## 移植综述

### 移植注意

1. LWIP协议内部使用的数据类型的定义。
2. 实现与信号量和邮箱操作相关的函数。 只要我们将UC/OS-II中的相关函数做响应的封装，就可满足LWIP的需求。
3. LWIP使用邮箱和信号量来实现上层应用与协议栈间、下层硬件驱动与协议栈间的信息交互。LWIP完成相关初始化后，会阻塞在一个邮箱上，等待接收数据进行处理。这个邮箱内的数据可能来自硬件驱动接收到的数据包，也可能来自应用程序。进入该邮箱内的数据，LWIP会对数据进行解析，然后再依次调用协议栈内部上层相关处理函数处理数据。处理完成后，LWIP继续阻塞在邮箱上等待下一批数据。
4. 与等待超时相关的函数。等待邮箱的过程中可以设置超时时间，如果设置超时，需要提供一个函数：该函数能够返回当前进程中超时事件链表的首地址。
5. 如果LWIP建立在多线程操作系统之上的话，则要实现创建一个新线程的函数。一个典型的LWIP应用系统包括这样的三个进程：上层应用程序进程，LWIP协议栈进程，底层硬件数据包接收发送进程。通常LWIP协议栈进程是在应用程序中调用LWIP协议栈初始化函数来创建的。
6. 临界区保护函数。禁止地址自动对齐。
7. 底层网络驱动函数的实现。最起码的，发送一个数据包和接收一个数据包函数需要实现。

## 动态内存管理

1. LWIP的动态内存管理机制有三种：C运行库自带的内存分配策略，动态内存堆（HEAP）分配策略和动态内存池（POOL）分配策略。
2. 通过lwippools.h中的宏定义MEM\_LIBC\_MALLOC来实现C运行库自带的内存分配策略或动态内存堆（HEAP）分配策略的选择。为1表示C运行库自带的内存分配策略，为0反之。

### 2.1 内存堆方式

1、动态内存堆分配策略是在一个实现定义好大小的内存块中进行管理，其内存分配的策略是采用最快合适的方法，只要找到一个比所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动态内存堆中。要求分配的大小不能小于MIN\_SIZE，一般为12字节

2、对于动态内存的使用，比较推荐的方法是分配---释放---分配---释放，这种使用方法可减少内存碎片。

1. Mem\_int()内存堆的初始化函数，主要告知内存堆的起止地址，以及初始化空闲列表，由lwip初始化时调用。
2. Mem\_malloc()申请分配内存。返回的内存是没有初始化的，内存堆是全局变量要做好线程安全保护。
3. Mem\_calloc()是对mem\_malloc的封装，它申请的内存全部完成初始化。

### 2.2 动态内存池

## 数据包pbuf

1. LWIP中常用到的内存分配策略有两种：内存堆分配，内存池分配。
2. LWIP必须有个高效的数据包管理核心，既能海纳百川似的兼容各种类型的数据，又能避免在各个层之间的幅值数据的巨大开销。
3. 数据包管理机构采用数据结构pbuf来描述数据包。

Struct pbuf {

/\* next指向下一个pbuf组成队列存放数据，有可能接收和发送的数据包很 大，pbuf管理的数据比较少，所以需要链接在一个列表上\*/

Struct pbuf \*next;

/\*数据指针，指向pbuf管理的数据的起始地址\*/

Void \*payload;

/\*表示当前pbuf和其后所有pbuf的有效数据的长度\*/

U16\_t tot\_len;

/\*表示当前pbuf中的有效长度\*/

U16\_t len;

/\*pbuf的类型\*/

U8\_t type;

/\*初始化pbuf时，将flags=0\*/

U8\_t flags;

/\*初始化pbuf时，ref=1，后续如果有指针指向pbuf时，该值+1，删除一个 pbuf，该值-1\*/

U16\_t ref;

};

### 3.1 pbuf类型

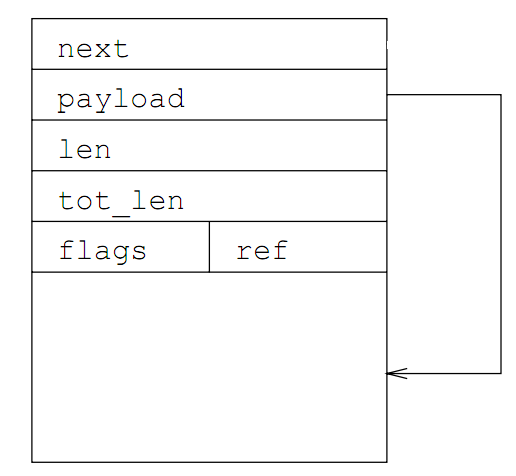
pbuf有四类：PBUF\_RAM、PBUF\_ROM、PBUF\_REF、PBUF\_POOL。

#### 3.1.1 PBUF\_RAM

PBUF\_RAM类型的pbuf主要通过内存栈分配得到，申请PBUF\_RAM类型时，协议栈会 在内存中分配响应的大小（包括pbuf结构体头的大小+相应数据缓冲区）。

P = (struct pbuf\*)mem\_malloc(LWIP\_MEM\_ALIGN\_SIZE(SIZEOF\_STRUCT\_PBUF + offet) +LWIP\_MEM\_ALIGN\_SIZE(length));

它们是一片连续的内存区。

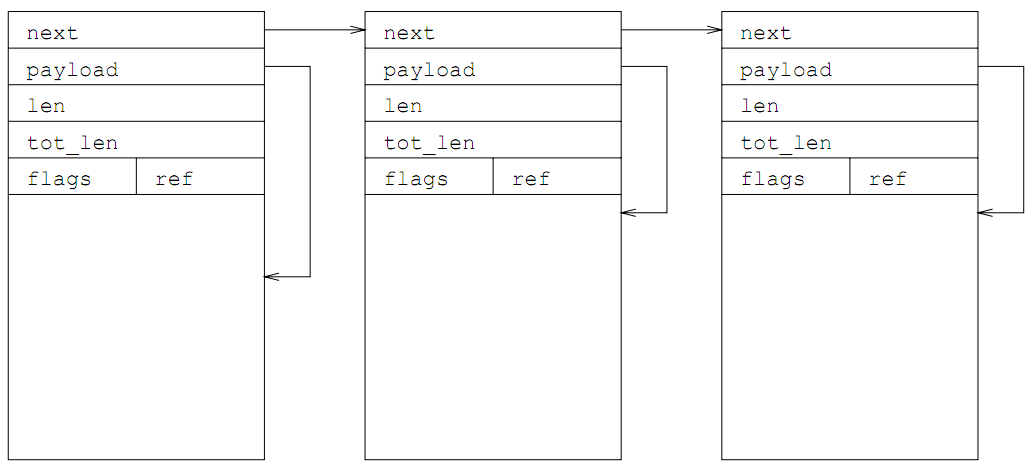


#### 3.1.2 PBUF\_POOL

PBUF\_POOL和PBUF\_RAM原理基本一样，不同点是PBUF\_POOL在内存池上分配，因为内存池已经实现创建好，所以PBUF\_POLL创建速度很快，但必须与创建内存池的大小匹配。

P = mem\_malloc(MEMP\_PBUF\_POOL);

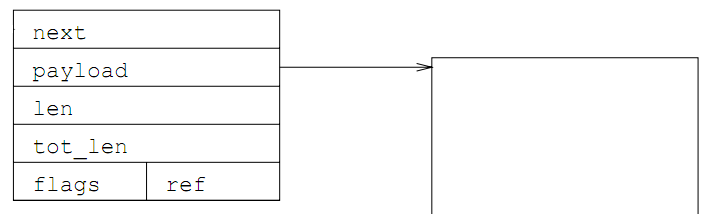
成功分配后pbuf如下：



#### 3.1.3 PBUF\_ROM和PBUF\_REF

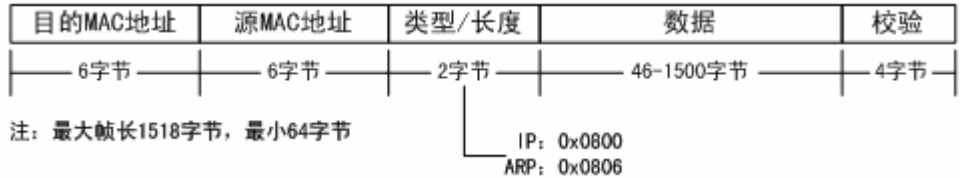
PBUF\_ROM和PBUF\_REF类型的pbuf基本相同，它们的申请都是在内存堆中分配一个响应的pbuf结构头，而不申请数据区的空间。

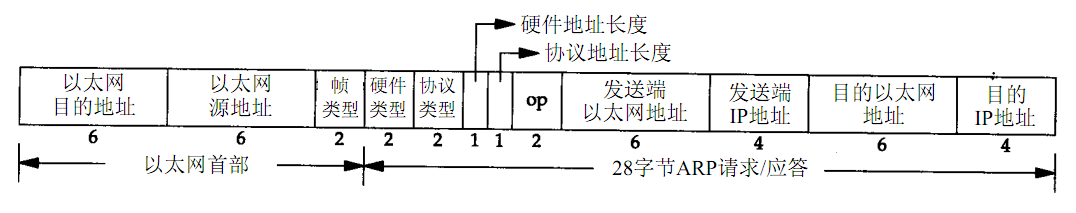
P = mem\_malloc(MEMP\_PBUF);



## pbuf释放

当pbuf的ref=1时，该pbuf才能被删除，假设我们的pbufs有A、B、C三个pbuf结构连在一起，





## 网络接口结构

LWIP从逻辑上看分为四层：链路层、网络层、传输层和应用层。应用层和协议栈的代码可以使用相同的内存，应用可以直接读写内部缓存。

## 以太网数据接收

LWIP中实现了接收一个数据包和发送一个数据包函数的框架，这两个函数分别是low\_level\_input和low\_level\_output，

数据包的接收最好使用中断方式：在无数据包接收到时，数据包接收进程阻塞在一个信号量下，当有数据包到来时，网卡芯片产生一个中断信号，处理器进入中断处理，并释放一个信号量。中断退出后，数据包接收进程得到信号量，并从网卡芯片中读取数据包，并将数据包递交给上层进行处理。

大小端是针对内存低位进行说明的，大端指高位在低端，小端指低位在低端。

数据包发送时只需要调用low\_level\_output函数即可。

## ARP表

ARP缓存表定义：

Struct etharp\_entry {

#if APP\_QUEUEING

Struct etharp\_q\_entry \*q; //数据包缓冲队列指针

#endif

Struct ip\_addr ipaddr; //目标IP地址

Struct eth\_addr ethaddr; //MAC地址

Enum etharp\_stare state; //描述entry的状态

U8\_t ctime; //描述该entry的时间信息

Struct netif \*netif; //相应网络接口信息

};

ARP\_QUEUEING是编译选项，表示是否允许缓存表象有数据包缓冲队列，在opt.h中设置。

iPaddr和ethaddr字段分别保存IP地址和MAC地址。State是个枚举类型，表示缓存表项的状态，定义如下：

enum etharp\_state{

ETHARP\_STATE\_EMPTY = 0，//没有任何信息

ETHARP\_STATE\_PENDING， //处于不稳定状态

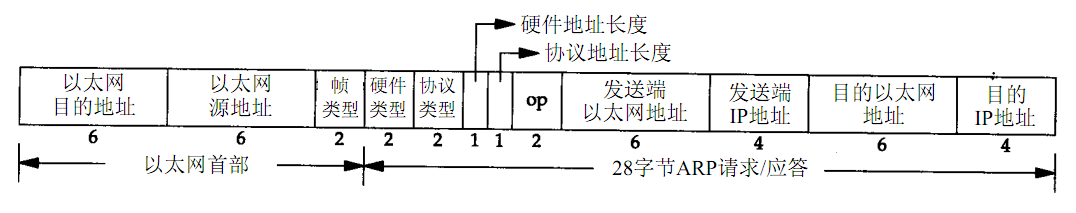
ETHARP\_STATE\_STABLE}; //就绪状态

ETHARP\_STATE\_PENDING：很有可能的情况是只有IP没有MAC。在该状态下，LWIP内核会发送一个广播的ARP请求到数据链路上，以让对应的IP地址的主机回应其MAC地址，源主机接收到MAC地址时，更新对应的ARP表项。此时将状态改为ETHARP\_STATE\_STABLE。

当表项在PENDING状态时，要发往该表项中IP地址处的数据包会被连接在表项对应的数据包缓冲队列上，当等到表项稳定后，这些数据包才会被发送出去，这就是为什么每个表项西药数据包缓冲队列指针了。

Ctime记录某个状态的时间，当某表项的ctime值大于规定的表项最大生存值时，该表项会被内存删除。

ARP数据包分为请求数据包和应答数据包，ARP数据包到达底层链路时会被加上以太网数据包头发送出去，最终呈现在链路上的数据报头格式如下图：



目的地址全为1的特殊地址是广播地址。

两个字节长度的以太网帧类型表示后边的数据类型，ARP--0x0806，IP数据包--0x0800。

硬件类型字段表示硬件的地址类型，=1表示以太网的MAC地址，长度为6个字节。

协议类型字段表示要映射的协议地址类型，=0x0800表示要映射为IP地址。

硬件地址长度和协议地址长度分别为6和4。

Op字段详解见下表：

|  |  |
| --- | --- |
| OP | 说明 |
| 1 | ARP请求 |
| 2 | ARP应答 |
| 3 | RARP请求 |
| 4 | RARP应答 |

当主机收到一份给自己的ARP请求报文后，它就白自己的硬件地址填进去，然后将请求数据包的源主机信息和目的主句信息交换位置，并把操作字段op置为2。

## 第 8 章 ARP表查询

ARP攻击：在局域网中，ARP病毒收到广播的ARP请求包，能够解析出其它节点的（IP，MAC）地址，然后病毒伪装为目的主机，高祖源主机一个假的MAC地址，使得源主机发送给目的主机的所有数据包都被病毒软件截取。

find\_entry--输入一个IP地址，返回值是该IP地址对应的ARP缓存表项索引。

Static s8\_t find\_entry(struct ip\_addr\*ipaddr, u8\_t flags)

该函数的主要功能是寻找一个匹配的ARP表项或者是一个新的ARP表项，并返回该表项的索引号。

find\_entry查询完成后已经知道了

第一个empty状态表项的索引

生存时间最老的pending状态且有数据缓冲表项的索引

生存时间最老的pending状态且无数据缓冲表项的索引

生存时间最长的stable状态表项的索引

如果flags==ETHARP\_TRY\_HARD，那么会按照a-->d-->c-->b的顺序选择一个合适的索引。

Etharp\_query：该函数的功能是向给定的IP地址发送一个数据包或者发送一个ARP请求。

处理过程：

1. 如果给定的IP地址不在ARP表中，则一个新的ARP表项会被创建，此时该表项处于pending状态，同时一个关于该IP地址的ARP请求包会被广播出去，同事将要发送的数据包挂接到该表项的数据缓冲指针上。
2. 如果IP地址在ARP表中有相应的表项存在但该表项为pending，则操作和前者一样。
3. 如果IP地址在ARP表项中有相应的表项存在，且表项处于stable状态，此时再判断给定的数据包是否为空，不为空则字节将该数据包发送出去，为空则向该IP地址发送一个ARP请求。

err\_t etharp\_query(struct netif\*netif，struct ip\_addr\*ipaddr，struct pbuf\*q)

对于PBUF\_REF、PBUF\_POOL、PBUF\_RAM型的数据包不能直接挂在等待发送链表上，因为这些数据包在被挂接后并不会立刻发送出去。

## 第 9 章 ARP层流程

static err\_t

update\_arp\_entry(struct netif \*netif, struct ip\_addr \*ipaddr, struct eth\_addr \*ethaddr, u8\_t flags)

{

s8\_t i; //

u8\_t k;

i = find\_entry(ipaddr, flags); //查找或新建一个ARP表项，返回其索引值

if (i < 0) return (err\_t)i; //如果为不合法索引，返回失败

arp\_table[i].state = ETHARP\_STATE\_STABLE; // 否则将对应表项状态改为

arp\_table[i].netif = netif; // 记录网络接口

k = ETHARP\_HWADDR\_LEN; //跟新缓存表项中的MAC地址

while (k > 0) {

k--;

arp\_table[i].ethaddr.addr[k] = ethaddr->addr[k];

}

arp\_table[i].ctime = 0; // 生存时间值设置为0

#if ARP\_QUEUEING //该ARP表项上有未发送的队列，则把这些队列发送出去

while (arp\_table[i].q != NULL) { // 只要缓冲链表上有数据则循环

struct pbuf \*p;

struct etharp\_q\_entry \*q = arp\_table[i].q; //记录下缓冲链表表头

arp\_table[i].q = q->next; // 缓冲链表表头指向下一个链表

p = q->p; // 去的记录下的缓冲链表表头指向的数据包

memp\_free(MEMP\_ARP\_QUEUE, q); //释放记录下的缓冲链表表头

etharp\_send\_ip(netif, p, (struct eth\_addr\*)(netif->hwaddr), ethaddr); //发送数

pbuf\_free(p); //释放数据包缓存空间

}

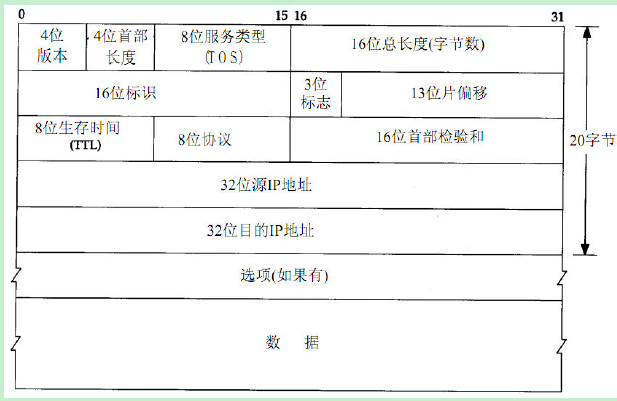
#endif

return ERR\_OK;

}

## 第 10 章 IP层输入

通常IP数据报头长度为20个字节。



第一个字段是4bit的版本号，对于IPv4，该值为4；对于IPv6，该值为6。

接下来的4bit是记录首部长度，以字节为单位。对于不焊任何选项字段的IP报头，该值为5，由于该字段最大值为15，所以其能描述的最大IP报头长度为60字节。

再下来的8bit的服务类型字段，该字段描述该IP数据包继续的服务类型，如最小延时、最大吞吐量、最高可靠性、最小费用等，在LWIP中无效。

16位的总长度字段描述了整个IP数据报，包括IP数据报头的总字节数。

16位的标识字段用于标识IP层发送出去的每一份IP数据报，没发送一份数据报文，该值加1。3位标识和13位片偏移用于在IP数据包分片时使用。

TTL字段藐视该IP数据包最多能被转发的次数。

8位协议字段用来描述该IP数据包是来自上层的哪个协议，1——ICMP，2——IGMP，6——TCP，17——UDP。

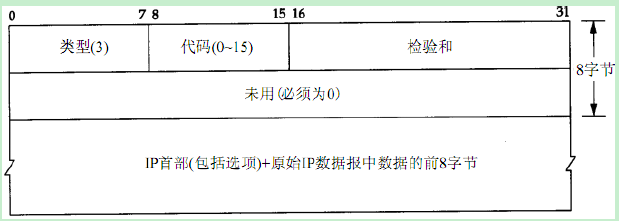
16位首部校验和只针对IP首部做校验。

从以太网底层进来的数据包经过ethernet\_input函数分发给IP模块或者ARP模块，分给IP模块是通过调用ip\_input函数实现，ethernet\_input需要将数据包去掉以太网头，然后调用ip\_input。

Ip\_input根据IP数据包头内部的协议字段判断数据包应该被递交给哪个上层协议，并调用响应的函数递交数据包。

## 第 11 章 IP分片重装1

## 第 13 章 ICMP处理

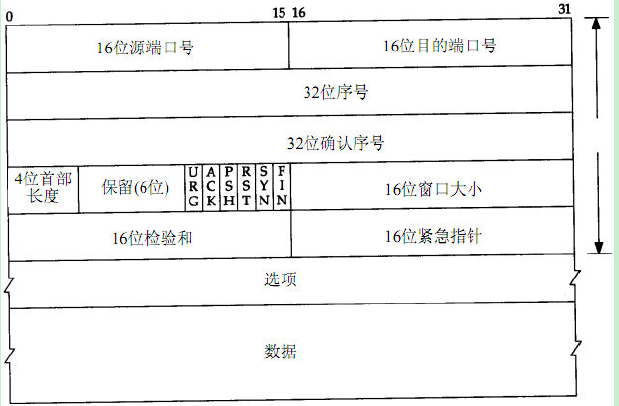


代码段值应为（0~15），目前LWIP只支持（0~5）分别表示网络不可达、主机不可达、协议不可达、端口不可达、需进行分片但设置了不分片位、源站选择失败。

## 第 14 章 TCP建立与断开

应用数据被分割成TCP认为最合适的发送的数据块，当TCP发出一个段后，它启动一个定时器，等待目的端确认收到这个报文，如果不能及时收到一个确认，将重发这个报文段；当TCP收到发自TCP连接另一端的数据，它会推迟几分之一秒发送确认，来保证接收数据的首部和校验和。

IP数据报的到达可能是失序的，所以TCP数据的到达也可能是失序的。



32位序号字段用来标识从TCP发送端到TCP接收端的数据字节流，用它来标识这个报文段中的第一个数据字节的序号。

32位确认序号只有ACK标志为1时才有效，它包含发送确认的一段所期望收到的下一个序号。确认序号应当是上次已成功收到的数据字节序号加1.当一个TCP连接被正确建立后，ACK字段总是被设置为1。

4位首部长度给出首部中32bit字的数目，TCP最多60字节的首部。

在TCP首部中有6个标志比特。他们中的饿多个可同时被设置为1。

|  |  |
| --- | --- |
| 名称 | 属性 |
| URG | 紧急指针有效标识 |
| ACK | 确认序号有效标识 |
| PSH | 接收方应尽快将这个报文段交给应用层 |
| RST | 重建连接 |
| SYN | 同步序号，用来发起一个连接 |
| FIN | 发端完成发送任务 |

16位窗口大小字段通过声明自身的窗口大小来实现流量控制，窗口大小表示还能接收的字节数。

16位校验和覆盖了整个的TCP报文段：TCP首部和TCP数据。

16位紧急指针是一个正的偏移量，和序号字段中的值相加表示紧急数据最后一个字节的序号，只有当UURG标志置1时紧急指针才有效。

最常见的可选字段是最长报文大小，又称MSS。每个连接方通常都在通信的第一个报文段（为建立连接而置SYN标志的那个段）中指明这个选项。

## 第 15 章 TCP状态转换

## 第 16 章 TCP控制块

struct tcp\_pcb {

IP\_PCB; //􄖭􁰃􀏔􀏾􁅣􀋈􁦣􄗄􀑚􄖲􁥹􂱘 IP 􂳌􀝇􀖵􁙃􀋈􀣙􁣀􀦠􁮍 IP 􀴄􀴔􀋈TTL 􃄝􀖵􁙃

struct tcp\_pcb \*next; //􂫼􀑢􄖲􁥹􀧘􀏾 TCP 􁥻􀠊􀴫􂱘􄫒􃸼􁣛􄩜

enum tcp\_state state; //TCP 􄖲􁥹􂱘􂢊􁗕􀋈􀥇􀐎􂢊􁗕􀳒 􀐁􁦣􄗄􂱘􄙷􀑯􂢊􁗕

u8\_t prio; //􄆹􁥻􀠊􀴫􂱘􀓬􀜜􃑻

void \*callback\_arg;//

u16\_t local\_port; //􁴀􀴄􃃃􀦷

u16\_t remote\_port; //􄖰􂿟􃃃􀦷

u8\_t flags;// 􄰘􀡴􂢊􁗕􀖵􁙃􀋈􀽖􄖲􁥹􁰃􁖿􄗳􁘶􀻡􀇃􀏔􀏾􃹿􁓊􄖳􂱘 ACK 􁰃􀧺􃹿􀦥􄗕􃄝

#define TF\_ACK\_DELAY (u8\_t)0x01U /\* Delayed ACK. \*///􄖭􀑯􁅣􁅮􀐝􁰃􀐎 flags 􁄫􂆉

#define TF\_ACK\_NOW (u8\_t)0x02U /\* Immediate ACK. \*///􁅮􀐝􂱘􁥽􂷕

#define TF\_INFR (u8\_t)0x04U /\* In fast recovery. \*/

#define TF\_RESET (u8\_t)0x08U /\* Connection was reset. \*/

#define TF\_CLOSED (u8\_t)0x10U /\* Connection was sucessfully closed. \*/

#define TF\_GOT\_FIN (u8\_t)0x20U /\* Connection was closed by the remote end. \*/

#define TF\_NODELAY (u8\_t)0x40U /\* Disable Nagle algorithm \*/

// 􁥹􁬊􂳌􀝇􁄫􂆉

u32\_t rcv\_nxt; //􁳳􁳯􁥹􁬊􂱘􀏟􀏔􀏾􁄫􃡖􀋈􀥇􁅗􀧥􀦥􄗕􃃃 ACK 􂱘􁑣􀧋

u16\_t rcv\_wnd; //􁥹􁬊􃁫􀦷

u16\_t rcv\_ann\_wnd; //􄗮􀨞􃁫􀦷􀻻􁇣􀋈􄕗􀔢􂠜􁴀􀐁􁮴􄆹􁄫􂆉

u32\_t tmr; // 􄆹􁄫􂆉􄆄􁔩􄆹 PCB 􃹿􀟯􁓎􂱘􁯊􀠏

u8\_t polltmr, pollinterval; // 􀏝􀏾􁅮􁯊􀰼􀋈􀧢􃓁􄆆􃾷

u16\_t rtime; //􄞡􀓴􁅮􁯊􀋈􄆹􀘐􄱣􁯊􄯈􀹲􀡴􀋈􁔧􀻻􀑢 rto 􂱘􀘐􁯊􀟭􄞡􀓴􀦥􂫳

u16\_t mss; //􁳔􀻻􁭄􁥂􂆉􀻻􁇣

//RTT 􀔄􄅵􂳌􀝇􂱘􀦖􁭄

u32\_t rttest; //􀔄􄅵􁕫􀠄􂱘 500ms 􂓈􃄨􁭄

u32\_t rtseq; //􂫼􀑢􂌟􄆩 RTT 􂱘􀣙􂱘􁑣􀧋

s16\_t sa, sv; //RTT 􀔄􄅵􀟎􂱘􁑇􀴛􀘐􀦞􀝊􁯊􄯈􁏂

u16\_t rto; // 􄞡􀦥􄍙􁯊􁯊􄯈􀋈􀟽􂫼􀠡􄴶􂱘􀞴􀏾􀘐􄅵􃅫􀟎􁴹

u8\_t nrtx; // 􄞡􀦥􂱘􂃵􁭄􀋈􄆹􁄫􂆉􀳼􁭄􁥂􀣙􀻮􂃵􄍙􁯊􁯊􃹿􀕓􂫼􀠄􀋈􀏢􄆒􃕂 rto 􂱘􀘐􂳌􀝇

// 􁖿􄗳􄞡􀓴/􁘶􀻡􂳌􀝇􂱘􀦖􁭄

u32\_t lastack; // 􁳔􀻻􂱘􂹂􄅸􁑣􀧋􀋈􄆹􁄫􂆉􀏡􃾷

u8\_t dupacks; // 􀏞􄴶􄖭􀏾􁑣􀧋􃹿􄞡􀓴􂱘􂃵􁭄

// 􄰏􀸲􁥻􀠊􂳌􀝇􀦖􁭄

u16\_t cwnd; //􄖲􁥹􂱘􁔧􀠡􄰏􀸲􃁫􀦷

u16\_t ssthresh; // 􁜶􄗳􀨃􀡼􄯜􀘐

// 􀦥􄗕􂳌􀝇􁄫􂆉

u32\_t snd\_nxt, // 􀏟􀏔􀏾􁇚􃽕􀦥􄗕􂱘􁄫􃡖􁑣􀧋

snd\_max, // 􁳔􅂬􂱘􀦥􄗕􁄫􃡖􁑣􀧋

snd\_wnd, // 􀦥􄗕􃁫􀦷

snd\_wl1, snd\_wl2, // 􀏞􂃵􃁫􀦷􁳈􁮄􁯊􂱘􁭄􁥂􁑣􀧋􀩠􂹂􄅸􁑣􀧋

snd\_lbb; // 􀦥􄗕􄯳􀟫􀐁􁳔􀧢􀏔􀏾􁄫􃡖􂱘􁑣􀧋

u16\_t acked; //

u16\_t snd\_buf; // 􀧃􂫼􂱘􀦥􄗕􃓧􀞆􁄫􃡖􁭄

u8\_t snd\_queuelen; // 􀧃􂫼􂱘􀦥􄗕􀣙􁭄

struct tcp\_seg \*unsent; // 􁳾􀦥􄗕􂱘􁭄􁥂􂆉􄯳􀟫

struct tcp\_seg \*unacked; // 􀦥􄗕􀑚􁳾􁬊􀠄􂹂􄅸􂱘􁭄􁥂􄯳􀟫

Struct tcp\_seg \*ooseq; // 􁥹􁬊􀠄􁑣􀟫􀒹􀻪􂱘􁭄􁥂􀣙􄯳􀟫

#if LWIP\_CALLBACK\_API // 􀲲􄇗􀟑􁭄􀋈􄚼􀟚􀟑􁭄􀳼􄕗􀔢􂠜􁴀􂉵􁅮􀐝

err\_t (\* sent)(void \*arg, struct tcp\_pcb \*pcb, u16\_t space);

err\_t (\* recv)(void \*arg, struct tcp\_pcb \*pcb, struct pbuf \*p, err\_t err); // 􁭄􁥂􀣙􁥹􁬊􀲲􄇗􀟑􁭄

err\_t (\* connected)(void \*arg, struct tcp\_pcb \*pcb, err\_t err);

err\_t (\* accept)(void \*arg, struct tcp\_pcb \*newpcb, err\_t err);

err\_t (\* poll)(void \*arg, struct tcp\_pcb \*pcb);

void (\* errf)(void \*arg, err\_t err);

#endif /\* LWIP\_CALLBACK\_API \*/

// 􀠽􀏟􂱘􁠔􁳝􁄫􂆉􀳼􄕗􀔢􂠜􁴀􀐁􀴛􁳾􁅮􀐝􀋈􂫼􀠄􁯊􀝡􄆆􃾷

u32\_t keep\_idle;

#if LWIP\_TCP\_KEEPALIVE

u32\_t keep\_intvl; // 􀖱􂌏􁅮􁯊􀰼􀋈􂫼􀑢􁺔􂌟􃁎􄯆􄖲􁥹􂱘􀦺􀏔􃃃􁰃􀧺􁋽􂑗

u32\_t keep\_cnt;

#endif /\* LWIP\_TCP\_KEEPALIVE \*/

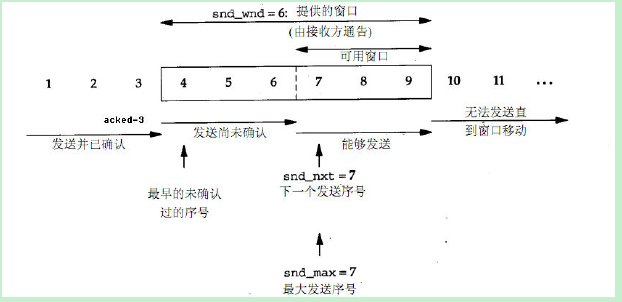
u32\_t persist\_cnt; // 􄖭􀏸􀏾􁄫􂆉􀧃􀒹􀕓􃁫􀦷􀻻􁇣􀖵􁙃􀖱􁣕􀏡􁮁􂌕􀡼

u8\_t persist\_backoff;

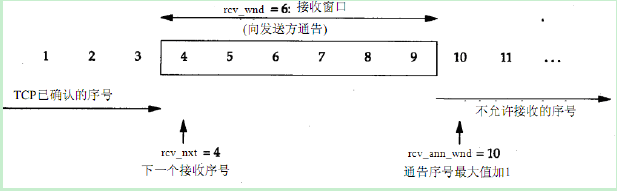
u8\_t keep\_cnt\_sent;

};

发送窗口：



接收窗口：



在接收方，rev\_wnd表示了自己接收窗口的大小，它可以在给发送方的ACK包中通告自己的窗口大小值，发送方接收到该值后，就设置自己的发送窗口snd\_wnd为该值。

## 第 17 章 TCP建立流程

tcp\_new函数：条用tcp\_alloc函数为一个连接分配一个TCP控制块tcp\_pcb。

tcp\_bind：当一个新建的PCB被初始化好后，tcp-bind函数就会被调用，将TCP控制块中的local\_ip和local\_port进行赋值。

Pcb链表有好几种：

处于侦听状态的链表tcp\_listen\_pcbs

处于稳定状态的链表tcp\_active\_pcbs

已经绑定完毕的pcb链表 tcp\_bound\_pcbs

处于TIME\_WAIT状态的PCB链表tcp\_tw\_pcbs。

tcp\_bound\_pcbs链表用来连接新创建的控制块，可以认为新建的控制块处于closed状态。

tcp\_listen\_pcbs链表用来链接处于LISTEN状态的控制块。

tcp\_tw\_pcbs链表用来连接处于TIME-WAIT状态的控制块。

tcp\_active\_pcbs链表用于连接处于TCP庄陶转换图中其它所有状态的控制块。

服务器需进入LISTEN状态等待客户端的连接。过程先将其从tcp\_bound\_\_pcbs链表上取下来，将其state字段置为LISTEN，最后再将PCB挂接到链表tcp\_listen\_pcbs上。

但事实是LWIP重新定义了tcp\_pcb\_listen的结构体，

## 第 21 章 TCP滑动窗口

## 第 22 章 TCP超时重传

TCP重传机制与TCP控制块tcp\_pcb内部的相关字段rtime、rttest、rtseq、sa、sv、rto、nrtx相关。

与超时时间间隔密切相关的是往返时间（RTT）的估计。

在某段时间内发送方可能会连续发送多个数据包，但发送方只能选择一个发送包启动定时器，估计器RTT值。

tcp\_output从unsent队列上取下第一个数据段，并调用函数tcp\_output\_segment将数据段发送出去，发送完毕后，tcp\_output将该数据段挂在unacked队列上。

## 第 23 章 TCP慢启动与拥塞避免

慢启动为发送方的TCP增加了另一个窗口：拥塞窗口（congestion window），记为cwnd。