM，则需要在内存中分配MEM\_SIZE大小的内存（连续或离散）用作堆内存，供堆内存管理器管理。**但因为**整个堆内存在分配时会进行对齐处理。**所以**实际分配的内存大小=MEM\_SIZE+用于对齐的内存大小。

管理堆内存时，Lwip内部的DMM又有两种实现策略。通过两个用户配置项决定使用哪种策略：

MEM\_USE\_POOLS：用于通知堆内存管理器lwip内部DMM使用固定内存块的池内存策略，编译相应的代码；

MEMP\_USE\_CUSTOM\_POOLS：是池内存管理器的配置项，用于通知该管理器编译用于堆内存的用户自定义内存池相关的代码（前提是用户也准备了lwippools.h头文件）；

只有当上面两个配置项都为1时，才能基于固定内存块的池内存策略正常使用lwip内部的DMM。

##### 实现策略1：（默认）基于可变内存块的堆内存策略

###### 原理：

*<参考《数据结构(c语言版)》严蔚敏 吴伟民，第195页可利用空间表的三种结构中的第三种以及8.3节“边界标识法”>*

堆内存空间分配：

堆内存可以是用户自定义的一块内存，也可以是内核内部自行分配的一块内存。由宏常量LWIP\_RAM\_HEAP\_POINTER表示（未对齐的）实际起始地址。为了实现这一功能，在管理器内部访问堆内存中的数据时，都是使用相对于堆内存（对齐后的）起始地址的偏移量来定位。这样只要改变LWIP\_RAM\_HEAP\_POINTER的值就可以改变堆内存的位置，而不用修改代码。

堆内存的首地址：

实际分配空间首地址（未对齐）：LWIP\_RAM\_HEAP\_POINTER；

实际使用首地址：存在全局变量ram中，等于初始化时将上面的地址作对齐处理的结果（结果比上面的地址大，相当于在实际分配的空间中开头的（ram- LWIP\_RAM\_HEAP\_POINTER）个字节不会被用）。

堆内存的大小：

用户配置的大小：MEM\_SIZE；

实际分配空间大小：（对齐处理后的大小，比配置的MEM\_SIZE大）

=MEM\_SIZE\_ALIGNED + (2\*SIZEOF\_STRUCT\_MEM) + MEM\_ALIGNMENT。

（详解：**MEM\_SIZE\_ALIGNED**是MEM\_SIZE对齐后的大小，用于用户的申请与释放；**两个SIZEOF\_STRUCT\_MEM**是两个mem结构体对齐后的大小，用于内部管理堆内存使用；**MEM\_ALIGNMENT**是1个对齐字，用途与堆内存首地址的对齐处理有关：堆内存实际使用的首地址ram是实际分配空间首地址的对齐处理的结果，且前者比后者大，这就导致实际分配空间的开头有几个（肯定小于MEM\_ALIGNMENT*【与对齐方法有关】*）字节不能用，如果不多分配MEM\_ALIGNMENT个字节用于由于地址对齐处理造成的空间减少，那么剩余的部分无法足够被用来作为堆内存和两个mem结构。）

堆内存管理算法：（类似于“边界标识法”）

堆内存管理器将堆内存中的所有内存块（已用内存块和空闲内存块）组织成一个双向循环链表结构的空间利用表。链表中的每个节点头部边界上设有标识，用于标识该节点为已用内存块还是空闲内存块，使得在回收用户释放的内存块时易于判别在物理位置上与其相邻的内存块是否为空闲块，以便将所有地址连续的空闲内存块组织成一个尽可能大的空闲块。

空间利用表的结构：

双向循环链表。传统链表中节点间的逻辑上前驱后继关系并不代表实际物理位置关系，与此不同的是，堆内存管理使用的双向链表中某个节点的前驱和后继结点，就是其物理位置上的前驱和后继结点。链表中的节点是已用节点和空闲节点混杂在一起的。

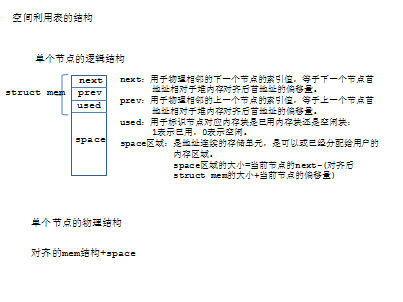
**从起始地址ram向高地址方向看：**

物理上第一个节点的首地址：由全局指针变量ram的值表示，是8位无符号的字节地址，使用时转换成mem结构体类型指针；

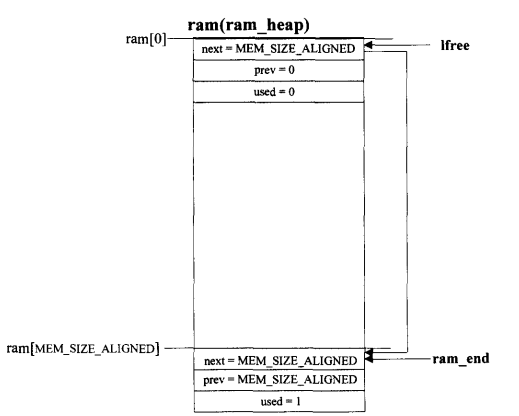
物理上最后一个节点的首地址：由全局指针变量ram\_end的值表示，是mem结构体类型指针；

物理上第一个可用的空闲节点的首地址：由全局指针变量lfree的值表示，是mem结构体类型指针，随内存块的申请释放而变化；

空间利用表单个节点的结构：



初始态时空间利用表的结构



分配算法：

分配算法可以采用first-fit也可以是best-fit。lwip使用的是first-fit。每次分配内存块时，从lfree指针所指示的空闲内存节点开始查找，找到第一个space区域容量不小于请求分配的size的空闲节点，然后进行分配，剩余部分作为一个新的空闲节点插入链表中。为了使整个堆内存管理器更有效的运行，实现时还做了如下两条约定：

1. 假设找到的待分配的空闲节点的容量（对齐的mem+space区域的大小）为m个字节，若每次分配只是从中分配n个字节大小作为已用节点（对齐的mem+申请的size），剩余的m-n个字节作为一个新的空闲节点仍留在链表中，则在若干次分配后，链表中会出现一些容量极小总也分配不出去的空闲节点，这就大大减慢了查找及分配的速度。

**因此：**选定一个常量e，当剩余字节大小（m-n）<=e时，就将容量为m的整个空闲节点分配给用户；反之，只分配其中的n个字节。lwip的堆内存管理器在实现时，规定剩余的字节大小e=STRUCTOF\_STRUCT\_MEM+MIN\_SIZE\_ALIGNED。

1. 分配时如果lfree所指向的空闲节点被分配，则从物理第一个节点开始查找，修改lfree使其指向新的第一个空闲节点。

回收算法：

一旦用户释放已用内存块，堆内存管理器就需要判断其物理位置相邻的两个节点的当前状态。如果至少有一个是空闲节点，则释放的内存块所在的节点就需要与其合并成一个更大的空闲节点；如果都不是空闲节点，则释放的内存块只是简单的插入链表中即可。

优点：内存浪费小，比较简单，适合小内存的管理；

缺点：频繁的内存分配和释放，可能会导致严重的内存碎片，如果严重的话，则可能导致内存分配失败。

###### 实现：

依赖：

平台环境移植文件（arch.h,cc.h）

基本数据类型、格式化输出说明符等；

操作系统模拟层文件（sys.h,sys\_arch.h）

1、操作系统层的临界代码区的实现机制（用于共享资源的并行访问控制）：

SYS\_ARCH\_DECL\_PROTECT

SYS\_ARCH\_PROTECT

SYS\_ARCH\_UNPROTECT等

错误码文件（err.h）

用户配置项

MEM\_SIZE（opt.h）

表示可用于分配的堆内存的（未对齐）大小。在opt.h中有对应的默认配置。如果用户自定义内核配置文件opts.h中没有定义，则使用默认配置。

LWIP\_RAM\_HEAP\_POINTER（在opt.h中无对应默认配置项，但可在opts.h中定义）

表示堆内存的实际未对齐的起始地址。因为堆内存管理器在使用本策略实现时，堆内存的位置是可以改变的，即要不使用用户分配的一块的内存，要不使用管理器内部自动分配的一块内存。如果使用前者，则需要用户将其分配的用于堆内存的实际首地址，通过宏定义的形式，告知堆内存管理器。

一般都是使用管理器内部自动分配的堆内存，并且通过MEM\_SIZE控制堆内存的大小。

MIN\_SIZE（在opt.h中无对应默认配置项，但可在opts.h中定义）

用于限制空间利用表中空闲内存块的space区域的最小大小。当将一个空闲节点分割分

配出去时，如果剩余的内存块大小不足以构成一个空间节点，则将整个空闲节点分配出去，而不分割。（源码和《哪些事》中都说是用于限制分配的内存块的最小大小，有这个作用，但也不全能限制）

MIN\_SIZE取较小值，则可以节省空间；取较大值，则可以预防too small blocks to fragment the RAM too much。

默认值是12。

？？？宏常量

LWIP\_RAM\_HEAP\_POINTER

**功能：**宏常量，表示堆内存的实际未对齐的起始地址。

**实现：**

条件预编译指令，如果该宏常量已定义，则表示堆内存由用户分配的；否则，需要管理器自动分配堆内存。并定义改宏常量为分配的堆内存的实际首地址。

**C代码：**

如果是使用的内核自行分配的内存，则

#ifndef LWIP\_RAM\_HEAP\_POINTER

/\*\* the heap. we need one struct mem at the end and some room for alignment \*/

u8\_t ram\_heap[MEM\_SIZE\_ALIGNED + (2\*SIZEOF\_STRUCT\_MEM) + MEM\_ALIGNMENT]; /\* 堆内存空间分配 \*/

**#define LWIP\_RAM\_HEAP\_POINTER ram\_heap**

#endif /\* LWIP\_RAM\_HEAP\_POINTER \*/

MEM\_SIZE\_ALIGNED（对齐的堆内存大小）

功能：表示对齐的可用于分配的堆内存的大小（因为堆内存实际分配的大小比这个值大）。

全局变量

堆内存空间分配（可配置）

分配策略：lwip内部的DMM在使用堆内存策略时，所管理的堆内存可以是用户自定义的一块内存，也可以是内核内部自行分配的一块内存。

首地址：由宏常量LWIP\_RAM\_HEAP\_POINTER表示；

分配策略配置：mem.c中根据是否已定义了宏常量LWIP\_RAM\_HEAP\_POINTER，来决定是使用哪种方式：如果已经定义了则使用用户自定义的一块内存，否则内核自行分配一块内存。

1）如果是用户自定义的一块内存，则要做以下两步：*（在哪个文件中？在cc.h还是直接在mem.c中？俩也许都可以）*

1、在内存中以全局变量的形式分配堆内存，大小参考“堆内存空间实际大小”；

2、定义宏常量LWIP\_RAM\_HEAP\_POINTER，且等于堆内存的实际起始地址；

2）如果是使用内核自行分配的方式，则在mem.c编译相应的代码：

#ifndef LWIP\_RAM\_HEAP\_POINTER

u8\_t ram\_heap[MEM\_SIZE\_ALIGNED + (2\*SIZEOF\_STRUCT\_MEM) + MEM\_ALIGNMENT];

#define LWIP\_RAM\_HEAP\_POINTER ram\_heap

#endif /\* LWIP\_RAM\_HEAP\_POINTER \*/

堆内存空间实际大小

功能：计算堆内存空间实际分配的内存大小（字节单位）；

实现C代码：

对齐的MEM\_SIZE+两个对齐的mem结构体大小+1个对齐字；

MEM\_SIZE\_ALIGNED + (2\*SIZEOF\_STRUCT\_MEM) + MEM\_ALIGNMENT

ram

功能：无符号8位字节指针，指向堆内存对齐后的首地址；

数据结构：

结构体mem

功能：

定义：

struct mem {

/\*\* index (-> ram[next]) of the next struct \*/

mem\_size\_t next;

/\*\* index (-> ram[prev]) of the previous struct \*/

mem\_size\_t prev;

/\*\* 1: this area is used; 0: this area is unused \*/

u8\_t used;

};

对齐大小：SIZEOF\_STRUCT\_MEM

接口实现：

堆内存空间分配（可配置）

##### 实现策略2：基于固定内存块的池内存策略

###### 原理：

*<参考《数据结构(c语言版)》严蔚敏 吴伟民，第195页可利用空间表的三种结构中的第二种，uCOS-II中的内存管理其实也是这种情况>*

本策略的实现机制实际是通过池内存管理器实现的，由用户通过宏定义确定若干个内存

块大小和个数不同的内存池，这些池被置于池管理器下来管理。堆内存管理器在申请或释放

内存块时只是自己找到合适的自定义内存池，再通过调用池内存器管理的相关函数来从对应

内存池中申请或释放内存块。

1、首先用户根据项目实际需要自定义一个lwippools.h的头文件，文件中定义内存块大小及个数不相同的若干个内存池，并且定义时，按照内存池中内存块的大小从小到大排列各个内存池。这些用户自定义内存池被置于池内存管理器的管理，在初始化时，同池内存管理器管理的其他内存池一并初始化；

在lwippoos.h中的定义格式：

若Define three pools with sizes 256, 512, and 1512 bytes，则：

LWIP\_MALLOC\_MEMPOOL\_START

LWIP\_MALLOC\_MEMPOOL(20, 256)

LWIP\_MALLOC\_MEMPOOL(10, 512)

LWIP\_MALLOC\_MEMPOOL(5, 1512)

LWIP\_MALLOC\_MEMPOOL\_END

2、动态内存的申请和释放：

每次动态内存的分配与释放都是以自定义内存池中固定大小的内存块为单位，在相应

的内存池中分配和回收。

**申请内存：mem\_malloc**

分配时采用first-fit（首次拟合）算法， 用户输入的是*size*大小，实际要分配的是一

个memp\_malloc\_helper+*size*大小的内存，从第一个用户自定义池开始查找池中内存块满足需求的第一个内存池，如果还有空闲内存块则取出一个空闲内存块来(*取内存块的操作由动态内存池管理器的memp\_malloc函数完成*)。如果已经空，且**MEM\_USE\_POOLS\_TRY\_BIGGER\_POOL**宏开关打开，则从内存块更大的自定义内存池中申请。最终申请的空闲内存块分为

memp\_malloc\_helper+*size*两部分，实际给用户返回的是*size*部分的首地址，用户只能使用*size*大小的内存部分。helper里面存储的是分配的*size*大小的内存所属用户自定义内存池的内存池类型。*（内存池类型是池内存管理器为其所管理的所有内存池分配的一个类型名）*

**释放内存时：mem\_free**

用户只提供当初申请时返回的size大小内存的首地址（也是当初分配时返回的），并没有显式的提供该内存是属于哪个用户自定义内存池的信息。这就是memp\_malloc\_helper的作用，并且helper结构当初在分配size内存时就隐式的在size内存首地址前定义好并存储了相应的内存池类型信息，释放时堆管理器提取出内存池类型信息，然后调用池内存管理器的释放函数，由其根据内存池类型信息和内存块的首地址，将size内存块+其前面的memp\_malloc\_helper结构所占的内存释放到相应用户自定义内存池空闲元素链表的头部。

###### 性能：

优点：实现简单，分配与释放较快；

缺点：内存浪费相对较多；

###### 实现：

依赖：

1. 平台环境移植文件：基本数据类型、格式化输出说明符等；
2. 池内存管理器提供的用于的用户自定义内存池管理的内存池类型、memp\_malloc\_helper结构体类型，第一个和最后一个用户自定义内存池在所有内存池中的定位宏（MEMP\_POOL\_FIRST,MEMP\_POOL\_LAST）等；

3、用户配置MEM\_SIZE；

内部控制结构：（无）

本身没有定义（即mem.h中没有），而是使用池内存管理器为管理用户自定义内存池使用的各个数据类型、数据结构、宏等；

接口实现：

数据类型和格式化输出说明符（mem\_size\_t和MEM\_SIZE\_F）

设计详解：

**因为**mem\_size\_t类型具体是哪种无符号整型数与为堆内存实际分配的大小（MEM\_SIZE+用于对齐的内存大小）有关；

**所以**理论上当（MEM\_SIZE+用于对齐的内存大小）大于16无符号能表示的最大数（65535）时，mem\_size\_t就用32位无符号整数表示，否则用16位2无符号整型数表示；

**但是因为**“用于对齐的内存大小”不确定，

**所以**在实现时，**假设**“用于对齐的内存大小”最大值是1535个字节，并且使用MEM\_SIZE进行比较，如果MEM\_SIZE>(65535-1535)=64000，则使用32位无符号整型数，否则使用16位无符号整型数。

**代码：**

**参考src\include\lwip\mem.h。**

操作接口：

mem\_init(初始化堆内存管理模块)

lwip不用实现，所以将该函数宏替换成空。lwip\_init的代码中就不包含调用mem\_init的代码。

mem\_malloc(分配一个内存块)

功能：输入申请的内存的字节大小，从用户自定义的内存池中寻找能满足需求的内存池，然后从其中分配一个空闲内存块。

输入：

申请的字节大小size（mem\_size\_t类型）

输出：

分配的内存的首地址或NULL

基本流程：

1、计算实际所需的内存大小：size+ LWIP\_MEM\_ALIGN\_SIZE(sizeof(struct memp\_malloc\_helper))；

2、在用户自定义的动态内存池中从第一个池开始，查找内存块大小满足所需大小的动态内存池，如果找到则退出循环，否则继续循环直到最后一个动态内存池（退出循环有两种情况，一种是：找到了内存块大小足够大的动态内存池；二是循环直到最后一个动态内存池都没有）；

3、判断有没有找到合适的动态内存池，没有则返回NULL，退出程序；

4、如果找到合适的动态内存池，则从中申请一个空闲块(memp\_malloc())，如果成功，则返回内存块地址并退出程序；

5、如果没有可用空闲块，则判断是否需要从更大内存池中申请，然后从更大的动态内存池中循环寻找合适的内存池，重复3、4步，甚至5步。

mem\_free（释放一个内存块）

功能：

将内存块释放到其原先分配时所属的用户自定义内存池中；

输入：

内存首地址；

输出：

无

基本流程：

1. 根据输入的内存块首地址计算其对应的memp\_malloc\_helper结构的首地址（方法：减去LWIP\_MEM\_ALIGN\_SIZE(sizeof(struct memp\_malloc\_helper))）；

2、调用memp\_free()，将内存块与memp\_malloc\_helper结构所占的内存释放；

mem\_calloc（分配一个内存块，并清零）

功能：

按数组方式申请一块连续内存，并清零。

输入：

元素个数（mem\_size\_t类型）；元素字节大小（mem\_size\_t类型）；

输出：

申请到的内存首地址或NULL；

基本流程：

1、计算内存大小=元素个数\*元素字节大小；

2、调用mem\_malloc()；

3、如果申请成功，则调用memset()清零；否则直接退出程序；

mem\_trim（调小一个内存块）

**与C库实现策略时一样，因为：**

mem\_trim is not used when using pools instead of a heap: we can't free part of a pool element and don't want to copy the rest。

## 子模块---池内存管理

# lwip---动态内存管理参考手册

本部分内容来源：

1. 从上面设计手册复制；
2. 基于上面设计手册的内容的总结；

## 堆内存管理

### 用户配置项（opt.h）（总结）

#### 实现策略配置：

注释：

1. 动态内存管理器的两个模块实现策略的配置只能是下面表格中的几种情况，绿色部分是默认配置。
2. 关于红色标记部分的说明：

2.1）如果MEM\_LIBC\_MALLOC=0，MEM\_USE\_POOLS=1&& MEMP\_USE\_CUSTOM\_POOLS=1，则MEMP\_MEM\_MALLOC必须等于0！！！！

2.2）如果MEMP\_MEM\_MALLOC=1，那么必须有MEM\_LIBC\_MALLOC=0，MEM\_USE\_POOLS=0&& MEMP\_USE\_CUSTOM\_POOLS=0！！！！！

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 动态堆内存管理器 | | | 动态池内存管理器  memp\_mem\_malloc | |
| C库策略  mem\_libc\_malloc | LWIP内部自带  mem\_use\_pools | |
|  | 基于固定内存块通过池内存管理器的实现策略 | 基于可变内存块的堆内存实现策略 | 固定内存块的池内存策略 | 使用堆内存管理器 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 |
|  |  |  |  |  |

#### 内部实现参数配置