# LwIP协议栈

## TCP/IP协议栈基础

### TCP/IP协议栈模型

### TCP/IP协议栈的数据及数据操作特点：

TCP/IP协议的基本原理是在各协议层之间对数据分组进行封装或解复用，而数据分组的长度是不确定的，大的可达KB字节，小的只有几字节。而且分组在协议层之间流动时，需要在分组头或尾动态地添加或删除数据。

### 实现：

#### 协议栈进程模型（7.1.2协议栈与应用程序、操作系统的关系）

#### 数据访问模式（7.1.1）

**存储区：**

在包含TCP/IP协议栈的嵌入式系统中，一般可以把整个系统的内存分为用户程序存储区和协议栈存储区。

*（这里用户程序指所有使用协议栈进行网络通信的程序，包括应用程序，如果有操作系统的话，也包括操作系统中使用协议栈的进程）*

存储区又分静态存储区（存储一些全局数据）和动态存储区（存储动态数据）。

**缓冲存储区：**

当这些存储区的某些部分中的数据被用来网络通信时，这些存储区从功能上就称为缓冲存储区（可能是一大块连续的内存区，更常见的是各个离散的内存块在功能上构成的“缓冲存储区”）。分两种：

用户程序存储缓冲区：位于用户程序存储区的buffer。

协议栈存储缓冲区：位于协议栈存储区的buffer。

**数据访问模式：**

在进行网络通信时，一般而言用户程序管理自己的存储缓冲区，为自己收发的数据分配缓存，协议栈管理自己的存储缓冲区，用户收发数据分组。

但是在具体协议栈实现时，基于时空效率的考虑可以有以下三种数据访问模式：

**严格分层的数据访问模式**

特点：用户程序与协议栈，协议栈内部各层之间，都有各自的独立的缓冲存储区，当数据包传递时，需要从一层复制到另一层，当某一层需要其他层中的数据时，只能通过其他层提供的接口函数来实现。每一层在处理完成时，都要将分配的缓存回收重用。例如在TCP层在计算TCP报文段的校验和以及TCP在重装无序报文段时，TCP都必须直到报文段的源IP地址和目的IP地址。为了得到这些信息，TCP层只能调用IP层地相关接口函数。

缺点：数据包在各层之间传递时因为涉及大量的内存复制，将使协议变得非常慢。

**稍微不严格分层的数据访问模式**

特点：

保持用户程序与协议栈之间的严格分层，但在协议栈内部数据包管理时就不在遵循严格的分层。

（1）前者是指，用户程序并不知道协议栈使用的数据包的结构和机制，收发数据时，数据包必须在用户程序缓冲存储区与协议栈缓冲存储区之间进行复制。

方法是：协议内核在外部提供了API层，用户程序只能使用几个API函数发送或接收数据；或者协议栈作为操作系统内核代码的一部分，由操作系统将协议函数或API函数进行完全封装，用户程序可以像操作简单文件那样处理网络通信的数据。（例如常见的BSD socket）。

（2）后者是指，协议栈内部各层之间不采用分级访问的模式，数据包在协议栈缓存中只存在一份，在各层之间的传递只是传递数据包的指针，每一层都知道数据包的结构（也就是数据包中各个协议首部字段和数据部分的位置），利用指针根据偏移在数据包中直接读写自己的首部字段或读写自己需要的其它协议层中的数据。

**最不严格分层的数据访问模式（通常用于小型的嵌入式系统）**

特点：用户程序与协议栈内核（或者说操作系统），协议栈内部各层之间都不遵循严格的分层。

用户程序发送数据时，只是把自己缓冲存储区中数据的地址告诉协议栈；接收数据时，协议栈都是将数据存在自己的动态存储区中，协议栈内部向上传递和向用户程序传递时，也只是告诉数据的地址。

在这种实现中，假定用户程序和协议栈内部各层都了解协议栈内部的数据包结构以及缓冲管理机制，数据包的传递只是在传递指针，用户程序和协议栈内部各层都是通过指针访问数据。

## 功能：

是TCP/IP协议栈的一种独立的轻型的实现。

*【“独立”是指LwIP系统可以不依赖于操作系统而独立运行；*

*“轻型”是指LwIP系统的存储器资源使用及代码容量少。】*

## 设计目标（特点）：

1、在实现TCP/IP协议栈功能的基础之上，尽量减少LwIP系统使用的RAM资源及代码容量。

## 应用场景：

Suitable for use in embedded systems with **tens of kilobytes of free RAM** and room for around **40 kilobytes of code ROM**.

*【lwip-1.4.1 《readme》】*

## 内核配置：

### 目的：

在移植LWIP内核时，根据实际产品硬件及应用需求配置内核中的相关参数，例如关闭某些功能模块，或设置某些数据结构的个数等；

### 配置内容及方法：

可配置内容：

可配置参数（各种开关量及宏常量）：

1）全部的可配置参数都在opt.h中。

2）少数可配置参数在某些.c文件中*（具体内容等到了解全部代码再列出）*，不过这涉及到的是直接修改内核源码；（不建议配置！！！）；

配置方法：

用户在移植时自定义一个名字为“lwipopts.h”的头文件，在该头文件配置相关参数，

如果某些参数没有配置，则内核使用opt.h中的默认配置；

### 内核配置机制的实现：

为了避免用户在移植LwIP内核配置相关参数时直接修改源码文件*（虽然也可以直接修改完成移植）*，内核规定在移植时，需要用户自定义一个名字为lwipopts.h的配置文件，在其中进行宏定义修改参数，如果某些参数没有设置，则内核使用自带配置文件opt.h中的默认设置。

内核在编译时实际使用的配置文件是内核自带配置文件opt.h和用户自定义lwipopts.h合成的一个配置文件。

内核自带配置文件opt.h的实现：

1、首先条件预编译指令实现头文件的幂等性；

2、然后将用户自定义的配置文件lwipopts.h包含在本文件中（里面有所有可配置参数的宏定义）；

3、opt.h中定义所有可配置参数的宏定义，并且采用条件预编译指令#ifndef 宏 #define 宏 #endif；

## 模块----动态内存管理

*只考虑了基本功能。代码中的调试代码，内存对齐，临界区的实现，其它复杂的预编译指令的功能，*

*动态内存管理的完整清晰的架构，还有待完善整理。*

### 需求：

**概念定义：（什么是动态内存）**

“动态内存”是用于满足程序动态申请与释放内存的一块内存，用以存储及处理一些临时数据，所谓“临时”是相对于程序的整个运行时间而言的，①在程序的整个运行时间内程序可能要做很多事，在处理每一件事时都可能涉及临时数据，②还有一些数据或数据结构是用于整个程序运行期间的，当程序结束时需要释放所占的空间，所以也称为临时数据。

按照C语言的观点动态内存又被称为“堆”或“堆内存”。

**动态内存管理目的/意义/作用：**

（动态）内存管理的目的是为了用于处理一些“临时数据”分配内存并回收。而LWIP协议内核处理的“临时数据”其实就是从上层应用程序或底层网络硬件驱动接收的数据或分组以及处理它们时需要的临时的变量数据结构等。

**性能要求：（动态内存管理是什么样的）：**

目标是针对TCP/IP协议栈处理的数据的结构特点和各种操作的特点，提供一种相对高效合理的内存管理机制，用于协议栈在收发数据分组时的数据存储及处理。

**要求是**：简单，且尽量减少在操作数据时的数据复制操作。

### 公共基础：

#### 内存对齐相关宏/宏函数：

##### MEM\_ALIGNMENT(opt.h)

功能：定义特定CPU的内存对齐大小，默认是1字节；

实现：

#ifndef MEM\_ALIGNMENT

#define MEM\_ALIGNMENT 1

#endif

##### LWIP\_MEM\_ALIGN\_SIZE(size) （mem.h）

**功能：**

输入所需要的一块内存的实际字节大小，对输入的大小值进行向上取整，以满足CPU的对齐大小MEM\_ALIGNMENT的对齐方式。

例如当MEM\_ALIGNMENT=4时，LWIP\_MEM\_ALIGN\_SIZE(3)和LWIP\_MEM\_ALIGN\_SIZE(4)返回的都是4。

**实现：（原理？？？？）**

#define LWIP\_MEM\_ALIGN\_SIZE(size) (((size) + MEM\_ALIGNMENT - 1) & ~(MEM\_ALIGNMENT-1))

##### LWIP\_MEM\_ALIGN\_BUFFER(size)

##### LWIP\_MEM\_ALIGN(addr)

### 子模块实现策略的配置：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 动态内存堆管理器 | | | 动态内存池管理器  memp\_mem\_malloc | |
| C库策略  mem\_libc\_malloc | LWIP内部自带  mem\_use\_pools | |
|  | 固定内存块的动态内存池策略 | 可变内存块的内存堆策略 | 固定内存块的动态内存池策略 | 可变内存块的内存堆策略 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 |
|  |  |  |  |  |

注释：

动态内存管理器的两个模块实现策略的配置只能是上面表格中的几种情况，绿色部分是默认配置。

### 子模块---动态内存堆管理(opt.h,mem.h,mem.c)

#### 概述：

动态内存堆管理器（Dynamic Memory Manager）

有两种实现：①直接使用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中宏开关量**MEM\_LIBC\_MALLOC**决定使用其中的哪一种，当使用C库提供的DMM时，那么内部DMM相关代码就不会编译，从而减少内核大小；

#### 对外接口：

##### 数据类型：

mem\_size\_t

功能：

用于描述内存相关字节大小，具体数据类型与DMM的实现方法相关。

##### 宏：

MEM\_SIZE\_F（格式化输出的格式说明符）

##### 操作

###### 专供LWIP内部使用：

**mem\_init()：**

上层节点：LWIP的初始化函数；

使用场景及功能：

使用LWIP内部DMM时才会进行如下操作：内存堆的初始化函数，主要是告知内存堆的起始地址、大小、初始化空闲列表；

使用C库DMM则本函数相当于不存在。

**mem\_trim()：**

使用场景：

仅当使用LWIP内部DMM且其实现算法是基于可变内存块的堆内存管理策略时，才会使用。

###### LWIP内部及用户使用：

**mem\_malloc()：**申请内存。输入需要的内存的字节长度，返回最新分配到的内存的指针，如果没有足够的空闲内存则返回NULL。分配的空间大小会受到内存对齐的影响，可能会比申请的略大。而且返回的内存时没有“初始化”的，即里面可能包含任何随机的垃圾。

**mem\_calloc()：**对mem\_malloc()的简单封装，它有两个参数，分别为元素的个数和大小，两者乘积即为申请的内存的大小，申请完后它会把申请的内存清零。

**mem\_free()：**释放申请的内存。

#### 实现1：C库DMM（MEM\_LIBC\_MALLOC=1）

##### 用户配置：

MEM\_LIBC\_MALLOC=1；其它的使用默认配置；

##### 实现：(mem.h)

###### 宏：

MEM\_SIZE\_F

#define MEM\_SIZE\_F SZT\_F

###### 数据类型：

mem\_size\_t

typedef size\_t mem\_size\_t;

###### 操作：

接口统一：

C库DMM提供的接口是malloc()，free()，calloc()等，与动态内存管理器的对外接口不同，

为了实现统一，LWIP源代码中使用预处理指令，将LWIP内核中其他功能模块中使用的

mem\_alloc，mem\_free等宏展开成C库中的函数名字。而mem\_init()和mem\_trim()在C库中并不存在，所以宏定义成nothing。

例如：

#ifndef mem\_free

#define mem\_free free

#endif

内部实现：

C库的事，无须关心。

#### 实现2：LwIP内部DMM（MEM\_LIBC\_MALLOC=0）

##### 实现策略原理：

两种实现策略：

1、基于可变内存块的堆内存管理

算法原理：

*<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>*

实现定义好一定大小（MEM\_SIZE）的内存作为堆内存，动态分配的内存块的大小不固定，可以随请求而变（实际分配的大小受内存对齐的影响）。并且分配策略是first-fit策略，即只要找到比请求的内存大的空闲块，就从中切割出合适的块，并把剩余的部分返回到内存堆中。

**分配：**

分配的内存块有个最小的大小限制，要求请求分配的大小不能小于MIN\_SIZE，如果小于则分配的内存块大小都固定为MIN\_SIZE。一般MIN\_SIZE为12字节，在这12字节中前几个字节会存放管理用的私有数据，不能被用户程序修改，否则会导致致命问题。

**释放：**

释放一个内存块时，内部管理器会查看前后相邻的内存块是否是空闲的，如果空闲则合并成一个大的空闲块。

**性能：**

**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。

2、基于固定内存块的动态内存池管理

算法原理：

*<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>*

事先建立好内存块大小不同的若干内存池，每次动态内存的分配与回收都是以固定大小的内

存块为单位，在相应的内存池中分配和回收。分配时如果

**MEM\_USE\_POOLS\_TRY\_BIGGER\_POOL**宏开关打开，则在对应大小内存池中没有空闲块时，

会从大内存池中进行分配。分配策略：采用first-fit（首次拟合）；

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；

##### 实现策略的配置：(opt.h)

宏开关：

MEM\_USE\_POOLS和MEMP\_USE\_CUSTOM\_POOLS同时为1，则选择动态内存池管理策略。

##### 实现：(mem.h,mem.c,memp.h,memp.c等)

###### 宏和数据类型：(mem.h)

**原理：**

取决于用户配置位置opt.h中设置的内存堆的字节大小（**MEM\_SIZE**），当应用需要发送的数据较多时，设置大一些，否则小一些。

**实现：**

#if MEM\_SIZE > 64000L

typedef u32\_t mem\_size\_t;

#define MEM\_SIZE\_F U32\_F

#else

typedef u16\_t mem\_size\_t;

#define MEM\_SIZE\_F U16\_F

#endif /\* MEM\_SIZE > 64000 \*/

###### (未完)（系统默认）实现策略1：基于可变内存块的堆内存管理

实现算法：边界标识法原理

可利用空间表：

堆内存管理器维护一个可利用空间表，将所有的空闲块链接成一个双向循环链表，初始状态时，整个堆内存是一整块空闲内存块；

可利用空间表中节点结构：

每一个节点就是一个空闲块。

用户配置项：

**堆内存大小：（opt.h）**

功能：

由用户配置，设定LWIP内部DMM在使用堆内存管理策略时管理的堆内存的字节大小。

如果应用程序发送的数据较多，则该值应设置大一些。

**最小分配大小（mem.c）：**

功能：<*《数据结构(c语言版)》严蔚敏 吴伟民，第199页,8.3.2 <分配算法>(1)*>

与采用的首次拟合分配算法有关，用于避免在大量分配后出现一些容量极小总也分配不出去的空闲块。

用于设置从堆内存中分配内存时的最小的分配字节数，除了每个

基本数据结构：

mem

操作：

mem\_init()

mem\_malloc()

mem\_free()

//mem\_realloc()

mem\_calloc()

mem\_trim()

###### 实现策略2：基于固定内存块的动态内存池管理

原理：

首先用户自定义所需要的内存池，并置于动态内存池管理器下，初始化时一并初始化。

当用户申请内存时：

**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内存池，然后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*)。该空闲内存块分为上面两部分，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类型。

当用户释放内存时：

**当用户释放***size*大小的内存时，只提供了该内存的首地址（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，释放时根据存储的类型信息将释放的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。

实现算法：

这种方法是在下面动态内存池管理的基础上实现的，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，这些池被置于内存池管理器下来管理。动态内存管理器在申请或释

放内存块时只是简单的调用内存池管理的相关函数来实现。

宏开关：(opt.h)

MEM\_USE\_POOLS\_TRY\_BIGGER\_POOL

功能：

从用户自定义的内存池中分配一块内存时，如果在满足容量需求的内存池中没有空闲内存块，那么是否允许从容量更大的内存池中申请；

数据结构：

memp\_malloc\_helper

**功能：**用于释放一个内存块时，帮助确定该内存块在哪个用户内存池中。

**实现**：

该数据结构中只包含一个memp\_t类型的变量，用来存储每一个分配的内存块所属的用户自定义内存池的类型。

#if MEM\_USE\_POOLS

/\*\* This structure is used to save the pool one element came from. \*/

struct memp\_malloc\_helper

{

memp\_t poolnr;

};

**用法：**

**当用户申请***size*大小的内存时，实际分配的是一个memp\_malloc\_helper+*size*大小的内存，

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类型。

**当用户释放***size*大小的内存时，只提供了该内存的首地址（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，释放时根据存储的类型信息将释放的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。

宏常量

MEMP\_POOL\_FIRST和MEMP\_POOL\_LAST

功能：

分别用于表示用户自定义内存池中的第一个和最后一个在所有内存池中的位置（两位置中间的都是用户自定义内存池）。

实现：(参考memp.h)

操作：

用户自定义动态内存池的建立(memp.c)

实现方法：

用户自定义一个名为《lwippools.h》的头文件，其中用户以如下格式的宏定义方法来告知动态内存池管理器定义一些用户自定义的内存池，每个池中的内存块的大小和个数由用户根据应用情况自行确定。

LWIP\_MALLOC\_MEMPOOL\_START

LWIP\_MALLOC\_MEMPOOL(20, 256)

LWIP\_MALLOC\_MEMPOOL(10, 512)

LWIP\_MALLOC\_MEMPOOL(5, 1512)

.........................................................

LWIP\_MALLOC\_MEMPOOL\_END

申请一块内存(mem\_malloc)(mem.c)

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

输出：

分配的内存的首地址

基本流程：

1、计算实际所需的内存大小：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。

申请一块内存并清零(mem\_calloc)(memset函数并没有定义，是否是bug?) (mem.c)

功能：

按数组方式申请一块连续内存，并清零。

输入：

元素个数；元素字节大小；

输出：

申请到的内存首地址；

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；

释放一块内存(mem\_free) (mem.c)

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；

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

mem\_trim()(#define mem\_trim(mem, size) (mem) 什么都不做，实际可以去除)

### 子模块---动态内存池管理(opt.h,memp.h,memp.c)

#### 原理：

##### 功能：

主要管理各种数据结构构成的各种类型的内存池，主要用于**快速方便**的分配各种数据结构。

管理器所支持的内存池的类型、各种类型池的属性值、所有内存池的存储结构都取决于用户自定义配置文件opts.h和内核配置文件opt.h。

##### 存储结构：

宏开关：(opt.h)

MEMP\_SEPARATE\_POOLS

连续存储结构：

所有内存池占用连续的一大块内存，顺序排列*（具体顺序就是在memp\_std.h中各内存池定义的顺序）*；

非连续存储结构：

各个内存池各自独自占用一块连续的内存；

##### 内存池的分类：（memp\_std.h）

###### 三大分类：

*(“可选”是指内存池的有无取决于内核配置文件；“必有”是指内核固有的)*

1)(可选)**MEMPOOL**类型：标准类型的内存池，指由管理器管理的内核各个功能模块控制结构相关的内存池，各种控制结构池的有无以及池中元素的个数依赖于内核配置文件opt.h中的设置；

2)(可选)**MALLOC\_MEMPOOL**类型：用于动态内存堆管理器的内存池，仅在动态内存堆管理器使用基于固定内存块的内存池策略并且在用户自定义头文件lwippools.h中声明了自定义内存池时才会存在，并被动态内存池管理管理；

3)(必有)**PBUF\_MEMPOOL**类型：pbuf结构相关内存池，有两种：一种是pbuf头内存池(PBUF类型)，另一种是完整pbuf内存池(PBUF\_POOL类型)。

###### 具体类型：

内存池的具体类型值是memp\_t枚举类型中的一种枚举常量值。格式都是固定为：

MEMP\_*name*；（*name*是memp\_std.h中LWIP\_MEMPOOL宏函数中*name*，对于用户自定义

内存池，枚举类型是MEMP\_POOL\_*size*）

##### 内存池的属性：(memp\_std.h)

Name：

表示内存池的类型（名字）；

Num： 表示内存池中元素的个数。对于标准内存池及puf相关内存池，个数都在opt.h配置文件中；用于mem\_alloc的内存池的元素个数在用户自定义文件lwippoo.h的形如LWIP\_MALLOC\_MEMPOOL(20, 256)的宏定义中；

Size： 表示内存池中元素的大小，由sizeof来计算；

Disc： 内存池的描述字符串，主要用于调试；

##### 内存池中元素的结构：

MEMP\_SIZE大小的头部+ MEMP\_ALIGN\_SIZE(size)大小的数据部分；

**头部**：是预留用来在某些特殊场合可以利用该空间对内存池元素进行特殊处理，一般为0，即不使用这项功能。

**数据部分**：实际用来存储各中内存池中的各个数据结构，但是大小是对齐后的大小。

#### 对外接口：

数据类型

memp\_t

操作：

memp\_init()(如果使用基于内存堆的策略，则不用)

memp\_alloc()

memp\_free()

#### 实现：

##### 1）基于固定内存块的动态内存池策略；

###### 原理：

管理器为所支持的所有内存池分配空间，每个内存池都以数组的形式实现。

每个内存池中的元素分已用和空闲两种，内核初始化时，调用管理器提供的内存池初始化函数，将每个内存池中的空闲元素链成一个单向链表，每次申请或释放内存池元素都是在空闲元素链表的头部进行操作：

内存池元素的申请：直接将对应内存池空闲元素链表的头结点取出来；

内存池元素的释放：直接将释放的元素插入空闲元素链表的头部，作为新的头结点；

每个内存池的基本信息：

元素的个数、元素的大小、空闲链表的头指针。

管理器管理内存池基本信息的方法就是将所有内存池的同一类信息组织在一起构成数组，从而形成了下面的数据结构。

*(内存池(基本信息)的管理方法与uCOS-II不同，uCOS中是将属于一个内存池（称为内存分区）的信息汇聚在一起构成内存控制块MCB)。*

###### 基本数据结构：

(必有)memp\_t（内存池类型）(memp.h)

**功能：**

是DMPM定义的一种枚举数据类型，主要目的是为内核使用的所有内存池都取一个好看、

好听的名字（或者说是为每一种内存池分配一个编号），提高源码的可读性。申请或释放

某一种数据结构时，都是通过提供memp\_t类型的枚举常量值告知内核申请或释放到哪一种

内存池中，实际上DMPM就是用这些枚举常量的实际编号作为memp\_sizes[]，memp\_num[]，

memp\_tab[]等数组的索引值，来找到对应的内存池。

**实现：*详情参考代码memp.h***

memp\_t数据类型用来限定内存池的类型，该类型的变量取值只能是DMPM所支持的所有内存池类型。但是DMPM支持的所有内存池的类型数目是可变的，取决于用户自定义配置文件lwipopts.h和内核配置文件opt.h的。

在C语言中用于定义一种符号取值范围是限定范围内的数据类型的方法就是使用枚举数据类型。为了实现枚举数据类型中的枚举常量值数目可变，LWIP的作者充分使用了预处理指令，并且还保持到了代码的简洁。

(必有)内部/全局memp\_sizes[](memp.c)

**功能：**

u16\_t类型的一维数组，里面存储了DMPM所支持的各种内存池中元素的字节大小（对齐后的大小），元素个数是MEMP\_MAX。

**元素索引：**

数组元素的顺序就是在memp\_t中的顺序（其实是在memp\_std.h中的顺序，因为memp\_t也是根据memp\_std.h实现的）通过memp\_t枚举类型中的某一个枚举常量值来索引。内存池的元素也就是各种数据结构，它们的大小在该结构定义后都是固定的，唯一大小不确定的就是用户自定义内存池中内存块的大小取决于用户自定义配置文件lwipopts.h。

**作用域：局部还是全局数据结构：（取决于配置）**

如果DMPM不管理第二大类的内存池（即不管理用户自定义内存池）并且也不是通过DMHM实现的（即不是使用内存堆策略），那么memp\_sizes就是DMPM的内部数据结构，否则是全局数据结构（即可被DMHM访问）。

**实现：*详情参考代码memp.c***

(可选)内部数据结构memp\_num[](memp.c)（MEMP\_MEM\_MALLOC=0时存在）

**存在条件：**

只有当DMPM是基于固定内存块内存池策略时（MEMP\_MEM\_MALLOC=0）才存在。

**功能：**

u16\_t类型的一维数组，存储的是每个内存池中元素的个数。

**元素的索引：**

数组元素的顺序就是在memp\_t中的顺序（其实是在memp\_std.h中的顺序，因为memp\_t也是根据memp\_std.h实现的）通过memp\_t枚举类型中的对应枚举常量值来索引。

**实现：*详情参考代码memp.c***

(可选)内部数据结构memp\_desc[](memp.c)（MEMP\_MEM\_MALLOC=0且LWIP\_DEBUG=1时存在）

**存在条件：**

MEMP\_MEM\_MALLOC=0且LWIP\_DEBUG=1时存在

**功能：**

字符型指针数组，每一个数组元素都是一个字符指针，每一个指针都指向对应内存池的字符串名字。用于调试时显示字符信息。

**实现：*详情参考代码memp.c***

(可选)内部数据结构memp\_tab[MEMP\_MAX] (memp.c)(MEMP\_MEM\_MALLOC=0时存在）

功能：

struct memp类型的指针数组，由各个内存池中空闲元素链表头指针构成的指针数组。每一个指针指向一个内存池中的空闲元素链表中的第一个空闲元素，如果某个内存池中没有空闲元素了，则指针值为NULL。

(可选)内部数据结构memp(memp.c)(MEMP\_MEM\_MALLOC=0时存在）

功能：

主要包含一个本结构类型的指针，用于将各个内存池中的空闲元素链成链表。

###### 操作：

基本操作：

建立所有内存池(数组的方式实现)

**功能及原理：**

为DMPM支持的所有内存池都分配内存，并且根据MEMP\_SEPARATE\_POOLS配置项决定在连续存储结构还是非连续存储结构。

**输入：**

各个内存池的元素的个数和大小；

**输出：**

在内存中为各个内存池分配内存；

**实现：*详情参考代码memp.c***

如果是连续存储结构：

u8\_t类型的一维数组，大小等于所有内存池的大小和；

如果是非连续存储结构：

则分别定义，共MEMP\_MAX个u8\_t类型的一维数组，每个数组大小等于对应类型内存池的大小；

初始化各内存池(memp\_init)

**功能：（不包含统计及溢出检查功能）**

将各个内存池中的空闲元素链成一个单向链表，并将memp\_tab[]中的各个头指针分别指向对应内存池中空闲元素链表的头结点。

**输入：**

无；隐式输入：memp\_sizes[],memp\_num[]

**输出：**

无

从一个内存池中申请一个空闲元素(memp\_malloc)

**功能：不包含溢出检查功能**

输入内存池类型，从对应的内存池的空闲元素链表头部返回一个空闲元素，如果链表已空，则返回NULL，表示申请失败；

**输入：**

memp\_t类型的值（内存池类型），用于指定内存池；

**输出：**

失败：NULL；

成功：找到的空闲元素的数据部分的首地址。

**基本流程：（整个都是在临界区内执行）**

1、根据输入的内存类型作为索引，在内存池空闲元素链表头指针数组memp\_tab[]中找到对应内存池空闲元素链表的头指针；

2、判断空闲元素链表是否为空，是直接退出程序，返回NULL；

3、如果不为空，则修改空闲元素链表的头指针，使其指向下一个空闲元素，退出程序并返回找到的**空闲元素的数据部分**的首地址***（参考内存池元素的结构）***；

释放一个已用元素(memp\_free)

**功能：（不包含溢出检查功能）**

将释放的元素放回的指定内存池中的空闲元素链表的头部。

**输入：**

内存池类型；已用元素数据部分的地址；

**输出：**

无；

**基本流程：（整个都是在临界区内执行）**

1、检查输入的地址是否为空，是直接退出程序；

2、将输入的元素数据部分的地址，计算元素的实际首地址；

(struct memp \*)(void \*)((u8\_t\*)mem - MEMP\_SIZE)

3、将释放的已用元素放到空闲元素链表头部，并修改memp\_tab[]中对应空闲元素链表的

头指针；

##### 2）基于可变内存块的内存堆策略；(memp.h)

实现：

不需要预先建立内存池，也不需要初始化，而是需要某种数据结构时直接从动态内存堆中申请（类似于c库函数malloc()和free()的功能）。

#define memp\_init() /\* 此函数不需要 \*/

#define memp\_malloc(type) mem\_malloc(memp\_sizes[type])

#define memp\_free(type, mem) mem\_free(mem)

## 模块----数据包缓冲管理

### 功能：

针对TCP/IP协议栈的数据及数据操作的特点，提供一种高效的机制，使LWIP内核能对流经协议栈的数据分组进行灵活的处理，同时减少数据再各层间传递时的时间与空间开销。

负责管理系统中用于数据包缓冲的缓冲存储区，因为Lwip采用不严格的分层数据访问模式，所以缓冲存储区包括用户程序和协议栈的缓冲存储区。

（所谓缓冲存储区是指所有用于数据包存储的内存区，可能是一大块连续的内存区，更常见的是各个离散的内存块在功能上构成的“缓冲存储区”）

### 特点：

Lwip协议内核的数据包缓冲管理机制就是基于“协议栈分层特点”一节中第三种“最不严格分层”方法实现的。

### 实现原理：

Lwip的数据包缓冲管理是基于一种pbuf数据结构。

协议栈收发的每一个数据包被组织成一种pbuf结构链表，链表中的每一个节点是一个pbuf结构。根据数据包的大小，一个pbuf链表可能包含多个pbuf，也可能只包含一个pbuf。

**pbuf在接收发送时的基本用法：**

发送时：某一层协议从上层接收到数据，然后封装自己的协议首部，然后向下层协议传递。具体到pbuf就是：从上层接收到pbuf（链）指针，此时第一个pbuf的payload指向从上层接收到的数据，本层通过pbuf\_header函数调整payload指针到首部空间中为它预留的协议首部字段，进行填写。最后将payload调整到首部空间中自己协议首部字段的起始位置，然后将pbuf传递给下层协议。

接收时：某一层协议从下层接收到数据，然后读写自己的协议首部，并向上层协议提交数据。

具体到pbuf就是：从下层接收到pbuf（链）指针，此时第一个pbuf的payload指针指向本层协议首部字段在首部空间中的起始位置，本层协议读写首部字段验证之后，将payload指针调整到本层的有效数据区（也就是上层协议协议首部字段在首部空间中的起始位置）。

pbuf结构既支持在动态内存中为数据包分配空间，也支持通过引用在动态内存或ROM中存储的外部数据（externally managed data）作为pbuf的数据进行管理。（通过不同类型的pbuf实现）

单pbuf链结构数据包队列：（currently ,packet queues are not supported!!!Use helper struct to queue multiple packets）（当前版本不支持）

当多个数据包排成队列时，也是通过每个数据包的pbuf链表直接连接而成，即让前一个数据包的最后一个pbuf的next指针指向下一个数据包的第一个pbuf。

（在数据包队列中）判断一个数据包pbuf链中最后一个pbuf的方法：

如果一个pbuf的tot\_len == p->len，则表示该pbuf是当前数据包中的最后一个pbuf，而不是next == NULL。

#### 完整的pbuf结构：

一个完整的pbuf结构包括pbuf头和数据区两部分。根据pbuf类型的不同，pbuf的头和数据区两部分可能占用缓冲区中的一块连续的内存，也可能占用不同的内存*（详情参考pbuf的类型）*。

##### pbuf头：（struct pbuf及各成员的作用）

###### 功能：

通过其中包含的控制信息控制一个pbuf；

###### 成员及作用：

struct pbuf {

struct pbuf \*next;

void \*payload;

u16\_t tot\_len;

u16\_t len;

u8\_t type;

u8\_t flags;

u16\_t ref;

};

next ：pbuf结构体类型的指针，用于构造pbuf链时，指向下一个pbuf，如果已经是pbuf链表中的最后一个pbuf，则等于NULL。

payload ：空型指针（因为其实际指向的数据类型并不确定），指向该pbuf的数据区中有效数据部分的起始位置。

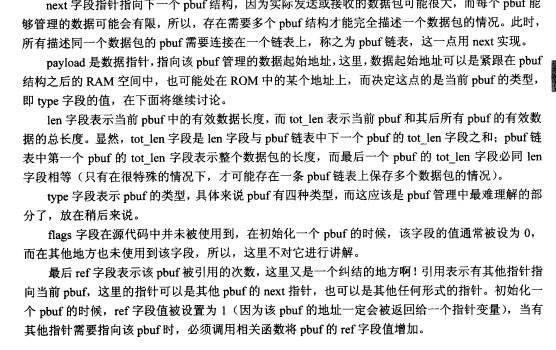
tot\_len ：total length，无符号16位数，表示该pbuf和其后面属于同一个数据包的所有pbuf数据区中有效数据部分的字节总长度。p->tol\_len == p->len + (p->next ? p->next->tol\_len : 0 )；

len ：无符号16位数，表示该pbuf数据区中有效数据部分的字节长度；

type ：无符号8位数*（没有使用枚举类型以节省空间，因为C语言中枚举类型的常量值是按照unsigned int型实现的）*，表示该pbuf的类型*（详情参考pbuf的类型）*。

flags ：无符号8位数，标识位，在源码中没有用到，初始化一个pbuf时设为0。

ref ：无符号16位数，表示该pbuf被引用的次数，值总是等于指向该pbuf的指针的个数，这些指针可以来自应用程序或堆栈或前一个pbuf的next成员。初始化时设为1*（因为该pbuf的地址一定会返回给一个指针变量）*。



##### 数据区（总长度是大小可变的）：

###### 首部空间(可选)

功能：

存储一个数据包从某一层产生并向上或上下流经协议栈时，所经过的各层协议首部字段的总和。存储一个数据包的pbuf链表中只能是第一个pbuf中包含“首部空间”，而且整个“首部空间”只能在一个pbuf中，不能跨多个pbuf。

大小：

申请pbuf时根据所在协议层的不同，在第一个pbuf中预留的首部空间的大小不同，pbuf是根据输入的pbuf\_layer参数，函数内部自动计算。

switch (layer) {

case PBUF\_TRANSPORT:

/\* add room for transport (often TCP) layer header \*/

offset = PBUF\_LINK\_HLEN + PBUF\_IP\_HLEN + PBUF\_TRANSPORT\_HLEN;

break;

case PBUF\_IP:

/\* add room for IP layer header \*/

offset = PBUF\_LINK\_HLEN + PBUF\_IP\_HLEN;

break;

case PBUF\_LINK:

/\* add room for link layer header \*/

offset = PBUF\_LINK\_HLEN;

break;

case PBUF\_RAW:

offset = 0;

break;

default:

LWIP\_ASSERT("pbuf\_alloc: bad pbuf layer", 0);

return NULL;

}

使用：

它是在数据包一开始产生申请pbuf时，在第一个pbuf中为各个首部字段预留好的空间。其它各层协议在处理该数据包时，从获得的pbuf链表的第一个pbuf的首地址处，通过pbuf\_header自动偏移payload指针到其首部字段在“首部空间”中的位置，读写自己的首部字段。当需要为其他协议首部字段准备数据时，也是自己直接偏移到对应位置读写。

当处于pbuf链表中的某一个pbuf（通常就是链表中的第一个pbuf）中包含有数据包的所有的、各种的首部字段时，则这个pbuf的pbuf体中存在“首部空间”，“首部空间”的长度等于pbuf头中的offset成员的值。

###### 有效数据区(payload区)

功能：

存储协议数据单元的数据部分。对不同的协议层其数据部分不同（某一层的数据部分是上层的整个协议数据单元，所以有效数据区会随着协议层不同而变化，表现在pbuf中就是payload指针的值变化）。

位置：

pbuf头中payload指针指向的位置，就是有效数据区的起始位置。

长度：

在单个pbuf中长度等于p->len，对pbuf链表来说长度=p->tot\_len。

#### pbuf类型

##### 区别：

pbuf分四种类型：PBUF\_RAM,PBUF\_POOL,PBUF\_ROM,PBUF\_REF。

区别：

从结构来说：区别在于pbuf头和数据区是作为整体占用一块连续的内存，还是头和数据区分离；

PBUF\_RAM和PBUF\_POOL是连续的；PBUF\_ROM和PBUF\_REF是分离的

从是否包含首部空间来说：

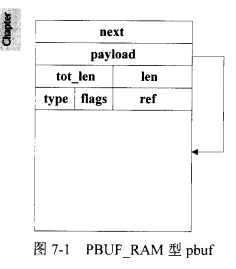
PBUF\_RAM类型的pbuf都包含一个offset大小的首部空间；

PBUF\_POOL类型的pbuf数据区在预定义大小时都保留了首部空间的大小，但申请时因为返回的可能是多个PBUF\_POOL类型的pbuf构成的pbuf链，在链中只有第一个pbuf的数据区存在首部空间，后面的pbuf的数据区全都用来作为有效数据部分。

PBUF\_ROM和PBUF\_REF类型的pbuf头和数据区是分离的，数据区完全由用户程序控制。

##### PBUF\_RAM类型

**结构：**



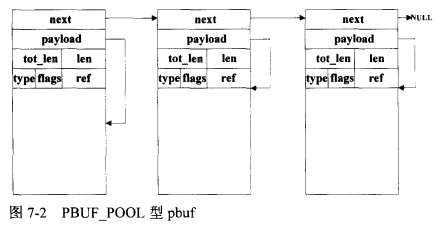
PBUF\_RAM类型的pbuf是占用连续的内存完整的pbuf结构，同时包含pbuf头和数据区。空间是从协议内核管理的动态内存堆中分配的。申请的每一个PBUF\_RAM类型的pbuf数据区中预留了首部空间，首部空间的大小offset根据申请时所在的协议层计算。数据区有效数据部分的长度等于申请时输入的长度。

**用途：**

在用户程序或协议栈发送动态数据时通常使用这种类型，这种类型的pbuf使用最多；

##### PBUF\_POOL类型

结构：



PBUF\_POOL类型的pbuf是占用连续的内存的完整的pbuf结构，是直接从MEMP\_PBUF\_POOL类型的内存池中分配的。同时包含pbuf头和数据区。数据区的大小是在内存池建立时按照最大值预定好的（PBUF\_POOL\_BUFSIZE=TCP\_MSS+40+PBUF\_LINK\_HLEN）。

但申请时因为返回的可能是多个PBUF\_POOL类型的pbuf构成的pbuf链，在链中只有第一个pbuf的数据区存在首部空间，后面的pbuf的数据区全都用来作为有效数据部分。

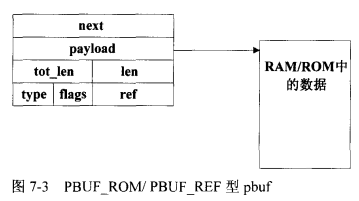
*(之所以返回的可能是多个PBUF\_POOL类型的pbuf构成的pbuf链，是因为第一个pbuf的数据区在留出首部空间后，剩下的有效数据部分长度小于申请时输入的数据长度，所以需要继续申请PBUF\_POOL类型的pbuf，直到有效数据部分的总长度满足要求，并且将多个pbuf链成链表。)*

用途：

主要用在网络驱动层接收数据时，快速申请pbuf结构存储接收的数据；

##### PBUF\_ROM类型和PBUF\_REF类型

结构：



- PBUF\_ROM: no buffer memory is allocated for the pbuf, even for

\* protocol headers. Additional headers must be prepended

\* by allocating another pbuf and chain in to the front of

\* the ROM pbuf. It is assumed that the memory used is really

\* similar to ROM in that it is immutable and will not be

\* changed. Memory which is dynamic should generally not

\* be attached to PBUF\_ROM pbufs. Use PBUF\_REF instead.

\* - PBUF\_REF: no buffer memory is allocated for the pbuf, even for

\* protocol headers. It is assumed that the pbuf is only

\* being used in a single thread. If the pbuf gets queued,

\* then pbuf\_take should be called to copy the buffer.

用途：

ROM只能用于发送静态数据，REF可用于收发动态数据；

### 实现：(不带内存对齐和custom\_pbuf，带操作系统的分析)

#### 依赖：

##### MEMCPY（pbuf\_copy等用到，opt.h）

功能：按字节复制数据，具体实现取决于用户硬件平台，默认是使用内核配置文件opt.h的配置，采用c函数库中的memcpy()函数；

实现：(opt.h)

##### 公共词：

###### LWIP\_MIN

###### 有符号加减法

LWIP\_+

LWIP\_-

##### 进出临界代码区

##### 配置：

###### 自定义PBUF相关(opt.h)

IP\_FRAG

IP\_FRAG\_USES\_STATIC\_BUF

LWIP\_NETIF\_TX\_SINGLE\_PBUF

###### Free ooseq PBUF\_POOL pbufs相关(宏开关)LWIP\_TCP(opt.h)(用于free ooseq PBUF\_POOL pbufs一级开关)

功能：

控制协议栈内核是否包含tcp协议功能的宏开关。

实现：

/\*\*

\* LWIP\_TCP==1: Turn on TCP.

\*/

#ifndef LWIP\_TCP

#define LWIP\_TCP 1

#endif

(宏开关)TCP\_QUEUE\_OOSEQ(opt.h) (用于free ooseq PBUF\_POOL pbufs二级开关)

功能：

控制TCP协议在接收到多个失序(out of seq number不连续)报文段时是否将其暂时存储（以队列的形式），还是直接扔掉discard）。（当接收端主机内存容量较小时，建议关闭。）

*[这一功能是应对这样一种情况：因为TCP接收端可以接收其接收窗口内的所有序号的报文段，但只能对按序收到的数据中最高序号给出确认，如果已经收到了较好序号的报文段，但低序号的报文段可能因为丢失或滞留在网络中的某个节点，无法或晚到达，则此时先前接收到的较高序号的报文段可以选择丢弃，也可以选择存储。]*

实现：

/\*\*

\* TCP\_QUEUE\_OOSEQ==1: TCP will queue segments that arrive out of order.

\* Define to 0 if your device is low on memory.

\*/

#ifndef TCP\_QUEUE\_OOSEQ

#define TCP\_QUEUE\_OOSEQ (LWIP\_TCP)

#endif

###### (宏常量)TCP\_MSS(opt.h)(用于计算PBUF\_POOL类型的pbuf数据区首部空间大小)

功能：

为TCP最大报文段长度，但实际表示的是TCP报文段中的数据部分的最大长度。

实现：

#ifndef TCP\_MSS

#define TCP\_MSS 536

#endif

###### (宏常量)ETH\_PAD\_SIZE(opt.h)(用于计算PBUF\_POOL类型的pbuf数据区首部空间大小)

功能：

/\*\* ETH\_PAD\_SIZE: number of bytes added before the ethernet header to ensure

\* alignment of payload after that header. Since the header is 14 bytes long,

\* without this padding e.g. addresses in the IP header will not be aligned

\* on a 32-bit boundary, so setting this to 2 can speed up 32-bit-platforms.

\*/

实现：

#ifndef ETH\_PAD\_SIZE

#define ETH\_PAD\_SIZE 0

#endif

###### (宏常量)PBUF\_LINK\_HLEN(opt.h)(用于计算PBUF\_POOL类型的pbuf数据区首部空间大小)

功能：

/\*\*

\* PBUF\_LINK\_HLEN: the number of bytes that should be allocated for a

\* link level header. The default is 14, the standard value for

\* Ethernet.

\*/

实现：

#ifndef PBUF\_LINK\_HLEN

#define PBUF\_LINK\_HLEN (14 + ETH\_PAD\_SIZE)

#endif

###### (宏常量)PBUF\_POOL\_BUFSIZE(opt.h)(用于表示PBUF\_POOL内存池中pbuf数据区的大小)

功能：

完整pbuf内存池中，pbuf数据区中的有效数据部分的字节长度；

实现：

#ifndef PBUF\_POOL\_BUFSIZE

#define PBUF\_POOL\_BUFSIZE LWIP\_MEM\_ALIGN\_SIZE(TCP\_MSS+40+PBUF\_LINK\_HLEN)

#endif

###### (宏开关) LWIP\_CHECKSUM\_ON\_COPY(opt.h)(暂时不用，因为此功能没实现)

功能：

控制当从用户程序存储缓冲区复制数据到pbuf中时，是否进行CHECKSUM验证，默认是关闭；

实现：

#ifndef LWIP\_CHECKSUM\_ON\_COPY

#define LWIP\_CHECKSUM\_ON\_COPY 0

#endif

##### 其它模块函数

###### 自定义PBUF相关

###### Free ooseq PBUF\_POOL pbufs相关

无论有无操作系统：

(全局变量)tcp\_active\_pcbs读写词（tcp.h）

用于pbuf\_free\_ooseq函数。

(函数)tcp\_segs\_free(tcp.c)

用于pbuf\_free\_ooseq函数。

(结构体)tcp\_pcb的成员读写词：

用于pbuf\_free\_ooseq函数。

需求：

next成员的读；ooseq成员的写；

有操作系统

tcpip\_callback\_with\_block(tcpip.h)(有操作系统环境时)

###### CHECKSUM相关(pbuf.c,inet\_chksum.h)(暂不需要)

###### C库字符串操作(pbuf.c,string.h)

##### 错误码（err.h）

ERR\_OK ：正常；

ERR\_ARG ：参数错误；

ERR\_VAL

#### 内部宏/数据类型/数据结构

##### (ok!!!)具体功能相关(pbuf.h,pbuf.c)（暂不完成）

###### (ok!!!)自定义PBUF功能

宏开关LWIP\_SUPPORT\_CUSTOM\_PBUF

实现：

#define LWIP\_SUPPORT\_CUSTOM\_PBUF (IP\_FRAG && !IP\_FRAG\_USES\_STATIC\_BUF && !LWIP\_NETIF\_TX\_SINGLE\_PBUF)

语符：LWIP\_SUPPORT\_CUSTOM\_PBUF

语法：( ---t/f )

语义：返回pbuf管理器的custom pbuf功能开关的状态。

语用：

C表达式(IP\_FRAG && !IP\_FRAG\_USES\_STATIC\_BUF && !LWIP\_NETIF\_TX\_SINGLE\_PBUF)为真，则返回t，否则返回f；

（ASYM版不用实现）数据类型：函数指针类型pbuf\_free\_custom\_fn

功能：？？？

实现：

typedef void (\*pbuf\_free\_custom\_fn)(struct pbuf \*p);

结构体pbuf\_custom

struct pbuf\_custom {

struct pbuf pbuf; /\*\* The actual pbuf \*/

pbuf\_free\_custom\_fn custom\_free\_function;/\*\* 函数指针，This function is called when buf\_free deallocates this pbuf(\_custom) \*/

};

各个成员读写词：

在原pbuf结构体各个成员读写词基础上增加对函数指针成员custom\_free\_function的读写词；

pbuf\_custom\_free\_function@

语符：pbuf\_custom\_free\_function@

语法：( custompbuf\_addr---custompbuf->custom\_free\_function )

语义：读custom pbuf的函数指针成员的值；

语用：1、计算地址=custom pbuf起始地址+pbuf结构体大小；

2、取值；

pbuf\_custom\_free\_function!

语符：pbuf\_custom\_free\_function!

语法：(函数指针 custompbuf\_addr--- )

语义：修改custom pbuf的函数指针成员的值；

语用：1、计算地址=custom pbuf起始地址+pbuf结构体大小；

2、修改值；

###### Ok!!!free ooseq PBUF\_POOL pbufs相关

功能概述：

PBUF\_POOL内存池中的pbuf是用来接收数据包的，严格来说在上层处理完成pbuf中的数据之前，该pbuf是不能释放回到内存池中的。

如果此时内存池空了，而又有系的数据包到来则（lwip是什么样的，还不清楚）：

1）直接将接收的新包丢掉；

2）申请别的类型的pbuf来缓冲；

3） 释放一个正在使用的PBUF\_POOL类型的pbuf:在这种情况下，能被释放的只有一种pbuf，就是TCP协议中接收的那些out of order sequence number segments。

因为丢掉它们可以确保触发TCP的重传机制，所以最终仍能保证接收到全部数据。而TCP下层或同层的UDP等都是best-effort的。

之所以说只有这一种是因为：虽然在协议栈中即使下层的协议都是best-effort，但在上层肯定有一层是带有确认重传机制的，以确保传输的可靠性和完整性。当出现上面的问题时，如果采用释放一个正在使用的pbuf的方法，那么协议栈唯一能确定的是只有TCP的ooseq的报文段是可以丢掉的。但是前提是协议栈在传输层使用了TCP协议并且使用了一种机制以缓存哪些ooseq的报文段。两个条件必须同时具备，否则即使使用TCP协议但并没有缓存这些ooseq报文段（如接收终端内存太小），仍然不行。

这里pbuf缓冲管理器提供了第三种方法：前提是传输层使用TCP协议，且TCP协议提供了缓存ooseq的机制（就是将它们组织成队列的结构）。（另外在pbuf管理器中又额外设了一个这种功能的开关PBUF\_POOL\_FREE\_OOSEQ，所以开关也必须打开）

依赖：

一级宏开关：LWIP\_TCP：(opt.h)

功能：用来控制协议栈传输层是否使用TCP协议。

因为pbuf管理器的free ooseq功能释放的是TCP协议中ooseq队列中的ooseq。所以，这是第一级宏开关。

二级宏开关：TCP\_QUEUE\_OOSEQ：(opt.h)

功能：用来控制当使用TCP协议时，是否支持将queue ooseq的功能。

使用：通常是当主机内存容量较大时，开启此功能，否则关闭。

**默认开关状态**是与LWIP\_TCP开关状态相同，即当协议栈使用TCP协议时，queue ooseq的功能也自然开启，当不使用TCP协议时，此功能也自然关闭。

**单独配置开关状态**：如果使用TCP协议，但不想使用此功能则单独将其关闭：

#define TCP\_QUEUE\_OOSEQ 0

实现：

#ifndef TCP\_QUEUE\_OOSEQ

#define TCP\_QUEUE\_OOSEQ (LWIP\_TCP)

#endif

OK!!!(三级宏开关) PBUF\_POOL\_FREE\_OOSEQ

功能：

控制当PUF\_POOL类型的内存池空时，是否允许释放PBUF\_POOL内存池中那些正用于TCP失序报文段队列中的完整pbuf。默认是开启，表示允许。

实现：

#if LWIP\_TCP && TCP\_QUEUE\_OOSEQ

/\*\* Define this to 0 to prevent freeing ooseq pbufs when the PBUF\_POOL is empty \*/

#ifndef PBUF\_POOL\_FREE\_OOSEQ

#define PBUF\_POOL\_FREE\_OOSEQ 1

#endif /\* PBUF\_POOL\_FREE\_OOSEQ \*/

词汇设计：

语法：PBUF\_POOL\_FREE\_OOSEQ

语法：（ --- t/f ）t表示开，f表示关

语义：如果LWIP\_TCP 和 TCP\_QUEUE\_OOSEQ都为非零，则返回真，否则返回假

语用：

1、取LWIP\_TCP和TCP\_QUEUE\_OOSEQ；

2、分别做不等于0比较；

3、如果两次比较的结果都为真，则返回1，否则返回0；

(辅助全局变量) (volatile类型)pbuf\_free\_ooseq\_pending（ASYM版先定义但不使用）

功能：

用于表示是否需要执行pbuf缓冲管理器的free ooseq功能，值为非零则表示需要执行，为零则不需要。

当PBUF\_POOL内存池没有空闲pbuf时，则将本全局变量置1，表示需要到上层TCP协议中释放ooseq的报文段。

使用：

当PBUF\_POOL内存池没有空闲pbuf时，由pbuf\_pool\_is\_empty置1。

由pbuf\_free\_ooseq()释放ooseq并清0。

实现：

声明(pbuf.h)：extern volatile u8\_t pbuf\_free\_ooseq\_pending;

定义(pbuf.c)：volatile u8\_t pbuf\_free\_ooseq\_pending;

##### OK!!!宏常量

###### PBUF\_TRANSPORT\_HLEN(pbuf.h)

功能：

表示为传输层的首部字段预留的最小空间大小，值为0x14等于tcp协议首部字段的固定部分的长度。用于pbuf\_alloc中计算首部空间大小。

###### PBUF\_IP\_HLEN(pbuf.h)

功能：

表示为IP层的首部字段预留的最小空间大小，值为0x14等于IP协议首部字段的固定部分的长度。用于pbuf\_alloc中计算首部空间大小。

###### pbuf标志(怎么用???)

/\*\* indicates this packet's data should be immediately passed to the application \*/

#define **PBUF\_FLAG\_PUSH** 0x01U

/\*\* indicates this is a custom pbuf: pbuf\_free and pbuf\_header handle such a

a pbuf differently \*/

#define **PBUF\_FLAG\_IS\_CUSTOM** 0x02U

/\*\* indicates this pbuf is UDP multicast to be looped back \*/

#define **PBUF\_FLAG\_MCASTLOOP** 0x04U

/\*\* indicates this pbuf was received as link-level broadcast \*/

#define **PBUF\_FLAG\_LLBCAST** 0x08U

/\*\* indicates this pbuf was received as link-level multicast \*/

#define **PBUF\_FLAG\_LLMCAST** 0x10U

/\*\* indicates this pbuf includes a TCP FIN flag \*/

#define **PBUF\_FLAG\_TCP\_FIN**  0x20U

词汇设计：

参考代码《PBUF\_头文件.sfl》

##### OK!!!数据类型

###### pbuf\_layer(pbuf协议层类型，枚举数据类型pbuf.h)

功能：用于指示当前所处的协议层类型，如在申请pbuf时需要指明。

实现：

typedef enum {

PBUF\_TRANSPORT,

PBUF\_IP,

PBUF\_LINK,

PBUF\_RAW

} pbuf\_layer;

词汇设计：

参考代码《PBUF\_头文件.sfl》

###### pbuf\_type(pbuf类型，枚举类型，pbuf.h)

功能：

表示pbuf的类型，不同的类型具有不同的结构；

实现：

typedef enum {

PBUF\_RAM, /\* pbuf data is stored in RAM \*/

PBUF\_ROM, /\* pbuf data is stored in ROM \*/

PBUF\_REF, /\* pbuf comes from the pbuf pool \*/

PBUF\_POOL /\* pbuf payload refers to RAM \*/

} pbuf\_type;

词汇设计：

参考代码《PBUF\_头文件.sfl》

##### OK!!!数据结构

###### OK!!!pbuf结构体

定义：（参考“实现原理”）

操作：

pbuf\_sizeof (单个pbuf结构体大小)

功能：

计算单个pbuf结构的大小。（C语言版本中都是对齐后的大小）

实现：

#define SIZEOF\_STRUCT\_PBUF LWIP\_MEM\_ALIGN\_SIZE(sizeof(struct pbuf))

读各个成员

pbuf\_next@

语符：pbuf\_next@

语法：(pbuf\_addr--- pbuf->next )

语义：读指定pbuf结构体中的next成员的值；

语用：1、计算目的地址=pbuf\_addr+next的偏移量；

2、读值；

pbuf\_payload@

语符：pbuf\_payload@

语法：(pbuf\_addr--- pbuf->payload )

语义：读指定pbuf结构体中的payload成员的值；

语用：1、计算目的地址=pbuf\_addr+ payload的偏移量；

2、读值；

pbuf\_tot\_len@

语符：pbuf\_tot\_len@

语法：(pbuf\_addr--- pbuf->tot\_len )

语义：读指定pbuf结构体中的tot\_len成员的值；

语用：1、计算目的地址=pbuf\_addr+ tot\_len的偏移量；

2、读值；

pbuf\_len@

语符：pbuf\_len@

语法：(pbuf\_addr--- pbuf->len )

语义：读指定pbuf结构体中的len成员的值；

语用：1、计算目的地址=pbuf\_addr+len的偏移量；

2、读值；

pbuf\_type@

语符：pbuf\_type@

语法：(pbuf\_addr--- pbuf->type )

语义：读指定pbuf结构体中的type成员的值；

语用：1、计算目的地址=pbuf\_addr+type的偏移量；

2、读值；

pbuf\_flags@

语符：pbuf\_flags@

语法：(pbuf\_addr--- pbuf->flags )

语义：修改指定pbuf结构体中的flags成员的值；

语用：1、计算目的地址=pbuf\_addr+flags的偏移量；

2、读值；

pbuf\_ref@

语符：pbuf\_ref@

语法：(pbuf\_addr--- pbuf->ref )

语义：读指定pbuf结构体中的ref成员的值；

语用：1、计算目的地址=pbuf\_addr+ref的偏移量；

2、读值；

改各个成员

pbuf->next!

语符：pbuf\_next!

语法：( 值 pbuf\_addr--- )

语义：修改指定pbuf结构体中的next成员的值：pbuf\_addr->next = 值 ；

语用：1、计算目的地址=pbuf\_addr+next的偏移量；

2、修改值；

pbuf\_payload!

语符：pbuf\_payload!

语法：( 值 pbuf\_addr--- )

语义：修改指定pbuf结构体中的payload成员的值：pbuf\_addr->payload = 值 ；

语用：1、计算目的地址=pbuf\_addr+ payload的偏移量；

2、修改值；

pbuf\_tot\_len!

语符：pbuf\_tot\_len!

语法：( 值 pbuf\_addr--- )

语义：修改指定pbuf结构体中的tot\_len成员的值：pbuf\_addr->tot\_len = 值 ；

语用：1、计算目的地址=pbuf\_addr+ tot\_len的偏移量；

2、修改值；

pbuf\_len!

语符：pbuf\_len!

语法：( 值 pbuf\_addr--- )

语义：修改指定pbuf结构体中的len成员的值：pbuf\_addr->len = 值 ；

语用：1、计算目的地址=pbuf\_addr+len的偏移量；

2、修改值；

pbuf\_type!

语符：pbuf\_type!

语法：( 值 pbuf\_addr--- )

语义：修改指定pbuf结构体中的type成员的值：pbuf\_addr->type = 值 ；

语用：1、计算目的地址=pbuf\_addr+type的偏移量；

2、修改值；

pbuf\_flags!

语符：pbuf\_flags!

语法：( 值 pbuf\_addr--- )

语义：修改指定pbuf结构体中的flags成员的值：pbuf\_addr->flags = 值 ；

语用：1、计算目的地址=pbuf\_addr+flags的偏移量；

2、修改值；

pbuf\_ref!

语符：pbuf\_ref!

语法：( 值 pbuf\_addr--- )

语义：修改指定pbuf结构体中的ref成员的值：pbuf\_addr->ref = 值 ；

语用：1、计算目的地址=pbuf\_addr+ref的偏移量；

2、修改值；

#### 操作：(pbuf.c,pbuf.h)

##### (ok! +暂不)具体功能相关：

###### （ok!+暂不）自定义pbuf：

(ASYM版暂不实现，定为空函数)pbuf\_alloced\_custom

功能：

实现：

struct pbuf \*pbuf\_alloced\_custom(pbuf\_layer l, u16\_t length, pbuf\_type type,

struct pbuf\_custom \*p, void \*payload\_mem,

u16\_t payload\_mem\_len);

###### ok!!!free ooseq PBUF\_POOL pbufs：（实现无操作系统的，建议PBUF\_POOL内存池配置多点，以减少本功能的执行）

基础词：

OK!!!(宏函数)PBUF\_POOL\_IS\_EMPTY(只实现无操作系统环境的)

PBUF\_POOL free ooseq功能关闭时，函数不存在；

PBUF\_POOL free ooseq功能打开时，则是pbuf\_pool\_is\_empty

功能:宏扩展成pbuf\_pool\_is\_empty函数，由其完成实际功能。

词汇设计：

语符：PBUF\_POOL\_IS\_EMPTY

语法：（ --- ）

语义：如果PBUF\_POOL\_FREE\_OOSEQ非零，则调用pbuf\_pool\_is\_empty，将全局变量

pbuf\_free\_ooseq\_pending置1；否则什么都不做；

语用：

1、调用《PBUF\_POOL\_FREE\_OOSEQ》返回本功能开关状态

2、如果开，则调用《pbuf\_pool\_is\_empty》，否则什么都不做；

(模块内部自定义函数)pbuf\_pool\_is\_empty

功能：将pbuf\_free\_ooseq\_pending置1。根据有无PBUF\_POOL\_FREE\_OOSEQ\_QUEUE\_CALL，则行为也不同。

输入：无；

输出：

基本流程：（分有无PBUF\_POOL\_FREE\_OOSEQ\_QUEUE\_CALL）：

无操作系统：

进入临界代码区，然后将pbuf\_free\_ooseq\_pending置1，出临界代码区；

有操作系统：（需要调用PBUF\_POOL\_FREE\_OOSEQ\_QUEUE\_CALL）

1、进入临界代码区；

2、将pbuf\_free\_ooseq\_pending的原值取出来，再将pbuf\_free\_ooseq\_pending置1；

3、判断pbuf\_free\_ooseq\_pending原值是否时零，是则执行

PBUF\_POOL\_FREE\_OOSEQ\_QUEUE\_CALL()，不是则直接退出程序。

词汇设计：（无操作系统）

语符：pbuf\_pool\_is\_empty

语法：( --- )

语义：进入临界代码区，然后将pbuf\_free\_ooseq\_pending置1，出临界代码区；

语用：进入临界代码区，然后将pbuf\_free\_ooseq\_pending置1，出临界代码区；

(函数)pbuf\_free\_ooseq (带操作系统则变成局部函数)

功能：

完成实质性的TCP协议中ooseq pbufs的释放：在所有当前活跃tcp连接的tcp\_pcb构成队列中从第一个开始，找到第一个ooseq队列中存在ooseq的TCP连接，然后将所有的ooseq都释放。

/\*\*

\* Attempt to reclaim some memory from queued out-of-sequence TCP segments

\* if we run out of pool pbufs. It's better to give priority to new packets

\* if we're running out.

\*

\*注意！！（有操作系统）： This must be done in the correct thread context therefore this function can only be used with NO\_SYS=0 and through tcpip\_callback.

\*/

输入：无

输出：无

基本流程：

1、进临界代码区；

2、将pbuf\_free\_ooseq\_pending清0；

3、出临界代码区；

4、不定循环。在所有当前活跃tcp连接的tcp\_pcb构成的队列中从第一个开始，找到第一个ooseq队列中存在ooseq的TCP连接，然后将所有的ooseq都释放。

词汇设计：

语符：pbuf\_free\_ooseq

语法：（ --- ）

语义：在所有当前活跃tcp连接的tcp\_pcb队列中从第一个开始，找到第一个ooseq队列中存在ooseq的TCP连接，然后将所有的ooseq都释放。

语用：

1、进临界代码区；

2、将pbuf\_free\_ooseq\_pending清0；

3、出临界代码区；

4、不定循环。在所有当前活跃tcp连接的tcp\_pcb构成的队列中从第一个开始，找到第一个ooseq队列中存在ooseq的TCP连接，然后将所有的ooseq都释放。

有操作系统环境

实现：

当pbuf\_pool\_is\_empty时，会调用PBUF\_POOL\_FREE\_OOSEQ\_QUEUE\_CALL，

由宏函数通过pbuf\_free\_ooseq\_callback调用pbuf\_free\_ooseq实现。

(宏函数)PBUF\_POOL\_FREE\_OOSEQ\_QUEUE\_CALL

pbuf\_free\_ooseq\_callback(带操作系统则变成局部函数)

无操作系统环境

(对外接口)PBUF\_CHECK\_FREE\_OOSEQ(宏函数)(有疑点)

功能及使用：

在无操作系统环境下，使用pbuf缓冲管理器的PBUF\_POOL内存池free ooseq功能的方法是，在**主函数（这是疑点）**中周期性的调用本函数，以检查是否有ooseq pbufs需要释放（判断辅助全局变量pbuf\_free\_ooseq\_pending是否为真），如果有则调用子函数pbuf\_free\_ooseq()。

/\*\* When not using sys\_check\_timeouts(), call PBUF\_CHECK\_FREE\_OOSEQ()

at regular intervals from main level to check if ooseq pbufs need to be

freed! \*/

实现：

#define PBUF\_CHECK\_FREE\_OOSEQ() do { if(pbuf\_free\_ooseq\_pending) { \

/\* pbuf\_alloc() reported PBUF\_POOL to be empty -> try to free some \

ooseq queued pbufs now \*/ \

pbuf\_free\_ooseq(); }}while(0)

词汇设计：

语符：PBUF\_CHECK\_FREE\_OOSEQ

语法：（ --- ）

语义：检查是否需要free ooseq，需要则调用《pbuf\_free\_ooseq》释放；

语用：

如果pbuf\_free\_ooseq\_pending非零，则调用《pbuf\_free\_ooseq》；否则直接退出程序；

##### OK!!!pbuf\_init()(当前版本函数为空，ASYM版则不实现)

语符：pbuf\_init

语法：（---）

语义：初始化pbuf管理器；

语用：lwip1.4.1及以前版本都不支持，函数被定义为空；

##### OK!!!pbuf\_alloc()(申请指定类型的pbuf)

###### 设计维护用：

语符：pbuf\_alloc

语法：( pbuflayer length pbuftype---pbuf\_addr/NULL )

pbuflayer：申请时所处的协议层类型；

length：pbuf数据区有效数据的字节长度，16位无符号数;

pbuftyep：pbuf类型

语义：

分配一个指定类型的pbuf结构（如果是PBUF\_POOL类型，则返回的可能是一个pbuf链）

Allocates a pbuf of the given type (possibly a chain for PBUF\_POOL type)

语用：

1、根据输入的协议层类型计算offset（首部空间大小），如果协议类型无效则返回NULL退出程序：

PBUF\_TRANSPORT：offset = PBUF\_LINK\_HLEN + PBUF\_IP\_HLEN + PBUF\_TRANSPORT\_HLEN

PBUF\_IP ：offset = PBUF\_LINK\_HLEN + PBUF\_IP\_HLEN；

PBUF\_LINK ：offset = PBUF\_LINK\_HLEN；

PBUF\_RAW ：offset = 0；

2、根据输入的pbuf类型申请相应的pbuf：

2.1 申请PBUF\_POOL类型：

2.1.1 调用memp\_malloc()申请一个pbuf，判断返回的pbuf结构体指针p是否是空，是则调用PBUF\_POOL\_IS\_EMPTY()在free ooseq功能开启情况下，释放一个TCP的失序报文段，返回NULL，退出程序；否则继续；

2.1.2 初始化返回的pbuf头中的各个成员：

p->type = type;

p->next = NULL;

p->payload = p+(SIZEOF\_STRUCT\_PBUF +offset );

p->tot\_len = length;

p->len =LWIP\_MIN(length, PBUF\_POOL\_BUFSIZE - offset);

p->ref = 1;

p->flags = 0;

2.1.3 循环继续申请PBUF\_POOL类型的pbuf，直到所有pbuf的len之和能满足length。并将申请到的所有pbuf初始化以及链成链表，如果中间有任何一个pbuf申请失败，则将先前已申请的都释放，退出程序；

2.1.3.1 判断是否需要继续申请PBUF\_POOL类型的pbuf：计算length - p->len，如果结果大于0，则表示需要继续申请；否则表示申请完成，退出程序；

2.1.3.2调用memp\_malloc()申请下一个pbuf，判断返回的pbuf结构体指针q是否是空，是则PBUF\_POOL\_IS\_EMPTY(),调用pbuf\_free(p)，将先前的返回的pbuf构成的链释放，返回NULL，退出程序；

2.1.3.3 本次循环申请pbuf成功，则初始化pbuf头中的各个成员并与先前的链成链表：

q->type = type;

q->next = NULL;

q->payload = q+(SIZEOF\_STRUCT\_PBUF +offset );(offset==0)

q->tot\_len = rem\_len = length - 已申请pbuf的len之和;

q->len =LWIP\_MIN(rem\_len, PBUF\_POOL\_BUFSIZE-offset );(offset==0)

q->ref = 1;

q->flags = 0;

前一个pbuf->next=q;

2.2 申请PBUF\_RAM类型：

2.2.1 调用mem\_malloc(SIZEOF\_STRUCT\_PBUF + offset+ length)，从动态内存堆中申请空闲内存p。申请失败，则返回NULL，退出程序；否则继续；

2.2.2 将空闲内存的前几个字节作为pbuf的头，初始化各个成员：

p->payload = p + SIZEOF\_STRUCT\_PBUF + offset;

p->len = p->tot\_len = length;

p->next = NULL;

p->type = type;

p->ref = 1;

p->flags = 0;

2.3 申请PBUF\_ROM或PBUF\_REF类型：

2.3.1 调用memp\_malloc(MEMP\_PBUF)，从MEMP\_PBUF类型的内存池中申请一个pbuf头结构，申请失败则返回NULL，退出程序；否则继续；

2.3.2 初始化返回pbuf头结构中的各个成员：

p->payload =NULL;

p->len = p->tot\_len = length;

p->next = NULL;

p->type = type;

p->ref = 1;

p->flags = 0;

2.4 如果哪种类型都不是，则返回NULL，退出程序；

###### 用户用：

语符：pbuf\_alloc

语法：( pbuflayer length pbuftype---pbuf\_addr/NULL )

pbuflayer：申请时所处的协议层类型；

length：pbuf数据区有效数据的字节长度，16位无符号数；

pbuftype：pbuf类型；

语义：为用户分配一个指定类型的pbuf结构（如果是PBUF\_POOL类型，则返回的可能是一个pbuf链）。成功时，返回pbuf结构的首地址，失败则返回NULL。

失败情况：参数错误；没有可用的pbuf结构；

语用：

1 根据协议层类型计算offset；

2 根据pbuf类型分配相应的pbuf；

3 如果参数错误，没有空闲的pbuf则返回NULL；

###### 子词：

Pbuf\_计算offset

语符：Pbuf\_计算offset

语法：(pbuflayer ---offset t)或(pbuflayer ---offset t)

语义：根据输入的pbuf协议层类型，计算pbuf的首部空间的大小offset，，如果协议类型无效则返回错误标识。

语用：

1、pbuflayer== PBUF\_TRANSPORT：

offset = PBUF\_LINK\_HLEN + PBUF\_IP\_HLEN + PBUF\_TRANSPORT\_HLEN;

2、pbuflayer== PBUF\_IP：

offset = PBUF\_LINK\_HLEN + PBUF\_IP\_HLEN;

3、pbuflayer== PBUF\_LINK：

offset = PBUF\_LINK\_HLEN;

4、pbuflayer== PBUF\_RAW：

offset = 0;

公共子词：

初始化pbuf头

语符：初始化pbuf头

语法：(length offset type pbuf\_addr--- pbuf\_addr )

语义：pbuf\_malloc申请pbuf时，调用本次初始化申请到的pbuf的pbuf头中的各成员值；

(length表示从当前pbuf开始的后面所有pbuf（如果有）的有效数据区总长度)

语用：

1 pbuf\_addr->type = type;

2 pbuf\_addr->next = NULL;

3 pbuf\_addr->tot\_len = length;

4 pbuf\_addr->ref = 1;

5 pbuf\_addr->flags = 0;

6 如果是PBUF\_ROM或PBUF\_REF或PBUF\_RAM类型：则pbuf\_addr->len = length;

否则是PBUF\_POOL类型：则pbuf\_addr->len = LWIP\_MIN(length, PBUF\_POOL\_BUFSIZE -offset);

7 如果是PBUF\_RAM或PBUF\_POOL类型：则pbuf\_addr->payload = pbuf\_addr + SIZEOF\_STRUCT\_PBUF + offset;

否则是PBUF\_ROM或PBUF\_REF类型：则pbuf\_addr->payload = NULL；

分配PBUF\_POOL

语符：分配PBUF\_POOL

语法：(length offset---pbuf\_addr/NULL )

语义：分配数据区有效数据长度满足要求的PBUF\_POOL类型的pbuf（或pbuf链）

语用：

1 调用memp\_malloc()申请一个pbuf，判断返回的pbuf结构体指针p是否是空，是则调

用PBUF\_POOL\_IS\_EMPTY()在free ooseq功能开启情况下，释放一个TCP的失序报文段，返回NULL，退出程序；否则继续；

2 初始化返回的pbuf头中的各个成员：

p->type = type;

p->next = NULL;

p->payload = p+(SIZEOF\_STRUCT\_PBUF +offset );

p->tot\_len = length;

p->len =LWIP\_MIN(length, PBUF\_POOL\_BUFSIZE - offset);

p->ref = 1;

p->flags = 0;

3 调用《更多PBUF\_POOL》:循环继续申请PBUF\_POOL类型的pbuf，直到所有pbuf的len之和能满足length。并将申请到的所有pbuf初始化以及链成链表;

4 判断申请是否成功，不是则将先前已申请的都释放，退出程序；

初始化PBUF\_POOL

语符：初始化PBUF\_POOL

语法：( length offset pbuf\_addr---)

语义：初始化pbuf\_addr指向的PBUF\_POOL类型的pbuf头中的成员值；

语用：

1 pbuf\_addr->type = type;

2 pbuf\_addr->next = NULL;

3 pbuf\_addr->tot\_len = length;

4 pbuf\_addr->ref = 1;

5 pbuf\_addr->flags = 0;

6 pbuf\_addr->len = LWIP\_MIN(length, PBUF\_POOL\_BUFSIZE -offset);

7 pbuf\_addr->payload = pbuf\_addr + SIZEOF\_STRUCT\_PBUF + offset;

更多PBUF\_POOL

语符：更多PBUF\_POOL

语法：( r rem\_len---t/f )

r =p; rem\_len=length - p->len；p是申请的第一个PBUF\_POOL类型pbuf的首地址；

t表示成功，用0表示，f表示失败，用非0值表示；

语义：循环继续申请PBUF\_POOL类型的pbuf，直到所有pbuf的len之和能满足length。并

将申请到的所有pbuf初始化以及链成链表，如果中间有任何一个pbuf申请失败，则

将先前已申请的都释放。返回t/f用于表示执行成功还是失败。

语用：

循环：

1 判断是否需要继续申请PBUF\_POOL类型的pbuf：如果rem\_len大于0，则表示需要继续申请；否则表示申请完成，退出程序；

2 调用memp\_malloc()申请下一个pbuf，判断返回的pbuf结构体指针q是否是空，是则PBUF\_POOL\_IS\_EMPTY(),调用pbuf\_free(p)，将先前的返回的pbuf构成的链表释放，退出程序；

3 本次循环申请pbuf成功，则初始化pbuf头中的各个成员并与先前的链成链表：

q->type = type;

q->next = NULL;

q->payload = q+(SIZEOF\_STRUCT\_PBUF +offset );(offset==0)

q->tot\_len = rem\_len = length – 已申请pbuf的len之和;

q->len =LWIP\_MIN(rem\_len, PBUF\_POOL\_BUFSIZE-offset );(offset==0)

q->ref = 1;

q->flags = 0;

前一个pbuf->next=q;

分配PBUF\_RAM

语符：分配PBUF\_RAM

语法：(length offset---pbuf\_addr/NULL )

语义：分配数据区有效数据长度满足要求的PBUF\_RAM类型的pbuf。

语用：

1 调用mem\_malloc(SIZEOF\_STRUCT\_PBUF + offset+ length)，从动态内存堆中申请空闲内存p。

申请失败，则返回NULL，退出程序；否则继续；

2 将空闲内存的前几个字节作为pbuf的头，初始化各个成员：

p->payload = p + SIZEOF\_STRUCT\_PBUF + offset;

p->len = p->tot\_len = length;

p->next = NULL;

p->type = type;

p->ref = 1;

p->flags = 0;

初始化PBUF\_RAM

语符：初始化PBUF\_RAM

语法：( length offset pbuf\_addr---)

语义：初始化pbuf\_addr指向的PBUF\_RAM类型的pbuf头中的成员值；

语用：

p->payload = p + SIZEOF\_STRUCT\_PBUF + offset;

p->len = p->tot\_len = length;

p->next = NULL;

p->type = PBUF\_RAM;

p->ref = 1;

p->flags = 0;

分配PBUF\_ROM

语符：分配PBUF\_ROM

语法：( length offset---pbuf\_addr/NULL )

语义：分配一个PBUF\_ROM类型的pbuf。

语用：

1、装PBUF\_ROM类型名；

2、调用《分配PBUF\_ROMorREF》；

子词：分配PBUF\_ROMorREF

语符：分配PBUF\_ROMorREF

语法：( length offset pbuftype---pbuf\_addr/NULL )

语义：根据pbuftype，分配PBUF\_ROM或PBUF\_REF类型的pbuf；

语用：

1 调用memp\_malloc(MEMP\_PBUF)，从MEMP\_PBUF类型的内存池中申请一个pbuf头结构，申请失败则返回NULL，退出程序；否则继续；

2 初始化返回pbuf头结构中的各个成员：

p->payload =NULL;

p->len = p->tot\_len = length;

p->next = NULL;

p->type = type;

p->ref = 1;

p->flags = 0;

分配PBUF\_REF

语符：分配PBUF\_REF

语法：( length offset---pbuf\_addr/NULL )

语义：分配一个PBUF\_REF类型的pbuf。

语用：

1、装PBUF\_REF类型名；

2、调用《分配PBUF\_ROMorREF》；

##### OK!!!pbuf\_free() (包含custom pbuf功能代码但不执行任何功能) (释放pbuf或pbuf链或pbuf队列)

原理：

pbuf释放的前提：

pbuf头中的ref成员表示该pbuf被引用的次数，申请pbuf时初始值是1，代表被引用一次；释放pbuf时，函数会首先将pbuf头的ref成员减1，只有当pbuf的ref成员等于0时，该pbuf所占的内存才会被释放。

基本过程：

某个pbuf链表的首节点删除成功后，该pbuf链的第二个节点自然地成为了首节点，此时，该节点的ref值可能变为0（该节点没有在其它地方被引用了，而是只被已删除的首节点引用过），在这种情况下，该节点也会被删除，因为lwip认为它和第一个节点一起存储同一个数据包。当第二个节点被删除后，函数又会查看第三个节点是否满足删除条件.....就这样一直删下去。当然，如果首节点删除后，第二个节点的ref值大于0，表示该节点还在其它地方被引用，不能再被删除，删除工作至此结束。

###### 设计维护用：

语符：pbuf\_free

语法：( pbuf\_addr---counter )

pbuf\_addr：表示指定的pbuf的首地址；counter：表示返回的被删除的pbuf个数，8

位无符号数；

语义： 释放指定的pbuf，解除一次对它的引用。如果指定pbuf还被其他对象引用，则pbuf\_free只是将ref减1，否则将真正删除该pbuf。如果该pbuf被删除，则也会将其后面（如果有）满足ref=1的所有连续的pbuf删除，直到某个pbuf的ref>1时，停止删除，返回实际删除的pbuf的个数。

删除某个pbuf时，根据pbuf类型的不同，调用不同的内存释放函数进行删除。PBUF\_POOL类型来自内存池MEMP\_PBUF\_POOL,PBUF\_ROM和PBUF\_REF类型来自内存堆，它们都需要memp\_free函数来释放相应内存空间；PBUF\_RAM类型来自内存堆，所以需要通过mem\_free函数删除；

**(PBUF\_REF和PBUF\_ROM类型的pbuf的数据区部分的内存由上层自行释放)**

语用：

1、判断输入的地址p是否等于0，是返回0，退出程序，否则继续；

2、释放的pbuf个数计数器count清零；

3、循环，从指定的pbuf开始将pbuf链中满足ref=0的所有pbuf删除；

3.1 进入临界代码区；

3.2临时变量ref = --(p->ref)；

3.3 出临界代码区；

3.4 如果临时变量ref 不等于0，结束循环，返回count，退出程序；

3.5 如果临时变量ref 等于0：

3.5.1 判断当前pbuf的flags中PBUF\_FLAG\_IS\_CUSTOM位是否有效，是则调用专门的函数custom\_free\_function()将其释放；（*如果不支持custom pbuf则实际不执行*）

3.5.2 根据pbuf的类型，调用各自专用的函数将其释放：

3.5.2.1 PBUF\_POOL类型：调用memp\_free(MEMP\_PBUF\_POOL,当前pbuf地址);

3.5.2.2 PBUF\_ROM或PBUF\_REF类型：memp\_free(MEMP\_PBUF,当前pbuf地址);

3.5.2.3 其它类型：mem\_free(当前pbuf地址);

3.5.3 计数器count加1；

3.5.4 进行下一循环；

###### 用户用：

语符：pbuf\_free

语法：( pbuf\_addr---counter )

pbuf\_addr：表示指定的pbuf的首地址；counter：表示返回的被真正删除的pbuf个

数，8位无符号数；

语义： 释放指定的pbuf，解除一次对它的引用。如果指定pbuf还被其他对象引用，则pbuf\_free只是将ref减1，否则将真正删除该pbuf。如果该pbuf被删除，则也会将其后面（如果有）满足ref=1的所有连续的pbuf删除，直到某个pbuf的ref>1时，停止删除，返回实际删除的pbuf的个数。

语用：

返回值counter == 0 的情况：

1 ：输入参数为NULL；（异常）

2 ：已解除引用，但指定pbuf还被其他节点引用，即ref>1；（正常）

###### 子词：

ref=--(p->ref)

语符：ref=--(p->ref)

语法：( p---临时变量ref )

语义：将指定puf的ref成员值减1，并将减1后的结果作为临时变量ref的初始值；

语用：

1、进入临界代码区；

2、将指定puf的ref成员值减1；

3、出临界代码；

释放custompbuf

语符：释放custompbuf

语法：( pbuf\_addr--- )

语义：判断指定pbuf的flags中PBUF\_FLAG\_IS\_CUSTOM位是否有效，是则调用专门的函数custom\_free\_function()将其释放；（如果不支持custom pbuf则实际不执行）

语用：

1、判断LWIP\_SUPPORT\_CUSTOM\_PBUF功能开关是否开启，是则继续执行，否则退出程序；

2、取pbuf的flags成员值p->flags；

3、PBUF\_FLAG\_IS\_CUSTOM位是否有效：(p->flags & PBUF\_FLAG\_IS\_CUSTOM) != 0

4、有效，则表明指定pbuf是custom pbuf，则执行函数指针指向的函数；

分类freepbuf

语符：分类freepbuf

语法：( pbuf\_addr--- )

语义：根据pbuf的类型不同，调用特定的函数将其释放

语用：PBUF\_POOL类型来自MEMP\_PBUF\_POOL内存池,PBUF\_ROM和PBUF\_REF类型来自内存堆，它们都需要memp\_free函数来释放相应内存空间；PBUF\_RAM类型来自内存堆，所以需要通过mem\_free函数删除

##### OK!!!pbuf\_clen()（计算pbuf链中pbuf的个数）

功能：指定起始pbuf，计算从其开始到最后一个pbuf的pbuf链中pbuf的个数。

输入：pbuf链中第一个pbuf的地址；

输出：pbuf个数；

基本流程：

循环直到某个pbuf的next指针为NULL，若每次循环p->next 不等于NULL，则计数器len加1。

###### 设计维护用和用户用：

语符：pbuf\_clen

语法：( pbuf\_addr---len )

len是8位无符号数；

语义：指定起始pbuf，计算从其开始到最后一个pbuf的pbuf链中pbuf的个数；

语用：循环直到某个pbuf的next指针为NULL，若每次循环p->next 不等于NULL，则计数器len加1。

##### Ok! !!!pbuf\_header()(调整pbuf->payload指针，将header隐藏到有效数据区或从中显示出来)

功能：调整指定pbuf中的payload指针，以在数据区的有效数据区显示或隐藏首部空间，当显示时，就可以通过payload指针读写首部空间。*（翻译自源码注释）*

显示首部空间：将payload指针从数据区的有效数据部分向首部空间方向偏移（偏移

量为正）；

隐藏首部空间：将payload指针从首部空间向有效数据部分方向偏移（则偏移量为负）；

输入：

指定pbuf首地址p；

有符号偏移量header\_size\_increment；

输出：

成功或失败标识（成功标识用0表示，失败标识用非0值表示）；

基本流程：

1、参数检查：p == NULL或header\_size\_increment == 0，则什么都不做，返回成功标识，退出程序；

2、如果是隐藏首部字段，则从首部空间向有效数据区方向偏移时，不能超出有效数据区的范围，否则返回失败标识，退出程序：（方法：header\_size\_increment小于0且其绝对值大于p->len则什么都不做）；

3、根据pbuf类型判断并调整：

3.1 如果pbuf类型是PBUF\_RAM或PBUF\_POOL：

3.1.1显示首部字段调整时边界检查：判断调整后的payload是否指向pbuf头范围内，是则超界，返回失败标识，退出程序；

3.1.2没有超界，则调整payload指针并修改len和tot\_len，返回成功标识，退出程序：

p->payload = p->payload - header\_size\_increment;

p->len += header\_size\_increment;

p->tot\_len += header\_size\_increment;

3.2 如果pbuf类型是PBUF\_REF或PBUF\_ROOM：

3.2.1 只能是从首部空间向有效数据部分方向偏移且不能超出有效数据部分范围，不是则返回失败标识，退出程序；

（方法：表达式(header\_size\_increment < 0) && (abs(header\_size\_increment)<= p->len)必须为真）;

3.2.2 是则，调整payload指针并修改len和tot\_len，返回成功标识，退出程序：

p->payload = p->payload - header\_size\_increment；

p->len += header\_size\_increment;

p->tot\_len += header\_size\_increment;

3.3 pbuf类型错误，则返回失败标识，退出程序；

###### 设计维护用：

语符：pbuf\_header

语法：( pbuf\_addr header\_size\_increment--- t/f )

header\_size\_increment：**原码方法表示的**16位有符号数；

成功标识用0表示，失败标识用非0值表示；

语义：调整指定pbuf中的payload指针，以在数据区的有效数据区显示或隐藏首部字段，当显示时，就可以通过payload指针读写首部字段。*（翻译自源码注释）*

在有效数据区显示首部字段：将payload指针从数据区的有效数据部分向首部空间方向调整（偏移量为正）；

在有效数据区隐藏首部字段：将payload指针从首部空间向有效数据部分方向调整（则偏移量为负）；

成功的情况t：

1、什么都没做：pbuf\_addr == NULL或header\_size\_increment == 0

2、完成调整；

失败的情况f：

1、无论pbuf类型从首部空间向有效数据区方向调整时，超出有效数据区的范围（即header\_size\_increment小于0且其绝对值大于p->len）；

2、若是PBUF\_RAM或PBUF\_POOL类型pbuf：调整后payload指向pbuf头范围内；

3、若是PBUF\_REF或PBUF\_ROOM类型pbuf：不是从首部空间向有效数据区方向偏移或调整后的payload超出有效数据区的范围；

语用：

1、参数检查：p == NULL或header\_size\_increment == 0，则什么都不做，返回成功标识，退出程序；

2、调用《HideHeaders判断》：如果是隐藏首部字段，即从首部空间向有效数据区方向偏移时，则不能超出有效数据区的范围，否则返回失败标识，退出程序：（方法：header\_size\_increment小于0且其绝对值大于p->len则什么都不做）；

3、调用《分类调整payload》，根据pbuf类型判断并调整，返回成功或失败标识；

###### 用户用：

语符：pbuf\_header

语法：( pbuf\_addr header\_size\_increment--- t/f )

header\_size\_increment：16位有符号数；

成功标识用0表示，失败标识用非0值表示；

语义：调整指定pbuf中的payload指针，以在数据区的有效数据区显示或隐藏首部字段，当显示时，就可以通过payload指针读写首部字段。*（翻译自源码注释）*

在有效数据区显示首部字段：将payload指针从数据区的有效数据区向首部空间方

向调整（偏移量为正）；

在有效数据区隐藏首部字段：将payload指针从首部空间向有效数据区方向调整（则偏移量为负）；

语用：

成功的情况t：

1、什么都没做：p == NULL或header\_size\_increment == 0

2、完成调整；

失败的情况f：

1、无论pbuf类型从首部空间向有效数据区方向调整时，超出有效数据区的范围（即header\_size\_increment小于0且其绝对值大于p->len）；

2、若是PBUF\_RAM或PBUF\_POOL类型pbuf：调整后payload指向pbuf头范围内；

3、若是PBUF\_REF或PBUF\_ROOM类型pbuf：不是从首部空间向有效数据部分方向调整或调整后的payload超出有效数据部分的范围；

###### 子词：

HideHeaders判断

语符：HideHeaders判断

语法：(pbuf\_addr header\_size\_increment---t/f )

语义：如果是在有效数据区隐藏首部字段但header\_size\_increment的绝对值大于pbuf\_addr->len，则返回参数无效标识t,否则正确标识f；

语用：如果C表达式为真，则返回参数无效标识t，否则返回有效标识f：

(header\_size\_increment < 0) && (abs(header\_size\_increment) > p->len)；

分类调整payload

语符：分类调整payload

语法：( pbuf\_addr header\_size\_increment---t/f )

语义：根据pbuf类型判断并调整payload指针，

语用：

1 如果pbuf类型是PBUF\_RAM或PBUF\_POOL：调用《调pbufRAMorPOOL》

调整payload指针；

2 如果pbuf类型是PBUF\_REF或PBUF\_ROOM：调用《调pbufREForROM》

调整payload指针；

3 pbuf类型错误，则返回失败标识，退出程序；

调pbufRAMorPOOL

语符：调pbufRAMorPOOL

语法：(pbuf\_addr header\_size\_increment---t/f )

语义：调整PBUF\_RAM或PBUF\_POOL类型的pbuf的payload指针；

语用：

1显示首部字段调整时边界检查：判断调整后的payload是否指向pbuf头范围内，是则超界，返回失败标识，退出程序；

2 没有超界，则调用《调整payload》，调整payload指针并修改len和tot\_len：

调pbufREForROM

语符：调pbufREForROM

语法：(pbuf\_addr header\_size\_increment---t/f )

语义：调整PBUF\_REF或PBUF\_ROM类型的pbuf的payload指针；

语用：

1 只能是从首部空间向有效数据部分方向偏移且不能超出有效数据部分范围，不是则返回失

败标识，退出程序；

（方法：表达式(header\_size\_increment < 0) && (abs(header\_size\_increment)<= p->len)必须为真）;

2 是，则调用《调整payload》，调整payload指针并修改len和tot\_len：

调整payload

语符：调整payload

语法：( pbuf\_addr header\_size\_increment--- )

语义：调整payload指针并修改len和tot\_len

语用：

p->payload = p->payload - header\_size\_increment；

p->len += header\_size\_increment;

p->tot\_len += header\_size\_increment;

##### ok!!!pbuf\_ref（pbuf的引用次数加1）

功能：将指定pbuf的ref成员加1；

输入：pbuf首地址p；

输出：无；

基本流程：

1、判断输入的地址是否无效：p ==NULL则无效，退出程序；

2、进去临界代码区；

3、p->ref加1；

4、出临界代码区；

语符：pbuf\_ref

语法：( pbuf\_addr--- )

语义：将指定pbuf的ref成员加1；

语用：

1、判断输入的地址是否无效：p ==NULL则无效，退出程序；

2、进去临界代码区；

3、p->ref加1；

4、出临界代码区；

##### OK!!!pbuf\_cat()(将两个pbuf或pbuf链，链成单链)

功能：指定两个pbuf，从它们开始的后面所有pbuf（如果有）构成两个pbuf链head和tail，将两个链成单链。（只有当上层节点保证后一个pbuf链tail不再单独使用时，才能通过pbuf\_cat()将head和tail链成单链，否则使用pbuf\_chain()）。

输入：

前一个pbuf（链）的首地址h；后一个pbuf（链）的首地址t；

输出：

无；

基本流程：

1、参数检查：两个指针参数均无效，则直接退出程序；(LWIP\_ERROR)

2、循环，将前一个pbuf链h中所有pbuf的tot\_len成员加上后一个pbuf链t的第一个pbuf的tot\_len的值；

3、令前一个pbuf链h的最后一个pbuf的next指向后一个pbuf链t的第一个pbuf；

###### 设计维护和用户用：

语符：pbuf\_cat

语法：( head\_pbufaddr tail\_pbufaddr--- )

语义：指定两个pbuf，从它们开始的后面所有pbuf（如果有）构成两个pbuf链head和tail，将两个链成单链。（只有当上层节点保证后一个pbuf链tail不再单独使用时，才能通过pbuf\_cat()将head和tail链成单链，否则使用pbuf\_chain()）

语用：（tail的第一个pbuf的ref没有加1）

1、参数检查：两个指针参数均无效，则直接退出程序；

2、循环，将前一个pbuf链h中所有pbuf的tot\_len成员加上后一个pbuf链t的第一个pbuf的tot\_len的值；

3、令前一个pbuf链h的最后一个pbuf的next指向后一个pbuf链t的第一个pbuf；

##### OK!!!pbuf\_chain()(将两个pbuf或pbuf链，链成单链)

功能：指定两个pbuf，从它们开始的后面所有pbuf（如果有）构成两个pbuf链head和tail，将两个链成单链。与pbuf\_cat()不同的是上层节点仍可单独使用后一个pbuf（或pbuf链）tail，并且在不再使用时，必须调用一次pbuf\_free(t)。

输入：

前一个pbuf（链）的首地址h；后一个pbuf（链）的首地址t；

输出：

无；

注意：

1）两个pbuf（链）必须是属于同一数据包。构造数据包队列时不建议调用此函数。

2）chain完之后，当调用者不再单独使用后面的pbuf（链）t时，需要调用pbuf\_free(t)将其释放；

（也就是说，chain完之后，调用pbuf\_free(新完整链的首地址)，并不会将t释放掉。）

基本流程：

1、调用pbuf\_cat(h,t)；

2、调用pbuf\_ref(t);

###### 设计维护和用户用：

语符：pbuf\_chain

语法：( head\_pbufaddr tail\_pbufaddr--- )

语义：指定两个pbuf，从它们开始的后面所有pbuf（如果有）构成两个pbuf链head和tail，将两个链成单链。与pbuf\_cat()不同的是上层节点仍可单独使用后一个pbuf（或pbuf链）tail，并且在不再使用时，必须调用一次pbuf\_free(t)。

语用：（tail的第一个pbuf的ref加1）

1、调用pbuf\_cat(h,t)；

2、调用pbuf\_ref(t);

##### Ok!!!pbuf\_dechain()（代码完成，但原理有不懂的地方）

功能：指定一个pbuf，将它与它后面的pbuf断开，返回后面第一个pbuf的首地址，如果被删除则返回NULL；（建议不要对(单pbuf链结构的)数据包队列执行此函数，主要是指当指定的pbuf是前一个数据包的最后一个pbuf时）。

输入：

pbuf的首地址p；

输出：

后面第一个pbuf的首地址；

或

NULL（ <1>如果指定pbuf后面没有pbuf；<2>后面第一个pbuf在断链解除指定pbuf对它的引用后被删除）；

基本流程：

1、判断指定的pbuf后面是否还有pbuf，没有则返回NULL，退出程序；有则继续；

2、强制令后面第一个pbuf的tot\_len= pbuf\_addr->tot\_len - pbuf\_addr->len；（为什么）

3、断链，返回后面第一个pbuf是否存在标识tail\_gone；

3.1 pbuf\_addr->next=NULL；

3.2 pbuf\_addr->tot\_len = pbuf\_addr->len；

3.3 调用《pbuf\_free》释放后面第一个pbuf，解除指定pbuf对它的引用，并将返回的删除pbuf个数，作为后面第一个pbuf是否存在的标识tail\_gone，如果为0表示存在，否则表示不存在；

4、判断tail\_gone，如果等于0，则返回后面第一个pbuf的首地址，否则返回NULL，退出程序；

###### 设计维护用和用户用：

语符：pbuf\_dechain

语法：( pbuf\_addr---下一个pbuf\_addr/NULL )

语义：指定一个pbuf，将它与它后面的pbuf断开，返回后面第一个pbuf的首地址，如果被删除则返回NULL；

（建议不要对(单pbuf链结构的)数据包队列执行此函数，主要是指当指定的pbuf是前一个数据包的最后一个pbuf时）

语用：

1、判断指定的pbuf后面是否还有pbuf，没有则返回NULL，退出程序；有则继续；

2、强制令后面第一个pbuf的tot\_len= pbuf\_addr->tot\_len - pbuf\_addr->len；（为什么）

3、断链，返回后面第一个pbuf是否存在标识tail\_gone；

3.1 pbuf\_addr->next=NULL；

3.2 pbuf\_addr->tot\_len = pbuf\_addr->len；

3.3 调用《pbuf\_free》释放后面第一个pbuf，解除指定pbuf对它的引用，并将返回的删除pbuf个数，作为后面第一个pbuf是否存在的标识tail\_gone，如果为0表示存在，否则表示不存在；

4、判断tail\_gone，如果等于0，则返回后面第一个pbuf的首地址，否则返回NULL，退出程序；

###### 子词：

dechain\_断链

语符：dechain\_断链

语法：( pbuf\_addr--- )

语义：将指定pbuf与其后面的pbuf断开。

语用：

1 pbuf\_addr->next=NULL；

2 pbuf\_addr->tot\_len = pbuf\_addr->len；

3 调用《pbuf\_free》释放后面第一个pbuf，解除指定pbuf对它的引用，并将返回的删除pbuf个数，作为后面第一个pbuf是否存在的标识tail\_gone，如果为0表示存在，否则表示不存在；

##### OK!!!pbuf\_copy()(将任意类型pbuf的payload数据复制到PBUF\_RAM类型pbuf中)

功能：将任意类型pbuf构成的pbuf链的payload数据复制到PBUF\_RAM类型pbuf链中。

1、（目的pbuf链中的pbuf只能是PBUF\_RAM类型的，因为在LWIP中PBUF\_ROM是只读的，PBUF\_REF可能只读也可能可写，PBUF\_POOL只用来主机接收新的数据分组并向上层协议传递时使用（即只能作为pbuf\_copy的源pbuf））

2、源pbuf链和目的pbuf链只能是构成一个数据包的完整或从指定pbuf开始的部分pbuf链，并且都不能是数据包队列中的某个数据包的pbuf链（即不支持单pbuf链结构的数据包队列中数据包pbuf链的复制）。

实际代码的功能：（源和目的pbuf链没有类型限制）

将一个pbuf链中的有效数据区的数据复制到另外一个pbuf链的有效数据区；

源pbuf链和目的pbuf链只能是构成一个数据包的完整或从指定pbuf开始的部分pbuf链，并且都不能是数据包队列中的某个数据包的pbuf链（即不支持单pbuf链结构的数据包队列中数据包pbuf链的复制）。

输入：

目的pbuf链首地址；源pbuf链首地址；

输出：

ERR\_OK ：成功；

ERR\_ARG：参数错误（源地址和目的地址无效，且目的pbuf链数据区的有效数据区长度不够）

ERR\_VAL：非法值（源pbuf链或目的pbuf链是单pbuf链结构的数据包队列中的pbuf链）

基本流程：

1、参数检查：必须满足( p\_to != NULL) && (p\_from != NULL) && (p\_to->tot\_len >= p\_from->tot\_len；否则返回ERR\_ARG，退出程序；(LWIP\_ERROR)

2、局部变量offset\_to和offset\_from清零。分别表示循环时当前源pbuf和目的pbuf有效数据部分累计复制的数据长度；

2、循环，从第一个源pbuf的开始复制有效数据区数据到目的pbuf的有效数据区空间：

2.1 判断当前目的pbuf的有效数据区的剩余空间是否足够当前源pbuf有效数据区剩

余的数据；if ((p\_to->len - offset\_to) >= (p\_from->len - offset\_from))

2.1.2 足够，则本次复制的数据长度就是当前源pbuf剩余的数据；

len = p\_from->len - offset\_from;

2.1.3 不够，则本次复制的数据长度是当前目的pbuf有效数据区剩余空间长度；

len = p\_to->len - offset\_to;

2.2 复制数据；目的地址=(p\_to->payload + offset\_to，源地址=p\_from->payload + offset\_from；数据长度=len；(MEMCPY)

2.3 更新本次复制后，当前目的pbuf和当前源pbuf有效数据区累计已复制数据长度；

offset\_to += len; offset\_from += len;

2.4 判断当前源pbuf的有效数据区是否全部复制完成；if (offset\_from >= p\_from->len)

2.4.1 是，则准备复制下一源pbuf的有效数据区，p\_from指向下一源pbuf；

offset\_from = 0; p\_from = p\_from->next;

2.4.2 不是，则什么都不做；

2.5 判断当前目的pbuf的有效数据区空间是否已全部用完；if (offset\_to == p\_to->len)

2.5.1 是，则

2.5.1.1 准备向下一目的pbuf复制数据，p\_to指向下一目的pbuf；

offset\_to = 0; p\_to = p\_to->next;

2.5.1.2 如果没有下一目的pbuf(p\_to==NULL)，而源pbuf还有数据要复制

(p\_from!=NULL)，则表示目的pbuf链的有效数据区空间不够，

返回ERR\_ARG，退出程序；(LWIP\_ERROR)

2.5.2 不是，则什么都不做；

2.6 判断当前源pbuf或下一个源pbuf是不是一个数据包pbuf链的最后一个pbuf;

if((p\_from != NULL) && (p\_from->len == p\_from->tot\_len))

2.6.1 是，则判断在最后一个pbuf后面是否还有pbuf，有则表示源pbuf链是数据包队列中的一个pbuf链，则返回ERR\_VAL，退出程序；(LWIP\_ERROR)

2.6.2 不是，则什么都不做；

2.7 判断当前目的pbuf或下一目的pbuf是不是最后一个目的pbuf；

if((p\_to != NULL) && (p\_to->len == p\_to->tot\_len))

2.7.1 是，则判断在最后一个目的pbuf后面是否还有pbuf，有则表示目的pbuf链是数据包队列中的一个pbuf链，则返回ERR\_VAL，退出程序；(LWIP\_ERROR)

2.7.2 不是，则什么都不做；

2.8继续循环，直到p\_from == NULL；

3、返回ERR\_OK，退出程序；

###### 设计维护用：(受限于MEMCPY词，ASYM版源pbuf链只是有条件支持全部类型)

**(支持源pbuf链中的pbuf是PBUF\_RAM或PBUF\_POOL类型的，如果是PBUF\_REF和PBUF\_ROM类型的则payload应该在DMemory中，否则不支持PBUF\_REF和PBUF\_ROM类型的)**

语符：pbuf\_copy

语法：( pbuf\_addr\_to pbuf\_addr\_from---err )

ERR\_OK ：成功；

ERR\_ARG：参数错误（源地址和目的地址无效，且目的pbuf链数据区的有效数据区长度不够）

ERR\_VAL：非法值（源pbuf链或目的pbuf链是单pbuf链结构的数据包队列中的pbuf链）

语义：将任意类型pbuf构成的pbuf链的payload数据复制到PBUF\_RAM类型pbuf链中。

1. **目的pbuf链**中的pbuf只能是PBUF\_RAM类型的。【因为在LWIP中PBUF\_ROM是只读的；PBUF\_REF可能只读也可能可写（即pbuf\_copy无法判断）；PBUF\_POOL只用来主机接收新的数据分组并向上层协议传递时使用（即只能作为pbuf\_copy的源pbuf）】
2. **受限于《MEMCPY》，源pbuf链**中的pbuf并不是任意类型的：必须满足是PBUF\_RAM或PBUF\_POOL类型的，或如果有PBUF\_REF和PBUF\_ROM类型的则payload应该在DMemory中，否则不支持PBUF\_REF和PBUF\_ROM类型的。

3、**源pbuf链和目的pbuf链**只能是构成一个数据包的完整或从指定pbuf开始的部分pbuf链，并且都不能是数据包队列中的某个数据包的pbuf链（即不支持单pbuf链结构的数据包队列中数据包pbuf链的复制）。

语用：

1、调用《pbufcopy参数检查》，必须满足( p\_to != NULL) && (p\_from != NULL) && (p\_to->tot\_len >= p\_from->tot\_len，否则参数错误；

2、如果参数正确，继续执行，否则返回ERR\_ARG,退出程序；

3、调用《循环pbufcopy》，从第一个源pbuf开始复制有效数据区数据到目的pbuf链的有效

数据区，直到复制完成源pbuf链，返回错误码；

###### 用户用：

语符：pbuf\_copy

语法：( pbuf\_addr\_to pbuf\_addr\_from---err )

语义：将任意类型pbuf构成的pbuf链的payload数据复制到PBUF\_RAM类型pbuf链中。

1. **目的pbuf链**中的pbuf只能是PBUF\_RAM类型的。【因为在LWIP中PBUF\_ROM是只读的；PBUF\_REF可能只读也可能可写（即pbuf\_copy无法判断）；PBUF\_POOL只用来主机接收新的数据分组并向上层协议传递时使用（即只能作为pbuf\_copy的源pbuf）】
2. **受限于《MEMCPY》，源pbuf链**中的pbuf并不是任意类型的：必须满足是PBUF\_RAM或PBUF\_POOL类型的，或如果有PBUF\_REF和PBUF\_ROM类型的则payload应该在DMemory中，否则不支持PBUF\_REF和PBUF\_ROM类型的。

3、**源pbuf链和目的pbuf链**只能是构成一个数据包的完整或从指定pbuf开始的部分pbuf链，并且都不能是数据包队列中的某个数据包的pbuf链（即不支持单pbuf链结构的数据包队列中数据包pbuf链的复制）。

语用：

输出：

ERR\_OK ：成功；

ERR\_ARG：参数错误（源地址和目的地址无效，且目的pbuf链数据区的有效数据区长度不

够）

ERR\_VAL：非法值（源pbuf链或目的pbuf链是单pbuf链结构的数据包队列中的pbuf链）

###### 子词：

pbufcopy参数检查

语符：pbufcopy参数检查

语法：(pbuf\_addr\_to pbuf\_addr\_from---t/f )

语义：C表达式为真，返回t（非零值），否则返回f（零值）：

( p\_to != NULL) && (p\_from != NULL) && (p\_to->tot\_len >= p\_from->tot\_len)

语用：

执行以下C表达式的功能：

( p\_to != NULL) && (p\_from != NULL) && (p\_to->tot\_len >= p\_from->tot\_len)

循环pbufcopy

语符：循环pbufcopy

语法：(pbuf\_addr\_to pbuf\_addr\_from---错误码 )

ERR\_OK ：成功；

ERR\_ARG：参数错误（目的pbuf链数据区的有效数据区长度不够）

ERR\_VAL：非法值（源pbuf链或目的pbuf链是单pbuf链结构的数据包队列中的pbuf链）

语义：从第一个源pbuf开始复制有效数据区数据到目的pbuf链的有效

数据区，直到复制完成源pbuf链，返回错误码；

语用：

1 局部变量offset\_to和offset\_from清零。分别表示循环时当前源pbuf和目的pbuf有效数

据部分累计复制的数据长度；

2 循环复制数据：

2.1 调用《计算本次复制len》，计算本次循环复制的数据的长度；

2.2 调用《pbuf复制数据》复制数据；目的地址=(p\_to->payload + offset\_to，源地址=p\_from->payload + offset\_from；数据长度=len；

2.3 调用《更新累计复制长度》，更新本次复制后当前目的pbuf和当前源pbuf有效数据区累计已复制数据长度；

2.4 调用《下次源和目的地址》，计算下一次复制的源地址和目的地址；

2.5 判断新地址是否有效：如果没有下一目的pbuf(新p\_to==NULL)，而源pbuf还有数据要复制(新p\_from!=NULL)，则表示目的pbuf链的有效数据区空间不够，返回ERR\_ARG，退出程序；(LWIP\_ERROR)

2.6 调用《NoPacket队列?》，当新源pbuf地址或新目的pbuf地址是一个数据包pbuf链的最后一个pbuf时，判断后面是否还有pbuf，是则表示一开始指定的源或目的pbuf链时是数据包队列中的，因为pbuf\_copy函数不支持此功能，所以返回ERR\_VAL，退出程序；否则继续循环复制数据；

2.7继续循环，直到新p\_from == NULL；

3 循环正常退出，返回ERR\_OK，退出程序；

计算本次复制len

语符：计算本次复制len

语法：(p\_to p\_from offset\_to offset\_from---len )

语义：计算本次复制长度；

语用：

1 判断当前目的pbuf的有效数据区的剩余空间是否足够当前源pbuf有效数据区剩

余的数据；（方法：判断(p\_to->len - offset\_to) >= (p\_from->len - offset\_from)）

2 足够，则本次复制的数据长度就是当前源pbuf剩余的数据；

len = p\_from->len - offset\_from;

3 不够，则本次复制的数据长度是当前目的pbuf有效数据区剩余空间长度；

len = p\_to->len - offset\_to;

pbuf复制数据

语符：pbuf复制数据

语法：(p\_to p\_from offset\_to offset\_from len--- p\_to p\_from offset\_to offset\_from len )

语义：复制数据，目的地址=p\_to->payload + offset\_to，源地址=p\_from->payload + offset\_from；数据长度=len；

语用：

1、计算目的地址= p\_to->payload + offset\_to；

2、计算源地址= p\_from->payload + offset\_from；

3、调用《MEMCPY》，复制数据；

更新累计复制长度

语符：更新累计复制长度

语法：(offset\_to offset\_from len---新offset\_to 新offset\_from )

语义：更新本次复制后，当前目的pbuf和当前源pbuf有效数据区累计已复制数据长度

语用：

新offset\_to += offset\_to + len;

新offset\_from += offset\_from + len;

下次源和目的地址

语符：下次源和目的地址

语法：( p\_to p\_from新offset\_to 新offset\_from ---新p\_to 新p\_from 新offset\_to 新offset\_from)

语义：计算下一次复制的源地址和目的地址，如果源或目的地址是下一Pbuf，则相应的offset\_from或offset\_to必须清零；

语用：

1 判断当前源pbuf的有效数据区是否全部复制完成；if (offset\_from >= p\_from->len)

1.1 是，则准备复制下一源pbuf的有效数据区，p\_from指向下一源pbuf；

offset\_from = 0; p\_from = p\_from->next;

1.2 不是，则什么都不做；

2 判断当前目的pbuf的有效数据区空间是否已全部用完；if (offset\_to == p\_to->len)

2. 1 是，则

2.2 准备向下一目的pbuf复制数据，p\_to指向下一目的pbuf；

offset\_to = 0; p\_to = p\_to->next;

新地址无效判断

语符：新地址有效判断

语法：(新p\_to 新p\_from---t/f )

语义：如果没有下一目的pbuf(新p\_to==NULL)，而源pbuf还有数据要复制(新p\_from!=NULL)，则表示目的pbuf链的有效数据区空间不够，则返回地址无效标识f，否则返回有效标识t；

语用：

1、新p\_to==NULL比较；

2、新p\_from!=NULL；

3、两个比较结果与运算，且结果作为最终返回的标识；

NoPacket队列?

语符：NoPacket队列?

语法：(新p\_to 新p\_from---t/f )

语义：因为buf\_copy函数不支持单pbuf链结构的数据包队列中数据包pbuf链的复制。所以需要判断新源pbuf地址或新目的pbuf地址是不是一个数据包pbuf链的最后一个pbuf，是则继续判断后面是否还有pbuf，是则表示一开始指定的源或目的pbuf链时是数据包队列中的，返回参数错误标识f，否则返回正确标识t。

语用：

1、调用《源NoPacket队列?》，判断源pbuf链是否是数据包队列中的链；

2、栈顶为假，则返回参数错误标识f，退出程序；

3、栈顶为真，调用《目的NoPacket队列?》，判断目的pbuf链是否是数据包队列中的链，不是返回真，是则返回假；

源NoPacket队列?

语符：源NoPacket队列?

语法：( 新puf\_from---t/f )

语义：源pbuf地址是不是一个数据包pbuf链的最后一个pbuf，是则继续判断后面是否还有pbuf，是则表示一开始指定的源或目的pbuf链时是数据包队列中的，返回参数错误标识f，否则返回正确标识t

语用：

1 判断新源地址pbuf是不是一个数据包pbuf链的最后一个pbuf;

if((p\_from != NULL) && (p\_from->len == p\_from->tot\_len))

1.1 是，则判断在最后一个pbuf后面是否还有pbuf，有则表示源pbuf链是数据包队列中

的一个pbuf链，则返回参错错误标识f；

1.2 不是，则返回正确标识t；

目的NoPacket队列?

语符：目的NoPacket队列?

语法：( 新pbuf\_to---t/f )

语义：源pbuf地址是不是一个数据包pbuf链的最后一个pbuf，是则继续判断后面是否还有pbuf，是则表示一开始指定的源或目的pbuf链时是数据包队列中的，返回参数错误标识f，否则返回正确标识t

语用：

1 判断当前目的pbuf或下一目的pbuf是不是最后一个目的pbuf；

if((p\_to != NULL) && (p\_to->len == p\_to->tot\_len))

1.1 是，则判断在最后一个目的pbuf后面是否还有pbuf，有则表示目的pbuf链是数据包

队列中的一个pbuf链，则返回参错错误标识f；

1.2 不是，则返回参数正确标识t；

##### Ok!!!pbuf\_copy\_partial()(将任意类型pbuf的payload数据复制到应用缓冲存储区)

功能：将指定pbuf开始的pbuf或pbuf链中的有效数据部分的数据，复制到指定的应用缓冲存储区中；

输入：

源pbuf链首地址buf；应用缓冲存储区目的地址dataptr；复制数据长度len；pbuf有效数据部分的偏移量offset；

输出：

实际复制的数据的字节长度

基本流程：

1、参数检查：(buf == NULL) || (dataptr == NULL)，则返回0，退出程序；

2、不定循环，分两个阶段：先根据偏移量，计算复制数据的起始地址；然后继续循环复制数据：循环终止的条件是：剩余复制数据长度为0（表示数据正常复制完成）；或当前pbuf

指针为NULL（在计算计算起始地址阶段终止，表示(输入offset)>=(buf->tot\_len);

在复制数据阶段终止，表示是(输入offset+len)>=(buf->tot\_len)）

2.1 根据offset判断当前是循环的哪个阶段：

offset !=0 ：表示是计算复制数据起始地址阶段；

offset ==0 ：表示是复制数据阶段；

2.1.1 计算复制数据起始地址阶段：

2.1.1 判断判断偏移量Offset是否超出当前pbuf，是则offset -= 当前pbuf->len；

2.1.2 否则继续循环，偏移计算复制数据起始阶段；

2.1.2 复制数据阶段：

2.1.2.1 初始化本次循环复制数据长度等于当前pbuf有效数据区剩余空间大小；

2.1.2.2判断本次循环复制数据长度是否大于剩余复制数据长度；

2.1.2.2.1 大于，则本次循环复制数据就等于剩余复制数据长度；

2.1.2.2.2 不大于，则本次循环复制的数据长度就是当前pbuf有效数据区剩余空间大小；

2.1.2.3 复制数据：源地址=(当前pbuf->payload+累计复制长度；目的地址=dataptr+累计复制长度；

长度=2.1.2.2的判断结果；

2.1.2.4 更新累计复制数据长度；

2.1.2.5 更新剩余复制数据长度；

2.1.2.6 offset清零，表示是在循环的复制阶段；

3、循环退出，返回累计复制数据长度（初始值为0）；

###### 设计维护用：(受限于MEMCPY词，ASYM版源pbuf链只是有条件支持全部类型，应用缓冲存储区也只能是DMemory中内存)

**(支持源pbuf链中的pbuf是PBUF\_RAM或PBUF\_POOL类型的，如果是PBUF\_REF和PBUF\_ROM类型的则payload应该在DMemory中，否则不支持PBUF\_REF和PBUF\_ROM类型的)**

语符：pbuf\_copy\_partial

语法：( pbuf\_addr dataptr len offset---real\_len )

输入参数：

pbuf\_addr：源pbuf链首地址；

dataptr：应用缓冲存储区目的地址；

len：复制数据长度，16位无符号数；

offset：pbuf有效数据区的偏移量，16位无符号数；

输出参数：

real\_len：实际复制的数据长度，16位无符号数；

语义：将指定pbuf或pbuf链的有效数据区的数据，复制到指定的应用缓冲存储区中；

1. **源pbuf链**：受限于《MEMCPY》词，源pbuf链中的pbuf并不是任意类型的：必须满足是PBUF\_RAM或PBUF\_POOL类型的，或如果有PBUF\_REF和PBUF\_ROM类型的则payload应该在DMemory中，否则不支持PBUF\_REF和PBUF\_ROM类型的。
2. **应用缓冲存储区**：受限于《MEMCPY》词，只能是DMemory中的内存。

语用：

1、参数检查：(buf == NULL) || (dataptr == NULL)，则返回0，退出程序；

2、不定循环，分两个阶段：先根据偏移量，计算复制数据的起始地址；然后继续循环复制数据：循环终止的条件是：剩余复制数据长度为0（表示数据正常复制完成）；或当前pbuf

指针为NULL（在计算计算起始地址阶段终止，表示(输入offset)>=(buf->tot\_len);

在复制数据阶段终止，表示是(输入offset+len)>=(buf->tot\_len)）

###### 子词：

辅助变量：copied\_total

pbufpart\_copydata

语符：pbufpart\_copydata

语法：( pbuf\_addr 目的dataptr len offset---pbuf\_addr 新目的dataptr 新len 新offse=0 )

语义：根据pbuf\_addr->len与输入的len的大小，确定本次复制的长度buf\_copy\_len，然后复制数据；更新累计复制的数据长度，更新len,offset；

语用：

1、计算本次复制的数据长度buf\_copy\_len；

2、调用《MEMCPY》复制数据；

3、更新累计复制的数据长度copied\_total，更新len,offset，目的dataptr；

##### Ok!!!pbuf\_realloc(调整payload区的大小，只支持减小的功能)

功能：调整指定pbuf（或pbuf链）的payload区的大小，只支持减少的功能；

输入：pbuf首地址；payload区新大小new\_len（16位无符号数）；

输出：无

基本流程：

1、如果是增大payload区，则直接退出程序；（方法：if (new\_len >= p->tot\_len) 为真则退出）

2、计算payload区的增量：grow = new\_len - p->tot\_len。如果grow为正表示是增大payload区，为负表示是减小payload区；（不可能为正）

3、根据新的payload区的大小，循环定位到新pbuf链的最后一个pbuf，并修改从第一个pbuf开始的中间pbuf的tot\_len（当前pbuf->tot\_len += (u16\_t)grow）

4、循环退出后，判断新pbuf链最后一个pbuf的类型是否是PBUF\_RAM类型并且要保留的payload区大小不等于其len的值；

4.1 是，则调用mem\_trim，将payload区调整为要保留的大小；

4.2 不是，则继续执行；

5、修改新pbuf链最后一个pbuf的len和tot\_len：q->len = rem\_len;q->tot\_len = q->len;

6、如果新pbuf链最后一个pbuf后面还有pbuf，则调用pbuf\_free解除最后一个pbuf对它的引用；如果没有则什么都不做；

7、将新pbuf链最后一个pbuf的next置NULL；

###### 设计维护用：

语符：pbuf\_realloc

语法：( pbuf\_addr new\_len--- )

语义：调整指定pbuf（或pbuf链）的payload区的大小，只支持减少的功能。根据指定的新的大小，pbuf链的前面的一些pbuf可能跳过（只修改tot\_len），新的last pbuf大小会被调整，并且后面的pbuf会被释放。如果last pbuf是ROM/REF/POOL类型，则只是调整tot\_len和len，如果是RAM类型，则会调用mem\_trim调整大小；（注意：指定的pbuf（或pbuf链）不能是数据包队列中的）

Depending on the desired length, the first few pbufs in a chain might be skipped and left unchanged. The new last pbuf in the chain will be resized, and any remaining pbufs will be freed.

注意： If the pbuf is ROM/REF, only the ->tot\_len and ->len fields are adjusted.

注意： May not be called on a packet queue.

语用：（与lwip中在grow的处理上不同，本词中的grow是作为无符号数）

1、如果是增大payload区或不调整大小，则直接退出程序；（方法：if (new\_len >= p->tot\_len) 为真则退出）

2、计算payload区的减少量：grow = p->tot\_len - new\_len。（不一样的地方）

3、调用《定位New\_last\_pbuf》，根据新的payload区的大小，循环定位到新pbuf链的最后一个pbuf，并修改从第一个pbuf开始的中间pbuf的tot\_len（当前pbuf->tot\_len -= grow）（不一样的地方）；

4、循环退出后，调用《pbufRAM类型?》判断新pbuf链最后一个pbuf的类型是否是PBUF\_RAM类型并且要保留的payload区大小不等于其len的值； 是，则调用mem\_trim，将payload区调整为要保留的大小；不是，则什么都不做；

5、修改新pbuf链最后一个pbuf的len和tot\_len：q->len = rem\_len;q->tot\_len = q->len;

6、如果新pbuf链最后一个pbuf后面还有pbuf，则调用pbuf\_free解除最后一个pbuf对它的引用；如果没有则什么都不做；

7、将新pbuf链最后一个pbuf的next置NULL；

###### 用户用：

语符：pbuf\_realloc

语法：( pbuf\_addr new\_len--- )

语义：调整指定pbuf（或pbuf链）的payload区的大小，只支持减少的功能。

语用：根据指定的新的大小，pbuf链的前面的一些pbuf可能跳过（只修改tot\_len），新的last pbuf大小会被调整，并且后面的pbuf会被释放。如果last pbuf是ROM/REF/POOL类型，则只是调整tot\_len和len，如果是RAM类型，则会调用mem\_trim调整大小；（注意：指定的pbuf（或pbuf链）不能是数据包队列中的(May not be called on a packet queue)）

###### 子词：

pbufRAM类型?

语符：pbufRAM类型?

语法：( lastpbuf\_addr rem\_len---lastpbuf\_addr rem\_len )

语义：判断新pbuf链最后一个pbuf的类型是否是PBUF\_RAM类型并且要保留的payload区大小不等于其len的值； 是，则调用mem\_trim，将payload区调整为要保留的大小；不是，则什么都不做；

语用：

1 lastpbuf\_addr->type == PBUF\_RAM判断；

2 rem\_len != lastpbuf\_addr->len判断；

3 两次判断结果与运算；

4 栈顶为真，则调用mem\_trim，输入参数是lastpbuf\_addr和lastpbuf的新的总大小=

lastpbuf\_addr->payload - lastpbuf\_addr + rem\_len

5 栈顶为假，什么也不做；

定位New\_last\_pbuf

语符：定位New\_last\_pbuf

语法：( grow pbuf\_addr new\_len---lastpbuf\_addr rem\_len )

语义：根据新的payload区的大小，循环定位到新pbuf链的最后一个pbuf，并修改从第一个pbuf开始的中间pbuf的tot\_len（当前pbuf->tot\_len -= grow），返回新pbuf链的最后一个pbuf的首地址和它的payload区应保留的长度（用循环结束后rem\_len的值表示）；

语用：

根据新的payload区的大小，循环定位到新pbuf链的最后一个pbuf，并修改从第一个pbuf开始的中间pbuf的tot\_len（当前pbuf->tot\_len -= grow）；

##### pbuf\_take()(复制用户存储区数据到pbuf中payload中)

功能：

将用户存储区的数据复制到pbuf（链）的有效数据空间；

输入：

目的pbuf首地址buf；用户存储区源地址dataptr；复制的数据长度len；

输出：

ERR\_OK：成功；

ERR\_ARG：失败；(buf == NULL) || (dataptr == NULL) || (buf->tot\_len < len)（代码注释是ERR\_MEM）

基本流程：

1、参数检查：满足(buf == NULL) || (dataptr == NULL) || (buf->tot\_len < len)，则返回ERR\_ARG，退出程序；

2、不定循环，复制数据，直到剩余复制数据长度等于0：

2.1 判断剩余复制数据长total\_copy\_len是否不等于0；

2.2 不等于0，继续循环复制数据；

2.2.1 本次循环复制数据长度初始化成剩余复制数据长度；

2.2.2 判断剩余复制数据长度是否大于当前pbuf有效数据区总大小；

2.2.2.1 大于，则本次循环复制数据长度等于有效数据去总大小；

2.2.2.2 不大于，则本次循环复制数据长度等于剩余复制数据长度；

2.2.3 复制数据：源地址=dataptr+累计已复制长度；目的地址=当前pbuf->payload;

长度=本次循环复制数据长度；

2.2.4 更新剩余复制数据长度；

2.2.5 更新累计已复制数据长度；

3、循环结束，返回ERR\_OK；

##### (ASYM可选功能，暂不考虑)pbuf\_coalesce()

##### (ASYM可选功能，暂不考虑)pbuf\_fill\_chksum()

##### (ASYM可选功能，暂不考虑)pbuf\_get\_at()

##### (ASYM可选功能，暂不考虑)pbuf\_memcmp()

##### (ASYM可选功能，暂不考虑)pbuf\_memfind

##### (ASYM可选功能，暂不考虑)pbuf\_strstr

# 附录：

## 笔记参考资料：

1、官方源码及附带文档：Lwip-1.3.0。

2、原作者文档：《Design and Implementation of the LwIP TCP/IP Stack》. Adam Dunkels , February 20 ,2001.中文版。

3、网络资料：《LwIP协议栈源码详解---TCP/IP协议的实现》，老衲五木。

4、《嵌入式网络那些事—LwIP协议深度剖析与实战演练》朱升林 著。