LwIP 协议栈详解

——TCP/IP 协议的实现

E-mail:for_rest@foxmail.com 老衲五木出品

前言

最近一个项目用到 LwIP，恰好看到网上讨论的人比较多，所以有了写这篇学习笔记的冲动，一是为了打发点发呆的时间，二是为了吹过的那些 NB。往往决定做一件事是简单的，而坚持做完这件事却是漫长曲折的，但终究还是写完了，时间开销大概为四个月，内存开销无法估计。。

这篇文章覆盖了 LwIP 协议大部分的内容，但是并不全面。它主要讲解了 LwIP 协议最重要也是最常被用到的部分，包括内存管理，底层网络接口管理，ARP 层，IP 层，TCP 层，API 层等，这些部分是 LwIP 的典型应用中经常涉及到的。而 LwIP 协议的其他部分，包括 UDP，DHCP,DNS,IGMP,SNMP,PPP 等不具有使用共性的部分，这篇文档暂时未涉及。

原来文章是发在空间中的，每节每节依次更新，后来又改发为博客，再后来就干脆懒得发了。现在终于搞定，于是将所有文章汇总。绞尽脑汁的想写一段空前绝后，人见人爱的序言，但越写越觉得像是猫儿抓的一样。就这样，PS:由于本人文笔有限，情商又低，下里巴人一枚，所以文中的很多语句可能让您很纠结，您可以通过邮箱与我联系。共同探讨才是进步的关键。

最后，欢迎读者以任何方式使用与转载，但请保留作者相关信息，酱紫！码字。。。世界上最痛苦的事情莫过于此。。。

——老衲五木

目录

1 移植综述

如果你认为所谓的毅力是每分每秒的“艰苦忍耐”式的奋斗，那这是一种很不足的心理状态。毅力是一种习惯，毅力是一种状态，毅力是一种生活。看了这么久的代码觉得是不是该写点东西了，不然怎么对得起某人口中所说的科研人员这个光荣称号。初见这如山如海的代码，着实看出了一身冷汗。现在想想其实也不是那么难，那么多革命先辈经过 N 长时间才搞出来的东东怎么可能让你个毛小子几周之内搞懂。我见到的只是冰川的一小角，万里长征的一小步，九头牛身上的一小毛…再用某人的话说，写吧，昏写，瞎写，胡写，乱写，写写就懂了。

我想我很适合当一个歌颂者，青春在风中飘着。你知道，就算大雨让这座城市颠倒，我会给你怀抱；受不了，看见你背影来到，写下我度秒如年难捱的离骚；就算整个世界被寂寞绑票，我也不会奔跑；逃不了，最后谁也都苍老，写下我，时间和琴声交错的城堡。我正在听的歌。扯远了…

正题，嵌入式产品连入 Internet 网，这个 MS 是个愈演愈烈的趋势。想想，你可以足不出户对你的产品进行配置，并获取你关心的数据信息，多好。这也许也是物联网世界最基本的雏形。当然，你的产品要有如此功能，那可不容易，至少它得有个目前很 Fashion 的 TCP/IP 协议栈。LWIP 是一套用于嵌入式系统的开放源代码 TCP/IP 协议栈。在你的嵌入式处理器不是很 NB，内部 Flash 和 Ram 不是很强大的情况下，用它还是很合适滴。

LWIP 的设计者为像我这样的懒惰者提供了详细的移植说明文档，当然这还不够，他们还尽可能的包揽了大部分工作，懒人们只要做很少的工作就功德圆满了。纵观整个移植过程，使用者需要完成以下几个方面的东西：

首先是 LWIP 协议内部使用的数据类型的定义，如 u8_t，s8_t，u16_t，u32_t 等等等等。由于所移植平台处理器的不同和使用的编译器的不同，这些数据类型必须重新定义。想想，一个 int 型数据在 64 位处理器上其长度为 8 个字节，在 32 位处理器上为 4 个字节，而在 16 位处理器上就只有两个字节了。因此这部分需要使用者根据处理器位数和和使用的编译器的特点来编写。所以在 ARM7 处理器上使用的 typedef unsigned int u32_t 移植语句用在 64 位处理器的移植过程中那肯定是行不通的了。

其次是实现与信号量和邮箱操作相关的函数，比如建立、删除、等待、释放等。如果在裸机上直接跑 LWIP，这点实现起来比较麻烦，使用者必须自己去建立一套信号量和邮箱相关的机制。一般情况下，在使用 LWIP 的嵌入式系统中都会有操作系统的支持，而在操作系统中信号量和邮箱往往是最基本的进程通信机制了。UC/OSII 应该算是最简单的嵌入式操作系统了吧，它也无例外的能够提供信号量和邮箱机制，只要我们将 UC/OSII 中的相关函数做相应的封装，就可满足 LWIP 的需求。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 协议模拟了 TCP/IP 协议的分层思想，表面上看 LWIP 也是有分层思想的，但从实现上看，LWIP 只在一个进程内实现了各个层次的所有工作。具体如下：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底层硬件驱动接收到的数据包，也可能来自应用程序。当在该邮箱内取得数据后，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。处理结束后，LWIP 继续阻塞在邮箱上等待下一批数据。当然 LWIP 还有一大串的内存管理机制用以避免在各层间交互数据时大量的时间和内存开销，这将在后续讲解中慢慢道来。当然，但这样的设计使得代码理解难度加大，这一点让人头大。信号量也

可以用在应用程序与协议栈的互相通信中。比如，应用程序要发送数据了，它先把数据发到LWIP 阻塞的邮箱上，然后它挂起在一个信号量上；LWIP 从邮箱上取得数据处理后，释放一个信号量，告诉应用程序，你要发的数据我已经搞定了；此后，应用程序得到信号量继续运行，而 LWIP 继续阻塞在邮箱上等待下一批处理数据。

其其次，就是与等待超时相关的函数。上面说到 LWIP 协议栈会阻塞在邮箱上等待接收数据的到来。这种等待在外部看起来是一直进行的，但其实不然。一般在初始化 LWIP 进程的时候，都会同时的初始化一些超时事件，即当某些事件等待超时后，它们会自动调用一些超时处理函数做相关处理，以满足 TCP/IP 协议栈的需求。这样看来，当 LWIP 协议栈阻塞等待邮箱之前，它会精明的计算到底应该等待多久，如果 LWIP 进程中没有初始化任何超时事件，那好，这种情况最简单了，永远的挂起进程就可以了，这时的等待就可以看做是天长地久的….有点暧昧了。如果 LWIP 进程中有初始化的超时事件，这时就不能一直等了，因为这样超时事件没有任何被执行的机会。LWIP 是这样做的，等待邮箱的时间设置为第一个超时事件的时间长度，如果时间到了，还没等到数据，那好，直接跳出邮箱等待转而执行超时事件，当执行完成超时事件后，再按照上述的方法继续阻塞邮箱。可以看出，对一个 LWIP 进程，需要用一个链表来管理这些超时事件。这个链表的大部分工作已经被 LWIP 的设计者完成了，使用者只需要实现的仅有一个函数：该函数能够返回当前进程个超时事件链表的首地址。LWIP 内部协议要利用该首地址来查找完成相关超时事件。

其其其次，如果 LWIP 是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。不支持多线程的操作系统，汗…表示还没听过。不过 UC/OSII 显然是支持多线程的，地球人都知道。这样一个典型的 LWIP 应用系统包括这样的三个进程：首先启动的是上层应用程序进程，然后是 LWIP 协议栈进程，最后是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。注意 LWIP 协议栈进程一般具有最高的优先级，以便实时正确的对数据进行响应。

其其其其次，其他一些细节之处。比如临界区保护函数，用于 LWIP 协议栈处理某些临界区时使用，一般通过进临界区关中断、出临界区开中断的方式来实现；又如结构体定义时用到的结构体封装宏，LWIP 的实现基于这样一种机制，即上层协议已经明确知道了下层所传上来的数据的结构特点，上层直接使用相关取地址计算得到想要的数据，而避免了数据递交时的复制与缓冲，所以定义结构体封装宏，禁止编译器的地址自动对齐是必须的；还有诸如调试输出、测量记录方面的宏不做讲解。

最后，也是比较重要的地方。底层网络驱动函数的实现。这取决于你嵌入式硬件系统所使用的网络接口芯片，也就是网卡芯片，常见的有 RTL8201BL、ENC28J60 等等。不同的接口芯片厂商都会提供丰富的驱动函数。我们只要将这些发送接收接口函数做相应的封装，将接收到得数据包封装为 LWIP 协议栈熟悉的数据结构、将发送的数据包分解为芯片熟悉的数据结构就基本搞定了。最起码的，发送一个数据包函数和接收一个数据包函数需要被实现。

那就这样了吧，虽然写得草草，但终于在撤退之前搞定。好的开始是成功的一半，那这暂且先算四分之一吧。不晓得一个月、两个月或者更多时间能写完否。预知后事如何，请见下回分解。

2 动态内存管理

最近电力局很不给力啊，隔三差五的停电，害得我们老是痛苦的双扣斗地主，不带这样的啊！今天还写吗？写，必须的。

昨天把 LWIP 的移植工作框架说了一下，网上也有一大筐的关于移植细节的文档。有兴趣的童鞋不妨去找找。这里，我很想探究 LWIP 内部协议实现的细节，以及所有盘根错节的问题的来龙去脉。以后的讨论研究将按照 LWIP 英文说明文档《Design and Implementation of the LWIP：TCP/IP Stack》的结构组织展开。

这里讨论 LWIP 的动态内存管理机制。

总的来说，LWIP 的动态内存管理机制可以有三种：C 运行时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。

动态内存堆分配策略和 C 运行时库自带的内存分配策略具有很大的相似性，这是 LWIP 模拟运行时库分配策略实现的。这两种策略使用者只能从中选择一种，这通过头文件 lwippools.h 中的宏定义 MEM_LIBC_MALLOC 来实现的，当它被定义为 1 时则使用标准 C 运行时库自带的内存分配策略，而为 0 时则使用 LWIP 自身的动态内存堆分配策略。一般情况下，我们选择使用 LWIP 自身的动态内存堆分配策略，这里不对 C 运行时库自带的内存分配策略进行讨论。

同时，动态内存堆分配策略可以有两种实现方式，纠结….第一种就是如前所述的通过开辟一个内存堆，然后通过模拟 C 运行时库的内存分配策略来实现。第二种就是通过动态内存池的方式来实现，也即动态内存堆分配函数通过简单调用动态内存池(POOL)分配函数来完成其功能(太敷衍了事了)，在这种情况下，用户需要在头文件 lwippools.h 中定义宏 MEM_USE_POOLS 和 MEM_USE_CUSTOM_POOLS 为 1，同时还要开辟一些额外的缓冲池区，如下：

```
LWIP_MALLOC_MEMPOOL_START
LWIP_MALLOC_MEMPOOL(20, 256)
LWIP_MALLOC_MEMPOOL(10, 512)
LWIP_MALLOC_MEMPOOL(5, 1512)
LWIP_MALLOC_MEMPOOL_END
```

这几句摘自 LWIP 源码注释部分，表示为动态内存堆相关功能函数分配 20 个 256 字节长度的内存块，10 个 512 字节的内存块，5 个 1512 字节的内存块。内存池管理会根据以上的宏自动在内存中静态定义一个大片内存用于内存池。在内存分配申请的时候，自动根据所请求的大小，选择最适合他长度的池里面去申请，如果启用宏 MEM_USE_POOLS_TRY_BIGGER_POOL，那么，如果上述的最适合长度的池中没有空间可以用了，分配器将从更大长度的池中去申请，不过这样会浪费更多的内存。晕乎乎…..就这样了，这种方式一般不会被用到。哎，就最后这句话给力。

下面讨论动态内存堆分配策略的第一种实现方式，这也是一般情况下被使用的方式。这部分讨论主要参照网上 Oldtom’s Blog，TA 写得很好(但是也有一点小小的错误)，所以一不小心被我借用了。

动态内存堆分配策略原理就是在一个事先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（First Fit）方式，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。分配的内存块有个最小大小的限

制，要求请求的分配大小不能小于 MIN_SIZE，否则请求会被分配到 MIN_SIZE 大小的内存空间。一般 MIN_SIZE 为 12 字节，在这 12 个字节中前几个字节会存放内存分配器管理用的私有数据，该数据区不能被用户程序修改，否则导致致命问题。内存释放的过程是相反的过程，但分配器会查看该节点前后相邻的内存块是否空闲，如果空闲则合并成一个大的内存空闲块。采用这种分配策略，其优点就是内存浪费小，比较简单，适合用于小内存的管理，其缺点就是如果频繁的动态分配和释放，可能会造成严重的内存碎片，如果在碎片情况严重的话，可能会导致内存分配不成功。对于动态内存的使用，比较推荐的方法就是分配->释放->分配->释放，这种使用方法能够减少内存碎片。下面具体来看看 LWIP 是怎么来实现这些函数的。

mem_init( ) 内存堆的初始化函数，主要是告知内存堆的起止地址，以及初始化空闲列表，由 lwip 初始化时自己调用，该接口为内部私有接口，不对用户层开放。

mem_malloc( ) 申请分配内存。将总共需要的字节数作为参数传递给该函数，返回值是指向最新分配的内存的指针，而如果内存没有分配好，则返回值是 NULL，分配的空间大小会收到内存对齐的影响，可能会比申请的略大。返回的内存是“没有“初始化的。这块内存可能包含任何随机的垃圾，你可以马上用有效数据或者至少是用零来初始化这块内存。内存的分配和释放，不能在中断函数里面进行。内存堆是全局变量，因此内存的申请、释放操作做了线程安全保护，如果有多个线程在同时进行内存申请和释放，那么可能会因为信号量的等待而导致申请耗时较长。

mem_calloc( ) 是对 mem_malloc( )函数的简单包装，他有两个参数，分别为元素的数目和每个元素的大小，这两个参数的乘积就是要分配的内存空间的大小，与 mem_malloc()不同的是它会把动态分配的内存清零。有经验的程序员更喜欢使用 mem_ calloc ()，因为这样的话新分配内存的内容就不会有什么问题，调用 mem_ calloc ()肯定会清 0，并且可以避免调用 memset()。

休息……

动态内存池(POOL)分配策略可以说是一个比较笨的分配策略了，但其分配策略实现简单，内存的分配、释放效率高，可以有效防止内存碎片的产生。不过，他的缺点是会浪费部分内存。

为什么叫 POOL？这点很有趣，POOL 有很多种，而这点依赖于用户配置 LWIP 的方式。例如用户在头文件 opt.h 文件中定义 LWIP_UDP 为 1，则在编译的时候与 UDP 类型内存池就会被建立；定义 LWIP_TCP 为 1，则在编译的时候与 TCP 类型内存池就会被建立。另外，还有很多其他类型的内存池，如专门存放网络包数据信息的 PBUF_POOL、还有上面讲解动态内存堆分配策略时提到的 CUSTOM_POOLS 等等等等。某种类型的 POOL 其单个大小是固定的，而分配该类 POOL 的个数是可以用户配置的，用户应该根据协议栈实际使用状况进行配置。把协议栈中所有的 POOL 挨个放到一起，并把它们放在一片连续的内存区域，这呈现给用户的就是一个大的缓冲池。所以，所谓的缓冲池的内部组织应该是这样的：开始处放了 A 类型的 POOL 池 a 个，紧接着放上 B 类型的 POOL 池 b 个，再接着放上 C 类型的 POOL 池 c 个….直至最后 N 类型的 POOL 池 n 个。这一点很像 UC/OSII 中进程控制块和事件控制块，先开辟一堆各种类型的放那，你要用直接来取就是了。注意，这里的分配必须是以单个缓冲池为基本单位的，在这样的情况下，可能导致内存浪费的情况。这是很明显的啊，不解释。

下面我来看看在 LWIP 实现中是怎么开辟出上面所论述的大大的缓冲池的(‘的’这个字，今天让我们一群人笑了很久)。基本上绝大部分人看到这部分代码都会被打得晕头转向，完全不晓得作者是在干啥，但是仔细理解后，你不得不佩服作者超凡脱俗的代码写能力，差一点用了沉鱼落雁这个词，罪过。上代码：

```
static u8_t memp_memory [ MEM_ALIGNMENT - 1
#define LWIP_MEMPOOL(name,num,size,desc) + ( (num) * (MEMP_SIZE + MEMP_ALIGN_SIZE(size) ) )
#include "lwip/memp_std.h"
];
```

上面的代码定义了缓冲池所使用的内存缓冲区，很多人肯定会怀疑这到底是不是一个数组的定义。定义一个数组，里面居然还有 define 和 include 关键字。解决问题的关键就在于头文件 memp_std.h，它里面的东西可以被简化为诸多条 LWIP_MEMPOOL(name,num,size,desc)。又由于用了 define 关键字将 LWIP_MEMPOOL(name,num,size,desc)定义为 +((num) * (MEMP_SIZE + MEMP_ALIGN_SIZE(size)))，所以，memp_std.h 被编译后就为一条一条的+（），+（），+（），+（）….所以最终的数组 memp_memory 等价定义为：

```
static u8_t memp_memory [ MEM_ALIGNMENT – 1
                          +（）
                          +（）….];
```

如果此刻你还没懂，只能说明我的表述能力有问题了。当然还有个小小的遗留问题，为什么数组要比实际需要的大 MEM_ALIGNMENT – 1？作者考虑的是编译器的字对齐问题，到此打住，这个问题不能深究啊，以后慢慢讲。

复制上面的数组建立的方法，协议栈还建立了一些与缓冲池管理的全局变量：

memp_num：这个静态数组用于保存各种类型缓冲池的成员数目

memp_sizes：这个静态数组用于保存各种类型缓冲池的结构大小

memp_tab：这个指针数组用于指向各种类型缓冲池当前空闲节点

接下来就是理所当然的实现函数了：

memp_init（）：内存池的初始化，主要是为每种内存池建立链表 memp_tab，其链表是逆序的，此外，如果有统计功能使能的话，也把记录了各种内存池的数目。

memp_malloc（）：如果相应的 memp_tab 链表还有空闲的节点，则从中切出一个节点返回，否则返回空。

memp_free（）：把释放的节点添加到相应的链表 memp_tab 头上。

从上面的三个函数可以看出，动态内存池分配过程时相当的简洁直观啊。

HC:百度说是胡扯的意思。哈哈……

3 数据包 pbuf

高的地方，总是很冷。孤独，可以让人疯狂。没人能懂你！昨天讲过了 LWIP 的内存分配机制。再来总之一下，LWIP 中常用到的内存分配策略有两种，一种是内存堆分配，一种是内存池分配。前者可以说能随心所欲的分配我们需要的合理大小的内存块(又是‘的’)，缺点是当经过多次的分配释放后，内存堆中间会出现很多碎片，使得需要分配较大内存块时分配失败；后者分配速度快，就是简单的链表操作，因为各种类型的 POOL 是我们事先建立好的，但是采用 POOL 会有些情况下会浪费掉一定的内存空间。在 LWIP 中，将这两种分配策略混合使用，达到了很好的内存使用效率。

下面我们将来看看 LWIP 中是怎样合理利用这两种分配策略的。这就顺利的过渡到了这节要讨论的话题：LWIP 的数据包缓冲的实现。

在协议栈中移动的数据包，最无疑的是整个内存管理中最重要的部分了。数据包的种类和大小也可以说是五花八门，数数，首先从网卡上来的原始数据包，它可以是长达上千个字节的 TCP 数据包，也可以是仅有几个字节的 ICMP 数据包；再从要发送的数据包看，上层应用可能将自己要发送的千奇百怪形态各异的数据包递交给 LWIP 协议栈发送，这些数据可能存在于应用程序管理的内存空间内，也可能存在于某个 ROM 上。注意，这里有个核心的东西是当数据在各层之间传递时，LWIP 极力禁止数据的拷贝工作，因为这样会耗费大量的时间和内存。综上，LWIP 必须有个高效的数据包管理核心，它即能海纳百川似的兼容各种类型的数据，又能避免在各层之间的复制数据的巨大开销。

数据包管理机构采用数据结构 pbuf 来描述数据包，其源码如下，

```
struct pbuf {

struct pbuf *next;

void *payload;

u16_t tot_len;

u16_t len;

u8_t  type;

u8_t flags;

u16_t ref;

};
```

这个看似简单的数据结构，却够我讲一大歇的了！next 字段指针指向下一个 pbuf 结构，因为实际发送或接收的数据包可能很大，而每个 pbuf 能够管理的数据可能很少，所以，往往需要多个 pbuf 结构才能完全描述一个数据包。所以，所有的描述同一个数据包的 pbuf 结构

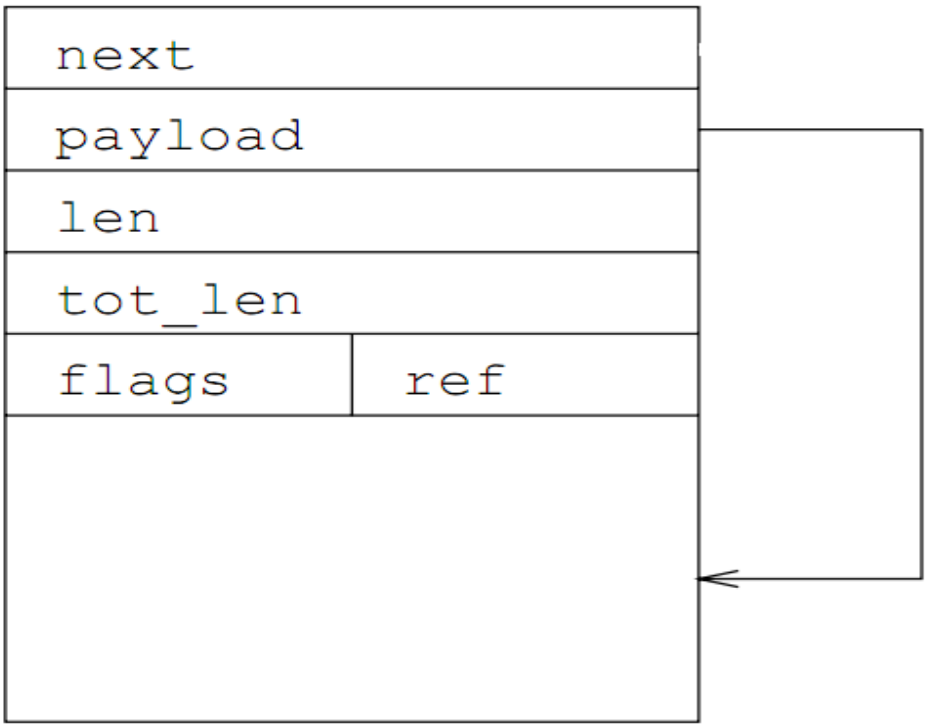
需要链在一个链表上，这一点用 next 实现。payload 是数据指针，指向该 pbuf 管理的数据的起始地址，这里，数据的起始地址可以是紧跟在 pbuf 结构之后的 RAM，也可能是在 ROM 上的某个地址，而决定这点的是当前 pbuf 是什么类型的，即 type 字段的值，这在下面将继续讨论。len 字段表示当前 pbuf 中的有效数据长度，而 tot_len 表示当前 pbuf 和其后所有 pbuf 的有效数据的长度。显然，tot_len 字段是 len 字段与 pbuf 链中随后一个 pbuf 的 tot_len 字段的和；pbuf 链中第一个 pbuf 的 tot_len 字段表示整个数据包的长度，而最后一个 pbuf 的 tot_len 字段必和 len 字段相等。type 字段表示 pbuf 的类型，主要有四种类型，这点基本上涉及到 pbuf 管理中最难的部分，将在下节仔细讨论。文档上说 flags 字段也表示 pbuf 的类型，不懂，type 字段不是说明了 pbuf 的类型吗？不过在源代码里，初始化一个 pbuf 的时候，是将该字段的值设为 0，而在其他地方也没有用到该字段，所以，这里直接忽略掉。最后 ref 字段表示该 pbuf 被引用的次数。这里又是一个纠结的地方啊。初始化一个 pbuf 的时候，ref 字段值被设置为 1，当有其他 pbuf 的 next 指针指向该 pbuf 时，该 pbuf 的 ref 字段值加一。所以，要删除一个 pbuf 时，ref 的值必须为 1 才能删除成功，否则删除失败。

pbuf 的类型，令人很晕的东西。pbuf 有四类：PBUF_RAM、PBUF_ROM、PBUF_REF 和 PBUF_POOL。下面，一个一个的来看看各种类型的特点。

PBUF_RAM 类型的 pbuf 主要通过内存堆分配得到的。这种类型的 pbuf 在协议栈中是用得最多的。协议栈要发送的数据和应用程序要传递的数据一般都采用这个形式。申请 PBUF_RAM 类型时，协议栈会在内存堆中分配相应的大小，注意，这里的大小包括如前所述的 pbuf 结构头大小和相应数据缓冲区，他们是在一片连续的内存区的。下面来看看源代码是怎样申请 PBUF_RAM 型的。其中 p 是 pbuf 型指针。

```
p = (struct pbuf*)mem_malloc(LWIP_MEM_ALIGN_SIZE(SIZEOF_STRUCT_PBUF + offset)
+ LWIP_MEM_ALIGN_SIZE(length));
```

可以看出，系统是调用内存堆分配函数 mem_malloc 进行内存分配的。分配空间的大小包括：pbuf 结构头大小 SIZEOF_STRUCT_PBUF，需要的数据存储空间大小 length，还有一个 offset。关于这个 offset，也有一大堆可以讨论的东西，不过先到此打住。总之，分配成功的 PBUF_RAM 类型的 pbuf 如下图：

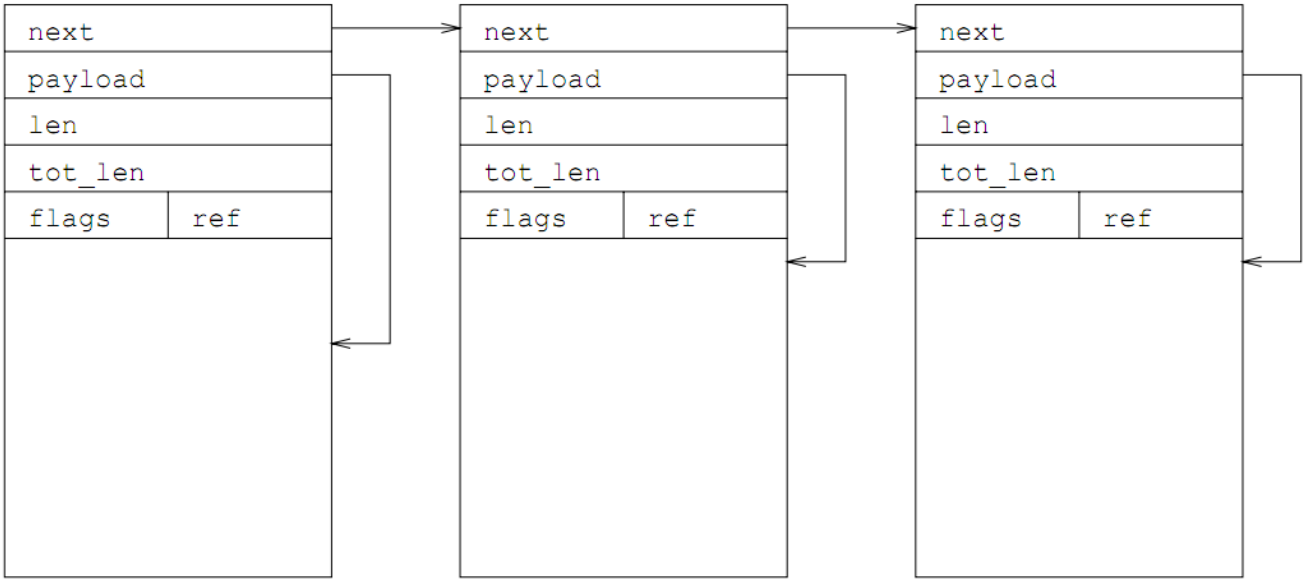


从图中可看出 pbuf 头和相应数据在同一片连续的内存区种，注意 payload 并没有指向 pbuf 头结束即 ref 字段之后，而是隔了一定的区域。这段区域就是上面的 offset 的大小，这段区域用来存储数据的包头，如 TCP 包头，IP 包头等。当然，offset 也可以是 0，具体值是多少，那就要看你是怎么个申请法了。如果还要深究，你肯定会更晕了。

PBUF_POOL 类型和 PBUF_RAM 类型的 pbuf 有很大的相似之处，但它主要通过内存池分配得到的。这种类型的 pbuf 可以在极短的时间内得到分配。在接受数据包时，LWIP 一般采用这种方式包装数据。申请 PBUF_POOL 类型时，协议栈会在内存池中分配适当的内存池个数以满足需要的申请大小。下面来看看源代码是怎样申请 PBUF_POOL 型的。其中 p 是 pbuf 型指针。

```
p = memp_malloc(MEMP_PBUF_POOL);
```

可以看出，系统是调用内存池分配函数 memp_malloc 进行内存分配的。分配成功的 PBUF_POOL 类型的 pbuf 如下图：

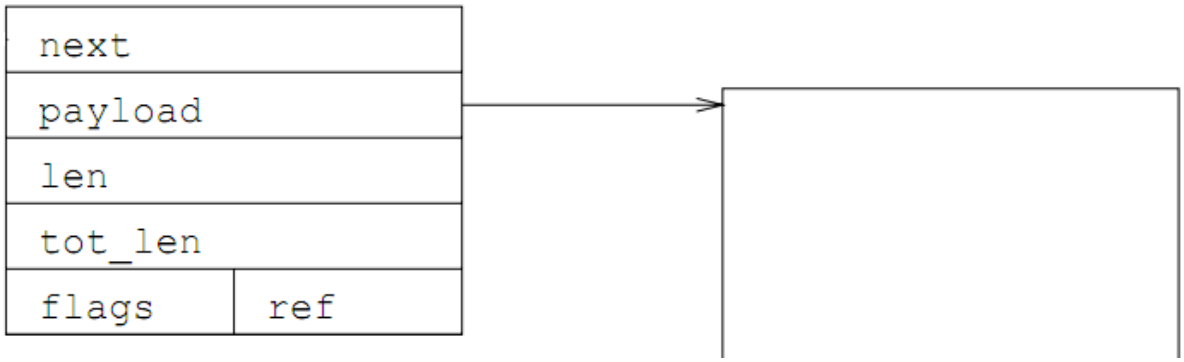


图中是分配指定大小的数据缓冲的结果，系统调用会分配多个固定大小的 PBUF_POOL 类型 pbuf，并把这些 pbufs 链成一个链表，以满足用户的分配空间请求。

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相应的 pbuf 结构头，而不申请数据区的空间。这就是它们与 PBUF_RAM 和 PBUF_POOL 的最大区别。PBUF_ROM 和 PBUF_REF 类型的区别在于前者指向 ROM 空间内的某段数据，而后者指向 RAM 空间内的某段数据。下面来看看源代码是怎样申请 PBUF_ROM 和 PBUF_REF 类型的。其中 p 是 pbuf 型指针。

```
p = memp_malloc(MEMP_PBUF);
```

可以看出，系统是调用内存池分配函数 memp_malloc 进行内存分配的。而此刻请求的内存池类型为 MEMP_PBUF，而不是 MEMP_PBUF_POOL，晕啊…这个太让人郁闷了。MEMP_PBUF 类型的内存池大小恰好为一个 pbuf 头的大小，因为这种池是 LWIP 专为 PBUF_ROM 和 PBUF_REF 类型的 pbuf 量身制作的。LWIP 还是真的很周到啊，它会为不同的数据结构量身定做不同类型的池。正确分配的 PBUF_ROM 或 PBUF_REF 类型的 pbuf，其结构如下图：



注：以上所有图片都来自文档《Design and Implementation of the LWIP：TCP/IP Stack》，这些图都有个共同的错误，即 len 和 tot_len 字段位置搞反了，窃喜。

最后说明，对于一个数据包，它可能使用上述的任意的 pbuf 类型，很可能的情况是，一大串不同类型的 pbufs 连在一起，用以保存一个数据包的数据。

广告……

下节看点，关于 pbuf 的内存释放问题。

4 pbuf 释放

牢骚发完，Go On。昨天说到了数据缓冲 pbuf 的内存申请，今天继续来探究一下它的内存释放过程。由于 pbuf 的申请主要是通过内存堆分配和内存池分配来实现，所以，pbuf 的释放也必须按照这两种情况分别讨论。

别慌，在展开讨论之前，还得说说某个 pbuf 能被释放的前提。在 LWIP 中这点很容易判断，因为前节说到 pbuf 的 ref 字段表示该 pbuf 被引用的次数，当 pbuf 被创建时，该字段的初始值为 1，由此可判断，当 pbuf 的 ref 字段为 1 时，该 pbuf 才可以被删除，所以位于 pbufs 链表中间的 pbuf 结构是不会被删除成功的，因为他们的 ref 值至少是 2。由此总之一下，能被删除的 pbuf 必然是某个 pbufs 链的首节点。当然 pbuf 的删除工作远不如此的简单，其中另一个需要特别注意的地方是，想想，很可能的情况是某个 pbufs 链的首节点删除成功后，该 pbufs 链的第二个节点就自然的成为该 pbufs 链的首节点，此时，该节点的 ref 值可能变为 1(该节点没有被引用了)，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数据包。当第二个节点也被删除后，LWIP 又会去看看第三个节点是否满足删除条件…就这样一直删一下去。当然，如果首节点删除后，第二个节点的 ref 值大于 1，表示该节点还在其他地方被引用，不能再被删除，删除工作至此结束。这段话写的很口水，不如我们举个例子来看看这个删除过程。假如现在我们的 pbufs 链表有 A，B，C 三个 pbuf 结构连接起来，结构为 A--->B--->C，利用 pbuf_free(A)函数来删除 pbuf 结构，下面用 ABC 的几组不同 ref 值来看看删除结果：

（1）1->2->3 函数执行后变为 ...1->3，节点 BC 仍在；

（2）3->3->3 函数执行后变为 2->3->3，节点 ABC 仍在；

（3）1->1->2 函数执行后变为.....1，节点 C 仍在；

（4）2->1->1 函数执行后变为 1->1->1，节点 ABC 仍在；

（5）1->1->1 函数执行后变为.....，节点全部被删除。

如果您能说醍醐灌顶，那将是我最大的动力。

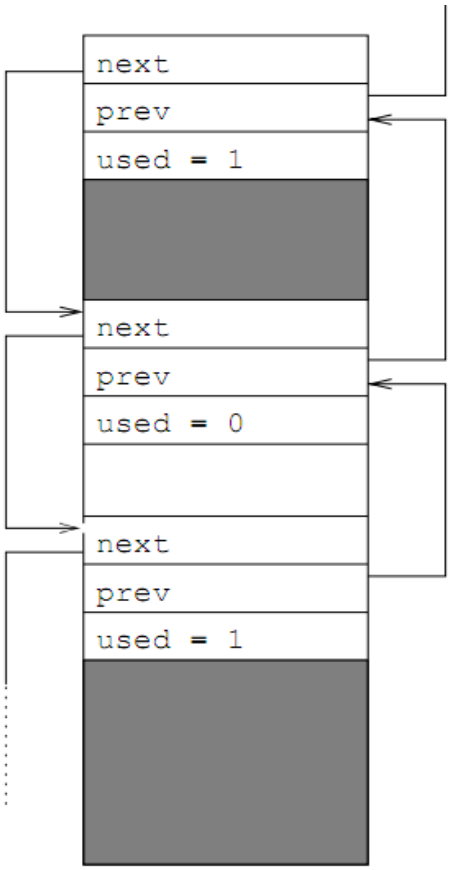
当可以删除某个 pbuf 结构时，LWIP 首先检查这个 pbuf 是属于前节讲到的四个类型中的哪种，根据类型的不同，调用不同的内存释放函数进行删除。PBUF_POOL 类型和 PBUF_ROM 类型、PBUF_REF 类型需要通过 memp_free()函数删除，PBUF_RAM 类型需要通过 mem_free()函数删除，原因不解释。

PBUF_RAM 类型来自于内存堆，所以需通过 mem_free()函数将 pbuf 释放回内存堆。这里，先得来看看内存堆的组织结构，见下图，在内存堆内部，内存堆管理模块通过在每一个内存分配块的顶部放置一个比较小的结构体来保存内存分配纪录(注意这个小小的结构体是内存管理模块自动附加上去的，独立于用户的申请大小)。这个结构体拥有三个成员变量，两个指针和一个标志，如图。next 与 prev 分别指向内存的下一个和上一个分配块，used 标志表示该内存块是否已被分配。图中需要注意的两个地方，first，图中每个内存块的大小是不同且可能随时变化的。Second，当系统初始化的时候，整个内存堆就是一个内存块，下图中是经过多次分配释放后内存堆呈现出来的结果。

内存堆管理模块根据所申请分配的大小来搜索所有未被使用的内存分配块，检索到的最先满足条件的内存块将分配给申请者，注意这里并不包括前面说到的那个小小结构体，所以用户得到的是 used 后的那个地址。当分配成功后，内存管理模块会马上在已经分配走了的数据区后面再插一个小小的结构体，并用 next 和 prev 指针将这个结构体串起来，以便于

下次分配。经过几次的申请与释放，我们就看到了图中的内存堆组织模型。

当内存释放的时候，为了防止内存碎片的产生，上一个与下一个分配块的使用标志会被检查，如果他们中的任何一个还未被使用，这个内存块将被合并到一个更大的未使用内存块中。内存堆管理模块是这样做的，它根据用户提供的释放地址退后几个字节去寻找这个小小的结构体，利用这个结构体来实现内存堆得合并等操作。已经分配的内存块被回收后，使用标志 used 清零。当然，如果上一个与下一个分配块都已被使用，这时的释放就是最简单的情况，但这也是产生内存堆碎片问题的根源。



哎，要了老命了。

接着来讲其他三种结构通过 memp_free()函数将 pbuf 释放回内存池的情况。前面的内容已经讲过了内存池 POOL 的结构，PBUF_POOL 型 pbuf 主要使用的是 MEMP_PBUF_POOL 类型的 POOL，PBUF_ROM 和 PBUF_REF 型 pbuf 主要使用的是 MEMP_PBUF 型的 POOL。这句话太绕了，你应该多读两遍。POOL 结构的起始处有个 next 指针，用于指向同类型的下一个 POOL，用于将同类型的 POOL 连接成一个单向链表，这里应该有必要仔细看看 POOL 池是怎样初始化的，代码很简单：

```
memp = LWIP_MEM_ALIGN(memp_memory);
for (i = 0; i < MEMP_MAX; ++i) {        //对各种类型的 POOL 依次操作
  memp_tab[i] = NULL;                   //空闲链表头初始为空
  for (j = 0; j < memp_num[i]; ++j) {   //把同类 POOL 链成链表
    memp->next = memp_tab[i];
    memp_tab[i] = memp;
    memp = (struct memp *)((u8_t *)memp + MEMP_SIZE + memp_sizes[i]);//取得下
  }       //一个 POOL 的地址
}
```

上面代码中有几个重要的全局变量，memp_memory 是缓冲池的起始地址，前面已有所讨论；MEMP_MAX 是 POOL 类型数；memp_tab 用于指向某类 POOL 空闲链表的起始节点；memp_num 表示各种类型 POOL 的个数；memp_sizes 表示各种类型单个 POOL 的大小，对于 MEMP_PBUF_POOL 和 MEMP_PBUF 型的 POOL，其大小是 pbuf 头和 pbuf 可装载数据大小的总和。

在这样的基础之上，POOL 池的释放就简单了，首先根据 POOL 的类型找到相应空闲链表头 memp_tab，将该 POOL 插在链表头上，并把 memp_tab 指向链表头，简单快捷。至于 POOL 池的的申请那自然而然的也就是对 memp_tab 头的操作了，这个相信你懂。

好了，到这里，LWIP 的内存相关机制就基本介绍完毕。当然，它不可能这样简单，在以后使用到的地方，我会再加说明。

整理整理，收工。周末来啦….冬眠一天是必不可少的….哈哈

5 网络接口结构

我只是不想，将这份心动付诸言语。前面还有一句：信任他人，并不意味着软弱。我只是假装对万物一无所知，好借此获得你所有的温柔。谢谢你所做的一切，现在一切又将重新开始。我只有将这份无法忘怀的思念送给你。

人们总说”黑夜会过去”，但那只是善意的谎言。我想就算一个人，应该也能生存下去，因为你的笑容已经永远铭刻在我心中，还有那应该已经被我舍弃的信任别人的心。

以上内容系剽窃于某某美女的歌词。

今天我们来讨论 LWIP 是怎样来处理与底层硬件，即网卡芯片间的关系的。为什么要首先讨论这个问题呢？与许多其他的 TCP／IP 实现一样，LWIP 也是以分层的协议为参照来设计实现 TCP／IP 的。LWIP 从逻辑上看分为四层：链路层、网络层、传输层和应用层。注意，虽然 LWIP 也采用了分层机制，但它没有在各层之间进行严格的划分，各层协议之间可以进行或多或少的交叉存取，即上层可以意识到下层协议所使用的缓存处理机制。因此各层可以更有效地重用缓冲区。而且，应用进程和协议栈代码可以使用相同的内存，应用可以直接读写内部缓存，因此节省了执行拷贝的开销。 我们将从 LWIP 的最底层链路层起步，开始整个 LWIP 内部协议之旅。

在 LWIP 中，是通过一个叫做 netif 的网络结构体来描述一个硬件网络接口的。这个接口结构比较简单，下面我们从源代码结构来分析分析这个结构：

```
struct netif {
  struct netif *next;     // 指向下一个 netif 结构的指针
  struct ip_addr ip_addr;         // IP 地址相关配置
  struct ip_addr netmask;
  struct ip_addr gw;

  err_t (* input)(struct pbuf *p, struct netif *inp);    //调用这个函数可以从网卡上取得一个
                                                         // 数据包
  err_t (* output)(struct netif *netif, struct pbuf *p,    // IP 层调用这个函数可以向网卡发送
        struct ip_addr *ipaddr);                           // 一个数据包

  err_t (* linkoutput)(struct netif *netif, struct pbuf *p);    // ARP 模块调用这个函数向网
                                                                // 卡发送一个数据包
  void *state;     // 用户可以独立发挥该指针，用于指向用户关心的网卡信息
  u8_t hwaddr_len; // 硬件地址长度，对于以太网就是 MAC 地址长度，为 6 各字节
  u8_t hwaddr[NETIF_MAX_HWADDR_LEN];     //MAC 地址
  u16_t mtu;     // 一次可以传送的最大字节数，对于以太网一般设为 1500
  u8_t flags;     // 网卡状态信息标志位

  char name[2]; // 网络接口使用的设备驱动类型的种类
  u8_t num;       // 用来标示使用同种驱动类型的不同网络接口
};
```

next 字段是指向下一个 netif 结构的指针。我们的一个产品可能会有多个网卡芯片，LWIP

会把所有网卡芯片的结构体链成一个链表进行管理，有一个 netif_list 的全局变量指向该链表的头部。next 字段就是用于链表用。

ip_addr、netmask、gw 三个字段用于发送和处理数据包用，分别表示 IP 地址、子网掩码和网关地址。前两个字段在数据包发送时有重要作用，第三个字段似乎没什么用。IP 地址和网卡设备必须一一对应。如果你连什么叫 IP 地址、子网掩码和它们的作用都不晓得，那你有必要去看看 TCP/IP 协议详解卷 1 第三章。

input 字段指向一个函数，这个函数将网卡设备接收到的数据包提交给 IP 层，使用时将 input 指针指向该函数即可，后面将详细讨论这个问题。该函数的两个参数是 pbuf 类型和 netif 类型的，返回参数是 err_t 类型。其中 pbuf 代表接收到的数据包。

output 字段向一个函数，这个函数和具体网络接口设备驱动密切相关，它用于 IP 层将一个数据包发送到网络接口上。用户需要根据实际网卡编写该函数，并将 output 字段指向该函数。该函数的三个参数是 pbuf 类型、netif 类型和 ip_addr 类型，返回参数是 err_t 类型。其中 pbuf 代表要发送的数据包。ipaddr 代表网卡需要将该数据包发送到的地址，该地址应该是接收实际的链路层帧的主机的 IP 地址，而不一定为数据包最终需要到达的 IP 地址。例如，当要发送 IP 信息包到一个并不在本地网络里的主机上时，链路层帧会被发送到网络里的一个路由器上。在这种情况下，给 output 函数的 IP 地址将是这个路由器的地址。

linkoutput 字段和上面的 output 基本上是起相同的作用，但是这个函数是在 ARP 模块中被调用的，这里不赘述了。注意这个函数只有两个参数。实际上 output 字段函数的实现最终还是调用 linkoutput 字段函数将数据包发送出去的。

state 字段可以指向用户关心的关于设备的一些信息，用户可以自由发挥，也可以不用。hwaddr_len 和 hwaddr[]表示 MAC 地址长度和 MAC 地址，一般 MAC 地址长度为 6。

mtu 字段表示该网络一次可以传送的最大字节数，对于以太网一般设为 1500，不多说。

flags 字段是网卡状态信息标志位，是很重要的控制字段，它包括网卡功能使能、广播使能、ARP 使能等等重要控制位。

name[]字段用于保存每一个网络网络接口的名字。用两个字符的名字来标识网络接口使用的设备驱动的种类，名字由设备驱动来设置并且应该反映通过网络接口表示的硬件的种类。比如蓝牙设备（bluetooth）的网络接口名字可以是 bt，而 IEEE 802.11b WLAN 设备的名字就可以是 wl，当然设置什么名字用户是可以自由发挥的，这并不影响用户对网络接口的使用。当然，如果两个网络接口具有相同的网络名字，我们就用 num 字段来区分相同类别的不同网络接口。

到这里，你可能一头雾水，太抽象的东西太容易让人纠结。好吧，我们举个例子来看看一个以太网网卡接口结构是这样被初始化，还有数据包是如何接收和发送的。先来看初始化过程，源码：

```
static struct netif enc28j60;                                   (1)
struct ip_addr ipaddr, netmask, gw;                             (2)

IP4_ADDR(&gw, 192,168,0,1);                                     (3)
IP4_ADDR(&ipaddr, 192,168,0,60);                                (4)
IP4_ADDR(&netmask, 255,255,255,0);                              (5)

netif_init();                                                   (6)
netif_add(&enc28j60, &ipaddr, &netmask, &gw, NULL, ethernetif_init, tcpip_input);   (7)
netif_set_default(&enc28j60);                                   (8)
netif_set_up(&enc28j60);                                        (9)
```

上面的(1)声明了一个 netif 结构的变量 enc28j60，由于在我的板子上使用的是网卡芯片 enc28j60，所以我选择使用了这个名字。(2)声明了三个分别用于暂存 IP 地址、子网掩码和网关地址的变量，它们是 32 位长度的。(3)~ (5)分别是对上述三个地址值的初始化，该过程简单。

(6)很简单，它只需初始化上面所述的全局变量 netif_list 即可：netif_list = NULL。

(7)调用 netif_add 函数初始化变量 enc28j60，其中比较重要的两个参数是 ethernetif_init 和 tcpip_input，前者是用户自己定义的底层接口初始化函数，tcpip_input 函数是向 IP 层递交数据包的函数，从前面的讲述中可以很明显的看出，该值会被传递给 enc28j60 的 input 字段。再来看看源码：

```
struct netif *
  netif_add(struct netif *netif, struct ip_addr *ipaddr, struct ip_addr *netmask,
  struct ip_addr *gw,
  void *state,
  err_t (* init)(struct netif *netif),
  err_t (* input)(struct pbuf *p, struct netif *netif))
{
  static u8_t netifnum = 0;
  netif->ip_addr.addr = 0;       //复位变量 enc28j60 中各字段的值
  netif->netmask.addr = 0;
  netif->gw.addr = 0;
  netif->flags = 0;              //该网卡不允许任何功能使能

  netif->state = state;          //指向用户关心的信息，这里为 NULL
  netif->num = netifnum++; //设置 num 字段，
  netif->input = input;          //如前所诉，input 函数被赋值

  netif_set_addr(netif, ipaddr, netmask, gw);   //设置变量 enc28j60 的三个地址

  if (init(netif) != ERR_OK) {       //用户自己的底层接口初始化函数
    return NULL;
  }

  netif->next = netif_list;          //将初始化后的节点插入链表 netif_list
  netif_list = netif;                // netif_list 指向链表头

  return netif;
}
```

上面的初始化函数调用了用户自己定义的底层接口初始化函数，这里为 ethernetif_init，再来看看它的源代码：

```
err_t    ethernetif_init(struct netif *netif)
{
      netif->name[0] = IFNAME0;       //初始化变量 enc28j60 的 name 字段
      netif->name[1] = IFNAME1;       // IFNAME 在文件外定义的，这里不必关心它的具体值
```

```
    netif->output = etharp_output;      //IP 层发送数据包函数
    netif->linkoutput = low_level_output;   // //ARP 模块发送数据包函数
    low_level_init(netif);                  //底层硬件初始化函数
    return ERR_OK;
}
```

天，还有函数调用！low_level_init 函数就是与我们使用的硬件密切相关的函数了。

啊啊啊啊啊啊啊，没写完，明天再来吧！！

6 以太网数据接收

少壮不努力，长大写程序。悲剧！

昨天说到 low_level_init 函数是与我们使用的与硬件密切相关初始化函数，看看：

```
static void low_level_init(struct netif *netif)
{
    netif->hwaddr_len = ETHARP_HWADDR_LEN; //设置变量 enc28j60 的 hwaddr_len 字段
    netif->hwaddr[0] = 'F';          //初始化变量 enc28j60 的 MAC 地址
    netif->hwaddr[1] = 'O';          //设什么地址用户自由发挥吧，但是不要与其他
    netif->hwaddr[2] = 'R';          //网络设备的 MAC 地址重复。
    netif->hwaddr[3] = 'E';
    netif->hwaddr[4] = 'S';
    netif->hwaddr[5] = 'T';

    netif->mtu = 1500;       //最大允许传输单元
                             //允许该网卡广播和 ARP 功能，并且该网卡允许有硬件链路连接
    netif->flags  = NETIF_FLAG_BROADCAST | \
                    NETIF_FLAG_ETHARP | NETIF_FLAG_LINK_UP;

    enc28j60_init(netif->hwaddr);    //与底层驱动硬件驱动程序密切相关的硬件初始化函数
    }
```

至此，终于变量 enc28j60 被初始化好了，而且它描述的网卡芯片 enc28j60 也被初始化好了，而且变量 enc28j60 也被链入链表 netif_list。

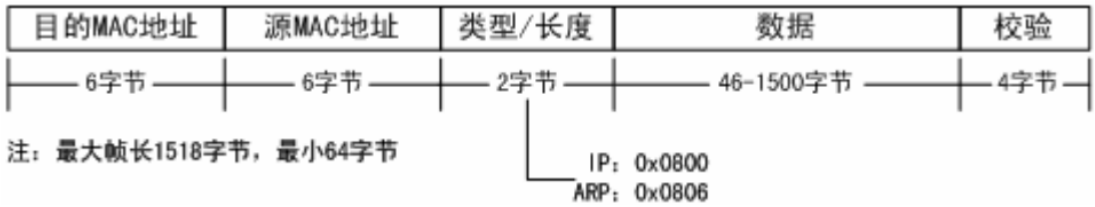
接着上上上上面的语句(8)调用 netif_set_default 函数初始化缺省网络接口。协议栈除了有个 netif_list 全局变量指向 netif 网络接口结构的链表，还有个全局变量 netif_default 全局变量指向缺省的网络接口结构。当 IP 层有数据发送时，它首先会以 netif_list 为索引选择满足某个条件的网络接口发送数据包，但是，当找不到这样的接口时，协议栈就会调用缺省的网络接口直接发送数据包，所以(8)中的意思是把变量 enc28j60 描述的网络接口设置为缺省的网络接口。

(9) 调用函数 netif_set_up 使能网络接口，这通过一个简单语句来实现：

```
netif->flags |= NETIF_FLAG_UP;
```

至此，网卡初始化完成，能正常接收和发送数据包了。下面我们来讨论讨论关于网卡数据包的接收和发送。

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，用户需要使用实际网卡驱动程序完成这两个函数。在第一篇中讲过，一个典型的 LWIP 应用系统包括这样的三个进程：首先是上层应用程序进程，然后是 LWIP 协议栈进程，最后是底层硬件数据包接收进程。这里我们就来讲讲第三个进程，看看数据包是怎样被接收并往上层传递的。但在这之前，有必要说说以太网网卡所收到的数据包的格式。如下图，



LWIP 使用了一个 eth_hdr 的数据结构来描述以太网数据包包头的 14 个字节。如下，

```
PACK_STRUCT_BEGIN
    struct eth_hdr {
    PACK_STRUCT_FIELD(struct eth_addr dest);     //目标 MAC 地址
    PACK_STRUCT_FIELD(struct eth_addr src);      //源 MAC 地址
    PACK_STRUCT_FIELD(u16_t type);                //类型
    } PACK_STRUCT_STRUCT;
PACK_STRUCT_END
```

其中 PACK_STRUCT_xxx 都是与编译器字对齐相关的宏定义，这里不作详细介绍了。上面的 dest、src 和 type 三个字段分别和上图中的目的 MAC 地址、源 MAC 地址和类型域字段对应。

在上面讨论的基础上，我们来看看这个数据包接收进程，源代码如下：

```
void   ethernetif_input(void *arg)     //创建该进程时，要将某个网络接口结构的 netif 结构指
{                                      //针作为参数传入
  struct eth_hdr *ethhdr;
  struct pbuf *p;

  struct netif *netif = (struct netif *)arg;

  while (1)
  {
    p = low_level_input (netif);   // 接收一个数据包
    if (p == NULL)          // 如果数据包为空，
          continue;       // 则循环结束，启动下次接收过程

    ethhdr = p->payload;     // 取得数据包内数据

    switch (htons(ethhdr->type))   // 判断数据包类型
      {                            // 只对 IP 数据包和 ARP 数据包进行处理
      case ETHTYPE_IP:             //   IP 数据包
      case ETHTYPE_ARP:            //   ARP 数据包
          if (netif->input(p, netif)!=ERR_OK)   // 将数据包发送到上层应用函数
            {
              pbuf_free(p);
              p = NULL;
            }
          break;
      default:
          pbuf_free(p);
```

```
                p = NULL;
                break;
        }     //switch
    }     //while
}     //main 函数
```

要创建上面的这个进程，需要把个网络接口结构的 netif 结构指针作为参数传入，在 UC/OSII 中要用到下面的语句实现，

```
OSTaskCreate(ethernetif_input,(void *)&enc28j60,
                    &T_ETHERNETIF_INPUT_STK[T_ETHERNETIF_INPUT_STKSIZE-1]
                     ETH_IF_TASK_PRIO);
```

在数据包接收进程中，有三个需要注意的地方。一是数据包接收的方法是查询方式，即处理器不断向网卡芯片中读取数据，如果读不到数据，则控制器会重新启动一个读取时序；如果能够成功读取到数据，则将数据通过网卡注册的 input 函数交往上层进行处理。使用查询方式实现的数据包接收进程其优先级必须低于系统中其他进程的优先级，否则它会阻塞比它优先级低的进程的运行。上面的程序有个可以改进的地方，即在读取到的数据包为空时，接收进程调用系统函数将自己延时一段时间再启动下一个读取过程，这样可以使其不能阻止优先级更低的进程的运行，缺点是数据包的接收得不到及时的响应。其实数据包的接收可以采用中断的方式来实现，这种方式是一种比较好的方式。一般的网卡芯片都有中断功能，即当网卡接收到一个数据包后，它可以产生中断信号告诉控制器自己接收到一个数据包。控制器此时启动一个读取数据包时序，就能有效的读取到非空数据包。所以可以这样来实现一个接收数据包进程：在无数据包收到时，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。

第二个需要注意的地方是 htons(ethhdr->type)函数的使用，htons 函数的功能是将一个半字长的数据从网络字节顺序转换到我们的处理器支持的字节顺序。解释一下，在计算机体系结构和计算机通信领域中，对于半字、字等的存储机制有可能不同。目前通常采用的存储机制主要有两种：big-endian 和 little-endian，即大端和小端。对于大端模式，某个半字或字数据的高位字节被在内存的低地址端，低位字节排放在内存的高地址端。对于小端模式，则恰好相反。由于我们使用的 ARM 处理器使用的是小端模式，而接收到的网络字节数据用的是大端模式，所以这里调用函数 htons 实现大端与小端的转换，实际就是将两个字节交换顺序即可。这样调用 htons(ethhdr->type)后，ethhdr->type 的值就为 0x0800 或 0x0806 等。

最后需要注意的地方，netif->input 在结构 enc28j60 初始化时已经被设置为指向 tcpip_input 函数，所以实际上上面是调用 tcpip_input 函数往上层递交数据包。tcpip_input 属于 IP 层函数，从这里我们可以看出 LWIP 的一个很大的特点，即各层之间没有明显的界限划分。像前面所讲的那样，LWIP 协议栈进程完成初始化相关工作后，会阻塞在一个邮箱上等待数据包的输入，这就对了，tcpip_input 函数就是向这个邮箱发送一条消息，且该消息中包含了收到的数据包存储的地址。LWIP 协议栈进程从邮箱中取到该地址后就可以对数据包进行处理了。

至此，数据包的接收可算大功告成，关于数据包的发送，这点很简单，因为它不必像数据包接收那样要使用一个专门的进程来实现，而是这样的：当上层有数据包要发送时，直接调用 netif->linkoutput 发送数据包就可以了。netif->linkoutput 在结构 enc28j60 初始化时已经被设置为指向 low_level_output 函数，该函数和底层硬件驱动密切相关，用于实现发送一个数据包的功能。用户应该结合具体网卡驱动实现该函数。

7 ARP 表

哈哈哈…..新的一年祝愿哥们和姐们一切都好。去年（嗯，确实是这个词）讲过了种种的种种，包括 LWIP 的移植要点、内存管理、数据包管理、网络接口管理等等。新的一年继续给力。

ARP，全称 Address Resolution Protocol，译作地址解析协议，是位于 TCP/IP 协议栈底层的协议。任何网络的通信都是基于底层硬件链路的，底层的数据链路有着自己的一套寻址机制，在以太网中，往往是通过一个 48 位的 MAC 地址来标示不同的网络通信设备的。而 TCP/IP 协议的上层是使用 IP 地址作为各个主机间通信寻址机制的。当源主机上层要向目标主机发送数据时，它只知道目标主机的 IP 地址，此时，源主机需要将该 IP 地址转换为目的主机对应的 MAC 地址，这样才能在数据链路上选择正确的通道将数据传送出去，这就是 ARP 的作用。哎，复杂了！

协议里面的一段描述可能更明了：在 ARP 背后有一个基本概念，那就是每个网络接口有一个硬件地址（一个 48 bit 的值，标识不同的以太网或令牌环网络接口），在硬件层次上进行的数据帧交换必须有正确的硬件接口地址。但是，TCP/IP 有自己的地址：32bit 的 IP 地址。知道主机的 I P 地址并不能让内核发送一帧数据给主机。内核（如以太网驱动程序）必须知道目的端的硬件地址才能发送数据。ARP 的功能是在 32 bit 的 I P 地址和采用不同网络技术的硬件地址之间提供动态映射。ARP 协议的基本功能就是通过目标设备的 IP 地址，查询目标设备的 MAC 地址，以保证通信的进行。

ARP 协议实现的核心是 ARP 缓存表，ARP 的实质就是对缓存表的建立、更新、查询等操作。ARP 缓存表是由一个个的缓存表项（entry）组成的，LWIP 中描述缓存表项的数据结构叫 etharp_entry，上源代码：

```
struct etharp_entry {
 #if ARP_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_state state;           // 描述该 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项有数据包缓冲队列，在 opt.h 里面设置。为什么要用数据包缓冲队列指针，随后慢慢道来。ipaddr 和 ethaddr 字段就是分别用于存储 IP 地址和 MAC 地址的，它们是 ARP 缓存表项的核心部分了。state 是个枚举类型，它表示该缓存表项的状态，一个表项有三个可能的状态，我们用枚举型 etharp_state 进行描述。

```
enum etharp_state {
    ETHARP_STATE_EMPTY = 0,
    ETHARP_STATE_PENDING,
    ETHARP_STATE_STABLE
    };
```

LWIP 内核通过数组的方式来创建 ARP 缓存表，如下，

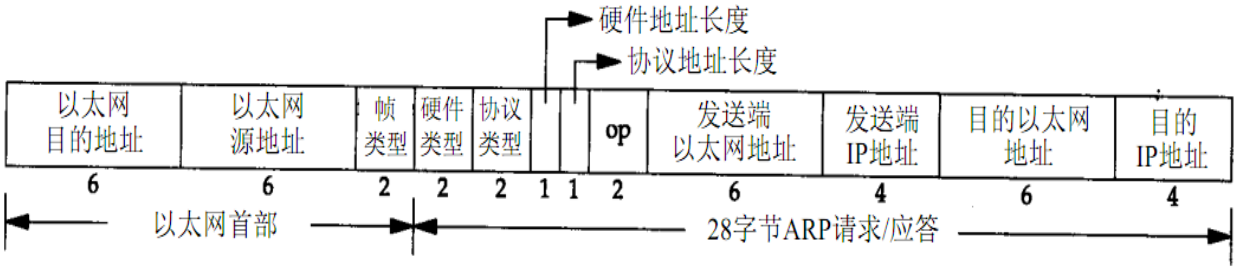
```
static struct etharp_entry arp_table[ARP_TABLE_SIZE];
```

初始化缓存表中的各个缓存表项都处于初始状态，没有记录任何信息，此时每个表项都处于 ETHARP_STATE_EMPTY 状态，ETHARP_STATE_PENDING 状态表示该表项处于不稳定状态，很可能的情况是，该表项只记录到了 IP 地址，但是还为记录到对应该 IP 地址的 MAC 地址，此时就该表项就处于 ETHARP_STATE_PENDING 状态。在该状态下，LWIP 内核会发出一个广播 ARP 请求到数据链路上，以让对应 IP 地址的主机回应其 MAC 地址，当源主机接收到 MAC 地址时，它就更新对应的 ARP 表项。当 ARP 表项得到更新后，它就完全记录了一对 IP 地址和 MAC 地址，此时该表项就处于 ETHARP_STATE_STABLE 状态。注意当某表项处在 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，当等到该表项稳定后，这些数据包才会被发送出去。这就是为什么每个表项需要有数据包缓冲队列指针了。

ctime 字段记录表项处于某个状态的时间，当某表项的 ctime 值大于规定的表项最大生存值时，该表项会被内核删除。在第一讲中，我们就说到了关于 LWIP 的超时事件，要使用 ARP 功能，就必须设置一个 ARP 超时事件，该超时事件的基本功能就是对每个表项的 ctime 字段值加 1，然后删除那些生存时间大于最大生存值的表项。

好了，下面讲讲能够正确建立 ARP 缓存的基础：ARP 数据包。要在源主机上建立关于目标主机的 IP 地址与 MAC 地址对应表项，则源主机和目的主机间的基本信息交互是必须的，简单的说就是，源主机如何告诉目的主机：我需要你的 MAC 地址；而目的主机如何回复：这就是我的 MAC 地址。ARP 数据包，这就派上用场了。

ARP 数据包可以分为 ARP 请求数据包和 ARP 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，



以太网包头中的前两个字段是以太网的目的 MAC 地址和源 MAC 地址，在前面一章已经有讲解。目的地址为全 1 的特殊地址是广播地址。在 ARP 表项建立前，源主机只知道目的主机的 IP 地址，并不知道其 MAC 地址，所以在数据链路上，源主机只有通过广播的方式将 ARP 请求数据包发送出去。电缆上的所有以太网接口都要接收广播的数据包，并检测数据包是否是发给自己的，这点通过对照目的 IP 地址来实现，如果是发给自己的，目的主机需要回复一个 ARP 应答数据包给源主机，以告诉源主机自己的 MAC 地址。

两个字节长的以太网帧类型表示后面数据的类型。对于 ARP 请求或应答数据包来说，该字段的值为 0x0806，对于 IP 数据包来说，该字段的值为 0x0800。

以太网数据报头说完，来说 ARP 数据报头。

硬件类型字段表示硬件地址的类型，它的值为 1 即表示以太网 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。它的值为 0x0800 即表示要映射为 IP 地址。它的值与包含 I P 数据报的以太网数据帧头中的类型字段的值相同。

接下来的两个 1 字节的字段，硬件地址长度和协议地址长度分别指出硬件地址和协议地址的长度，以字节为单位。对于以太网上 A R P 请求或应答来说，它们的值分别为 6 和 4。

操作字段 op 指出四种操作类型，它们是 A R P 请求（值为 1）、A R P 应答（值为 2）、R A R P 请求（值为 3）和 R A R P 应答（值为 4），这里我们只关心前两个类型。这个字段

必需的，因为 A R P 请求和 A R P 应答的帧类型字段值是相同的。

接下来的四个字段是发送端的以太网 MAC 地址、发送端的 I P 地址、目的端的以太网 MAC 地址和目的端的 I P 地址。

注意，这里有一些重复信息：在以太网的数据帧报头中和 A R P 请求数据帧中都有发送端的以太网 MAC 地址。对于一个 ARP 请求来说，除目的端 MAC 地址外的所有其他的字段都有填充值。当目的主机收到一份给自己的 ARP 请求报文后，它就把自己的硬件地址填进去，然后将该请求数据包的源主机信息和目的主机信息交换位置，并把操作字段 op 置为 2，最后把该新构建的数据包发送回去，这就是 ARP 响应。

最后，用源码来看看 LWIP 是如何描述上面的这个数据报头的：

```
struct etharp_hdr {
  PACK_STRUCT_FIELD(struct eth_hdr ethhdr);      // 14 字节的以太网数据报头
  PACK_STRUCT_FIELD(u16_t hwtype);               // 2 字节的硬件类型
  PACK_STRUCT_FIELD(u16_t proto);                // 2 字节的协议类型
  PACK_STRUCT_FIELD(u16_t _hwlen_protolen);   //  两个 1 字节的长度字段
  PACK_STRUCT_FIELD(u16_t opcode);               // 2 字节的操作字段 op
  PACK_STRUCT_FIELD(struct eth_addr shwaddr);    // 6 字节源 MAC 地址
  PACK_STRUCT_FIELD(struct ip_addr2 sipaddr);    // 4 字节源 IP 地址
  PACK_STRUCT_FIELD(struct eth_addr dhwaddr);    // 6 字节目的 MAC 地址
  PACK_STRUCT_FIELD(struct ip_addr2 dipaddr);    // 4 字节目的 IP 地址
} PACK_STRUCT_STRUCT;
```

不唐僧了，和前面的各个描述完全相符。PACK_STRUCT_FIELD()是防止编译器字对齐的宏定义，讲过了的。

8 ARP 表查询

ARP 攻击，是针对以太网地址解析协议（ARP）的一种攻击技术。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，告诉源主机一个假 MAC 地址，这样就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，而源主机和目的主机却浑然不知。ARP 攻击通过伪造 IP 地址和 MAC 地址实现 ARP 欺骗，能够在网络中产生大量的 ARP 通信量使网络阻塞，攻击者只要持续不断的发出伪造的 ARP 响应包就能更改目标主机 ARP 缓存中的 IP-MAC 条目。ARP 协议在设计时未考虑网络安全方面的特性，这就注定了其很容易遭受 ARP 攻击。黑客只要在局域网内阅读送上门来的广播 ARP 请求数据包，就能偷听到网内所有的 (IP, MAC)地址。而源节点收到 ARP 响应时，它也不会质疑，这样黑客很容易冒充他人。

这一节主要针对 ARP 讲解 ARP 表的创建，更新，查询等操作。这里我们先从几个简单的函数入手讲解 ARP 各个子模块功能，然后再将各个模块与上层协议结合起来，宏观的讲解 ARP 模块。

第一个需要迫不及待要说的函数是 find_entry，该函数最重要的输入是一个 IP 地址，返回值是该 IP 地址对应的 ARP 缓存表项索引。函数声明原型如下，

```
static s8_t find_entry(struct ip_addr *ipaddr, u8_t flags)
```

这里，很有必要翻译一下源代码中注释的内容：该函数主要功能是寻找一个匹配的 ARP 表项或者创建一个新的 ARP 表项，并返回该表项的索引号。如果参数 ipaddr 为给定的非空的内容，则函数需要返回一个处于 pending 或 stable 的索引表项，如果没有匹配的表项，则该函数需要返回一个 empty 的表项，但该表项的 IP 字段的值要被设置为 ipaddr 的值，这种情况下，find_entry 函数返回后，调用者需要将表项从状态 empty 改为 pending。还有一种情况，如果参数 ipaddr 为空值，同样返回一个状态为 empty 的表项。

返回状态为 empty 的表项，首先从状态标示为 empty 的空闲 ARP 表项中选取，如果这样的表项都用完了，同时参数 flags 的值被设置为 ETHARP_TRY_HARD，则 find_entry 就回收最老的 ARP 表项，将该表项设置为 empty 状态返回。

这个函数比较大，有将近 200 行代码，这里就不贴了，直接讲讲它的工作流程。这部分的讨论还是参考了网上某位大侠的博客，名字记不得了，对不起啊啊啊啊啊！网络，有时确实是个好东西，越发的明白。好了，看看 find_entry 的工作流程。

首先，lwip 有一个比较巧妙的地方，它并不是冲上去就是就把 arp 缓存中所有的表项搜索一遍，而是做了一个假设，假设这次的表项索引还是上一次的（在很多情况下就是这样的）。所以，LWIP 中有个全局的变量 etharp_cached_entry，它始终保存着上次用到的索引号，如果这个索引恰好就是我们要找的内容，且索引的表项已经处于 stable 状态，那就直接返回这个索引号就完成了，we're really fast！

如果情况不够理想，就必须去检索整个 ARP 表了，检索的过程是从 ARP 表的第一个表项开始，依次往后检索直至最后一个表项，过程较复杂。对于每个表项首先判断它是否为 empty 状态，find_entry 只关心第一个状态为 empty 的表项索引值，对该索引值以后的 empty 表项不感兴趣，忽略。如果一个表项不是 empty 状态，则判断它是不是 pending 状态。对于 pending 状态的表项，需要做以下的事情，先看看它里面存的 IP 地址和我们的 ipaddr 是否匹配，如果匹配，好返回该索引值，记住还要更新 etharp_cached_entry 为该索引值，如果不匹配，则判断该索引的数据包指针是否为空，find_entry 试图记录生存时间最长的 pending 状

态有数据缓冲或无数据缓冲的表项索引。如果一个表项也不是 pending 状态，则判断它是不是 stable 状态。对于 stable 状态的表项，与 pending 状态的表项处理过程相似， find_entry 试图记录生存时间最长的 stable 表项的索引。很晕吧，我也很晕，看了下面这段可能你会好点！

如果到这里都还没有找到匹配的表项，那就很杯具了，我们需要为 find_entry 调用者返回一个 empty 的表项索引。经过上面一段后，find_entry 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、生存时间最老的 stable 状态表项的索引，我们暂且先将这四个值假设为 a、b、c、d。如果参数 flags 的值被设置为 ETHARP_TRY_HARD，那么 find_entry 会按照 a-->d-->c-->b 的顺序选择一个合适的索引返回，为什么是这样的顺序？很明显，不解释。find_entry 首先判断 a 是否在 ARP 表项范围内，如果是，则选择 a，如果不是，则判断 b 是否在 ARP 表项范围内，依此类推。当选中一个索引后，随即就会将该索引对应的表项设置为 empty 状态，并且将该表项的 IP 地址设置为 ipaddr 的值，ctime 值设置为 0，最后返回索引。至此，find_entry 大功告成！

接下来，我很感兴趣的一个函数是 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，当然情况远不如此简单。还是很有必要翻译一下源代码中函数功能注释的内容：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，再同时要发送的数据包会被挂接在该表项的数据缓冲指针上；如果 IP 地址在 ARP 表中有相应的表项存在，但该表项处于 pending 状态，则操作与前者相同，即发送一个 ARP 请求和挂接数据包；如果 IP 地址在 ARP 表中有相应的表项存在，且表项处于 stable 状态，此时再来判断给定的数据包是否为空，不为空则直接将该数据包发送出去，为空则向该 IP 地址发送一个 ARP 请求。

etharp_query 函数原型如下所示，源代码在 150 行左右，这里主要讲解其流程：

```
err_t etharp_query(struct netif *netif, struct ip_addr *ipaddr, struct pbuf *q)
```

(1) 首先判断给定的 ipaddr 是否合法，对于空 IP 地址、广播 IP 地址、多播 IP 地址不予处理。

(2) 将 ipaddr 作为参数调用函数 find_entry，函数返回一个 ARP 表项索引，该表项可能是原来已经有的，此时该表项应该是 pending 或 stable 状态；该表项也可能是新申请得到的，此时该表项应该是 empty 状态。

(3) 根据返回的表项索引找到该 ARP 表项，判断该表项是否为 empty 状态，如果是，说明该表项是新申请的，则将该表项状态设置为 pending 状态。

(4) 判断要发送的数据包是否为空，或者判断 ARP 表项是否为 pending 状态，这两个条件只要有一个成立，就发送一个 ARP 请求出去，发送 ARP 请求的函数是 etharp_request。

(5) 如果待发送的数据包不为空，此刻就根据 ARP 表项的状态作不同的处理：若 ARP 表项处于 stable 状态，则直接调用函数 etharp_send_ip 发送数据包；若 ARP 表项处于 pending 状态，则需要将该数据包挂接到表项的待发送数据链表上，由于 pending 状态的表项必然在第(4)步中发出了一个 ARP 请求，当内核接收到 ARP 回应时，会将表项设置为 stable 状态，并将其链表上的数据全部发送出去，当然这项工作具体是怎样完成的那是后话了。

将数据包挂接在表项的发送链表上，这又是一个较复杂的过程：最重要的一点是判断该数据包 pbuf 的类型，对于 PBUF_REF、PBUF_POOL、PBUF_RAM 型的数据包不能直接挂在发送链表上，因为这些数据包在被挂接后并不会被立刻发送出去，这可能导致数据包在等待发送的过程中内部数据被改动。对于以上这些类型的待发送数据包，需要将数据拷贝至新的 pbuf 中，然后将新的 pbuf 挂接至发送链表。至此，etharp_query 函数功德圆满！

时间总是在不经意间走完。。。我却没写完。

9 ARP 层流程

前面一节重点说了 ARP 缓存表以及如何对其进行相关操作，关于 ARP，一共想说三个函数，前面已经讲过了两个。

最后要讲的一个函数是 update_arp_entry，该函数用于更新 ARP 缓存表中的表项或者在缓存表中插入一个新的表项。该函数会在收到一个 IP 数据包或 ARP 数据包后被调用。该函数原型如下，

```
static err_t
update_arp_entry(struct netif *netif, struct ip_addr *ipaddr, struct eth_addr *ethaddr, u8_t flags)
```

其中重要的两个参数 ipaddr 和 ethaddr 分别对应的 ip 地址和 mac 地址，函数利用这两个地址去更新或插入 ARP 表项。由于这个函数代码量较小，这里就列出源码来讲解，注意这个源码是经过我处理的，已经去掉了编译选项、源码注释、调试输出信息等非重点部分。

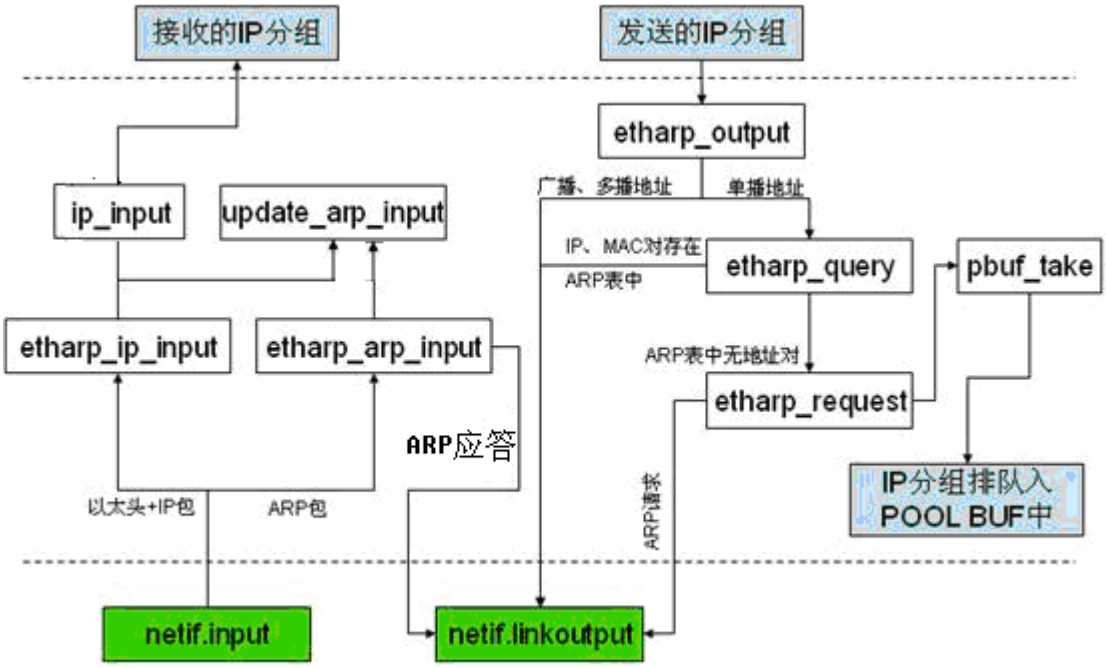
```
static err_t
update_arp_entry(struct netif *netif, struct ip_addr *ipaddr, struct eth_addr *ethaddr, u8_t flags)
{
  s8_t i;          // 两个变量，不解释
  u8_t k;
  i = find_entry(ipaddr, flags);     // 查找或新建一个 ARP 表项，返回其索引值
  if (i < 0)     return (err_t)i;        // 如果为不合法的索引值，则更新缓存表失败
  arp_table[i].state = ETHARP_STATE_STABLE; // 否则将对应表项状态改为 stable
  arp_table[i].netif = netif;        // 记录下网络接口

  k = ETHARP_HWADDR_LEN;   // 这一段是更新缓存表项中的 MAC 地址
  while (k > 0) {
    k--;
    arp_table[i].ethaddr.addr[k] = ethaddr->addr[k];
  }
  arp_table[i].ctime = 0;     //   生存时间值置 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) {   // 只要缓冲链表中海有数据则循环
    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节点
    p = q->p;       // 取得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); // 发送数据包
    pbuf_free(p);     // 释放数据包缓存空间
  }
#endif
  return ERR_OK;
}
```

从源程序中可以看出，update_arp_entry 的流程如下：先通过调用 find_entry 找到对应 ipaddr 对应的表项，并设置相应的 arp 表项的成员（主要是 state， netif， ethaddr， cttime），

最后如果定义了 ARP_QUEUEING，并且这个 arp 表项上有未发送的数据包的话，则把这些数据全部发送出去。虽然比较啰嗦，但是还是我们还是根据不同的 ipaddr 经过 find_entry 执行后，来看看 update_arp_entry 运行的几种不同情况。

首先可以肯定的是，update_arp_entry 的两个参数 ipaddr 和 ethaddr 必是互相匹配的，因为它们是从源主机发来的 IP 包或 ARP 包中解析出来的，代表了源主机的 MAC 地址和 IP 地址。find_entry 利用 ipaddr 作为参数执行后，返回一个 ARP 表项索引。如果该表项处于 empty 状态，那么该表项现在一定是新创建的，此时设置该表项为 stable 状态并设置该表项其他字段值后即结束。如果该表项是处于 pending 状态，由于此时已经有了和 ipaddr 匹配的 MAC 地址返回，所以该表项也被设置为 stable 状态并同时设置该表项其他字段值。如果该表项是处于 stable 状态，其实此时只需要将 ctime 的值复位即可，但是 LWIP 为了节省代码量，它还是选择像上面的情况一样做相同的处理，这样虽然有些步骤是多余的，但并不影响函数功能。最后都会检查该表项是否还有数据需要发送，如果有，则将所有数据包发送出去。

现在是时候从宏观上来看看到底 ARP 是怎么一个工作流程，以及它在整个 LWIP 协议栈当中发挥的重要作用。关于这点，不得不借鉴网上某位大侠的了，不过对 TA 的图做了一些小小的修改（脸红，用画图工具改的，Visio 表示不会用）。



该图简洁明了的解释了基本所有 LWIP 的数据包接收与发送的全过程。我们可以看到几个熟悉的身影：etharp_query、etharp_request、update_arp_entry。在前面已经讲过了的！

ARP 从功能上来说可以简单的分成两个部分：当有数据包输入时，更新 arp 表，如果是 ip 包则递交给 ip 层，如果是 arp 包，则针对不同的 arp 包类型做相应的响应；当向目的 ip 发送一个数据包的时候，需要通过 arp 实现 ip 到 MAC 地址的映射，必要时，需要发送广播数据包获得目标机器的 MAC 地址。

LWIP 利用 netif.input 指向的函数接收以太网数据包，通常这个函数是 ethernet_input。注意，这里并不是说 ethernet_input 直接与底层硬件交互接收数据包，而是更底层的函数接收到数据包后将数据包递交给 ethernet_input，ethernet_input 再对其进行处理。

以太网的帧类型可以是：IP，ARP，甚至可以是 pppoe， wlan 等。这里主要分析 IP 和

ARP 两种类型的数据包。ethernet_input 根据以太网首部的类型字段判断收到的数据包的类型，如果是 IP 包，则将该包递交给 etharp_ip_input，如果是 ARP 包，则将该包递交给 etharp_arp_input。

对于 ip 类型的数据包，etharp_ip_input 首先检查是否开启了 ETHARP_TRUST_IP_MAC 这个选项，如果开启了就是要用这个帧中的信息和 update_arp_entry 函数来更新 arp 表（利用帧首部的源 mac 地址和帧数据中 ip 报文中的源 ip 地址），然后丢弃以太网帧首部，将 IP 报文通过 ip_input 函数递交给 ip 层。

对于 arp 类型的数据包，etharp_arp_input 函数首先利用数据包头信息更新 arp 表的内容，然后再判断该 ARP 数据包的类型，如果是 ARP 请求包，则首先判断这个包是不是给自己的，如果是给自己的，则在原有包的基础上重组一个 ARP 应答包发送出去（注意此处并没有重新分配一个 pbuf，而是借用了原来的缓冲结构）。如果不是给自己的，则直接忽略。如果是 ARP 应答包，主要的工作就是更新 arp 表，但是这一步已经在 arp 包刚进来的时候就处理了，所以这里不需要再重复做，这样 ARP 包的处理也完毕。

LWIP 利用 netif.output 指向的函数发送 IP 数据包，通常这个函数是 etharp_output。注意，这里并不是说 etharp_output 直接与底层硬件交互发送数据包，而是将数据包做相应的处理，主要是将 IP 数据包打包成以太网帧数据，最终递交给 netif.linkoutput 函数来发送的。

etharp_output 函数接收 IP 层要发送的数据包，并将数据包发送出去。由于是发送 ip 数据包，所以函数一开始需要增加缓冲区大小，大小为以太网的数据首部的大小。然后检查 ip 地址，可以分为广播包，多播包，单播包(单播包又分为是局域网内部还是局域网外面)。

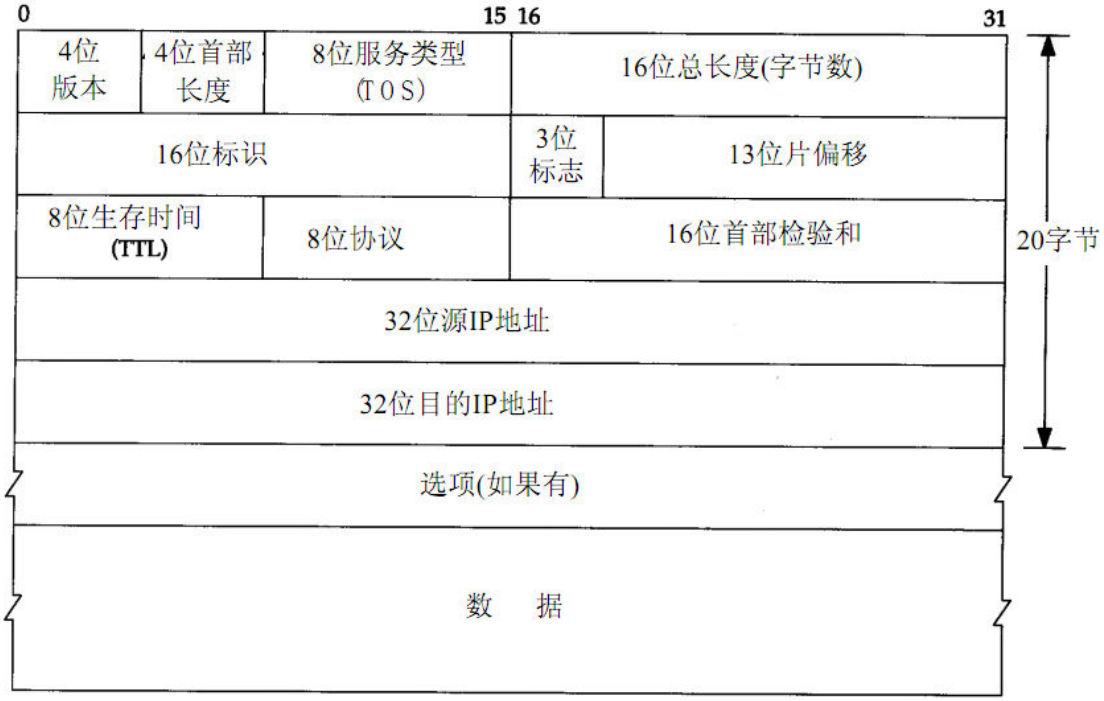
广播包：判断目的 IP 地址是不是为全 1，或者是全 0（老版本中使用的），如果是广播包则目的 IP 的 MAC 地址不需要查询 arp 表，直接将 MAC 地址设置为全 1 发送即可，即 MAC 六个字节值为 0xff，0xff，0xff，0xff，0xff，0xff。

多播包：判断目的 ip 地址是不是 d 类地址，即 0xexxxxxxx，如果是多播的话，mac 地址也是确定的，即将 MAC 地址 01-00-5e-00-00-00 的低 23 位设置为 IP 地址的低 23 位。对于以上的两种数据包，etharp_output 直接调用函数 etharp_send_ip 将数据包发送出去。

单播包：要比较目的 IP 和本地 IP 地址，看是否是局域网内的，不是局域网内的，则将目的 IP 地址设置为默认网关的地址，然后再统一调用 etharp_query 函数将数据包发送出去，注意这些数据包在这种情况下可能被连接在相关 ARP 表项的发送链表上，等待发送。

10　IP 层输入

对于 IP 层主要讨论信息包的接收、分片数据包重装、信息包的发送和转发三个内容。IP 数据报头结构如下所示，其中，选项字段是可以没有的，所以通常的 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 字段用于记录首部长度，以字为单位。所以对于不含任何选项字段的 IP 报头，则该长度值为 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 15*4=60 字节。

再下来是一个 8bit 的服务类型字段，该字段主要用于描述该 IP 数据包急需的服务类型，如最小延时、最大吞吐量、最高可靠性、最小费用等。这个字段在 LWIP 中没啥用处。

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。理论上说，IP 数据包总长度最大可达 65535 字节，但在实际应用中，底层链路可不允许这么大的数据包出现在链路上，因为这会大大增加数据出错的可能性，所以在链路层往往会对大的 IP 数据包进行分片，当然这些都是后话。

接下来的 16 位标识字段用于标识 IP 层发送出去的每一份 IP 数据报，每发送一份报文，则该值加 1。然后的 3 位标志和 13 位片偏移字段用于在 IP 数据包分片时使用，这里先不讨论。LWIP 的较高版本才支持 IP 分片功能。

TTL 字段描述该 IP 数据包最多能被转发的次数，每经过一次转发，该值会减 1，当该值为 0 时，一个 ICMP 报文会被返回至源主机。

8 位协议字段用来描述该 IP 数据包是来自于上层的哪个协议，该值为 1 表示为 ICMP 协议，该值为 2 表示 IGMP 协议，该值为 6 表示 TCP 协议，该值为 17 表 UDP 协议。

16 位首部校验和只针对 IP 首部做校验，它并不关心其内部数据在传输过程中出错与否，

对于数据的校验是上层协议负责的，如 ICMP、IGMP、TCP、UDP 协议都会计算它们头部以及整个数据区的长度。这里再 COPY 一段这个校验和是怎样生成以及在接收端是如何实验校验的。

在发送端为了计算一份数据报的 IP 检验和，首先把检验和字段置为 0。然后，对首部中每个 16 bit 进行二进制反码求和（整个首部看成是由一串 16 bit 的字组成），结果存在检验和字段中。当接收端收到一份 I P 数据报后，同样对首部中每个 16 bit 进行二进制反码的求和。由于接收方在计算过程中包含了发送方保存在首部中的检验和字段，因此，如果首部在传输过程中没有发生任何差错，那么接收方计算的结果应该为全 1。如果结果不是全 1（即检验和错误），那么 IP 就丢弃收到的数据报。但是不生成差错报文，由上层去发现丢失的数据报并进行重传。

接下来是两个 32 位的 IP 地址，不啰嗦了。最后一个字段是任选字段，不同的协议会选择性的使用该字段，这里也不讨论。

现在来看看 LWIP 中是怎么样来描述这个 IP 数据报头的，使用的结构体叫 ip_hdr：

```
struct ip_hdr {
  PACK_STRUCT_FIELD(u16_t _v_hl_tos);  // 前三个字段：版本号、首部长度、服务类型
  PACK_STRUCT_FIELD(u16_t _len);   // 总长度
  PACK_STRUCT_FIELD(u16_t _id);    // 标识字段
  PACK_STRUCT_FIELD(u16_t _offset); // 3 位标志和 13 位片偏移字段
#define IP_RF 0x8000            //
#define IP_DF 0x4000            // 不分组标识位掩码
#define IP_MF 0x2000            // 后续有分组到来标识位掩码
#define IP_OFFMASK 0x1fff   // 获取 13 位片偏移字段的掩码
  PACK_STRUCT_FIELD(u16_t _ttl_proto);  // TTL 字段和协议字段
  PACK_STRUCT_FIELD(u16_t _chksum);    // 首部校验和字段
  PACK_STRUCT_FIELD(struct ip_addr src);  // 源 IP 地址
  PACK_STRUCT_FIELD(struct ip_addr dest);  // 目的 IP 地址
} PACK_STRUCT_STRUCT;
```

注意结构体声明的时候定义了几个宏定义：IP_RF、IP_DF、IP_MF、IP_OFFMASK，它们是在求与分组相关两个字段时要用到的掩码，也可以在结构体的外面进行定义，无影响。

前面讲过，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，分发给 IP 模块是通过调用 ip_input 函数完成的，当然在递交前，ethernet_input 需要将数据包去掉以太网头。现在来看看数据包传递给 ip_input 后，该函数进行了哪些方面的工作。这里我们先不涉及其内部关于 DHCP 协议的相关处理。

第一件事是检查 IP 头部的版本号，如果该值不为 4，则立即丢弃该数据包。更高版本的 LWIP 协议栈可以支持 IPv6，但这里我们只讨论 IPv4。接下来函数检查 IP 数据报头是否只保存于一个 pbuf 中，如果不是 ，也直接丢弃该 IP 包，这是因为 LWIP 不允许 IP 数据包头被分装在不同的 pbuf 里面。同时，函数检查 IP 报头中的总长度字段是否大于递交上来的数据包总长度，如果是，则说明存在传输错误，直接丢弃数据包。

然后是对 IP 数据报头做校验，该工作是函数 inet_chksum 完成的，如果校验不通过则直接丢弃数据包。inet_chksum 函数在后续有需要时会详细讲解。

接着，需要在这里对数据包进行截断操作，按照 IP 包头记录的总长度字段截取数据包，因为经过 ethernet_input 传递上来的数据包只被去除了以太网数据包头部，而对于可能存在的以太网填充字段和一定存在的以太网校验字段(最后一字节)没做处理，我们在这里对它们进行截断，得到完整无冗余的 IP 数据包。

然后，函数检测 IP 数据包中的目的 IP 地址是否与本机的相符，本机的 IP 地址是保存在 netif 结构体变量中的，一个系统可能有着多个网卡设备，这就意味着它有多个 netif 结构体变量分别用于描述这些网卡设备，也意味着本机有着多个 IP 地址，这些 netif 结构体是被连接在 netif_list 链表上的。ip_input 函数会遍历 netif_list 链表上的 netif 结构以找到匹配的 IP 地址，并记录该 netif 结构体变量，也即记录该网卡。从这点看来，在 ARP 部分内容中，对于某个接收到的 ARP 请求包，也应该按照这种方式进行遍历后再给出 ARP 回应更好，而源代码并没有这样做，当然，这只是个人意见。当遍历完成后，如果依旧没有得到与匹配的 netif 结构体变量，这说明该数据包不是给本机的，此时需要对数据包进行转发或者丢弃工作，这是通过宏定义 IP_FORWARD 来完成的，这里注意不要对广播数据包进行转发。

再接下来，根据目标 IP 地址判断数据包是否为广播或多播 IP 数据包，LWIP 不对这些类型的数据包进行响应。

再接下来的工作可以说是 ip_input 函数中最复杂最难理解的部分，这就是 IP 分片数据包的重装，ip_input 函数通过数据包的 3 位标志和 13 位片偏移字段判断发给自己的该 IP 包是不是分片包，如果是，则需要将该分片包暂存，等到接收完所有分片包后，统一将整个数据包递交给上层应用程序。这是万言难尽的过程，先在这里打住，我们在以后的内容里面细细讨论。如果是分片包，且不是最后一片，则函数到这里就返回了。

终于，能到达这一步的数据包必然是未分片的或经过分片完整重装后的数据包。此时，ip_input 函数根据 IP 数据包头内部的协议字段判断该数据包应该被递交给哪个上层协议，并调用相应的函数递交数据包。是 UDP 协议，则调用 udp_input 函数；是 TCP 协议，则调用 tcp_input 函数；是 ICMP 协议，则调用 icmp_input 函数；是 IGMP 协议，则调用 igmp_input 函数；如果都不是，则调用函数 icmp_dest_unreach 返回一个协议不可达 ICMP 数据包给源主机，同时删除数据包。

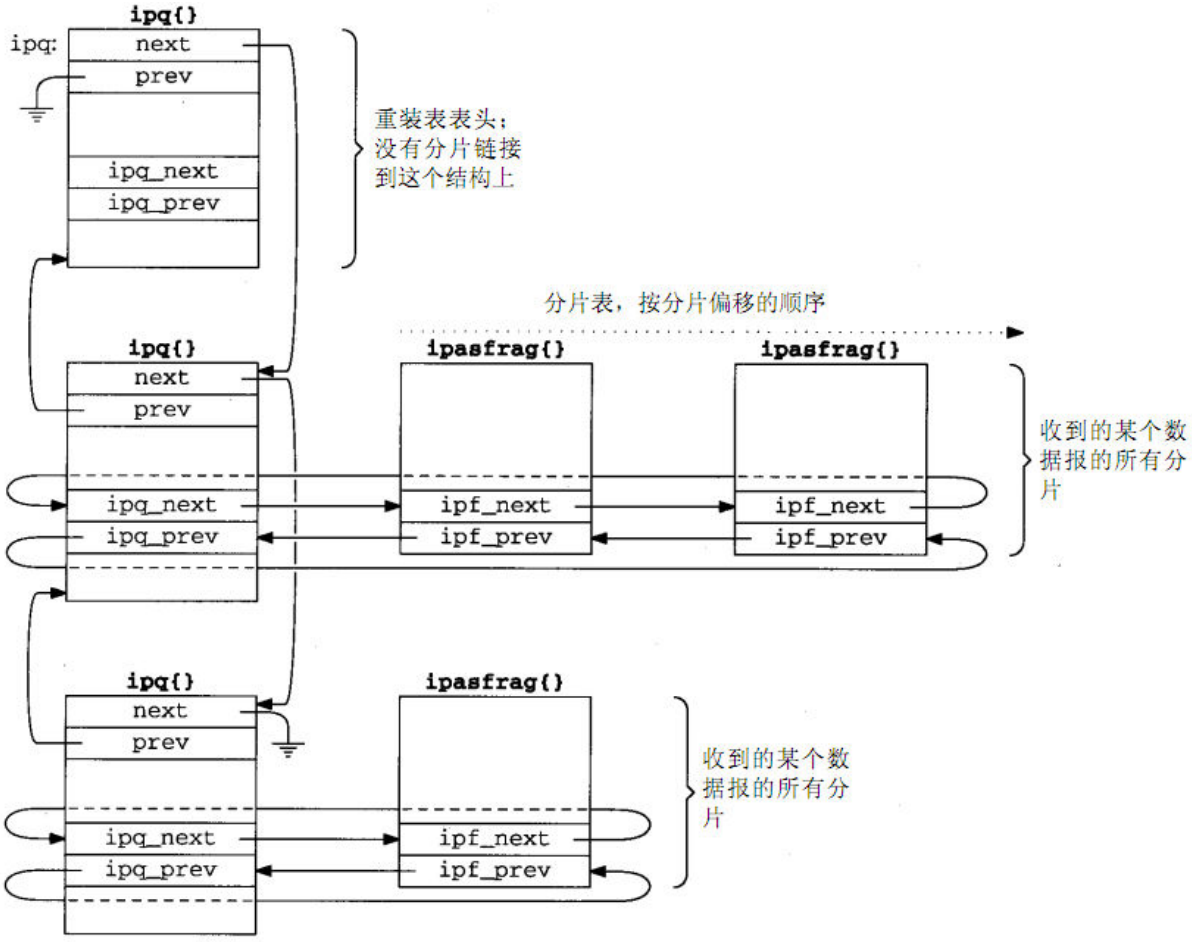
写完收工！

11　IP 分片重装 1

较低版本的 LWIP 协议并不支数据包持数据包的分片与重装功能，较高版本的 LWIP 协议，关于分片数据包的重装，采用了与标准协议中差别较大的方式来实现。它采用了一种更为简单的数据结构来组装分片包，但是这样导致的结果是组装过程源代码晦涩难懂，代码执行效率低。个人认为 LWIP 的这种实现机制不是太好，有时间的话想自己将整个数据包分片重装机制全部重新实现，指定更好。

需要注意的是，在一般的嵌入式产品中，数据量是比较小的，基本不会出现数据分片的情况，而且嵌入式产品也不会在网络中充当路由器，实现数据包的重组、转发等功能。因此，较低版本的 LWIP 依然能在大多数应用中发挥其功能。

首先，我们来看看标准协议中例举的 BSD 是怎么来实现数据包的重组的，再讲 LWIP 的重装机制与该机制对比讲解。



标准中主要采用了两个数据结构来实现数据包的重装，ipq 和 ipasfrag。其中 ipq 是为某个将要重组的数据包建立的节点信息数据结构，包括目标 IP 地址，数据包编号等信息，该结构还有指针 ipf_next、ipf_prev 使得该数据包的各个分片数据组成一个双向链表，图中 ipasfrag 用于包装各个分片信息包。同时，将所有 ipq 结构体够成一个双向链表，便于某个 ipq 结构的查找、插入和删除操作。整个 ipq 构成的链表有一个固定的链表头，该表头不存

储任何数据包的数据，只做标识用。

在 LWIP 中，结构体 ip_reassdata 与上面的 ipq 起着类似的作用，但结构成员大不相同，ip_reassdata 被用来构成的是单向链表，以实现对某个 ip_reassdata 结构的查找删除等操作。所有 ip_reassdata 结构构成的单向链表有一个头节点指针，即 reassdatagrams，用于指向链表头。同时 LWIP 中没有类似 ipasfrag 的结构对收到的各个分片数据包进行封装，而是直接将分片数据包挂接在对应的 ip_reassdata 结构之后。至于怎样挂接，那是后话。

现在来看看传说中的 ip_reassdata 结构，源代码：

```
struct ip_reassdata {
  struct ip_reassdata *next;       // 用于构建单向链表的指针
  struct pbuf *p;       // 该数据报的数据链表
  struct ip_hdr iphdr;   // 该数据报的 IP 报头
  u16_t datagram_len;   // 已经收到的数据报长度
  u8_t flags;   // 是否收到最后一个分片包
  u8_t timer;   // 设置超时间隔
  };
```

ip_reassdata 主要用于描述一份正在被组装的数据报，完成数据包组装的函数叫做 ip_reass，该函数以 IP 分片数据包为输入参数，输出组装好的数据包指针或空指针，这里我们来看看它的操作流程。

首先，判断该 IP 数据包的头部大小，目前 LWIP 不支持有 IP 选项的 IP 数据包，所以 IP 数据包头部大小只能为 20 个字节，如果大于该值，ip_reass 直接丢弃该数据包后返回。

然后，由于 LWIP 对所有 ip_reassdata 结构连接的 pbufs 个数总和有个上限限定，即 IP_REASS_MAX_PBUFS，所以 ip_reass 检查当前数据包分片占用的 pbufs 个数，假设如果这么多个 pbufs 被连接在某个 ip_reassdata 后，所使用了的 pbufs 个数是否大于了我们刚才提的上限值，如果是大于，LWIP 是不会允许这样的情况下将 pbufs 被连接入 ip_reassdata 的。此时用户可以有两种选择，一是直接删除数据包后返回，二是删除 ip_reassdata 链表中生存时间最长的 ip_reassdata 结构体及其相关 pbufs，直到有了足够的 pbufs 个数能使用。这种选择是通过宏定义 IP_REASS_FREE_OLDEST 来实现的。哎，这段说得太凌乱了！为什么要定义一个最大允许 pbufs 使用个数呢，不懂！

到这步就可以进行分片数据报的插入操作了，首先要做的就是在链表 reassdatagrams 中找到对应的 ip_reassdata 结构体，此时又有两种情况：没有找到匹配的结构体，即该分片是第一个到来的分片，此时需要创建一个新的 ip_reassdata 结构体并插入链表 reassdatagrams 中；找到匹配的结构体，即在该分片到来之前已经有分片到来，此时要判断该分片是不是整个数据包的第一个分片，如果是，则用对应 ip_reassdata 结构体的 iphdr 字段记录该 IP 数据包头部。

到这里，必然找到了分片数据包对应的 ip_reassdata 了，接着我们就判断该分片是不是整个数据包的最后一个分片包，因为最后一个分片数据包的 IP_MF 位为 0。如果是最后一个分片包，则设置对应 ip_reassdata 的 flag 标志和 datagram_len 值，表示收到最后一个分片和设置整个数据包的长度值。PS:第一个到来的分片包不一定是第一个分片包，最后一个到来的数据包也不一定是最后一个分片包。

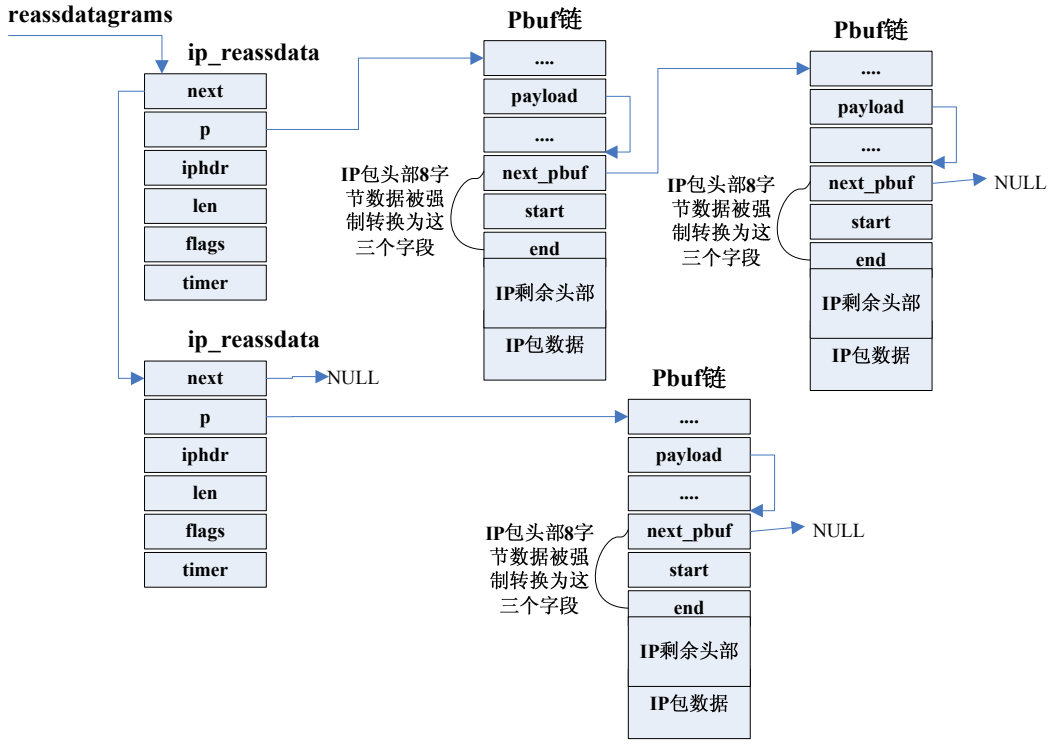
最后调用函数 ip_reass_chain_frag_into_datagram_and_validate 对分片数据包进行插入操作，这是我们后续会重点将的一个函数，先在这里打住。上面这个函数还会检测某个数据包是否被组装完毕，如果某个数据包被组装完毕，那 ip_reass 还要做以下工作：

根据 ip_reassdata 结构找到第一个分片数据包，将 ip_reassdata 结构中的 IP 报头字段 iphdr 拷贝至第一个分片数据包头部（第一个分片数据包头部头部数据已经被覆盖，在讨论函数

ip_reass_chain_frag_into_datagram_and_validate 时我们会讲到），并重新计算校验和。同时将第一个分片以后的各个分片数据包去掉头部信息，这个整个数据包就恢复出来了，然后将 ip_reassdata 结构从链表中删除，将重组后的数据包指针作为返回参数返回。

12　IP 分片重装 2

上一节还遗留有一个问题：IP 分片包是怎么被插入相应的组装链表的。这里用一个直观的图形来解释这一过程。



图中展示了有两个数据包正在被组装的过程，两个 ip_reassdata 结构分别用于两个数据包的组装过程。第一个数据包已经组装好了两个分片数据，第二个组装好了一个分片数据。LWIP 并不像标准协议中描述的那样，另外分配一个数据结构来描述各个分片数据，而是直接利用各个分片数据，将数据中 IP 头部的前八个字节拿来存储组装过程中的相关信息，因此每个进来的 IP 分片数据包头都被改变了，也因此要使用 ip_reassdata 结构中的 iphdr 字段来暂时保存 IP 数据包的完整头部。

这八个字节被变成了什么值呢，这就是结构体 ip_reass_helper 的内容了。这个结构体就是组装过程中最为重要的结构体了。看看，这恐怕是最简单的一个结构体了。

```
struct ip_reass_helper {
    struct pbuf *next_pbuf;
    u16_t start;
    u16_t end;
  };
```

next_pbuf 字段在组装过程中用来连接各个分片数据包；start 字段用来记录该分片数据包数据在整个 IP 包中的起始位置；同理，end 字段表示在 IP 包中的结束位置。

函数 ip_reass_chain_frag_into_datagram_and_validate(PS：神，这个函数名太长了)用来向一个 ip_reassdata 结构中插入一个 IP 分片包 pbuf，插入后检查该 IP 包是否被组装完毕，未组装完毕则返回 0，否则返回非 0 值。很明显，这个函数的输入参数有两个，ip_reassdata

结构和分片包 pbuf。下面仔细看看这个函数名最长的函数到底干了些什么工作。

首先它会从该 IP 分片包的头部中提取信息，包括分片数据的起始偏移地址 offset 和分片数据的长度 len，之后它将该 IP 分片包的头部得前八个字节强制转换为 ip_reass_helper 结构，并将 start 字段的值置为 offset；end 字段的值置为 offset+len；next_pbuf 字段的值置为 NULL。到这里，进来的这个分片包已经被改变面貌了，它的 IP 头部已经被毁坏，下面就会将这个改头换面的分片包进行插入操作了。插入操作主要是利用比较各个 ip_reass_helper 的 start 和 end 字段的值以确定这个分片包被插在链表中的哪个位置，插入位置的寻找是整个组装过程中最难理解的部分了，但从原理上看是很简单的一个过程，这里不再对查找插入位置及插入操作的源码做解析。

到这步，分片的数据包已经被插入到相应的 ip_reassdata 结构后的链表中了，此时这个函数名最长的函数要检查是否这个 ip_reassdata 对应的数据包已经组装完毕。这里要分两步来看，第一是判断该数据包的最后一个分片包是否已经到来，在上面一节中已经讲过，当收到的 IP 分片包为某个 IP 包的最后一个分片时，函数 ip_reass 会把对应 ip_reassdata 结构的 flags 字段置 1，所以，如果检测到某个 ip_reassdata 结构的 flags 字段仍为 0，说明最后一个分片还未被收到，IP 数据包组装肯定还未完成，此时，函数名最长的函数直接返回 0 即可。第二步，如果发现 flags 字段置 1，说明最后一个分片包已经收到，但是整个 IP 是否被组装完毕还是未知，因为在网络上，分片包不是每次都能按次序到达，因此，收到的最后一个分片数据包不一定是最后一个分片包。此时需要遍历 ip_reassdata 结构后面的各个分片包链表，以检测是否还有分片包未被接收到。

到这步，ip_reass_chain_frag_into_datagram_and_validate 函数就完成工作了，它将分片的数据插入了某个 ip_reassdata 结构的数据分片链表，并检测该 IP 数据包是否被组装完毕，组装完毕则返回一个非 0 值，否则返回 0 值。

这里，我们有回到了 ip_reass 函数，ip_reass 通过函数调用将某个分片数据包插入相应的个 ip_reassdata 结构后，通过函数的返回值来决定要做的下一步操作。如果数据包组装未完成，ip_reass 函数需要向调用它的函数返回一个空指针，如果数据包组装完成，它就要向调用它的函数返回这个组装好的数据包。从上面的图中可以看出，被组装好的数据包是全部分片被挂接在一个 ip_reassdata 结构上的，ip_reass 函数需要从这个 ip_reassdata 结构上取下相应的各个分片数据，并删除各个分片中的不必要信息，然后将整个数据包返回给调用者。为了完成这个任务，ip_reass 函数进行了下面的工作。

先将 ip_reassdata 结构中的 iphdr 字段各个值做相应的修正，如修正报文总长度、校验和，然后将 iphdr 字段全部拷贝到第一个分片包的 IP 头部字段中，这样整个 IP 数据包的头部就出现在第一个分片信息包中了，接下来从第二个分片包开始，将它们的 IP 头部信息删除，这样，一个完整的 IP 数据包就重新组装好了。最后，调用 ip_reass_dequeue_datagram 函数删除数据包组装过程中使用的 ip_reassdata 结构体，即从上图所述的 reassdatagrams 链表删除对应 reassdata 的节点。好了，功德圆满！

这个圈子绕得有点大，本来是在讲 ip_input 函数的，结果就讲到了 ip_reass 函数，后来又讲到了 ip_reass_chain_frag_into_datagram_and_validate 函数。没办法，谁让后面两个函数是为 ip_input 函数服务的呢！ip_input 函数使用 ip_reass 组装好的数据包递交到上层 TCP 或 UDP 等应用中去，在前面已经讲过了。

现在我们还是要回到原路上，走上 ip_input 这条道路。ip_input 的基本流程已经在前面讲得很清楚了，还有一个函数为它服务，即 ip_forward 函数，它主要是完成数据包的转发工作，当设备接收到的数据包不是给自己的时候，它就可以选择将该数据包转发出去，本来这里没有必要讲 ip_forward 函数的，因为在一般的应用中，这项功能会被禁止，设备收到不是给自己的数据包时，将在 ip_input 函数处理的初期被丢弃。但到目前，我们还未涉及到任何

关于 IP 数据包发送的内容，考虑了很久，还是觉得应该把 ip_forward 函数讲解一下，因为数据包的转发与数据包的发送是完全一样的原理，使用了完全相同的接口函数，因此讲解了 ip_forward 函数就等于讲解了 IP 层数据包发送的所有工作细节。在 TCP 层或 UDP 层必然涉及到数据包的发送工作，在这里就利用 ip_forward 函数将 IP 层数据包发送的整个过程讲解清楚，这样逻辑清楚，利于理解！

当收到一个 IP 数据包后，LWIP 会遍历所有网络接口的 IP 地址，判断这个数据包是不是给自己的，如果不是，就要调用收到该数据包的那个网络接口将数据包转发出去。但是不慌，转发前还要检测这个包是不是一个广播包，如果是，直接丢弃，不做处理。现在来看看数据包转发函数 ip_forward 做了哪些工作呢，这个函数的输入参数有三个：要转发的数据包指针，要转发的数据包的 IP 报头指针，收到该数据包的的网络接口数据结构 netif 指针。

首先，调用 ip_route 函数找到转发该数据包应该使用的网络接口，ip_route 函数以数据包 IP 报头中的目标地址为参数，查找应该使用的相关结构。如果找不到满足要求的接口，则选择缺省网络接口。ip_route 函数现在这里打住，在讲完 ip_forward 函数之后，再对它进行详细的讲解。

ip_forward 检查 ip_route 函数找到的网络接口是否为有效，所谓有效，即不能为空，也不能为接收到该 IP 包的那个接口。当判定网络接口为无效时，数据包不会被转发。当可以用某个网络接口转发数据包时，ip_forward 先将该 IP 报头中 TTL 字段值减 1，若 TTL 变为 0，则需要向源主机发送一份超时 ICMP 信息，表示当前数据包的生存周期到了，这个数据包在这里被丢弃，不会被转发出去。至于怎样发送这个超时的 ICMP 信息包，这就涉及到 IP 层数据包的发送函数 ip_output 了，我们将在后面慢慢道来。

接下来函数重新计算头部校验和，因为头部 TTL 字段的值已经被修改，最后调用 netif 结构注册的 output 函数，该函数将数据包组装成以太网数据帧并发送出去。前面说过了，这个函数就是 etharp_output。

到这里 ip_forward 函数的工作就完成了，还剩下两个问题，ip_route 函数和怎样发送一个超时的 ICMP 信息包出去。这里讲解第一个问题，第二个问题放在 ICMP 部分。ip_route 函数以目标 IP 地址为输入参数，然后在网络接口结构链表 netif_list 上找寻与该 IP 地址在同一子网上的网络接口，若找到则返回满足要求的网络接口，若找不到则返回缺省网络接口。如此的简单，不多说。

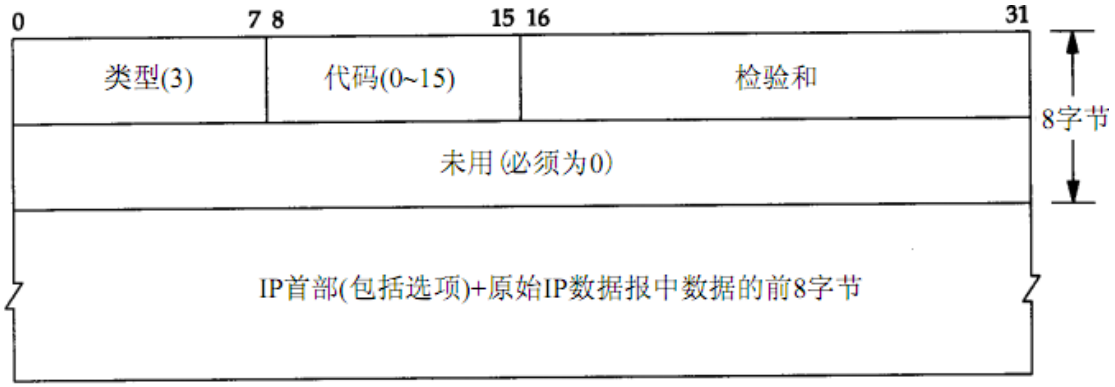
13　ICMP 处理

目前，IP 层的东西基本讲解完，数据包的发送或分片发送没有具体涉及到。数据包的发送，与上层协议密切相关，即传输层，后面的内容就是讨论传输层的东西了。这里先讲解传输层协议中比较简单的 ICMP 协议。ICMP（Internet Control Message Protocol）是 Internet 控制报文协议，用于在 IP 主机、路由器之间传递控制消息。控制消息是指网络通不通、主机是否可达、路由是否可用等网络本身的消息。这些控制消息虽然并不传输用户数据，但是对于用户数据的传递起着重要的作用。

在以前讲解 IP 层 ip_input 函数时，已经三次涉及到了 ICMP 的东西，第一次在数据包转发过程中，需要将数据包的 TTL 值减 1，若此时 TTL 值变为 0 则用 icmp_time_exceeded 函数向源主机返回一份超时 ICMP 信息；还有两次是 ip_input 函数通过 IP 报文头部的协议字段值判断该数据包是交给哪个上层协议的，若是 ICMP 协议，则调用 icmp_input 函数；若没有一个协议能接受这个数据包，则调用 icmp_dest_unreach 函数向源主机返回一个协议不可达 ICMP 差错控制包。这里先讲解 icmp_time_exceeded 和 icmp_dest_unreach 函数是怎样发送 ICMP 信息包的。

先来看看 ICMP 报文的格式，所有 ICMP 报文的前四个字节都是一样的，分别为 1 字节的类型字段，1 字节的代码字段和 2 字节的校验和字段。校验和字段的计算覆盖整个 ICMP 报文。类型字段和代码字段唯一确定了该 ICMP 报文属于那种类型：如回显、超时、时间戳请求等等，尽管各种类型的 ICMP 报文结构通常是不相同的，但它们开始的四个字节是相同的。ICMP 报文从大的方面来说可以分为 ICMP 查询报文和 ICMP 差错报文。ICMP 查询报文包括 ICMP 回显应答、回显请求、时间戳请求、地址掩码请求等等类型，LWIP 只实现了 ICMP 回显应答；ICMP 差错报文有目的不可达、超时、重定向等等类型，LWIP 只实现了目的不可达、超时两项 ICMP 处理功能。这里先讲解目的不可达和超时两种类型的 ICMP 处理。

目的不可达和超时两种 ICMP 报文均属于 ICMP 差错报文，协议中规定，ICMP 差错报文应始终包含产生 ICMP 差错报文的 IP 数据报的 IP 首部和数据前 8 个字节。所以，目的不可达和超时这两种 ICMP 报文均有下面的报文结构，这个结构与前面所述完全相符，不解释了。



先讲与 icmp_dest_unreach 函数所描述的目的不可达差错报文。该报文的类型字段值应为 3，代码字段值应为（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。很明显，在 ip_input 函数找不到将该数据包交到哪个上层协议时，应该产生一个协议不可达的差错报

文，即代码字段值为 2。函数原型如下，输入参数为接收到的目标不可达的 IP 数据包和应该用于填充 ICMP 头部代码字段的值。

```
void icmp_dest_unreach(struct pbuf *p, enum icmp_dur_type t)
```

现在来看看这个函数做了哪些工作，首先为要发送数去的 ICMP 数据包申请一个 pbuf 缓冲区，这个缓冲区的长度为上图所述结构的长度与一个 IP 数据报头大小之和，之所以要多申请 IP 数据报头大小的空间是为了当该 ICMP 数据包被递交给 IP 层发送时，IP 层不需要再去申请一个数据报头来封装该 ICMP 数据包，而是直接在已经申请好的报头中填入 IP 头部数据，注意这里申请好的 pbuf 的 payload 指针是指向 ICMP 数据报头处的。接下来，函数填写 ICMP 数据包的相关字段，将类型段填充为 3，代码段为输入参数 t，同时将不可达的 IP 数据包的 IP 报头和数据前 8 个字节拷贝到 ICMP 数据包相应字段，最后计算校验和字段的值，然后调用 ip_output 函数将组装好的 ICMP 包发送出去。

ip_output 函数通过调用 ip_output_if 函数完成数据包的发送，它在调用 ip_output_if 函数前，要先根据发送数据包的目的 IP 地址找到相应的发送网络接口结构，并将该结构作为调用 ip_output_if 函数的参数。ip_output_if 函数主要是填充 IP 报头各个字段的值，然后调用 netif->output 函数将封装好的 IP 数据包发送出去。

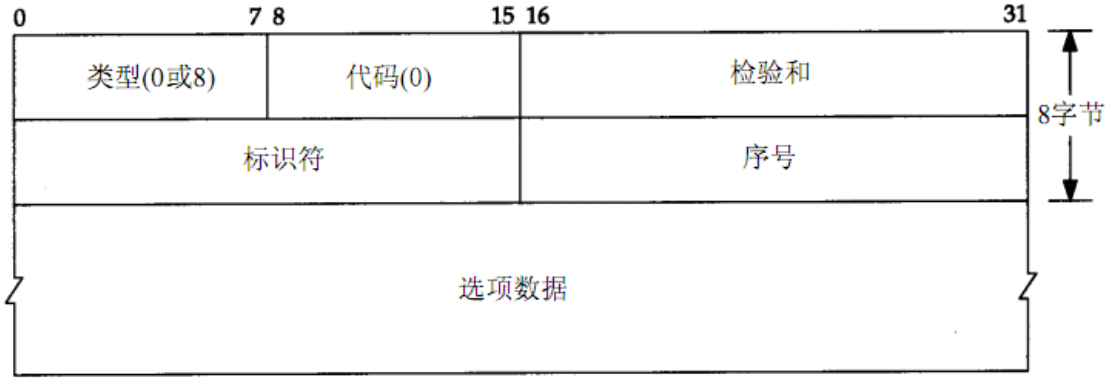
icmp_time_exceeded 函数用于产生一个超时类型的 ICMP 报文。该类型的报文与上面所述的报文有完全相同的报文结构，但类型字段值应为 11，代码字段应为 0 或 1，分别代表传输期间生存时间超时和数据组装期间生存时间超时。对于后者，在讨论数据包重组时，我们知道每个 ip_reassdata 结构体中都有个时间字段，当指定的时间到达而数据包还未被组装完毕，则内核会将该 ip_reassdata 结构相关的所有分片数据全部删除，并向源主机发送一个代码字段为 1 的超时 ICMP 报文。

从代码流程和内容来看，icmp_time_exceeded 函数和 icmp_dest_unreach 函数完全一样，只是在填充 ICMP 报文的类型和代码字段使用了不同的值，这里不再赘述。

接下来该讨论 ICMP 查询报文了，这部分在整个产品的设计调试过程中显示出非常重要的作用。即 Ping 命令，它与 ICMP 回显应答、请求报文密切相关。

这小段来自于协议：“Ping”这个名字源于声纳定位操作。目的是为了测试另一台主机是否可达。该程序发送一份 ICMP 回显请求报文给主机，并等待返回 ICMP 回显应答。一般来说，如果不能 Ping 到某台主机，那么就不能 Telnet 或者 FTP 到那台主机。反过来，如果不能 Telnet 到某台主机，那么通常可以用 Ping 程序来确定问题出在哪里。Ping 程序还能测出到这台主机的往返时间，以表明该主机离我们有“多远”。同时，Ping 程序也能在数据包的路由过程中记录下路由路径。

ICMP 回显请求和回显应答报文格式如下图，回显应答的类型值为 0，回显请求类型值为 8，二者代码字段值均为 0，Unix 系统在实现 ping 程序时是把 ICMP 报文中的标识符字段置成发送进程的 ID 号。这样即使在同一台主机上同时运行了多个 ping 程序实例，ping 程序也可以识别出返回的信息。序列号从 0 开始，每发送一次新的回显请求就加 1。



icmp_input 函数处理接收到的 ICMP 数据包，并根据包类型做相应的处理。目前 LWIP 只能处理 ICMP 回显请求包，对其他类型的 ICMP 包不做响应，这在嵌入式产品中是足够用了的。对于 ICMP 回显请求，icmp_input 需要生成一个回显应答报文返回给源主机。

来看看 icmp_input 函数做了哪些工作。首先将传进来的数据包 pbuf 的 payload 指针调整为指向 ICMP 头部，并判断 ICMP 头部长度是否小于 4 个字节，若是，则说明是个错误的 ICMP 数据包，该包被丢弃。对于正确的 ICMP 包，函数根据其头部类型字段的值判断该做什么样的处理。当前版本只实现了对 ICMP 回显请求的相关响应操作。

若当前的数据包为 ICMP 回显请求，则函数继续判断该数据包是否为广播或者多播包，对这两种数据包不做处理；接下来判断该数据包大小是否小于 ICMP 回显请求头部长度（如上图所示），是则丢弃数据包；接下来函数为这个数据包申请 IP 报头和以太网帧头内存空间，成功后将该 ICMP 包类型字段变为 0，从新计算校验和，并将 IP 报头的源 IP 地址和目的 IP 地址交换位置，最后将整个数据包利用 ip_output_if 函数将数据包发送出去。ICMP 回显应答将回显请求中的数据原样返回给源主机，源主机在收到回显应答后，通过处理回显应答中的数据可以得到相关信息，如计算往返时间等。

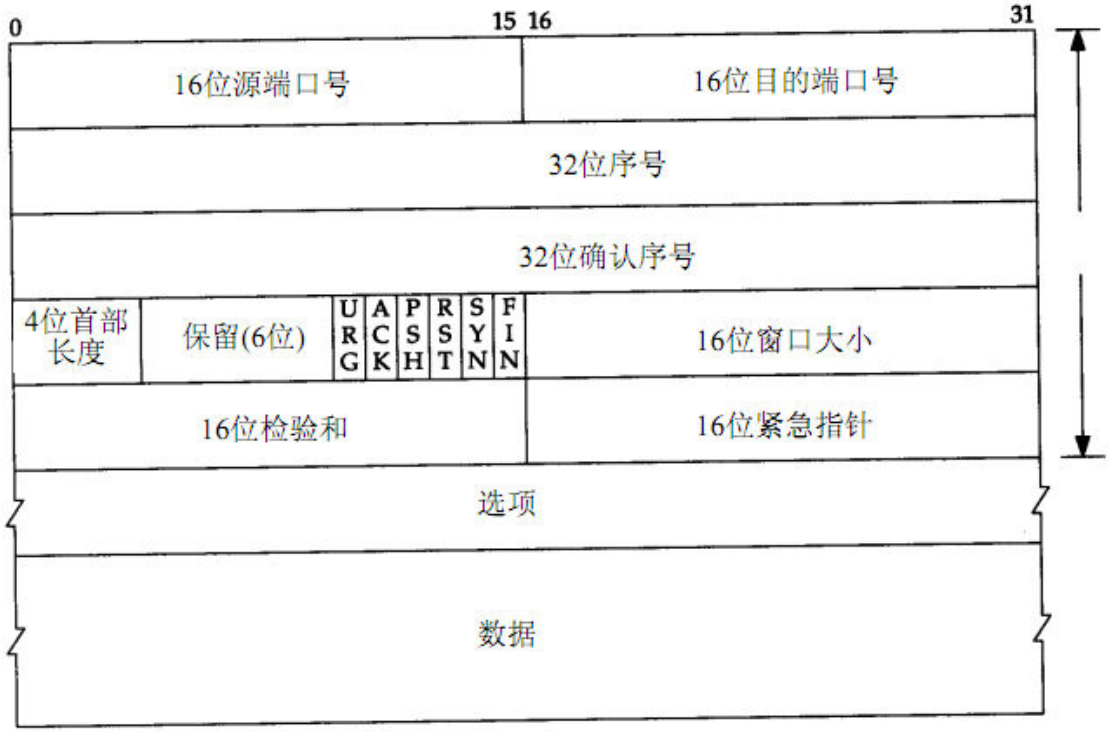
ICMP 就是这么多了。

14　TCP 建立与断开

TCP 部分是整个 LWIP 最庞大也是难理解的部分，其代码将近占了整个协议栈代码量的一半。看到如此大的一个工程真的不知道从哪里下口才能将它讲清楚。郁闷，看到啥就写啥吧先，等写完了再来慢慢整理。但我想参考的基本主线还是标准协议的 TCP 部分。

TCP 叫传输控制协议，它为上层提供一种面向连接的、可靠的字节流服务，（PS：这一段都剽窃自协议）。TCP 通过下面的一系列机制来提供可靠性：应用数据被分割成 TCP 认为最适合发送的数据块；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段，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，它将发送一个确认，这个确认不是立即发送，通常将推迟几分之一秒；TCP 将保持它首部和数据的检验和，如果收到段的检验和有差错，TCP 将丢弃这个报文段并且不发送确认收，以使发送端超时并重发；I P 数据报的到达可能会失序，因此 TCP 报文段的到达也可能会失序，如果必要，TCP 将对收到的数据进行重新排序，将收到的数据以正确的顺序交给应用层；I P 数据报会发生重复，TCP 的接收端必须丢弃重复的数据；T C P 还能提供流量控制。

下图是 TCP 首部结构，若不计任选字段，其大小为 20 字节，与 IP 报首部大小相同。



源端口号和目的端口号，用于标识发送端和接收端的应用进程。这两个值加上 IP 首部中的源 IP 地址和目的 IP 地址就能唯一确定一个 TCP 连接。一个 IP 地址和一个端口号也称为一个插口（socket）。

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。当建立一个新的连接时，SYN 标志置 1，序号字段包含由这个发送主机选择的该连接上的初始序号 ISN（Initial Sequence Number）。该主机要发送数据的第一个字节序号为 ISN+1。

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端所期望收到的下一个序号。因此，确认序号应当是上次已成功收到数据字节序号加 1。当一个 TCP 连接被正确建立后，ACK 字段总是被设置为 1 的。

4 位首部长度给出首部中 32 bit 字的数目。需要这个值是因为任选字段的长度是可变的。由于这个字段有 4 bit，因此 TCP 最多有 6 0 字节的首部。然而，若没有任选字段，正常的长度是 20 字节。

在 T C P 首部中有 6 个标志比特。它们中的多个可同时被设置为 1。在这里简单介绍它们的用法，在以后用到时会详加讲解：URG 紧急指针（urgent pointer）有效标识；ACK 确认序号有效标识；PSH 接收方应该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 位检验和覆盖了整个的 TCP 报文段：TCP 首部和 TCP 数据。这是一个强制性的字段，一定是由发送端计算和存储，并由收收端进行验证。 TCP 检验和使用的 TCP 头部并不包括实际头部中的所有字段，而是一个伪首部，具体关于伪首部的结构可参看 UDP 部分，或参看协议。

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 URG 标志置 1 时紧急指针才有效。TCP 的紧急方式是发送端向另一端发送紧急数据的一种方式。

最常见的可选字段是最长报文大小，又称为 MSS (Maximum Segment Size)。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。它指明本端所能接收的最大长度的报文段。

TCP 报文段中的数据部分有时是为空的。例如在一个连接建立和一个连接终止时，双方交换的报文段仅有 TCP 首部；又如一方没有数据要发送，则它需使用没有任何数据的首部来确认收到的数据；再在处理超时的许多情况中，也会发送不带任何数据的报文段。

上面的都是协议规定的内容，没有任何自由发挥的空间，下面来看看 LWIP 是如何描述这样一个 TCP 报头的，数据结构 tcp_hdr 如下，与上图描述的完全相符，不解释了。

```
PACK_STRUCT_BEGIN
 struct tcp_hdr {
  PACK_STRUCT_FIELD(u16_t src);     // 源端口
  PACK_STRUCT_FIELD(u16_t dest);    // 目的端口
  PACK_STRUCT_FIELD(u32_t seqno); // 序号
  PACK_STRUCT_FIELD(u32_t ackno); // 确认序号
  PACK_STRUCT_FIELD(u16_t _hdrlen_rsvd_flags); // 首部长度+保留位+标志位
  PACK_STRUCT_FIELD(u16_t wnd);       // 窗口大小
  PACK_STRUCT_FIELD(u16_t chksum);    // 校验和
  PACK_STRUCT_FIELD(u16_t urgp);      // 紧急指针
 } PACK_STRUCT_STRUCT;
PACK_STRUCT_END
```

哎，又一次茫然了，不晓得该写什么！TCP 这块真是太难啃了。先抛开 LWIP 的内容，仔细讲解详解卷中的内容。先讲讲一个 TCP 连接的建立过程。TCP 建立连接需要有三个报文段的交互过程，所以又称三次握手过程。

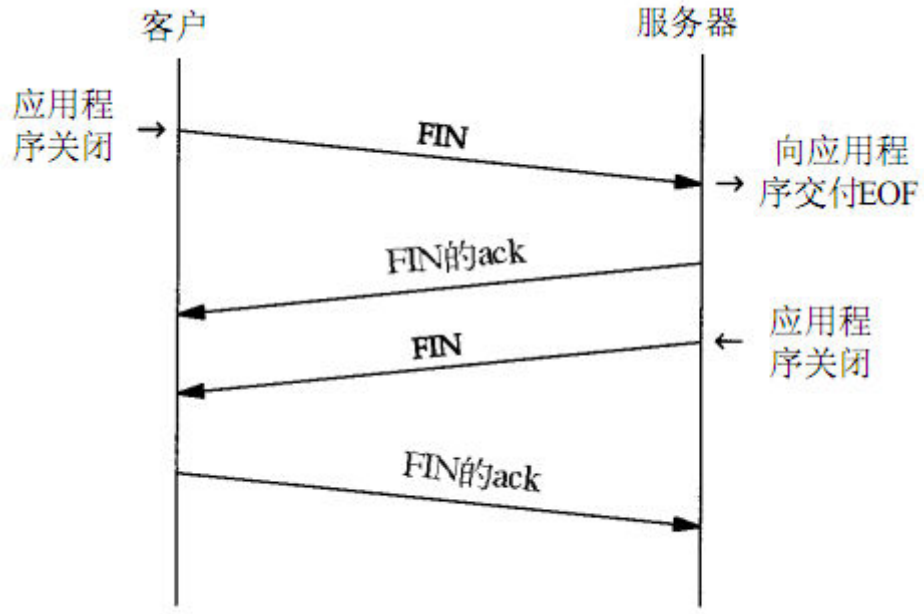
首先，请求端（通常称为客户）发送一个 SYN 标志置 1 的 TCP 数据报，数据包中指明自己的端口号及将连接的服务器的端口号，同时通告自己的初始序号 ISN。

当服务器接收到该数据包并解析后，也发回一个 SYN 报文段作为应答。该回应报文包含服务器自身选定的初始序号 ISN，同时，将 ACK 置 1，将确认序号设置为请求端的 ISN 加 1 以对客户的 SYN 报文段进行确认。这里的 ISN 也表示了服务器希望接收到的下一个字节的序号。由此可见，一个 SYN 将占用了一个序号。

最后，当请求端接收到服务器的 SYN 应答包后，会再次产生一个握手包，这个包中，ACK 标志置位，确认序号设置为服务器发送的 ISN 加 1，以此来实现对服务器的 SYN 报文段的确认。

通常存在这样的一种情况，两端同时发起连接，即同时发送第一个 SYN 数据包，这时，这两端都处于主动打开状态，在后续的讨论中我们将涉及这个内容。

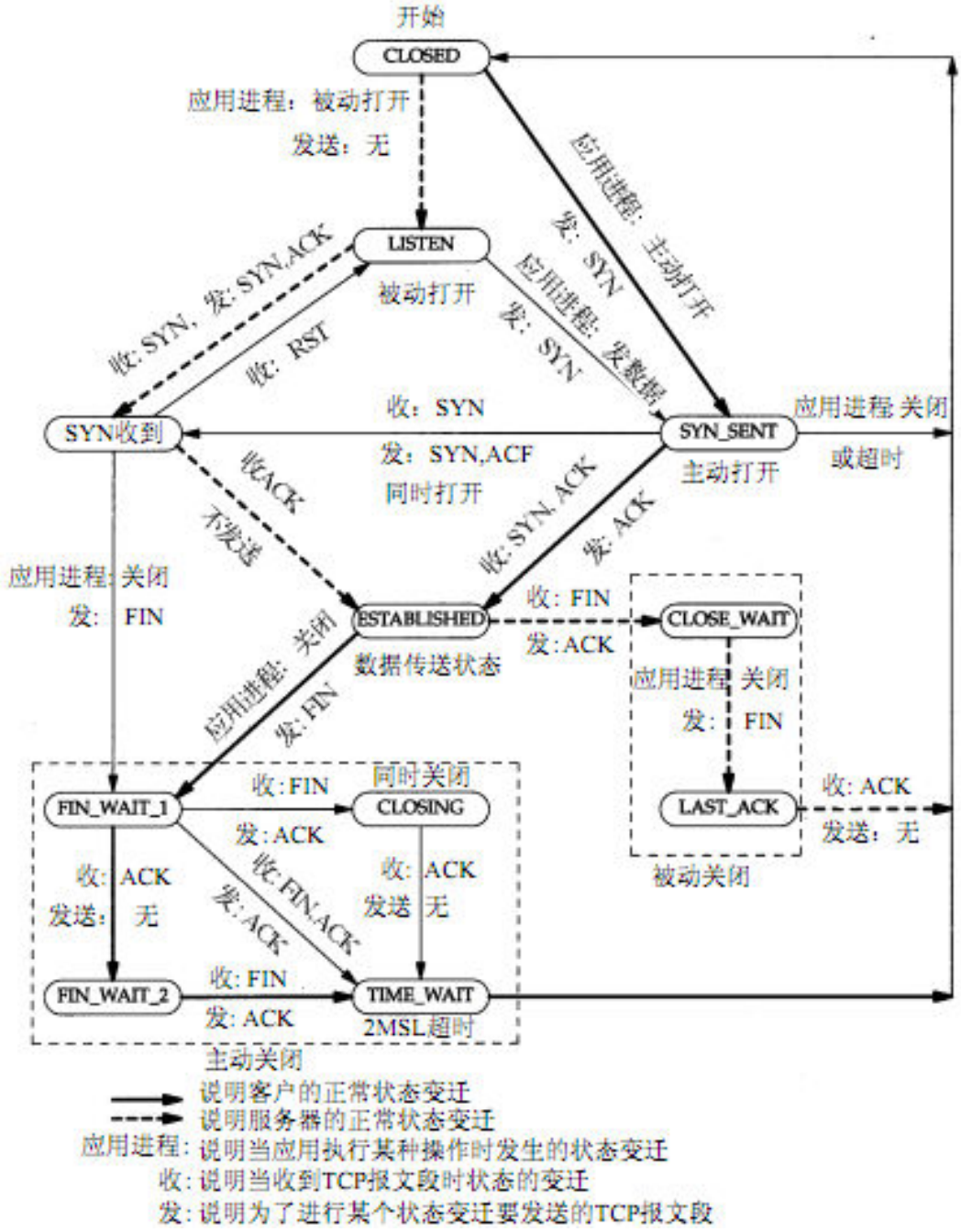
TCP 连接的断开需要四次握手过程。一个 TCP 连接是全双工（即数据在两个方向上能同时传递），因此每个方向必须单独地进行关闭。当发送数据的一方完成它的数据发送任务后它就可以发送一个 FIN 标志置 1 的握手包来终止这个方向连接。当另一端收到这个 FIN 包时，它必须通知应用层另一端已经终止了那个方向的数据传送。发送 F I N 通常是应用层进行关闭的结果，收到一个 FIN 意味着在这一方向上已经没有数据流动。一个 TCP 连接在收到一个 FIN 后仍能发送数据，此时的连接处于半关闭状态。通常首先进行关闭的一方（即发送第一个 FIN）将执行主动关闭，而另一方（收到这个 FIN）执行被动关闭。通常一方完成主动关闭而另一方完成被动关闭，但也存在双方都为主动关闭的情况，这将在后续讨论。断开一个连接的四次握手过程如下图所示，首先客户端应用程序主动执行关闭操作时，客户端会向服务器发送一个 FIN 握手包，用来关闭从客户到服务器的数据传送。当服务器收到这个 FIN，它发回一个 ACK，确认序号为收到的序号加 1。和 SYN 一样，一个 FIN 将占用一个序号。同时服务器还向其上层应用程序通告接收到结束动作，接着这个服务器程序就会关闭它的连接，导致它的 TCP 端发送一个 FIN 握手包，客户必须发回一个确认，并将确认序号设置为收到序号加 1。



在这个图中，发送 FIN 将导致应用程序关闭它们的连接，这些 FIN 的 ACK 是由 TCP 软件自动产生的。连接断开的主动发起方通常是客户端，但如果是服务器首先发起 FIN 包，上图的握手过程也是完全成立的。

15　TCP 状态转换

在理解了 TCP 连接建立于断开的过程后，再来看 TCP 的状态转换图就相对容易了。（PS：其实还是很有难度！！）

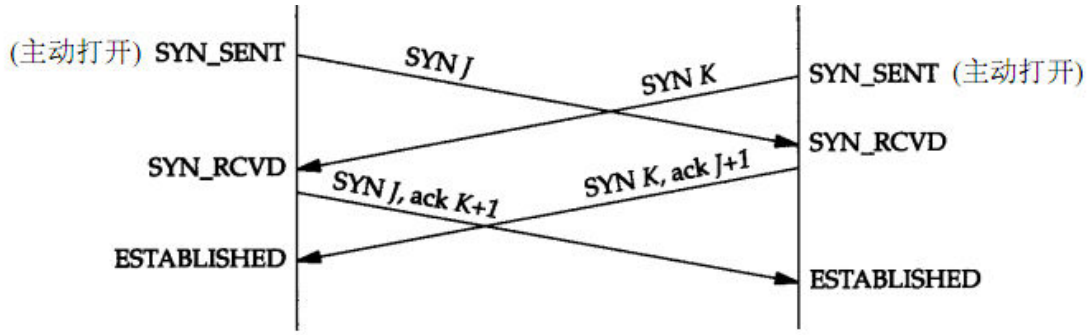


图中有两个典型的状态转换路径，第一个是客户端申请建立连接与断开连接的过程，如图中黑色粗线所示：与前面描述的一致，在客户端通过发送一个 SYN 包，主动向服务器申请一个连接，数据包发出后客户端进入 SYN_SENT 状态等待服务器的 ACK 和 SYN 包返回，当收到这个返回包后，客户端对服务器的 SYN 进行确认，然后自身进入 ESTABLISHED 状态，与前面描述的三次握手过程完全一致。当客户端申请断开连接时，它要发送 FIN 包

给服务器申请断开连接，当 FIN 包发送后，客户端进入 FIN_WAIT_1 状态等待服务器返回确认包，当收到这个确认包后，表明客户端到服务器方向的连接断开成功，此时客户端进入 FIN_WAIT_2 状态等待服务器到客户端方向的连接断开，此时当客户端收到服务器的 FIN 包时，即向服务器返回一个 ACK 包，表明服务器到客户端方向的连接断开成功，此后客户端进入 TIME_WAIT 状态，在该状态下等待 2MSL 后，客户端进入初始的 CLOSED 状态。在连接处于 2MSL 等待时，任何迟到的数据报文段将被丢弃。此过程与断开连接的四次握手过程完全相符。

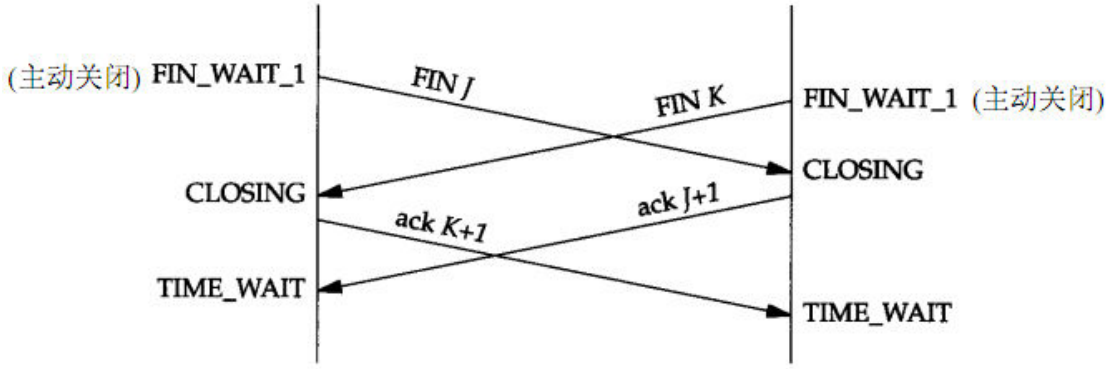
另一个典型的状态转换路径是描述服务器的，如图中虚线所示。服务器建立连接一般属于被动过程，它首先打开某个端口，进入 LISTEN 状态以侦听客户端的连接请求。当服务器收到客户端的 SYN 连接请求，则进入 SYN_REV 状态，并向客户端返回一个 ACK 及自身的 SYN 包，此后，服务器等待客户端返回一个确认包，收到该 ACK 包后，服务器进入 ESTABLISHED 状态，并可以和服务器进行稳定的数据交换过程。可见，连接建立的过程和前面描述的三次握手过程完全一致。当服务器收到客户端发送的一个断开数据包 FIN 时，则进入 CLOSE_WAIT 状态，并向上层应用程序通告这个消息，同时向客户端返回一个 ACK 包，此时客户端到服务器方向的连接断开成功；此后，当服务器上层应用处理完毕相关信息后会向客户端发送一个 FIN 包，并进入 LASK_ACK 状态，等待客户端返回一个 ACK 包，当收到返回的 ACK 包后，此时服务器到客户端方向的连接断开成功，服务器端至此进入初始的 CLOSED 状态。此过程与断开连接的四次握手过程完全相符。

上面这个转换图中还有两处较特殊的转换路线，它们分别用于处理 TCP 两端同时发起或断开连接的情况。两台主机同时执行打开操作的握手包交互如下图所示，两台主机在同时



发送自身的 SYN 请求包后各自进入 SYN_SENT 状态，等待对方的 ACK 包返回，但一定时间后，每个主机都收到对方的 SYN 包而不是 ACK 包，此时两端都判定已经遇到了同时打开的状况发生，两端都进入 SYN_RCVD 状态，并对对方的 SYN 进行确认并再次发送自己的 SYN 包，当接收到对方的 ACK+SYN 后，两边都进入 ESTABLISHED 状态。从这一点看，状态转换图中的从 SYN_RCVD 到 ESTABLISHED 转换的条件应该有两个，而图中只标出了一个。一个同时打开的连接需要交换 4 个报文段，比正常的三次握手多一个。要注意的是在这样的连接过程中，没有将任何一端称为客户或服务器，因为每一端既是客户又是服务器。

两台主机同时发起主动关闭操作的握手包交互如下，当两个主机的用户程序同时执行



关闭操作时，两主机都向对方发送一个 FIN 包，并进入 FIN_WAIT_1 状态等待对方的 ACK 返回，但一段时间后，双方各自都收到对方的 FIN 包，而不是 ACK 包，此时两主机都判定遇到了双方同时主动关闭的状况，此时，两个主机就没有必要进入 FIN_WAIT_2 状态等待对方的 FIN 包了，因为这个包刚刚已经收到，而是直接进入 CLOSING 状态，并向对方发送一个 FIN 包的确认，等到双方都收到对方的 ACK 包后，两边都各自进入 TIME_WAIT 状态。

再来详细讲解下 TIME_WAIT 状态，协议中是这样描述的：当 TCP 执行一个主动关闭，并发出最后一个 ACK 后，该连接必须在 TIME_WAIT 状态停留的时间为 2 倍的 MSL。这样可让 TCP 保证在最后的这个 ACK 丢失的情况下重新发送 ACK（另一端超时并重发最后的 FIN）。处于 TIME_WAIT 等待状态的 TCP 端口此刻还不能被其他新连接所使用。

虽然上面的状态转换图上指出，从一个连接从 LISTEN 状态转换到 SYN_SENT 状态是允许的，但是大多数的协议实现中均没有实现该转换，即执行被动打开的连接一般不要主动发起连接。

平静时间：如果主机在 TCP 状态转换过程中突然崩溃，在 TCP 重启后的一个 MSL 内，TCP 不能发送任何数据报文段，这段时间称为平静时间。设置平静时间是为了防止旧连接的延迟的数据报分组对新连接造成影响。

16 TCP 控制块

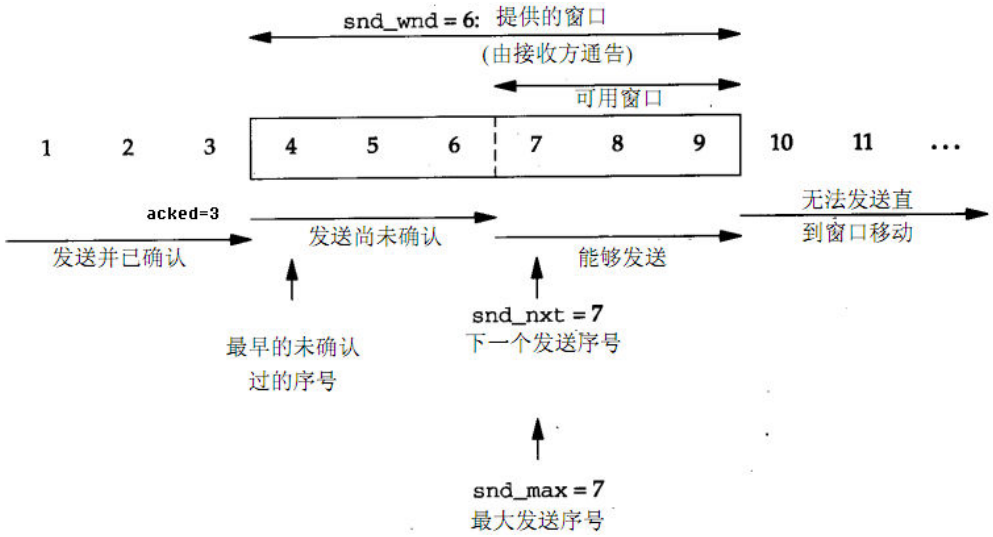
这一节正式踏入 LWIP 协议 TCP 部分的大门。先来看看它是怎样来描述一个 TCP 连接的。这个结构灰常的复杂，这里的简单描述，也并不全面，并不能清晰说明各个字段的作用，在后续的 TCP 相关内容中，会对每个用到的字段详加讲解。结构体 tcp_pcb 的源代码如下：

```
struct tcp_pcb {
  IP_PCB;  //这是一个宏，描述了连接的 IP 相关信息，包括双方 IP 地址，TTL 等信息
  struct tcp_pcb *next; //用于连接各个 TCP 控制块的链表指针
  enum tcp_state state; //TCP 连接的状态，即为状态图中描述的那些状态
  u8_t prio;        //该控制块的优先级
  void *callback_arg;//
  u16_t local_port;   //本地端口
  u16_t remote_port; //远程端口
  u8_t flags;// 附加状态信息，如连接是快速恢复、一个被延迟的 ACK 是否被发送等
#define TF_ACK_DELAY (u8_t)0x01U     /* Delayed ACK. *///这些宏定义是为 flags 字段
#define TF_ACK_NOW     (u8_t)0x02U     /* Immediate ACK. *///定义的掩码
#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 */
// 接收相关字段
  u32_t rcv_nxt;      //期望接收的下一个字节，即它向发送端 ACK 的序号
  u16_t rcv_wnd;     //接收窗口
  u16_t rcv_ann_wnd;   //通告窗口大小，较低版本中无该字段

  u32_t tmr;           // 该字段记录该 PCB 被创建的时刻
  u8_t polltmr, pollinterval; // 三个定时器，后续讲解

  u16_t rtime;     //重传定时，该值随时间增加，当大于 rto 的值时则重传发生

  u16_t mss;       //最大数据段大小

//RTT 估计相关的参数
  u32_t rttest;     //估计得到的 500ms 滴答数
  u32_t rtseq;     //用于测试 RTT 的包的序号
  s16_t sa, sv;     //RTT 估计出的平均值及其时间差

  u16_t rto;       // 重发超时时间，利用前面的几个值计算出来
  u8_t nrtx;       // 重发的次数，该字段在数据包多次超时时被使用到，与设置 rto 的值相关
```

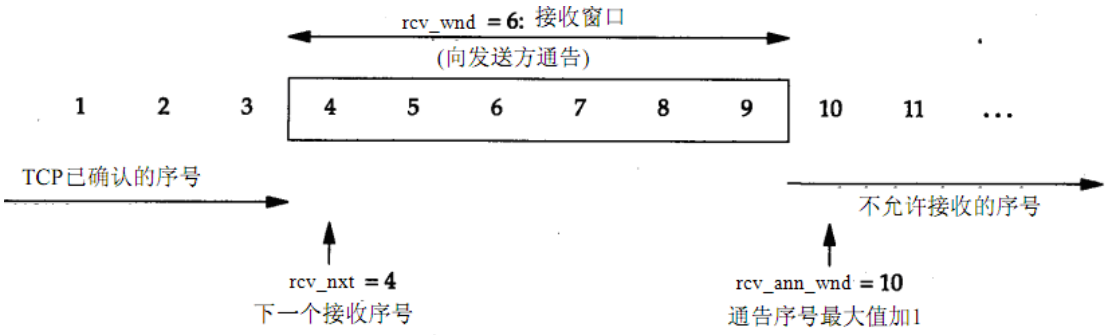
```
  // 快速重传/恢复相关的参数
  u32_t lastack; // 最大的确认序号，该字段不解
  u8_t dupacks; // 上面这个序号被重传的次数

  // 阻塞控制相关参数
  u16_t cwnd;   //连接的当前阻塞窗口
  u16_t ssthresh;   // 慢速启动阈值

  // 发送相关字段
  u32_t snd_nxt,           // 下一个将要发送的字节序号
    snd_max,               // 最高的发送字节序号
    snd_wnd,               // 发送窗口
    snd_wl1, snd_wl2,   // 上次窗口更新时的数据序号和确认序号
    snd_lbb;               // 发送队列中最后一个字节的序号
  u16_t acked;       //

  u16_t snd_buf;     // 可用的发送缓冲字节数
  u8_t snd_queuelen; // 可用的发送包数

  struct tcp_seg *unsent;     // 未发送的数据段队列
  struct tcp_seg *unacked;    // 发送了未收到确认的数据队列
  struct tcp_seg *ooseq;      // 接收到序列以外的数据包队列

#if LWIP_CALLBACK_API     // 回调函数，部分函数在较低版本没定义
  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); // 数据包接收回调函数
  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 */
                          // 剩下的所有字段在较低版本中均未定义，用到时再讲解
  u32_t keep_idle;
#if LWIP_TCP_KEEPALIVE
  u32_t keep_intvl;     // 保活定时器，用于检测空闲连接的另一端是否崩溃
  u32_t keep_cnt;
#endif /* LWIP_TCP_KEEPALIVE */

  u32_t persist_cnt;       // 这两个字段可以使窗口大小信息保持不断流动
  u8_t persist_backoff;
  u8_t keep_cnt_sent;
};
```

先说说和接收数据相关的字段 rcv_nxt， rcv_wnd， rcv_ann_wnd 和数据发送的相关字段 snd_nxt，snd_max，snd_wnd，acked。这些字段都和 TCP 中有名的滑动窗口协议有密切关

系。如下图所示，连接的双方都维持一个窗口用于数据的发送。滑动窗口把整个序列分成三部分：左边的是发送了并且被确认的分组，窗口右边是还没发送的分组，窗口内部是待确认的分组，窗口内部又分成已经发送待确认的，和未发送但将立即发送。TCP 是通过正面确认和重传技术来保证可靠性的，滑动窗口可以使发送方在收到前一个分组的确认信息前发送下一个分组，这样提高了网络的带宽利用率。



除了发送窗口外，TCP 连接的双方还各自维护了一个接收窗口，如下图，接收方的接收窗口和发送方的发送窗口对比起来看看数据包的交互过程。



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，就以此设子自己的发送窗口大小值 snd_wnd。发送方的发送窗口内包含的数据发送序列是与 ACK 序号密切相关的，即它将 ACK 序号以后的 snd_wnd 个字节序号包括在窗口内。发送方的 acked 字段就表示已经接收到的最高的 ACK 序号，snd_nxt 表示发送方即将发送数据的序号，acked 与 snd_nxt 之间的数据表示已经被发送但还未接收到 ACK，发送方也必须将他们包括在滑动窗内，以方便超时重发，snd_nxt 到发送窗口末端表示还未发送的数据。snd_max 表示不解，MARKKKKK 一下，后面再来看看。在接收方，接收处于滑动窗内编号的数据，当某个序号以前的所有序号都已经接收到后，则接收方可以 ACK 该序号，并将接收窗口向后滑动。发送方也接收到该 ACK 后，也将自己的发送窗口向后滑动。在接收方，re_nxt 表示希望接收到的下个字节序号，rev_ann_wnd 表示对方通告的窗口大小，这里也表示不解。凌乱，凌乱。。。

与发送相关的还有 cwnd 字段，这就涉及到慢启动的概念了。当发送方接收到接收方的窗口通告后，并不会一下子把窗口内允许的数据全部发送出去，因为这样做的话可能由于中间路由器转发拥塞等原因，造成网络吞吐量不稳定，带宽利用率低等不良现象。发送端的做法是用 cwnd 字段保存一个拥塞窗口，发送方取拥塞窗口与通告窗口中的最小值作为发送上

限，拥塞窗口初始值一般取 1，并在每次收到接收方的一个 ACK 后加上一个值。关于慢启动还会在后续内容中讲解。

17　TCP 建立流程

前面说了一大堆虚无缥缈的东西，而且大部分都借鉴于标准协议里面的内容，让人有点晕！这一节我们就看看如何在我们的 LWIP 上实现一个 http 服务器的过程，结合连接建立过程来理解 TCP 状态转换图和 TCP 控制块中各个字段的意义。这里先讲解一些与 TCP 相关的最基础的函数，至于是怎样将这些函数合理高效的组织起来以方便实际应用，这里先不涉及。

第一个函数是 tcp_new 函数，该函数简单的调用 tcp_alloc 函数为一个连接分配一个 TCP 控制块 tcp_pcb。tcp_alloc 函数首先为新的 tcp_pcb 分配内存空间，若内存空间不够，则函数会释放处于 TIME-WAIT 状态的 TCP 或者优先级更低的 PCB（在 PCB 控制块的 prio 字段）以为新的 PCB 分配空间。当内存空间成功分配后，函数会初始化新的 tcp_pcb 的内容，源码如下：

```
if (pcb != NULL) {
    memset(pcb, 0, sizeof(struct tcp_pcb));  // 清 0 所有字段的值
    pcb->prio = TCP_PRIO_NORMAL; // 设置 PCB 的优先级为 64，优先级在 1~127 之间
    pcb->snd_buf = TCP_SND_BUF; // TCP 发送数据缓冲区剩余大小
    pcb->snd_queuelen = 0;    // 发送缓冲中的数据包 pbuf 个数
    pcb->rcv_wnd = TCP_WND;   // 接收窗口大小
    pcb->rcv_ann_wnd = TCP_WND; // 通告窗口大小
    pcb->tos = 0;   // IP 报头部 TOS 字段
    pcb->ttl = TCP_TTL; // IP 报头部 TTL 字段
    pcb->mss = (TCP_MSS > 536) ? 536 : TCP_MSS; // 设置最大段大小，不能超过 536 字节
    pcb->rto = 3000 / TCP_SLOW_INTERVAL;// 初始超时时间值，为 6s
    pcb->sa = 0;      // 估计出的 RTT 平均值？？
    pcb->sv = 3000 / TCP_SLOW_INTERVAL;   // 估计出的 RTT 方差？？
    pcb->rtime = -1;   //重传定时器，当该值大于 rto 时则重传发生
    pcb->cwnd = 1;   // 阻塞窗口
    iss = tcp_next_iss();   // iss 为一个临时变量，保存该连接的初始数据序列号
    pcb->snd_wl2 = iss;   // 上一个窗口更新时收到的 ACK 号
    pcb->snd_nxt = iss;   // 下一个将要发送的数据编号
    pcb->snd_max = iss;   // 发送了的最大数据编号
    pcb->lastack = iss;    // 上一个 ACK 编号
    pcb->snd_lbb = iss;   // 下一个将要缓冲的数据编号

    pcb->tmr = tcp_ticks;   // tcp_ticks 是一个全局变量，记录了当前协议时钟滴答
    pcb->polltmr = 0;    // 未解？？？
#if LWIP_CALLBACK_API
    pcb->recv = tcp_recv_null;   // 注册默认的接收回调函数
#endif /* LWIP_CALLBACK_API */
    pcb->keep_idle  = TCP_KEEPIDLE_DEFAULT;
#if LWIP_TCP_KEEPALIVE      //保活定时器相关设置。。未解？？
```

```
    pcb->keep_intvl = TCP_KEEPINTVL_DEFAULT;
    pcb->keep_cnt   = TCP_KEEPCNT_DEFAULT;
#endif
    pcb->keep_cnt_sent = 0;
  }
```

上面有很多晕的地方，这些将在后续一一讲解。PCB 中的还有一些函数字段如发送、接收函数等是在具体应用中初始化的。

当一个新建的 PCB 被初始化好后，tcp_bind 函数将会被调用，用来将 IP 地址及端口号与该 TCP 控制块绑定。该函数的输入参数很明显有三个，即 TCP 控制块、IP 地址和端口号。tcp_bind 函数的工作也很简单，就是将两个参数的值赋值给 TCP 控制块中 local_ip 和 local_port 的字段。但这里有个前提，就是这个<IP 地址、端口>对没有被使用。所以，函数需要先遍历各个 pcb 链表，以保证这个<IP 地址、端口>对没有被其他 PCB 使用，这里的 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。如果遍历完这些链表后，都没有找到相应的<IP 地址、端口>对，则说明该<IP 地址、端口>对可用，则可进行上面说的赋值操作，最后，函数将这个 PCB 加入绑定完毕的 PCB 链表 tcp_bound_pcbs。

上面一共说了四种 PCB 链表，现在看看它们各自用来链接了处于哪种状态的 PCB 控制块。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 链表用来连接处于 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 链表用来连接处于 TCP 状态转换图中其他所有状态的控制块。

从状态转换图可以知，服务器端需进入 LISTEN 状态等待客户端的连接。因此，服务器端此时需要调用函数 tcp_listen 使相应 TCP 控制块进入 LISTEN 状态。可以直接的想象，要把一个控制块置为 LISTEN 状态很简单，先将其从 tcp_bound_pcbs 链表上取下来，将其 state 字段置为 LISTEN，最后再将该 PCB 挂接到链表 tcp_listen_pcbs 上。但事实上，LWIP 的实现有一定的区别，它引入了一个叫 tcp_pcb_listen 的结构，该结构与 tcp_pcb 结构相近，但是去掉了其中在 LISTEN 阶段用不到的传输控制字段，这样 tcp_pcb_listen 的结构更小，更可以节省内存空间。所以，其实 tcp_listen 是这样做的，先申请一个 tcp_pcb_listen 的结构，然后将 tcp_pcb 参数中的有用字段拷贝进来，然后将这个 tcp_pcb_listen 的结构挂接到链表 tcp_listen_pcbs 上。

到这里服务器就等待客户端发送来的 SYN 数据包进行连接了，要等待外面的数据包，这就和以前讨论过的 ip_input 函数相关了，ip_input 函数会判断 IP 包头部的协议字段，并把 TCP 数数据包通过 tcp_input 函数传递到 TCP 层。SYN 数据包当然是 TCP 层数据包，当然也要经过 tcp_input 函数进行处理并递交上层，现在就来看看 tcp_input 函数。

tcp_input 函数开始会对 IP 层递交进来的数据包进行一些基础操作，如移动数据包的 payload 指针、丢弃广播或多播数据包、数据和校验、提取 TCP 头部各个字段的值等等。接下来，函数根据接收到的 TCP 包的<IP 地址、端口>对遍历 tcp_active_pcbs 链表，寻找匹配的 PCB 控制块，若找到，则调用 tcp_process 函数对该数据包进行处理。若找不到，则再分别到 tcp_tw_pcbs 链表和 tcp_listen_pcbs 中寻找，找到则调用各自的数据包处理函数 tcp_timewait_input 和 tcp_listen_input 对数据包进行处理，若到这里都还未找到匹配的 TCP 控制块，则 tcp_input 函数会调用函数 tcp_rst 向源主机发送一个 TCP 复位数据包。

这里我们的 TCP 控制块处于 LISTEN 状态，连接在 tcp_listen_pcbs 上，正在等待一个 SYN 数据包。因此，当等到该数据包后，函数 tcp_listen_input 应该被调用。从状态转换图

上可以看出，处于 LISTEN 状态的 TCP 控制块只能响应 SYN 握手包，所以，tcp_listen_input 函数对非 SYN 握手包返回一个 TCP 复位数据包，若一个数据包不是 SYN 包，则其 TCP 包头中的 ACK 字段通常会被置 1，所以 tcp_listen_input 函数是通过检验该位来实现的。接下来，函数通过验证 SYN 位来确认该包是否为 SYN 握手包。若是，则需要新建一个 tcp_pcb 结构，因为处于 tcp_listen_pcbs 上的控制块结构是 tcp_pcb_listen 结构的，而其他链表上的控制块结构是 tcp_pcb 结构的，所以这里新建一个 tcp_pcb 结构，并将相应 tcp_pcb_listen 结构拷贝至其中，同时在 tcp_active_pcbs 链表中添加这个新的 tcp_pcb 结构。这样新的 TCP 控制块就处在 tcp_active_pcbs 中了，注意此时的这个 tcp_pcb 结构的 state 字段应该设置为 SYN_RCVD，表示进入了收到 SYN 状态。注意 tcp_listen_pcbs 链表中的这个 tcp_pcb_listen 结构还一直存在，它并不会被删除，以等待其他客户端的连接，服务器正是需要这样的功能。

到这里，函数 tcp_listen_input 还没完。它应该从收到的 SYN 数据报中提取 TCP 头部中选项字段的值，并设置自己的 TCP 控制块。这里要被调到用的函数叫 tcp_parseopt，它目前仅能够做的是提取选项中的 MSS（最长报文大小）字段，在 LWIP 以后的更高版本中，该函数将被扩充，以支持更多的 TCP 选项。此后，函数还可以调用 tcp_eff_send_mss 来设置控制块中 mss 字段的值，该函数可直译为“有效发送最长报文大小”，所谓有效，就是指收到 SYN 数据包中的 MSS 值不能大于我的硬件支持的最大发送报文长度，即硬件的 MTU。因此当收到的 MSS 值更大时，设置控制块中 mss 字段值会被设置为 MTU，而不是 MSS。

最后，函数需要向源端返回一个带 SYN 和 ACK 标志的握手数据包，并可以向源端通告自己的 MSS 大小。发送数据包是通过 tcp_enqueue 和 tcp_output 函数共同完成的。关于数据包的发送，将在以后介绍。

最最后，来看看函数 tcp_listen_input 内部的关键源代码部分，这几行代码涉及到 TCP 控制块内部各个字段值的设置，其中很重要的就是滑动窗口相关的字段。

```
ip_addr_set(&(npcb->local_ip), &(iphdr->dest)); //复制本地 IP 地址
npcb->local_port = pcb->local_port;   //复制本地端口
ip_addr_set(&(npcb->remote_ip), &(iphdr->src)); //复制源 IP 地址
npcb->remote_port = tcphdr->src;   //复制源端口
npcb->state = SYN_RCVD;   // 设置 TCP 状态
npcb->rcv_nxt = seqno + 1;   // 期望接收到的下一个字节序号
npcb->snd_wnd = tcphdr->wnd; // 设置发送窗口大小
npcb->ssthresh = npcb->snd_wnd;   //快速启动阈值设为和发送窗口大小相同？？
npcb->snd_wl1 = seqno - 1;//该字段？？
npcb->callback_arg = pcb->callback_arg; //该字段？？
#if LWIP_CALLBACK_API
  npcb->accept = pcb->accept;     //接收回调函数
#endif /* LWIP_CALLBACK_API */
```

其中 npcb 表示新建的 tcp_pcb 结构，还有很多不懂的地方，为啥仅仅拷贝保留了这几个字段，其他字段直接被忽略？

18 TCP 状态机

服务器端接收到 SYN 握手包，向客户端返回带 SYN 和 ACK 的握手包，并将相应 TCP 控制块置为 SYN_RCVD 状态，并挂在 tcp_active_pcbs 链表上。以后，继续等待客户端发送过来的握手包，这次，服务器期望的是接收一个 ACK 包以完成建立连接要求的三次握手操作。

还是和前几次一样，数据包进来通过 ip_input 传递给 tcp_input，后者在三个链表中查找一个匹配的连接控制块。这次进来的是客户端发送的 ACK 握手包，服务器端相应的 tcp 控制块一定是在 tcp_active_pcbs 链表上。接下来，以查找到的 tcp_pcb 结构为参数，调用 TCP 状态机函数 tcp_process 处理输入数据段。在讲解 tcp_process 函数之前，先来看看这个状态机函数要用到的一些重要的全局变量，这些变量是在 tcp_input 函数中，通过接收数据段的各个字段的值进行设置的。全局变量 tcphdr 指向收到的数据段的 TCP 头部；全局变量 seqno 记录了 TCP 头部中的数据序号字段；全局变量 ackno 存储确认序号字段；全局变量 flags 表示 TCP 首部中的各个标志字段；全局变量 tcplen 表示数据报中数据的长度，对于 SYN 包和 FIN 包，该长度为 1；全局变量 inseg 用于描述收到的数据内容，它是 tcp_seg 类型的，tcp_seg 这个结构体后面再讲；全局变量 recv_flags 用来标识 tcp_process 函数对数据段的处理结果，初始化为 0。

tcp_process 函数首先判断该数据段是不是一个复位数据段，若是则进行相应的处理，这里先跳过这小部分。直接到达 tcp_process 函数状态机部分，它就是对 TCP 状态转换图的简单代码诠释。必须要贴一大段代码上来了，这比任何语言更能说清楚问题，注意下面的代码已经被我去掉了相关注释和编译输出部分。

```
switch (pcb->state) {
  case SYN_SENT:              // 客户端将 SYN 发送到服务器等待握手包返回
  if ((flags & TCP_ACK) && (flags & TCP_SYN) //实际收到 ACK 和 SYN 包
          && ackno == ntohl(pcb->unacked->tcphdr->seqno) + 1) {
        pcb->snd_buf++;
        pcb->rcv_nxt = seqno + 1;
        pcb->lastack = ackno;
        pcb->snd_wnd = tcphdr->wnd;
        pcb->snd_wl1 = seqno - 1; /* initialise to seqno - 1 to force window update */
        pcb->state = ESTABLISHED;

        tcp_parseopt(pcb);        // 处理选项字段
      #if TCP_CALCULATE_EFF_SEND_MSS // 根据需要设置有效发送 mss 字段
        pcb->mss = tcp_eff_send_mss(pcb->mss, &(pcb->remote_ip));
      #endif
        pcb->ssthresh = pcb->mss * 10;//由于重新设置了 mss 字段值，所以要重设 ssthresh 值

        pcb->cwnd = ((pcb->cwnd == 1) ? (pcb->mss * 2) : pcb->mss);   // 设置阻塞窗口
        --pcb->snd_queuelen;   //要发送的数据段个数减 1？？
        rseg = pcb->unacked;      //取下要被确认的字段
```

```
        pcb->unacked = rseg->next;

        if(pcb->unacked == NULL) //没有字段需要被确认，则停止定时器
          pcb->rtime = -1;
        else {                    //否则重新开启定时器
          pcb->rtime = 0;
          pcb->nrtx = 0;
        }
        tcp_seg_free(rseg);   //释放已经被确认了的段
        TCP_EVENT_CONNECTED(pcb, ERR_OK, err);
        tcp_ack_now(pcb);   //返回 ACK 握手包
      }
      else if (flags & TCP_ACK) {   //仅仅只有 ACK 而无 SYN 标志
        tcp_rst(ackno, seqno + tcplen, &(iphdr->dest), &(iphdr->src),
          tcphdr->dest, tcphdr->src);//不支持半打开状态，所以返回一个复位包
      }
      break;
    case SYN_RCVD:     // 服务器端发送出 SYN+ACK 后便处于该状态
      if (flags & TCP_ACK && //  在 SYN_RCVD 状态接收到 ACK 返回包
          !(flags & TCP_RST)) {    //  判断 ACK 序号是否合法
        if (TCP_SEQ_BETWEEN(ackno, pcb->lastack+1, pcb->snd_nxt)) {
          u16_t old_cwnd;
          pcb->state = ESTABLISHED; //  进入 ESTABLISHED 状态
          old_cwnd = pcb->cwnd;     //  保存旧的阻塞窗口
          accepted_inseq = tcp_receive(pcb); //  若包含数据则接收数据段
          pcb->cwnd = ((old_cwnd == 1) ? (pcb->mss * 2) : pcb->mss);//重新设置阻塞窗口

          if ((flags & TCP_FIN) && accepted_inseq) { //  如果 ACK 包同时含有 FIN 位且
            tcp_ack_now(pcb);              //已经接收完了最后的数据，则响应 FIN
            pcb->state = CLOSE_WAIT;//进入 CLOSE_WAIT 状态
          }
        }
        else { //不合法的 ACK 序号则返回一个复位包
          tcp_rst(ackno, seqno + tcplen, &(iphdr->dest), &(iphdr->src),
                  tcphdr->dest, tcphdr->src);
        }
      }
      break;
    case CLOSE_WAIT: //服务器，TCP 处于半打开状态，在该方向上不会再接收到数据包
//服务器在此状态下会一直等待上层应用执行关闭命令 tcp_close，并将状态变为 LAST_ACK

case ESTABLISHED://稳定状态，客户端会一直保持稳定状态直到上层应用调用 tcp_close 函数关闭连接，将状态变为 FIN_WAIT_1
      accepted_inseq = tcp_receive(pcb); //直接接收数据
```

```
    if ((flags & TCP_FIN) && accepted_inseq) { //同时数据包有 FIN 标志，且接收到了最后
      tcp_ack_now(pcb);          的数据，则响应 FIN
      pcb->state = CLOSE_WAIT; //  进入 CLOSE_WAIT 状态
    }
    break;
  case FIN_WAIT_1://客户端独有的状态
    tcp_receive(pcb); //还可以接收来自服务器的数据
    if (flags & TCP_FIN) { //如果收到 FIN 包
      if (flags & TCP_ACK && ackno == pcb->snd_nxt) {//且还有 ACK，则进入 TIME_WAIT
        tcp_ack_now(pcb);   //  发 ACK
        tcp_pcb_purge(pcb);   //  清除该连接中的所有现存数据
        TCP_RMV(&tcp_active_pcbs, pcb);   //  从 tcp_active_pcbs 链表中删除
        pcb->state = TIME_WAIT;   //  置为 TIME_WAIT 状态
        TCP_REG(&tcp_tw_pcbs, pcb);   //  加入 tcp_tw_pcbs 链表
      } else {//无 ACK，则表示两端同时关闭的情况发生
        tcp_ack_now(pcb);   //  发送 ACK
        pcb->state = CLOSING; //  进入 CLOSING 状态
      }
    } else if (flags & TCP_ACK && ackno == pcb->snd_nxt) {//不是 FIN 包，而是有效 ACK
      pcb->state = FIN_WAIT_2; //则进入 FIN_WAIT_2 状态
    }
    break;
  case FIN_WAIT_2:   //客户端在该状态等待服务器返回的 FIN
    tcp_receive(pcb); //还可以接收来自服务器的数据
    if (flags & TCP_FIN) {//如果收到 FIN 包
      tcp_ack_now(pcb); //  发 ACK
      tcp_pcb_purge(pcb); //  清除该连接中的所有现存数据
      TCP_RMV(&tcp_active_pcbs, pcb); //  从 tcp_active_pcbs 链表中删除
      pcb->state = TIME_WAIT; //  置为 TIME_WAIT 状态
      TCP_REG(&tcp_tw_pcbs, pcb); //  加入 tcp_tw_pcbs 链表
    }
    break;
  case CLOSING: //  进入了同时关闭的状态，这种情况极少出现
    tcp_receive(pcb); //还可以接收对方的数据
    if (flags & TCP_ACK && ackno == pcb->snd_nxt) { //若是有效 ACK
      tcp_ack_now(pcb);          //  与上面类似的操作，不解释了
      tcp_pcb_purge(pcb);
      TCP_RMV(&tcp_active_pcbs, pcb);
      pcb->state = TIME_WAIT;
      TCP_REG(&tcp_tw_pcbs, pcb);
    }
    break;
  case LAST_ACK://服务器端(被动关闭端)能出现该状态
    tcp_receive(pcb); //还可以接收对方的数据
```

```
    if (flags & TCP_ACK && ackno == pcb->snd_nxt) {//接收到有效 ACK
      recv_flags = TF_CLOSED;// 全局变量 recv_flags 用于标识该数据段进行了哪些处理
    }// 以便 tcp_input 的后续处理，此时并没有把 pcb->state 的状态设置为 CLOSED 状态
    break;
  default:
    break;
  }
```

这就是 TCP 状态机的大致流程图，如果对照着 TCP 状态转换图来看，你会觉得它是如此的简单。不过还有很多搞不清楚的地方，比如控制块中各个字段的值特别是阻塞窗口和发送接收窗口等的值，它们代表了什么意义，为什么要如此设置，等等。

注意 TCP 状态转换图中的双方同时打开的情况，即从 LISTEN 状态到 SYN_SENT 状态，SYN_SENT 状态到 SYN_RCVD 状态，没有被实现。许多其他 TCP/IP 实现，如 BSD 中也是这样的，没有实现这部分功能。因为在实际应用中这种情况几乎不可见，所以实现并没有严格按照协议来实行。

19　TCP 输入输出函数 1

这节从 tcp_receive 函数入手，逐步深入了解控制块各个字段的意义以及整个 TCP 层的运行机制，足足 600 行，神想吐血。源码注释的该函数功能为：检查收到的数据段是不是对已发数据段的确认，如果是，则释放相应发送缓冲中的数据；接下来，如果该数据段中有数据，应将数据挂接到控制块的接收队列上（pcb->ooseq）。如果数据段同时也是对正在进行 RTT 估计的数据段的确认，则 RTT 计算也在这个函数中进行。我晕，陷入了恶性循环。越看越难，越看越说不清，TCP 的东西太多了。要讲清楚 tcp_receive 还得说清楚 tcp_enqueue，不管了，先硬着头皮写下去！源码注释对该函数的功能描述很简单：将数据包或者连接的控制握手包放到 tcp 控制块的发送队列上。这个函数的原型为

```
err_t
tcp_enqueue( struct tcp_pcb *pcb, void *arg, u16_t len,u8_t flags,
                        u8_t apiflags,u8_t *optdata, u8_t optlen )
```

其中有几个重要的输入参数：pcb 是相应连接的 TCP 控制块；arg 是要发送的数据的指针；len 是要发送的数据的长度，以字节为单位；flags 是 TCP 数据段头部中的标识字段，主要用于连接建立或断开的握手；apiflags 表示要对该数据段做的操作，包括是否拷贝数据、是否设置 PUSH 标志；optdata 表示 TCP 头部中的选项字段的值，optlen 表示选项字段的长度。

tcp_enqueue 首先确认要发送的数据长度 len 是否小于当前连接能用的数据发送缓冲区大小，即 pcb->snd_buf，若缓冲区不够，则不会对该数据进行任何处理（其实这个缓冲区并不存在，只是用 snd_buf 标识出连接还能缓存的数据量）。接着，将要发送的数据段的序号字段设置为 pcb->snd_lbb，然后判断 pcb->snd_queuelen 值是否超过了所允许挂接的数据包的上限值 TCP_SND_QUEUELEN，如果超过了该上限值，则函数也不会对这个要发送的数据段进行处理。接下来 tcp_enqueue 函数会将数据组装成为 tcp_seg 类型的数据段，根据数据长度的大小不同，可能需要几个 tcp_seg 类型结构才能描述完所有的数据，每个数据段中的 TCP 头部部分字段值要在这里都要被设置，包括数据序号、标志字段。最后，所有创建好的 tcp_seg 类型结构都是连接在 queue 队列上的，queue 是函数的一个临时变量。接下来，函数 tcp_enqueue 需要将 queue 队列上的数据段挂接到 TCP 控制块的 unsent 队列上，这里又有好几种情况，即 unsent 队列是否为空的情况，若为空，则直接挂接，若不为空，则需要将 queue 挂接在 unsent 队列的最后一个 tcp_seg 之后，如果挂接点处相邻两个 tcp_seg 所包含的数据大小小于最长发送段大小 pcb->mss，且相邻的两个段都不是 FIN 包或 SYN 包，则需要将两个段合并为一个段。最后，函数需要调整 TCP 控制块中的相关字段的值，这点也是我最关心的地方，

```
  if ((flags & TCP_SYN) || (flags & TCP_FIN)) {    //发送 SYN 或 FIN 包被认为数据长度为 1
      ++len;
    }
  if (flags & TCP_FIN) {          // 若为 FIN 包，则设置 flags 字段为相应值
    pcb->flags |= TF_FIN;
  }
  pcb->snd_lbb += len;    // 下一个要被缓冲数据的序号，注意与 snd_nxt 不同
  pcb->snd_buf -= len;    // 减小空闲的发送缓冲数，注意这个缓冲区并不是真正存在的
  pcb->snd_queuelen = queuelen;    // 未发送队列中的 pbuf 个数
```

因为在看滑动窗口时怎样实现的时候，这些字段是非常关键的。

凌乱凌乱，讲了 tcp_enqueue 函数，又不得不讲讲 tcp_output 函数。tcp_output 函数有个唯一的参数，即某个链接的 TCP 控制块指针 pcb，函数把该控制块的 unsent 队列上数据段发送出去或直接发送一个 ACK 数据段。如果调用该函数时，控制块的 flags 字段设置了 TF_ACK_NOW 标志，则函数必须马上发出去一个带有 ACK 标志。因此，如果此时 unsent 队列中无数据发送或者发送窗口此时不允许发送数据，则函数需要发出去一个不含任何数据的 ACK 数据报。当没有 TF_ACK_NOW 置位，或者 TF_ACK_NOW 置位但该 ACK 能和数据段一起发送出去时，则此时函数会取下 unsent 队列上的数据段发送出去（这里先暂时不考虑 nagle 算法）。发送一个具体的数据段是通过调用函数 tcp_output_segment 实现的，这个函数主要是填充待发送数据段的 TCP 头部中的确认序号为 pcb->rcv_nxt，通告窗口大小为 pcb->rcv_ann_wnd，校验和字段，最后 tcp_output_segment 将数据包递交给 IP 层发送。当然，tcp_output_segment 还有许多其他操作，这里我们先不关心。

好了，还是回到 tcp_output 这条正道上来，数据段被发送出去后，这个函数还需要设置控制块相关字段的值。这里我最关心的还是与滑动窗密切相关的字段，

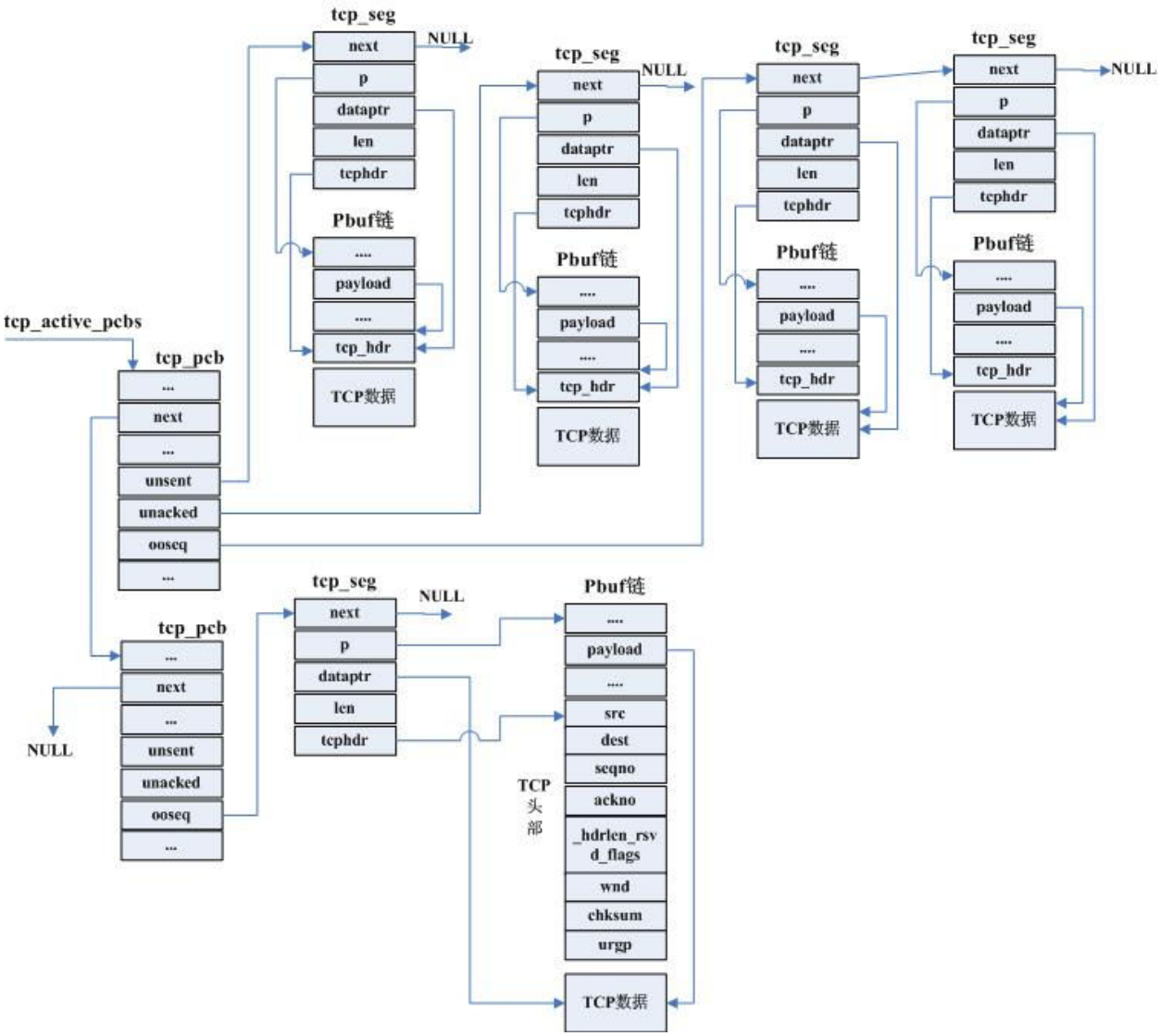
```
pcb->snd_nxt = ntohl(seg->tcphdr->seqno) + TCP_TCPLEN(seg); // 下一个要发送的字节序号
if (TCP_SEQ_LT(pcb->snd_max, pcb->snd_nxt)) {
    pcb->snd_max = pcb->snd_nxt; // 最大发送序号
  }
```

接下来，函数将发送出去的这个段挂接在控制块 unacked 链表上，以便后续的重发等操作。到这里，unsent 队列上的第一个数据段就处理完了，tcp_output 函数还会依次按照上述方法处理 unsent 队列上剩下的各个数据段，直到数据被全部发送出去或者发送窗口被填满。

现在可以来看看 tcp_receive 这个庞然大物了。这个函数简单的来说就是操作 TCP 控制块中的 unsent、unacked、ooseq 字段，这三个字段用于连接 TCP 的各种数据段。unsent 用于连接还未被发送出去的数据段、unacked 用于连接已经发送出去但是还未被确认的数据段、ooseq 用于连接接收到的无序的数据段。这个三个字段都是 tcp_seg 类型的指针，结构体 tcp_seg 用于描述一个 TCP 数据段，源代码如下：

```
struct tcp_seg {
    struct tcp_seg *next;      // 用来建立链表的指针
    struct pbuf *p;            // 数据段 pbuf 指针
    void *dataptr;             // 指向 TCP 段的数据区
    u16_t len;                 // TCP 段的数据长度
    struct tcp_hdr *tcphdr;    // 指向 TCP 头部
    };
```

掌握这个结构体很重要，这是理解 tcp_receive 函数的关键。从下面的图中可以看出，tcp_seg 结构时怎样描述一个 TCP 数据段的。能够进行数据段收发的 TCP 控制块都被连接在链表 tcp_active_pcbs 上，每个控制块的三个指针 unsent、unacked、ooseq 连接了该连接相关的数据。unsent、unacked 链表与 ooseq 链表上的 tcp_seg 结构描述数据段的方式不尽相同，从图上可知，unsent、unacked 链表的 tcp_seg 结构 dataptr 和 tcphdr 字段都指向 pbufs 的数据起始位置，即 TCP 头部位置；而 ooseq 链表上的 tcp_seg 结构 dataptr 指向了 TCP 数据段的开始位置，tcphdr 字段指向了 TCP 头部。且对于链表 ooseq 上的数据包 pbuf，其 payload 指针也是指向 TCP 数据段的开始位置，而不是指向 pbuf 的数据开始位置。这是因为链表 ooseq 上的 TCP 数据段都是从 IP 层递交上来的，TCP 层已经调用 tcp_input 函数将数据包的 payload 指针指向了 TCP 数据段的开始位置。


tcp_seg
next
NULL
p
dataptr
len
tcphdr
Pbuf链
....
payload
....
tcp_hdr
TCP数据
tcp_active_pcbs
tcp_pcb
...
next
...
unsent
unacked
ooseq
...
NULL
TCP
头
部
src
dest
seqno
ackno
_hdrlen_rsv
d_flags
wnd
chksum
urgp

20　TCP 输入输出函数 2

继续说 tcp_receive 函数。前面说过，当 tcp_input 函数接收到来自 IP 层的数据包后，会将相关 TCP 头部字段保存在一些全局变量中，全局变量 seqno 记录了 TCP 头部中的数据序号字段；全局变量 ackno 存储确认序号字段；全局变量 flags 表示 TCP 首部中的各个标志字段；全局变量 tcplen 表示数据报中数据的长度，对于 SYN 或 FIN 置为的数据包，该长度要加 1；全局变量 inseg 用于描述收到的数据内容，它是 tcp_seg 类型的。在 TCP 状态机的实现函数 tcp_process 中，会在不同地方来调用函数 tcp_receive 来处理输入的 TCP 数据段。

tcp_receive 首先判断是否可以根据接收到的数据段来跟新本地的发送窗口大小，因为数据段中有对方的窗口大小通告。与窗口跟新的 TCP 控制块中的字段是：snd_wnd、snd_wl1、snd_wl2，分别表示当前窗口大小、上一次窗口更新时接收到的数据序号（seqno）、上一次窗口更新时接收到的确认序号（ackno）。有三种情况可以导致本地发送窗口跟新：收到数据段中的 seqno 大于 snd_wl1（连接建立握手过程中使用这种方式）、新 seqno 等于 snd_wl1 且新确认号 ackno 大于 snd_wl2、新确认号 ackno 等于 snd_wl2 且数据段中有比 snd_wnd 更大的窗口通告。为什么是这三种情况呢，表示不懂，还需努力。当满足上面三种情况中任一种时，则需要进行窗口的更新，即重新设置 snd_wnd、snd_wl1、snd_wl2 三个值。

接下来，tcp_receive 函数根据接收数据段的确认序号 ackno 值进行相关操作。该值是对方返回的数据确认信息。TCP 控制块中的 lastack 字段记录了连接收到的上一个确认序号的值，函数比较 lastack 与 ackno 值判断该 ACK 是否为重复的 ACK，如果是，则说明网络发生了异常，此时应进行拥塞避免算法或慢启动算法，这里先不涉及这两个算法。

正常情况下 ackno 应该是处在 lastack+1 与 snd_max 之间的，在这种情况下，函数首先设置 TCP 控制块相关字段的值，其中最重要的是 snd_buf、lastack 字段，当然还有包括与超时重传、慢启动等相关的字段，这里先不讨论。函数接下来根据这个 ackno 来处理控制块的 unacked 队列，因为 unacked 队列连接的是发送后而未被确认的数据段，因此，当收到 ackno 后，应该遍历 unacked 队列，将数据段中所有数据编号都小于等于 ackno 的数据段移除。当所有满足要求的数据段移除成功后，函数应检查 unacked 队列是否为空，若是，则停止重传定时器，否则复位重传定时器。

接下来函数还要根据这个 ackno 去处理控制块的 unsent 队列，This may seem strange(源码注释中这样说)！因为对于需要重传的 TCP 段，LWIP 是直接将他们挂在 unsent 队列上的，所以收到确认后，可能是对已经发生了超时的数据段的确认，所以函数会遍历 unsent 队列，将数据段中所有数据编号都小于等于 ackno 的数据段移除，当然 ackno 的合法性在这种情况下也是一个考虑因素，就是 ackno 不能大于 snd_max 的值，即已经发送了的最大数据序号。

接下来，如果 ackno 确认了正在进行 RTT 估计的数据段，则 RTT 的估计在此处进行。控制块的 rtseq 字段记录了正在进行 RTT 估计的数据段的数据序号。关于 RTT，放在后面讨论。

再接下来，函数要对收到的数据段中的数据进行处理了，前面说过了，seqno 记录了接收到的数据的起始序号，tcplen 表示数据的长度，对于 SYN 包和 FIN 包，该长度要加 1，这两个全局变量在这部分中起着很重要的作用。接收窗口与接收数据密切相关，控制块中的三个重要字段 rcv_nxt、rcv_wnd 与 rcv_ann_wnd，分别表示期望接收的下一字节编号、接收窗口大小、向对方通告的接收窗口大小。对数据的处理主要做了以下三件事：如果收到的数据段编号包含 rcv_nxt，则说明这个数据段是顺序到达的段，这个段可被直接递交给上层；

否则这个数据段需要按照其数据编号被连接在 ooseq 链表上，在链表上插入数据段的过程中，可能出现数据重叠的现象，此时需要将重复的数据移除；最后，会检查 ooseq 链表上的第一个数据是否是顺序到达的段，若是，则将该数据段提交给上层。

首先，若数据起始编号 seqno 并不恰恰是 rcv_nxt，而是比 rcv_nxt 大，同时数据段中又有数据的编号是 rcv_nxt，此时需要将数据段截断，使其起始编号恰好是 rcv_nxt。若 seqno 小于 rcv_nxt，且数据段中所有数据的编号均小于 rcv_nxt，说明这是一个重复的数据段，对这类重复数据段只是响应一个 ACK 包给源端，并不做其他处理。

接下来，判断数据段是否在接收窗口内（seqno 的值可能已在上一步中被调整了），接收窗口的起始字节序号分别是 rcv_nxt 与 rcv_nxt + rcv_ann_wnd – 1。若在窗口内，则可以根据数据段在窗口内的位置对数据进行进一步的处理。

如果数据段处在窗口的起始位置，即 seqno 等于 rcv_nxt，则说明该数据段是连续到来的数据段，可以直接把它递交给上层。但这里 LWIP 不是直接将这段数据交给上层的，而是将数据段挂接在 ooseq 链表的第一个位置，由于 ooseq 链表上的其他已挂接的数据段可能因为这个数据段的到来而变为有序(可以这样认为，若比某个数据段起始编号小的数据都到了，则该数据段就是有序的了，可以递交给上层)，这样，协议就可以将尽可能多的数据递交给上层。数据段被插入到 ooseq 链表的第一个位置后，可能遇到链表上第一个和第二个数据段数据重叠的情况，此时需要将第一个数据的尾部截断，以避免数据的重复。全局变量 recv_data 指针用来指向可以上上层递交的数据包，所以，到这里 tcp_receive 函数将 recv_data 指向第一个数据段中的数据，并设置好 rcv_nxt、rcv_wnd、rcv_ann_wnd 的值，接下来，遍历 ooseq 链表后续的数据段，将所有有序的数据都挂接到 recv_data 指针上，挂接完成后，向源端返回一个确认包。实际源码实现和上面的过程完全相同，但是实现上和上述有一点小差别，从上面的描述可以看出，被插入到第一个位置的数据段会马上被取下来挂到 recv_data 指针上，所以这个插入操作是没有必要实现的，事实也如此。

如果数据段不是处在窗口的起始位置，则向源端返回一个立即确认数据段，并将该数据段挂接到 ooseq 链表上，挂接数据段主要根据数据的编号在链表中选择一个合适的位置插入，同时如果插入操作可能出现数据重叠的现象，则需要将数据进行截断操作。想起了伪代码这个词，就用伪代码来描述一下这个过程：

```
ZSL = 1；
For（从 ooseq 取下第 ZSL 个数据段；该数据段不为空；ZSL++）
{   if 该数据段起始编号==要插入的数据段起始编号
        if 要插入的数据段更长
            用要插入的数据段代替第 ZSL 个数据段；
            删除第 ZSL 个数据段；
            if   插入后的数据段与后一个数据段有数据重合
                 将插入数据段的尾部截断；
            end
            跳出循环；
        else 跳出循环；
        end
  else   if   ZSL==1     // 即是第一个数据段
            if 该数据段起始编号>要插入的数据段起始编号
              将新数据段插入到第一个位置；
              若与后一个数据段有数据重合，则截断数据段尾部；
              跳出循环；
```

```
            end
        else          // 不是第一个数据段
            if 要插入的数据段起始编号在第 ZSL-1 个和第 ZSL 个数据段起始编号之间
                将数据段插入到第 ZSL 位置上；
                若与后一个数据段有数据重合，则截断数据段尾部；
                若与前一个数据段有数据重合，则截断前一个数据段尾部；
                跳出循环；
            end
        end

        if   第 ZSL 个数据段是链表上最后一个数据段
            将新数据段插在链表尾上；
            若与前一个数据段有数据重合，则截断前一个数据段尾部；
            跳出循环；
        end
end
}
```

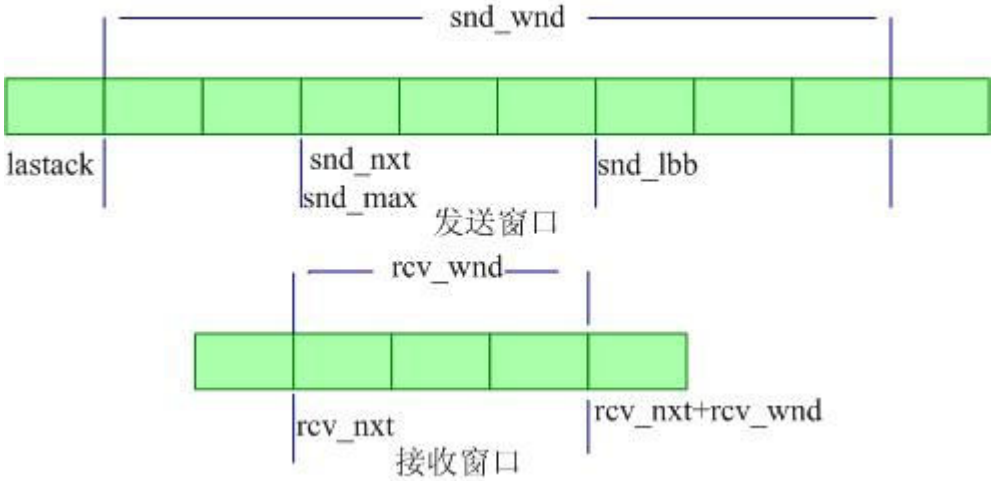
回过神来，如果数据段落在窗口之外，则直接向源端返回一个立即确认数据段。到这里函数 tcp_receive 就完成了，如果它将新的数据挂接到了数据指针 recv_data 上，则该函数返回一个 1，否则返回 0。

21 TCP 滑动窗口

前面讨论了 TCP 层所有基础性的东西，这里开始讲支撑 TCP 功能的各种 TCP 机制，包括滑动窗口机制、TCP 定时器、RTT 估计与超时重传机制、慢启动与拥塞避免算法、快速恢复与重传、Nagle 算法等。先说滑动窗口是怎样实现的，前面在 TCP 控制块中，我们已经涉及到了与发送和接收窗口相关的字段，这里再来看看。发送窗口相关字段有：snd_wl1、snd_wl2、snd_nxt、snd_max、lastack、snd_lbb、snd_wnd。That's too many!!

snd_wl1 和 snd_wl2 字段与窗口的更新密切相关，它们分别表示上一次窗口更新时所收到的数据段中的 seqno 和 ackno。当接收到新的数据段时，是否需要提取数据段中的窗口通告并进行窗口更新，就要看 snd_wl1 和 snd_wl2 字段的值与数据段中的 seqno 和 ackno 间的大小关系了，如果满足窗口更新条件，则发送窗口被更新；字段 snd_wnd 保存了窗口大小。lastack 字段表示最后一个有效确认的 ackno；snd_nxt 表示下一个将要发送的数据起始编号；snd_max 表示表示发送了的最大数据序号+1（这貌似和 snd_nxt 的值一样，但为什么要用 snd_max 字段，表示不解），snd_lbb 表示在下一个将被缓存且在发送窗口内的数据编号。后面三个字段太纠结了，看看下面的图理解可能好理解一点。

snd_max 和 snd_nxt 的值相同，从 lastack 到 snd_nxt 之间的数据是已经发送出去但未被确认的，它们被挂接在控制块的 unacked 链表上，从 snd_nxt 到 snd_lbb 间的数据是已经被应用程序递交给协议栈，但并未被协议栈发送出去的数据，这些数据挂接在控制块的 unsent 链表上，从 snd_lbb 到窗口末端表示可用的发送缓冲，但应用层还未递交数据，下一次应用层递交的数据将从 snd_lbb 开始编号。



接收窗口有三个字段 rcv_nxt、rcv_wnd、rcv_ann_wnd，其中 rcv_nxt 表示下一个将要接收的数据序号，rcv_wnd 表示接收窗口的大小，rcv_ann_wnd 表示向发送方通告的窗口大小。接收窗口相对于发送窗口来说更简单，我们也在上图中加以了描述。那么 TCP 连接两端的滑动窗口是如何实现的呢？现在以客户机与服务器之间连接建立过程来看看滑动窗是如何工作的。

（1）服务器端：首先调用函数 tcp_alloc 分配一个新的 TCP 控制块，在函数 tcp_alloc 中，控制块的 rcv_wnd 和 rcv_ann_wnd 被初始化为默认大小 TCP_WND，snd_wl2、snd_nxt、snd_max、lastack、snd_lbb 五个字段被初始化为 ISS（ISS 可以看成一个系统全局变量，其值随时间变化），这里假设这五个字段都被初始化为 ZSL1，即表示上次窗口更新时收到的确

认号为 ZSL1，最后一个有效确认的数据编号是 ZSL1，下一个将要发送的数据编号为 ZSL1，下一个将被缓存的数据编号是 ZSL1；然后调用函数 tcp_bind 进行端口的绑定，这一步不涉及窗口相关字段的操作；接下来，服务器调用函数 tcp_listen_with_backlog 进入监听状态，此时会新建一个 tcp_pcb_listen 结构来保存 TCP 控制块的相关字段，并把 tcp_pcb_listen 结构放入链表 tcp_listen_pcbs，此时这个 TCP 控制块就进入了 LISTEN 状态等待客户端的连接。

（2）客户端：前几步与服务器相同，首先调用函数 tcp_alloc 分配一个新的 TCP 控制块，将控制块的 rcv_wnd 和 rcv_ann_wnd 被初始化为默认大小 TCP_WND，snd_wl2、snd_nxt、snd_max、lastack、snd_lbb 五个字段被初始化为自身的 ISS（初始序列号），这里假设这五个字段都被初始化为 ZSL2，即表示上次窗口更新时收到的确认号为 ZSL2，最后一个有效确认的数据编号是 ZSL2，下一个将要发送的数据编号为 ZSL2，下一个将被缓存的数据编号是 ZSL2；然后调用函数 tcp_bind 进行端口的绑定，这一步不涉及窗口相关字段的操作；接下来，客户端调用函数 tcp_connect 向服务器端发起连接，在 tcp_connect 函数中，rcv_nxt 字段置为 0，snd_nxt 为 ZSL2，lastack 为 ZSL2-1，snd_lbb 为 ZSL2-1，rcv_wnd 和 rcv_ann_wnd 被置为默认大小 TCP_WND，接收窗口 snd_wnd 被置为默认大小 TCP_WND。接下来，tcp_enqueue 函数被调用来发送一个 SYN 数据包，这就是前面讲过的内容了，在函数 tcp_enqueue 中，数据包被组装后挂在 unsent 队列上，数据包的起始编号被设置为 snd_lbb 的值 ZSL2-1（*），然后调整发送窗口的相关字段，如下面的代码，

```
if ((flags & TCP_SYN) || (flags & TCP_FIN)) {    //发送 SYN 或 FIN 包被认为数据长度为 1
    ++len;
  }
pcb->snd_lbb += len;   // 下一个要被缓冲数据的序号，注意与 snd_nxt 不同
```

所以，tcp_enqueue 函数过后，snd_lbb 值变为 ZSL2，其他字段值不变。

tcp_connect 函数接下来还调用 tcp_output 将数据包发送出去，后者发送一个具体的数据段是通过调用函数 tcp_output_segment 实现的，这个函数主要是填充待发送数据段的 TCP 头部中的确认序号为 rcv_nxt 的值 0（*），通告窗口大小为 rcv_ann_wnd 的值 TCP_WND（*）。最后，tcp_output 通过下面的代码来更新窗口相关的字段：

```
pcb->snd_nxt = ntohl(seg->tcphdr->seqno) + TCP_TCPLEN(seg); // 下一个要发送的字节序号
if (TCP_SEQ_LT(pcb->snd_max, pcb->snd_nxt)) {
      pcb->snd_max = pcb->snd_nxt; // 最大发送序号
}
```

即 snd_nxt 和 snd_max 的值都变为 ZSL2。综上，客户端发送一个 SYN 包后进入 SYN_SENT 状态，此时控制块中窗口相关的各个字段 rcv_nxt 为 0，snd_nxt 为 ZSL2，lastack 为 ZSL2-1，snd_lbb 为 ZSL2，rcv_wnd 和 rcv_ann_wnd 为默认大小 TCP_WND，发送窗口 snd_wnd 为默认大小 TCP_WND；发出去的 SYN 包中的 seqno 为 ZSL2-1，ackno 为 0，通告窗口为 TCP_WND。

（3）服务器端：处于 LISTEN 状态的控制块接收到 SYN 包后，调用函数 tcp_listen_input 处理数据包，由于该数据包是 SYN 包，所以需要重新建立一个控制块，并将 tcp_pcb_listen 结构中的内容拷贝出来，并把控制块加入 tcp_active_pcbs 链表中。在这里，控制块的窗口相关字段被填充：rcv_nxt = seqno + 1= ZSL2-1+1= ZSL2，snd_wl1 = seqno – 1= ZSL2-2（设为减 2 是为了保证下次接到数据段后能进行窗口更新）。rcv_wnd 和 rcv_ann_wnd 为默认大小 TCP_WND，snd_wl2、snd_nxt、snd_max、lastack、snd_lbb 都维持不变，为 ZSL1。接下来服务器调用函数 tcp_enqueue 组装数据包挂在 unsent 队列上，数据包的起始编号被设置为 snd_lbb 的值 ZSL1（*），然后调整发送窗口的相关字段，如下面的代码，

```
if ((flags & TCP_SYN) || (flags & TCP_FIN)) {    //发送 SYN 或 FIN 包被认为数据长度为 1
```

```
        ++len;
      }
   pcb->snd_lbb += len;   // 下一个要被缓冲数据的序号，注意与 snd_nxt 不同
```

所以，tcp_enqueue 函数过后，snd_lbb 值变为 ZSL1+1，其他字段值不变。

接下来调用 tcp_output 将数据包发送出去，与客户端类似，填充待发送数据段的 TCP 头部中的确认序号为 rcv_nxt 的值 ZSL2（*），通告窗口大小为 rcv_ann_wnd 的值 TCP_WND（*）。最后，tcp_output 还要更新窗口相关的字段：

```
pcb->snd_nxt = ntohl(seg->tcphdr->seqno) + TCP_TCPLEN(seg); // 下一个要发送的字节序号
if (TCP_SEQ_LT(pcb->snd_max, pcb->snd_nxt)) {
        pcb->snd_max = pcb->snd_nxt; // 最大发送序号
}
```

即 snd_nxt 和 snd_max 的值都变为 ZSL1+1。综上，服务器发送一个 SYN+ACK 后进入 SYN_RCVD 状态，此时控制块中窗口相关的各个字段 rcv_nxt 为 ZSL2，snd_nxt 为 ZSL1+1，lastack 为 ZSL1，snd_lbb 为 ZSL1+1，rcv_wnd 和 rcv_ann_wnd 为默认大小 TCP_WND，发送窗口 snd_wnd 为默认大小 TCP_WND；发出去的 SYN+ACK 包中的 seqno 为 ZSL1，ackno 为 ZSL2，通告窗口为 TCP_WND。

（4）客户端：处于 SYN_SENT 状态的客户端调用函数 tcp_process 处理服务器返回的 SYN+ACK 包：rcv_nxt = seqno + 1= ZSL1+1;lastack = ackno= ZSL2; snd_wl1 = seqno – 1= ZSL1-1（这样设置 snd_wl1 使得下次接到数据段后能进行窗口更新）。到这里，客户端调用函数 tcp_ack_now 向服务器端返回一个立即确认数据包，tcp_ack_now 也是通过调用函数 tcp_output 将数据包发送出去，这里注意，我们并没有调用函数 tcp_enqueue 组装一个数据包，而是直接在 tcp_output 组装发送的，这里发送出去的数据包 seqno = (pcb->snd_nxt)=ZSL2, ackno = (pcb->rcv_nxt)= ZSL1+1，注意立即确认数据包不包含任何数据，也不占任何数据长度，所以 snd_nxt 字段值是保持不变的。发送完 ACK 包后，客户端进入 ESTABLISHED 状态。

（5）服务器端：处在 SYN_RCVD 状态的服务器端接收到客户端返回的确认后，调用函数 tcp_process 处理这个数据包。这个过程很简单，没有涉及窗口相关字段的操作，只是将控制块置为 ESTABLISHED 状态。

到这里，一条连接就建立起来了，来看看两端的窗口情况。服务器端：rcv_nxt 为 ZSL2，snd_nxt 为 ZSL1+1，lastack 为 ZSL1，snd_lbb 为 ZSL1+1，窗口大小默认；客户端：rcv_nxt 为 ZSL1+1，snd_nxt 为 ZSL2，lastack 为 ZSL2，snd_lbb 为 ZSL2，窗口大小默认。

比较上面的值，可以看出，两端的滑动窗口都正确的建立起来了。从整个过程来看，一片混乱，毫无规律。当两端的连接进入 ESTABLISHED 后，窗口的调整就相当规律了，函数 tcp_receive 和 rcv_nxt 密切相关，因为收到数据段会利用该函数进行处理；tcp_enqueue 函数和 snd_lbb 字段密切相关，用以缓存数据包；tcp_output 函数和 snd_nxt 密切相关，该函数用于向网络中发送数据包；tcp_output_segment 函数主要填充发送包的 ackno 字段。

22 TCP 超时与重传

在 TCP 两端交互过程中，数据和确认都有可能丢失。TCP 通过在发送时设置一个定时器来解决这种问题。如果当定时器溢出时还没有收到确认，它就重传该数据。对任何 TCP 协议实现而言，怎样决定超时间隔和如何确定重传的频率是提高 TCP 性能的关键。

这节讲解 TCP 的超时重传机制，TCP 控制块 tcp_pcb 内部的相关字段为 rtime、rttest、rtseq、sa、sv、rto、nrtx，太多了，先不要晕！

与超时时间间隔密切相关的是往返时间（RTT）的估计。RTT 是某个字节的数据被发出到该字节确认返回的时间间隔。由于路由器和网络流量均会变化，因此 RTT 可能经常会发生变化，TCP 应该跟踪这些变化并相应地改变其超时时间。

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，估计其 RTT 值，另外，一个报文段被重发和该报文的确认到来之前不应该更新估计器。协议中利用一些优化算法平滑 RTT 的值，并根据 RTT 值设置 RTO 的值，即下一个数据包的重传超时时间。

先来看看超时重传机制是怎样实现的，再来重点介绍与 RTT 估计密切相关的部分。前面讲过 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 将该数据段挂接到 unacked 队列上，至于挂在 unacked 队列上的什么位置，那是后话。tcp_output_segment 负责将数据段发送出去，发送出去后它要做的工作如下面的代码所示：

```
if(pcb->rtime == -1)
        pcb->rtime = 0;
if (pcb->rttest == 0) {
        pcb->rttest = tcp_ticks;
        pcb->rtseq = ntohl(seg->tcphdr->seqno);
}
```

rtime 用于重传定时器的计数，当其值为-1 时表示计数器未被使能；当值为非 0 时表示计数器使能，在这种情况下，rtime 的值每 500ms 被内核加 1，当 rtime 超过 rto 的值时，在 unacked 队列上的所有数据段将被重传。rto 已经提及过多次，就是我们为数据包所设置的超时重传时间。接下来的 rttest 字段与 RTT 估计密切相关，当 rttest 值为 0 时表示 RTT 估计未启动，否则若要启动 RTT 估计，则应在发送数据包出去后，将 rttest 的值设置为 tcp_ticks（全局变量，系统当前滴答数），并用 rtseq 字段记录要进行 RTT 估计的数据段的起始数据编号。当接收到对方返回的 ACK 编号后，就可以根据 rttest 与 rtseq 的值计算 RTT 了，字段 sa、sv 与 rto 值的计算密切相关，放在后续讨论。数据段发送就是这么多了，主要是针对发送出去的数据段启动重传定时器。当然如过数据段发送出去的时候，重传定时器是启动的，即 rtime 不等于-1，此刻不对重传定时器做任何操作。同理，如果 rttest 不等于 0，则说明 RTT 正在进行，此时不会对 RTT 的各个字段做任何操作。

TCP 慢 定时器每 500ms 产生一次中断处理，在中断处理中，若 TCP 控制块的重传计数器被启动（即 rtime 不为 0），则 rtime 值被加 1。同时，当 rtime 值大于 rto 时，调用重传函数对未被确认的数据进行重传，代码如下（仅列出了与重传相关的部分）：

```
if (pcb->unacked != NULL && pcb->rtime >= pcb->rto) {
    ……
```

```
    pcb->rtime = 0;   // 复位计数器
    tcp_rexmit_rto(pcb);   // 函数调用进行重传
    …….
}
```

tcp_rexmit_rto 函数实现重传的机制很简单，它将 unacked 链表上的所有数据段插入到 unsent 队列的前端，并将控制块重传次数字段 nrtx 加 1，最后调用 tcp_output 重发数据包。

这里你可能会发现一个问题，假设我们刚刚从 unsent 队列上取下一个数据段发送出去，并将该数据段挂接在 unacked 链表上等待确认，接着前面某个数据段处设置的重传定时器超时，这样整个 unacked 链表上又被放到了 unsent 队列上进行重传。不可避免，我们刚刚发送出去的那个数据段又回到了 unsent 队列上，这岂不是悲剧，它又得重发一遍？尽管这种情况是可能发生的，但是 LWIP 通过窗口的控制以及收到确认号后遍历 unsent 队列（下面讲解）这两种方式使得这种可能性降到了最小。这里注意，在较老版本的 LWIP 协议栈中，每个数据段结构 tcp_seg 中都对应有一个 rtime 字段，用于记录某个数据段的超时情况，这样可以避免重传时将整个 unacked 链表上放回到 unsent 队列上。而新版本的中，整个 TCP 控制块公用了一个 rtime 字段。

在数据接收上，tcp_receive 函数提取收到的数据段中的 ackno，并用该 ackno 来处理 unacked 队列，即当该 ackno 确认了某个数据段中的所有数据，则将该数据段从 unacked 队列中移除，并释放数据段占用的空间。同时，函数要检查 unacked 队列，如果 unacked 队列中没有被需要确认的数据段了，此时需要停止重传定时器，否则要复位重传定时器。很简单，用下面的代码：

```
if(pcb->unacked == NULL)
    pcb->rtime = -1;
else
    pcb->rtime = 0;
```

接下来，tcp_receive 函数还要根据收到的确认号遍历 unsent 队列，以处理那些正被等待重传的数据段。unsent 队列上那些是被重传的数据段？很明显就是数据段内数据编号小于控制块 snd_max 字段值的那些数据段。能被确认号确认的数据段会从 unsent 链表中移除，同时数据段占用的空间被释放。

关于数据段的超时设置与重传就是这么多了，下面到了很重要的内容，即 RTT 的估计。这里的代码有点难度啊！先来看看《TCP/IP 详解 1》里面是怎样描述 RTT 的。

TCP 超时与重传中最重要的部分就是对一个给定连接的往返时间（RTT）的测量。由于路由器和网络流量均会变化，因此我们认为这个时间可能经常会发生变化，TCP 应该跟踪这些变化并相应地改变其超时时间。TCP 必须测量在发送一个带有特别序号的字节和接收到包含该字节的确认之间的 RTT，同时应注意，发出去的数据段与返回的确认之间并没有一一对应的关系。

在往返时间变化起伏很大时，基于均值和方差来计算 RTO 能提供更好的响应，下面这个算法是 Jacobson 提出的，目前广泛应用在了 TCP 协议的实现中，当然 LWIP 也不例外。

$$Err = M - A$$

$$A \leftarrow A + gErr$$

$$D \leftarrow D + h(|Err| - D)$$

$$RTO = A + 4D$$

其中 M 表示某次测量的 RTT 的值，A 表示测得的 RTT 的平均值，A 值的更新如第二式所示，D 值为 RTT 的估计的方差，其更新如第三式所示。二式和三式中 g 和 h 都为常数，一般 g 取 1/8，h 取 1/4。这样取值是为了便于计算，从后面可以看出，通过简单的移位操作就可以完成上述计算了。RTO 的计算第四式所示，初始时，RTO 取值为 6，即 3s，A 值为 0，D 值为 6。

现在我们将上面的四个表达式做简单的变化，就得到了 LWIP 中计算 RTT 的表达式：

Err = M-A

A = A+ Err/8= A+(M-A)/8 ——> 8A = 8A + M-A

D =D + (|Err |-D)/4= D + (|M-A |-D)/4 ——> 4D = 4D + (|M-A |-D)

RTO = A+4D

令 sa = 8A,sv = 4D，这就是 TCP 控制块中的两个字段。带入上面变换后的表达式，得到：

sa = sa + M - sa>>3

sv = sv + (|M - sa>>3 |-sv>>2)

RTO = sa>>3 + sv

这样我们就得到了最关心的 RTO 值，还有一个疑问：M 值怎么得到？M 表示某次测量的 RTT 的值，在 LWIP 中它就是系统当前 tcp_ticks 值减去数据包被发送出去时的 tcp_ticks 值。tcp_ticks 也是在内核的 500ms 周期性中断处理中被加 1。

如果到这里你都还很清醒，那说明对 RTT 的理解就没什么问题了。这就来看看源代码是怎样进行 RTT 估算的，这也是在函数 tcp_receive 中进行的。

```
if (pcb->rttest && TCP_SEQ_LT(pcb->rtseq, ackno)) {    // 有 RTT 正在进行且该数据段被确认
    m = (s16_t)(tcp_ticks - pcb->rttest);     //计算 M 值
    m = m - (pcb->sa >> 3);   // M - sa>>3
    pcb->sa += m;     //更新 sa
    if (m < 0) {
      m = -m;       //   |M - sa>>3 |
     }
    m = m - (pcb->sv >> 2);   // (|M - sa>>3 |-sv>>2)
    pcb->sv += m;     // 更新 sv
    pcb->rto = (pcb->sa >> 3) + pcb->sv; //计算 rto
    pcb->rttest = 0;   // 停止 RTT 估计
}
```

这段代码基本是前面讲的公式的翻译了，不解释！还有这样一个问题，当以某个 RTO 为超时值发送数据包后，在 RTO 时间后未收到对该数据段的确认，则该数据包被重发，若重发后仍收不到关于该数据包的确认，这种情况下，协议栈该怎么办呢，是每次都按照原来的 RTO 重发数据包吗？答案是否定的，因为当多次重传都失败时，很可能是网络不通或者网络阻塞，如果这时再有大量的重发包被投入到网络，这势必使问题越来越严重，可能数据包永远的被阻塞在网络中，而无法到达目的端。与标准里面描述的一样，LWIP 是这样做的：如果重发的数据包超时，则接下来的重发包必须按照 2 的指数避让，即将 RTO 值设置为前一次的 2 倍，当重发超过一定次数后，不再对数据包进行重发。这是在 500ms 定时处理函数 tcp_slowtmr 中完成的，看看源代码。

```
if(pcb->rtime >= 0)        // 若重传定时器是开启的
    ++pcb->rtime;          // 则增加定时器的值
if (pcb->unacked != NULL && pcb->rtime >= pcb->rto) { //如果定时器超时，且有数据未确认
    if (pcb->state != SYN_SENT) { // 对处于 SYN_SENT 状态的数据不做避让处理
```

```
            pcb->rto = ((pcb->sa >> 3) + pcb->sv) << tcp_backoff[pcb->nrtx];   // rto 值退让
        }
    pcb->rtime = 0;       // 复位定时器
    ……
    tcp_rexmit_rto(pcb);   // 重传数据
}
```

上面这小段代码有三个地方想说一下：一是对处于 SYN_SENT 状态的控制块不进行超时时间的避让，可能是由于考虑到 SYN_SENT 状态一般发送出去的是 SYN 握手包，每次按照固定的 RTO 值进行重发，为什么要这样呢？不解，貌似标准里面也是进行避让了的啊！第二点是避让使用一个数组 tcp_backoff 通过移位的方式实现，tcp_backoff 定义如下：

```
const u8_t tcp_backoff[13] = { 1, 2, 3, 4, 5, 6, 7, 7, 7, 7, 7, 7, 7};
```

在这里面，当重传次数多于 6 次时，RTO 值将不再进行避让。最后一点是函数 tcp_rexmit_rto，该函数真正完成数据包的重传工作：

```
void tcp_rexmit_rto(struct tcp_pcb *pcb)
{
  struct tcp_seg *seg;
  if (pcb->unacked == NULL) {
    return;
  }
  for (seg = pcb->unacked; seg->next != NULL; seg = seg->next);     // 将 unacked 队列全部放到
  seg->next = pcb->unsent;                                          // 将 unsent 队列前端
  pcb->unsent = pcb->unacked;
  pcb->unacked = NULL;
  pcb->snd_nxt = ntohl(pcb->unsent->tcphdr->seqno);   // 下一个要发送的数据编号指向队列
                                                      //   unsent 首部的数据段
  ++pcb->nrtx;          // 重传次数加 1
  pcb->rttest = 0;      // 重发数据包期间不进行 RTT 估计
  tcp_output(pcb);     // 发送一个数据包
}
```

从这个代码和上面的避让算法里你可以很清楚的看到字段 nrtx 的作用了，它是多次重传时设置 rto 值的重要变量。另外注意，在重传期间不应该进行 RTT 估计，因为这种情况下的估计值往往是不准确的。这就是传说中的 Karn 算法，Karn 算法认为由于某报文即将重传，则对该报文的计时也就失去了意义。即使收到了 ACK，也无法区分它是对第一次报文，还是对第二次报文的确认。因此，TCP 只对未重传报文计时。还有一个要注意，经过上面的代码，snd_nxt 的值很可能就会小于 snd_max 的值咯，相信你隐约中感觉到 snd_max 的作用了吧，哈哈，收工！

23 TCP 慢启动与拥塞避免

这节讨论慢启动算法与拥塞避免算法。版权声明：下面这几段的内容大部分来自于《TCP/IP 详解，卷 1：协议》，若有雷同，纯非巧合！

假如按照前面所有的讨论，发送方一开始便可以向网络发送多个报文段，直至达到接收方通告的窗口大小为止。当发送方和接收方处于同一个局域网时，这种方式是可以的。但是如果在发送方和接收方之间存在多个路由器和速率较慢的链路时，就有可能出现一些问题。中间路由器缓存分组，可能消耗巨大的时间和存储器空间开销，这样 TCP 连接的吞吐量会严重降低。

要解决这种问题，TCP 需要使用慢启动(slow start)算法。该算法通过观察新分组进入网络的速率应该与另一端返回确认的速率相同而进行工作。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。当与另一个网络的主机建立 TCP 连接时，拥塞窗口被初始化为 1 个报文段（即另一端通告的报文段大小）。每收到一个 ACK，拥塞窗口就增加一个报文段。发送方取拥塞窗口与通告窗口中的最小值作为发送上限。

拥塞窗口是发送方使用的流量控制，而通告窗口则是接收方使用的流量控制。发送方开始时发送一个报文段，然后等待 ACK。当收到该 ACK 时，拥塞窗口从 1 增加为 2，即可以发送两个报文段。当收到这两个报文段的 ACK 时，拥塞窗口就增加为 4。这是一种指数增加的关系。

说到慢启动算法，就不得不说拥塞避免算法，在实际中这两个算法通常是在一起实现。通常慢启动算法是发送方控制数据流的方法，但有时发送方的数据流会达到中间路由器的极限，此时分组将被丢弃。拥塞避免算法是一种处理丢失分组的方法。该算法假定由分组受到损坏引起的丢失是非常少的（远小于 1%），因此在发送方看来，分组丢失就意味着在源主机和目的主机之间的某处网络上发生了拥塞。发送方可以通过两种方式来判断分组丢失：发生数据包发送超时和接收到重复的确认。

当拥塞发生时，我们希望降低分组进入网络的传输速率，发送方通过慢启动与拥塞避免算法主动调节分组进入网络的速率，同时发送方也通过接收方通告的窗口大小来被动调节分组发送速率。

拥塞避免算法和慢启动算法需要对每个连接维持两个变量：一个拥塞窗口 cwnd 和一个慢启动门限 ssthresh。这样算法一般按照如下过程进行：

1) 对一个给定的连接，初始化 cwnd 为 1 个报文段，ssthresh 为 65535 个字节。

2) TCP 输出例程的输出不能超过 cwnd 和接收方通告窗口的大小。拥塞避免是发送方使用的流量控制，而通告窗口则是接收方进行的流量控制。前者是发送方感受到的网络拥塞的估计，而后者则与接收方在该连接上的可用缓存大小有关。

3) 当拥塞发生时（超时或收到重复确认），ssthresh 被设置为当前窗口大小的一半（cwnd 和接收方通告窗口大小的最小值，但最少为 2 个报文段）。此外，如果是超时引起了拥塞，则 cwnd 被设置为 1 个报文段（这就是慢启动）。

4) 当新的数据被对方确认时，就增加 cwnd，但增加的方法依赖于我们是否正在进行慢启动或拥塞避免。如果 cwnd 小于或等于 ssthresh，则正在进行慢启动，否则正在进行拥塞避免。慢启动一直持续到我们回到当拥塞发生时所处位置的一半时候才停止（因为我们记录了在步骤 2 中给我们制造麻烦的窗口大小的一半），然后转为执行拥塞避免。

慢启动算法初始设置 cwnd 为 1 个报文段，此后每收到一个确认就加 1，这会使窗口按

指数方式增长：发送 1 个报文段，然后是 2 个，接着是 4 个……。拥塞避免算法要求每次收到一个确认时将 cwnd 增加 1 /cwnd。与慢启动的指数增加比起来，这是一种加性增长(additive increase)。我们希望在一个往返时间内最多为 cwnd 增加 1 个报文段（不管在这个 RTT 中收到了多少个 ACK），然而慢启动将根据这个往返时间中所收到的确认的个数增加 cwnd。

说了半天用一句话来概括慢启动算法和拥塞避免算法：每个 TCP 控制块有两个字段 cwnd 和 ssthresh，当数据发送时，发送方只能取 cwnd 和接收方通告窗口大小中的较小者作为发送上限。当有确认返回时，若此时 cwnd 值小于等于 ssthresh，则做慢启动算法，即每收到一个确认，cwnd 都加 1；若此时 cwnd 值大于 ssthresh，则做拥塞避免算法，即每收到一个确认，cwnd 都加 1/cwnd，这保证了在一个 RTT 估计内，cwnd 增加值不超过 1。当 cwnd 增加到某个值时拥塞发生，则按照 3）所示来更新 cwnd 和 ssthresh 的值。晕，MS 不是一句话，是一段话！

这里我们按照上面算法所述的步骤来看看 LWIP 具体是怎样来描述整个过程的。

1) 给定一个连接，初始化其 cwnd 与 ssthresh。cwnd 都被初始化为 1，而 ssthresh 字段在客户端与服务器端略有不同，在客户端执行主动连接，调用函数 tcp_connect，在其中完成相关字段值的设置：

```
pcb->cwnd = 1;
pcb->ssthresh = pcb->mss * 10;
pcb->state = SYN_SENT;
```

在发送出 SYN 包并收到服务器的返回 SYN+ACK 后，客户机在函数 tcp_process 中对相关字段进行进一步的设置：

```
pcb->ssthresh = pcb->mss * 10;
pcb->cwnd = ((pcb->cwnd == 1) ? (pcb->mss * 2) : pcb->mss);
```

而在服务器端，进行被动连接，在函数 tcp_listen_input 中设置相关字段的值：

```
npcb->snd_wnd = tcphdr->wnd;
npcb->ssthresh = npcb->snd_wnd;
```

可以看到在服务器端，ssthresh 被设置为了对方通告的接收窗口大小。

2) TCP 输出例程的输出不能超过 cwnd 和接收方通告窗口的大小。在 tcp_output 函数中是用下面的代码来发送数据段的：

```
wnd = LWIP_MIN(pcb->snd_wnd, pcb->cwnd);    // 取有效发送窗口大小值
seg = pcb->unsent;        //取得第一个发送数据段
……
while (seg != NULL &&ntohl(seg->tcphdr->seqno) - pcb->lastack + seg->len <= wnd)
    {                                        // 数据段在有效发送窗口内
     pcb->unsent = seg->next;
     tcp_output_segment(seg, pcb);      // 则发送数据段
     ……
     seg = pcb->unsent;              // 取得下一个发送数据段
    }
```

3）发生拥塞时，需要更新 cwnd 和 ssthresh 的值。若拥塞是由超时引起的，则相应的更新代码在 tcp_slowtmr 中被实现，如下，这段熟悉的代码我们在超时重传中也见到过：

```
if(pcb->rtime >= 0)
    ++pcb->rtime;
if (pcb->unacked != NULL && pcb->rtime >= pcb->rto) {      // 超时发生
    ……       // 与超时重传相关的部分
```

```
            eff_wnd = LWIP_MIN(pcb->cwnd, pcb->snd_wnd); // 取得超时的有效发送大小
            pcb->ssthresh = eff_wnd >> 1;     // ssthresh 设置为有效大小的一半
            if (pcb->ssthresh < pcb->mss)   { // 修正 ssthresh 至少为 2 个报文段
              pcb->ssthresh = pcb->mss * 2;
            }
            pcb->cwnd = pcb->mss;       // 设置 cwnd 大小为 1 个报文段大小
        }
```

若拥塞是由重复确认引起的，则相应的更新代码在 tcp_receive 函数中实现，这段代码涉及到快速恢复与重传部分的知识，我们重点放在后面讲解。

```
          if (pcb->lastack == ackno) {
              ……
              ……
              if (pcb->cwnd > pcb->snd_wnd)         // ssthresh 设置为有效发送窗口的一半
                  pcb->ssthresh = pcb->snd_wnd / 2;
              else
                  pcb->ssthresh = pcb->cwnd / 2;
              if (pcb->ssthresh < 2*pcb->mss) {     // 修正 ssthresh 至少为 2 个报文段
                   pcb->ssthresh = 2*pcb->mss;
                }
              pcb->cwnd = pcb->ssthresh + 3 * pcb->mss; // 这里不解？
              ……
            }
```

4) 当新的数据被对方确认时，更新 cwnd 和 ssthresh 的值。

```
if (TCP_SEQ_BETWEEN(ackno, pcb->lastack+1, pcb->snd_max)){
……
      if (pcb->state >= ESTABLISHED) {
        if (pcb->cwnd < pcb->ssthresh) { //小于的时候执行慢启动，标准里是小于等于哦？
          if ((u16_t)(pcb->cwnd + pcb->mss) > pcb->cwnd) {    // 能否增加
            pcb->cwnd += pcb->mss;     // 增加一个段大小
          }
        } else {       // 以下执行拥塞避免
          u16_t new_cwnd = (pcb->cwnd + pcb->mss * pcb->mss / pcb->cwnd);
          if (new_cwnd > pcb->cwnd) { // 能否增加
            pcb->cwnd = new_cwnd;
          }
        }
      }
……
}
```

这段代码基本就是上面所说的第四步的直接翻译了。需要注意的是 new_cwnd 为什么要用上面的等式计算出来呢？那是因为 cwnd 的大小是按照 mss 的大小为单位增加减少的，其他不解释。

24　TCP 快速恢复重传和 Nagle 算法

前面介绍过，在收到一个失序的报文段时，该报文段会被挂接到 ooseg 队列上，同时向发送端返回一个 ACK（期待的下一个字节），很明显，这个 ACK 一定是个重复的 ACK，且这个重复的 ACK 被发送出去的时候不会有任何延迟。接收端利用该重复的 ACK，目的在于让对方知道收到一个失序的报文段，并告诉对方自己希望收到的序号。

但是在发送方看来，它不可能知道一个重复的 ACK 是由一个丢失的报文段引起的，还是由于仅仅出现了几个报文段的重新排序引起。因此我们需要等待少量重复的 ACK 到来。假如这只是一些报文段的重新排序，则在重新排序的报文段被处理并产生一个新的 ACK 之前，只可能产生 1 ~ 2 个重复的 ACK。如果一连串收到 3 个或 3 个以上的重复 ACK，就非常可能是一个报文段丢失了。于是我们就重传丢失的数据报文段，而无需等待超时定时器溢出。这就是快速重传算法。在上节讲超时重传时说到，当超时发生后，ssthresh 会被设置为有效发送窗口的一半，而 cwnd 被设置为一个报文段大小，即执行的是慢启动算法。而在这里，当执行完快速重传后，接下来执行的不是慢启动算法而是拥塞避免算法，这就是所谓的快速恢复算法了。

在快速重传后没有执行慢启动的原因在于，由于收到重复的 ACK 不仅仅告诉我们一个分组丢失了。而且由于接收方只有在收到另一个报文段，并将该报文段挂接到 ooseg 队列后，才会产生重复的 ACK，这就说明，在收发两端之间仍然有流动的数据，而我们不想执行慢启动来突然减少数据流。

卷一中描述的该算法步骤如下：

1) 当收到第 3 个重复的 ACK 时，将 ssthresh 设置为当前拥塞窗口 cwnd 的一半。重传丢失的报文段。设置 cwnd 为 ssthresh 加上 3 倍的报文段大小。

2) 每次收到另一个重复的 ACK 时，cwnd 增加 1 个报文段大小并发送 1 个分组（如果新的 cwnd 允许发送）。

3) 当下一个确认新数据的 ACK 到达时，设置 cwnd 为 ssthresh（在第 1 步中设置的值）。这个 ACK 应该是在进行重传后的一个往返时间内对步骤 1 中重传的确认。另外，这个 ACK 也应该是对丢失的分组和收到的第 1 个重复的 ACK 之间的所有中间报文段的确认。

LWIP 也是在函数 tcp_receive 中实现快速恢复与重传的，如下所示，整个过程与上面算法所述基本相同。

```
if (pcb->lastack == ackno) {      // 如果该 ACK 是个重复的 ACK
  pcb->acked = 0;       // 则被该 ACK 确认的数据个数为 0
  if (pcb->snd_wl1 + pcb->snd_wnd == right_wnd_edge){ // 如果未进行窗口更新
      ++pcb->dupacks;   // 收到重复确认的次数加 1
      if (pcb->dupacks >= 3 && pcb->unacked != NULL) { //如 1)所述,三个以上充复 ACK
        if (!(pcb->flags & TF_INFR)) { // 此时快速重传未开启，即 dupacks 为 3 次
          tcp_rexmit(pcb);     // 调用函数重传丢失的报文段
          if (pcb->cwnd > pcb->snd_wnd)             // ssthresh 设置为有效发送窗口的一半
              pcb->ssthresh = pcb->snd_wnd / 2;
          else
              pcb->ssthresh = pcb->cwnd / 2;
          if (pcb->ssthresh < 2*pcb->mss) {      // 修正 ssthresh 值，最小为 2 个报文段
            pcb->ssthresh = 2*pcb->mss;
          }
```

```
                pcb->cwnd = pcb->ssthresh + 3 * pcb->mss;// cwnd 为 ssthresh+3*报文段大小
                pcb->flags |= TF_INFR;     // 设置快速重传标志
            }
            else   // 快速重传已经开始，即 dupacks 大于 3 次
            {
                if ((u16_t)(pcb->cwnd + pcb->mss) > pcb->cwnd)   // 快速重传已经开始，如 2）
                    pcb->cwnd += pcb->mss; // 每收到一个重复 ACK，cwnd 增加 1 个报文段大
            }              //这与 2）中描述的有区别，这里收到重复 ACK 后没有发送 1 个分组
        } // if dupacks 大于 3
    } //if 如果未进行窗口更新
}     // if 如果是重复的 ACK
else if (TCP_SEQ_BETWEEN(ackno, pcb->lastack+1, pcb->snd_max)){ //确认了新的数据
        if (pcb->flags & TF_INFR) {       // 正处于快速重传状态，
          pcb->flags &= ~TF_INFR;       // 清除快速重传标志
          pcb->cwnd = pcb->ssthresh;     // 如 3）所示，设置 cwnd 的值
         }
        …….
        pcb->dupacks = 0;     // 清除重复确认标志
        pcb->lastack = ackno; // 记录 ackno
        …….
       }
```

上面这段代码有两个地方需要说明一下：一是调用函数 tcp_rexmit 重传丢失的报文段，这个函数和上一节讲到的函数 tcp_rexmit_rto 相类似，都是重传数据包。这个函数功能是将 unacked 队列上的第一个数据段放到 unsent 队列首部，并调用函数 tcp_output 输出数据包。第二个需要注意的地方是：在收到三个以上的重复 ACK 后，代码只是将 cwnd 的值增加一个报文段大小，而没像向上面 2）中所述的那样发送一个数据包。

快速重传与恢复就这么多了，下面是 Nagle 算法部分。

基于窗口的流量控制方案，会导致一种被称为“糊涂窗口综合症 SWS (Silly Window Syndrome）”的状况。当 TCP 接收方通告了一个小窗口并且 TCP 发送方立即发送数据填充该窗口时，SWS 就会发生，当一个小的报文段被确认，窗口再一次以较小单元被打开而发送方将再一次发送一个小的报文段填充这个窗口。这样就会造成 TCP 数据流包含一些非常小的报文段情况的发生，而不是满长度的报文段。糊涂窗口综合症是一种能够导致网络性能严重下降的 TCP 现象，因为小单元的数据段中 IP 头部和 TCP 头部这些字段占了大部分空间，而真正的 TCP 数据却很少。

该现象可能由 TCP 连接两端中的任何一端引起。这是由于接收方可以通告一个小的窗口（而不是一直等到有大的窗口时才通告），而发送方也可以发送少量的数据（而不是等待其他的数据以便发送一个大的报文段）。

为了避免 SWS 的发生，在发送方和接收方必须设法消除这种情况。接收方不必通告小窗口更新，并且发送方在只有小窗口提供时不必发送小的报文段。可以在任何一端采取措施避免出现糊涂窗口综合症的现象。

接收方解决 SWS 的方法是接收方不通告小窗口。通常的算法是接收方不通告一个比当前窗口大的窗口 （可以为 0），除非窗口可以增加一个报文段大小（也就是将要接收的 MSS）或者可以增加接收方缓存空间的一半，不论实际有多少。

发送方避免出现糊涂窗口综合症的措施是只有以下条件之一满足时才发送数据：

(a)可以发送一个满长度的报文段；

(b)可以发送至少是接收方通告窗口大小一半的报文段；

(c)可以发送任何数据并且不希望接收 ACK（也就是说，我们没有还未被确认的数据）或者该连接上不能使用 Nagle 算法。

条件(b)主要对付那些总是通告小窗口（也许比 1 个报文段还小）的主机，它要求发送方始终监视另一方通告的最大窗口大小，这是一种发送方猜测对方接收缓存大小的企图。虽然在连接建立时接收缓存的大小可能会减小，但在实际中这种情况很少见。

条件 (c)使我们在有尚未被确认的数据（正在等待被确认）以及在不能使用 Nagle 算法的情况下，避免发送小的报文段。如果应用进程在进行小数据的写操作（例如比该报文段还小），条件(c)可以避免出现糊涂窗口综合症。

这三个条件也可以让我们回答这样一个问题：在有尚未被确认数据的情况下，如果 Nagle 算法阻止我们发送小的报文段，那么多小才算是小呢？从条件 (a)中可以看出所谓“小”就是指字节数小于报文段的大小 MSS。

在 LwIP 中，SWS 在发送端就被自然的避免了，因为 TCP 报文段在建立和排队时不知道通告的接收器窗口。在大数据量发送中，输出队列将包括最大尺寸的报文段。这意味着，如果 TCP 接收方通告了一个小窗口，发送方将不会发送队列中的第一个报文段，因为它比通告的窗口要大。相反，它会一直等待直至窗口有足够大的空间容下它。当作为 TCP 接收方时，LwIP 将不会通告小于连接允许的最大报文段尺寸的接收器窗口（这点未懂）。

来看看 LWIP 在数据段发送的时候是如何让来避免糊涂窗口的，下面的代码经过一定的删减，只保留了与算法相关的部分。

```
 seg = pcb->unsent; //  取得第一个数据段
 while (seg != NULL && ntohl(seg->tcphdr->seqno) - pcb->lastack + seg->len <= wnd) { //整个
                                                          //  数据段是否在有效发送窗口内
        if((tcp_do_output_nagle(pcb) == 0) &&    // nagle 算法阻止发送数据包
           ((pcb->flags & (TF_NAGLEMEMERR | TF_FIN)) == 0)){ //  有内存错误标志和
           break;              //    FIN 包标志时，nagle 算法失效
         }
      ……
      pcb->unsent = seg->next;    //  记录下一个数据段
      tcp_output_segment(seg, pcb); //  发送数据段
      ……
      seg = pcb->unsent; //  取得下一个数据段
    }
```

这短短的几句就实现了 nagle 算法？有太多需要解释！第一个是 while 的循环条件里面，它要求将要发送的数据段内所有数据序号都必须在有效发送窗口内。这样的话，如果对方通告了一个小窗口，且发送的数据段很大的话则数据段不会被发送出去，这可以看作是上述条件(a) 和条件(b)的变形。对于其他情况，则可以利用 Nagle 算法来判断是否输出数据包。同时如果 flags 的 TF_NAGLEMEMERR 和 TF_FIN 标志置位时，Nagle 算法失效，即数据包不会被 Nagle 算法阻止。TF_NAGLEMEMERR 可以理解为表示内存错误，它是在 tcp_enqueue 函数组装数据包时出现发送缓存空间不足时被置位的，当这种情况发生时，已经组装好的数据包需要被尽快发送出去，所以 Nagle 算法在该标志置位时失效；TF_FIN 标志置位时说明上层应用发出了关闭连接命令，所以此时也应尽快将该连接上的数据段发送出去，Nagle 算法在这种情况下也失效。Nagle 算法是通过宏 tcp_do_output_nagle 来实现的，如下：

```
#define tcp_do_output_nagle(tpcb) ((((tpcb)->unacked == NULL) || \
```

```
((tpcb)->flags & TF_NODELAY) || \
(((tpcb)->unsent != NULL) && ((tpcb)->unsent->next != \
  NULL))) ?  1 : 0 )
```

不要被如此多的括号吓着，我们只关心上式等于 0，即 Nagle 算法阻止数据包发送时的情况。具体为 unacked 队列不为空，且 Nagle 算法已使能（TF_NODELAY 未置位），且 unsent 发送队列上有少于两个的待发送数据段时，tcp_do_output_nagle 取值为 0，tcp_output 跳出 while 循环，不进行任何数据段的发送。

上面这段也可以按照（c）的说法来描述，当没有还未被确认的数据（unacked 队列为空），或者 Nagle 算法未使能，或者 unsent 队列上有两个或两个以上的数据段时，数据段可以被发送出去。这里比（c）中多了一个 unsent 队列上数据包个数的限制，此时我们可以把这个多出的限制看作是对条件(a) 和条件(b)的另一种解释。

很多情况下需要禁止 Nagle 算法，这是通过设置控制块 flags 字段中的 TF_NODELAY 标志来实现的。

前面说过，接收方解决 SWS 的方法是不通告小窗口。但是若使用 LwIP 作为接收端，它貌似还未实现此功能，即它可能向发送端通告小窗口信息。难道我还没看懂。。。

25 TCP 坚持与保活定时器

这节讲解 TCP 的坚持定时器和保活定时器，先看坚持定时器。

TCP 的接收方通过通告窗口大小来告诉发送方自己可以接收的数据字节数，接收方采用这种方式来进行流量控制。假如接收方通告的窗口大小为 0 会发生什么情况呢？这将有效地阻止发送方传送数据，直到通告窗口变为非 0 为止。

发送方接到 0 窗口通告时，则会停止数据段的发送，直到接收方通过非 0 的窗口。很重要的一点，TCP 必须能够处理含新非 0 窗口通告的数据包丢失的情况，通常这个非 0 窗口通告是在一个不含任何数据的 ACK 包中发送的。ACK 的传输并不可靠，也就是说， TCP 不对 ACK 报文段进行确认（很明显，也就不会存在该 ACK 报文段的重发），TCP 只确认那些包含有数据的 ACK 报文段。

如果一个确认丢失了，则双方就有可能因为等待对方而使连接终止：接收方等待接收数据（因为它已经向发送方通告了一个非 0 的窗口） ，而发送方在等待允许它继续发送数据的非 0 窗口更新。为防止这种死锁情况的发生，发送方使用一个坚持定时器 (persist timer) 来周期性地向接收方查询，以便发现窗口是否已增大。这些从发送方发出的报文段称为窗口探查 (window probe)。

控制块中有两个字段与坚持定时器有关：persist_cnt 和 persist_backoff。persist_cnt 用于坚持定时器计数，当计数值超过某个值时，则发出窗口探查数据包。persist_backoff 表示坚持定时器是否被启动（是否>0）以及已经发出去了几个探查数据包（persist_backoff 为大于 0 的整数时）。若坚持定时器已经被启动，则在内核 500ms 中断处理函数 tcp_slowtmr 会进行如下处理：

```
if (pcb->persist_backoff > 0) {    // 如果坚持定时器已经开启
  pcb->persist_cnt++;     // 增加计数值
  if (pcb->persist_cnt >= tcp_persist_backoff[pcb->persist_backoff-1]) { // 计数值超过
                                                    // 某个计数值上限时则进行窗口探查
    pcb->persist_cnt = 0; // 复位计数值
    if (pcb->persist_backoff < sizeof(tcp_persist_backoff)) { // 增加计数值上限
      pcb->persist_backoff++;
    }
    tcp_zero_window_probe(pcb); // 发送一个窗口探查包
  }
}
```

有两点需要提及的。一是数组 tcp_persist_backoff，它保存了一系列的坚持定时器的计数值上限，persist_backoff 是该数组的索引。即发送第一个探测包的时间为 3 次 500ms（1.5s）中断后，发送第二个探测包的时间为 6 次（3s）后，当第六次及其以上发送探测包时，时间间隔都变为 120 次中断（60s）。

```
const u8_t tcp_persist_backoff[7] = { 3, 6, 12, 24, 48, 96, 120 };
```

再来看看函数 tcp_zero_window_probe 是如何进行窗口探查的。tcp_zero_window_probe 函数很简单，组装一个含一字节数据的 TCP 报文段发送出去，这个字节的数据是从 unacked 或从 unsent 队列上取得的，且窗口探查包的数据序号字段被填成这个字节的数据序号。所以当这两个队列都为空时，则表示没有任何数据需要处理，当然窗口探查也没有必要进行；当

这个字节的数据是从 unacked 队列中得到时，由于该队列是已经被发送过的，对应窗口探查到达接收端时，会被看做是重复报文而不进行相关数据处理，只向发送方是返回一个 ACK 包；当这个字节的数据是从 unsent 队列中得到时，则这个字节数据到达接收端时会被挂接在 ooseq 队列或直接递交给上层应用，当发送方窗口允许时，unsent 队列中的第一个数据段的第一个字节被发送后在接收方看来是重复的，接收方能够检测出这个重复的字节，并直接删除该字节数据。从整个过程可以看出，窗口探查包里面的 1 字节数据并不影响整个数据传输过程。

什么时候启动一个窗口探查呢？这是在函数 tcp_output 最后完成的。当发送完能够发送的数据段后，unsent 队列还不为空，且此时窗口探查未启动，且当前窗口太小以至不能发送下一个数据段，此时要启动窗口探查。

```
if (seg != NULL && pcb->persist_backoff == 0 &&
    ntohl(seg->tcphdr->seqno) - pcb->lastack + seg->len > pcb->snd_wnd) {
  pcb->persist_cnt = 0;       // 复位计数值
  pcb->persist_backoff = 1;   // 开始窗口探查
}
```

什么时候停止一个窗口探查呢？从前面已经知道，在函数 tcp_receive 刚开始的部分，就会根据接收数据包的情况更新发送窗口，也即是在这里若检测到一个非 0 窗口，则停止窗口探查，如下所示。

```
if (TCP_SEQ_LT(pcb->snd_wl1, seqno) ||
    (pcb->snd_wl1 == seqno && TCP_SEQ_LT(pcb->snd_wl2, ackno)) ||
    (pcb->snd_wl2 == ackno && tcphdr->wnd > pcb->snd_wnd)) { // 若满足窗口跟新条件
     pcb->snd_wnd = tcphdr->wnd;   // 窗口更新
     pcb->snd_wl1 = seqno;
     pcb->snd_wl2 = ackno;
     if (pcb->snd_wnd > 0 && pcb->persist_backoff > 0) {   检测到非 0 窗口且探查开启
          pcb->persist_backoff = 0;   // 停止窗口探查
     }
 }
```

再来看保活定时器。如果一个已经处于稳定状态的 TCP 连接双方都没有向对方发送数据，则在两个 TCP 模块之间不交换任何信息。然而很多时候，连接的双方都希望知道对方的是否处于非活动状态。常见的状况是一个服务器希望知道客户主机是否崩溃并关机或者崩溃又重新启动，许多 TCP/IP 实现中提供的保活定时器可以提供这种检测功能。

保活功能主要是为服务器应用程序提供的。服务器应用程序希望知道客户主机是否崩溃，从而可以合理分配客户使用资源。如果一个给定的连接在两个小时之内没有任何动作，则服务器就向客户发送一个探查报文段。客户主机必处于以下 4 个状态之一：

1) 客户主机依然正常运行，并从服务器可达。客户的 TCP 响应正常，而服务器也知道对方是正常工作的。服务器在两小时以后将保活定时器复位，并发送探查报文。如果在两个小时定时器到时间之前有应用程序的通信量通过此连接，则定时器在交换数据后的未来 2 小时再复位，发送探查报文。

2) 客户主机已经崩溃，并且关闭或者正在重新启动，在这些情况下，客户的 TCP 都不会有任何响应。服务器将不能够收到对探查报文的响应，并在等待 75 秒后超时，以后服务器还会发送 9 个这样的探查报文，每个间隔 75 秒。如果服务器没有收到一个响应，它就认为客户主机已经关闭并终止连接。

3) 客户主机崩溃并已经重新启动。这时服务器将收到一个对其保活探查的响应，但是这个

响应是一个复位，使得服务器终止这个连接。
4) 客户主机正常运行，但是从服务器不可达。这与状态 2 相同，因为 TCP 不能够区分状态 4 与状态 2 之间的区别，它所能发现的就是没有收到探查的响应。

在第 1 种情况下，服务器的应用程序没有感觉到保活探查的发生。TCP 层负责一切，这个过程对应用程序都是不可见的。当第 2、3 或 4 种情况发生时，服务器应用程序将收到来自它的 TCP 层的差错报告（通常服务器应用程序向网络发出了读操作请求，然后等待来自客户的数据。如果保活功能返回一个差错，则该差错将作为读操作的返回值返回给应用程序）。在第 2 种情况下，差错是诸如“连接超时”之类的信息，而在第 3 种情况则为“连接被对方复位” 。第 4 种情况看起来像是连接超时，也可根据是否收到与连接有关的 ICMP 差错报文来判断是否是目的不可达引起的。

至此又会涉及 TCP 控制块中四个字段：keep_idle、keep_intvl、keep_cnt 和 keep_cnt_sent。其中 keep_intvl 和 keep_cnt 与编译选项 LWIP_TCP_KEEPALIVE 相关，当该编译选项为 1 时，keep_intvl 和 keep_cnt 字段分别用于保存用户自定义的保活时间选项值，这点在后面介绍。实际应用中使用系统默认的保活时间选项值即可，所以我们将 LWIP_TCP_KEEPALIVE 设置为 0，则 keep_intvl 和 keep_cnt 字段不会被编译，自然也不在我们的讨论范围之内了。

keep_idle 字段记录了在多久后进行保活探测，一般为 2 小时，keep_cnt_sent 字段表示已经发送的保活数据包的个数。除了这两个字段外，还有几个与默认保活时间选项值宏定义：

```
#define   TCP_KEEPIDLE_DEFAULT      7200000UL   // 保活时间毫秒数（2 小时）
#define   TCP_KEEPINTVL_DEFAULT     75000UL //连续保活包的时间间隔毫秒数（75s）
#define   TCP_KEEPCNT_DEFAULT       9U   // 保活包被重复发送的次数
#define   TCP_MAXIDLE   TCP_KEEPCNT_DEFAULT * TCP_KEEPINTVL_DEFAULT
                                                    //执行保活探测需要消耗的时间
```

用户可以通过宏 LWIP_TCP_KEEPALIVE 允许 keep_intvl 和 keep_cnt 字段，它们分别用于记录用户自定义的保活包时间间隔与保活包个数。当不使用自定义值时，就用上面的两个 DEFAULT 值作为保活选项值。

在这里，我们使用系统默认的保活选项值来分析保活的整个过程，此时字段 keep_idle 被设置为 TCP_KEEPIDLE_DEFAULT 的值。保活处理也是在内核 500ms 中断处理函数 tcp_slowtmr 中进行的。TCP 控制块中还有一个字段要重新提及一下，即 tmr 记录了该 TCP 连接上最近一个数据段到来时的系统时间 tcp_ticks 值。

```
if((pcb->so_options & SOF_KEEPALIVE) &&   // 如果开启了保活功能，稳定数据交互状态
                  ((pcb->state == ESTABLISHED) || (pcb->state == CLOSE_WAIT))) {
    if((u32_t)(tcp_ticks - pcb->tmr) >    // 2 小时+9*75 秒后断开连接
                  (pcb->keep_idle + TCP_MAXIDLE) / TCP_SLOW_INTERVAL)
      {
        tcp_abort(pcb);     // 断开连接
      }
    else if((u32_t)(tcp_ticks - pcb->tmr) > // 在 2 小时+9*75 秒内则发送保活包
              (pcb->keep_idle + pcb->keep_cnt_sent * TCP_KEEPINTVL_DEFAULT)
              / TCP_SLOW_INTERVAL)
      {
        tcp_keepalive(pcb);         // 发送保活包
        pcb->keep_cnt_sent++;   // 保活包次数加 1
      }
  }
```

tcp_abort 函数用于释放一个连接，主要工作包括将控制块从相应的 TCP 链表中删除，若该连接上还有数据则释放数据所占用的内存空间，最后向对方发送一个 RST 数据包。tcp_keepalive 函数用于发送一个保活包，保活包只是一个 TCP 首部，并不包含任何数据，所以不会对对方的数据接收造成影响。

关于保活选项的两个小时的空闲时间是可以改变。用户只要自己定义 keep_idle 的值就可以了，但是系统一般不建议修改这些值。Don't change this unless you know what you're doing！哈哈。。。

26 TCP 定时器

这节讨论 TCP 的定时处理函数。在前面的讨论中，我们看到了与 TCP 的各种定时器，包括重传定时器、持续定时器和保活定时器，此外 TCP 中还有几个定时器我们还未涉及。这里总的来看看 TCP 中的各个定时器。TCP 为每条连接总共建立了七个定时器，依次为：

1)“连接建立(connection establishment)”定时器在发送 SYN 报文段建立一条新连接时启动。如果在 75 秒内没有收到响应，连接建立将中止。

2)“重传(retransmission)”定时器在 TCP 发送某个数据段时设定。如果该定时器超时而对端的确认还未到达，TCP 将重传该数据段。重传定时器的值 (即 TCP 等待对端确认的时间)是动态计算的，与 RTT 的估计值密切相关，且还取决于该报文段已被重传的次数。

3)“延迟 ACK(delayed ACK)”定时器在 TCP 收到必须被确认但无需马上发出确认的数据时设定。如果在 200ms 内，有数据要在该连接上发送，延迟的 ACK 响应就可随着数据一起发送回对端，称为捎带确认。如果 200ms 后，该确认未能被捎带出去，则定时器超时，此时需要发送一个立即确认。

4)“持续 (persist)”定时器在连接对端通告接收窗口为 0，阻止 TCP 继续发送数据时设定。由于连接对端发送的窗口通告不可靠(只有数据才会被确认，ACK 不会被确认)，允许 TCP 继续发送数据的后续窗口更新有可能丢失。因此，如果 TCP 有数据要发送，但对端通告接收窗口为 0，则持续定时器启动，超时后向对端发送 1 字节的数据，判定对端接收窗口是否已打开。

5)“保活(keep alive)”定时器在 TCP 控制块的 so_options 字段设置了 SOF_KEEPALIVE 选项时生效。如果连接的连续空闲时间超过 2 小时，则保活定时器超时，此时应向对端发送连接探测报文段，强迫对端响应。如果收到了期待的响应， TCP 可确定对端主机工作正常，在该连接再次空闲超过 2 小时之前，TCP 不会再进行保活测试。如果收到的是 RST 复位响应， TCP 可确定对端主机已重启。如果连续若干次保活测试都未收到响应， TCP 就假定对端主机已崩溃，但它无法区分是主机故障还是连接故障。

6) FIN_WAIT_2 定时器，当某个连接从 FIN_WAIT_1 状态变迁到 FIN_WAIT_2 状态并且不能再接收任何新数据时，FIN_WAIT_2 定时器启动，设为 10 分钟。定时器超时后，重新设为 75 秒，第二次超时后连接被关闭。加入这个定时器的目的是为了避免如果对端一直不发送 FIN，某个连接会永远滞留在 FIN _ WAIT_ 2 状态（假设 TCP 不选用半打开功能）。

7) TIME_WAIT 定时器，一般也称为 2MSL 定时器。2MSL 指两倍的 MSL，即最大报文段生存时间。当连接转移到 TIME_WAIT 状态，即连接主动关闭时，定时器启动。状态转换图那一节中已经详细说明了需要 2MSL 等待状态的原因。连接进入 TIME_WAIT 状态时，定时器设定为 1 分钟，超时后，TCP 控制块被删除，端口号可重新使用。

前面的 7 个定时器中，重传定时器使用 rtime 字段计数，持续定时器使用 persist_cnt 字段计数，其他五个定时器除延迟 ACK 定时器外都使用 rtime 字段计数，从上面的描述中可以看出，这四个定时器是 TCP 处于四种不同的状态时使用的，因此四个定时器完全独立的使用 rtime 字段而不会互相影响。延迟 ACK 定时器使用系统 250ms 周期性定时来完成的。

LWIP 中包括两个定时器函数：一个函数每 250 ms 调用一次(快速定时器)；另一个函数每 500ms 调用一次(慢速定时器)。延迟 ACK 定时器与其他 6 个定时器有所不同：如果某个连接上设定了延迟 ACK 定时器，那么下一次 250ms 定时器超时后，延迟的 ACK 必须被发送(实际的 ACK 延迟时间在 0~250ms 之间)。其他的 6 个定时器每 500 ms 增加 1，当计数值

超过某些阈值时，则相应的动作被触发。

先看简单的快速定时器处理函数：

```
void tcp_fasttmr(void)
{
  struct tcp_pcb *pcb;
  for(pcb = tcp_active_pcbs; pcb != NULL; pcb = pcb->next) { //遍历整个 active 链表
    if (pcb->refused_data != NULL) { //  如果某个控制块还没有数据未接收
      err_t err;
      TCP_EVENT_RECV(pcb, pcb->refused_data, ERR_OK, err); //调用上层函数接收数据
      if (err == ERR_OK) {
        pcb->refused_data = NULL; //  成功接收则复位指针
       }
     }
    if (pcb->flags & TF_ACK_DELAY) {//若控制块开启了延迟 ACK 定时器
      tcp_ack_now(pcb);       //  发送一个立即确认
      pcb->flags &= ~(TF_ACK_DELAY | TF_ACK_NOW); //清除标志位
     }
  }// if
}// for
```

从上面可以看出，快速定时器处理函数主要做了两方面的工作，一是向上层递交上层一直未接收的数据，二是发送该连接上的立即确认数据段。与快速定时器相比，慢速定时器处理函数就显得相当的庞大了，这里我们只列出和各个定时器相关的部分，而对其他部分采用伪代码的方式加以描述，当然，这里所谓的其他部分就是我们在前面已经讲解过的部分了。

```
void tcp_slowtmr(void)
{
  ++tcp_ticks;
  pcb = tcp_active_pcbs;
  while (pcb != NULL) {
    pcb_remove = 0;
    该控制块的零窗口探查处理；
    该控制块的超时重传处理；
    if (pcb->state == FIN_WAIT_2) {    // FIN_WAIT_2 定时器超时
      if ((u32_t)(tcp_ticks - pcb->tmr) >
            TCP_FIN_WAIT_TIMEOUT / TCP_SLOW_INTERVAL) {
        ++pcb_remove;
      }
    }
   该控制块的超时保活处理；
#if TCP_QUEUE_OOSEQ
   if (pcb->ooseq != NULL &&    //  丢弃在 ooseq 队列中长时间未被处理的数据
            (u32_t)tcp_ticks - pcb->tmr >= pcb->rto * TCP_OOSEQ_TIMEOUT) {
      tcp_segs_free(pcb->ooseq);
      pcb->ooseq = NULL;
   }
```

```
#endif /* TCP_QUEUE_OOSEQ */
    if (pcb->state == SYN_RCVD) { // SYN_RCVD 状态超时
            if ((u32_t)(tcp_ticks - pcb->tmr) >
                    TCP_SYN_RCVD_TIMEOUT / TCP_SLOW_INTERVAL) {
                ++pcb_remove;
            }
    }
    if (pcb->state == LAST_ACK) { // LAST_ACK 定时器超时
                if ((u32_t)(tcp_ticks - pcb->tmr) > 2 * TCP_MSL / TCP_SLOW_INTERVAL) {
                    ++pcb_remove;
            }
    }
    若有超时则删除控制块；
    无超时则周期性的外发数据包（poll）；
    取得 active 链表上的下一个控制块；
 }// while
 pcb = tcp_tw_pcbs;   //处理 TIME-WAIT 链表
 while (pcb != NULL) {
      pcb_remove = 0;     // TIME-WAIT 定时器超时
      if ((u32_t)(tcp_ticks - pcb->tmr) > 2 * TCP_MSL / TCP_SLOW_INTERVAL) {
         ++pcb_remove;
      }
      若超时则删除控制块；
      取得 TIME-WAIT 链表上的下一个控制块；
 }// while
}//函数尾
```

可以看出各个定时器的实现都是利用全局变量 tcp_ticks 与 tmr 字段的差值来实现的。当 TCP 进入某个状态时，就会将相应 tmr 字段设置为当前的全局时钟 tcp_ticks 的值，所以上面的差值可以有效表示出 TCP 处于某个状态的时间。各个定时器超时后的处理也很相似，即将变量 pcb_remove 加 1，pcb_remove 变量是超时处理中最核心的变量了，当针对某个 PCB 控制块作完超时判断之后，函数通过判断 pcb_remove 的值来处理 TCP 控制块，当 pcb_remove 值大于 1 时，则表示该控制块上有超时事件发生，该控制块或被删除或被挂起。注意伪代码中的重传定时器超时并不会影响 pcb_remove 的值。如果细心，你还可以看到，上面的代码多了两个超时事件，即 SYN_RCVD 状态超时和 ooseq 队列数据超时，当然，这两个超时事件并不影响协议栈功能的实现。

最后来看看系统为每个超时时间设置的超时时间，从上面的代码中可以看出，它们是在各个宏定义里面实现的。

```
#define TCP_TMR_INTERVAL          250     //250ms
#define TCP_FAST_INTERVAL         TCP_TMR_INTERVAL   // 快速定时器
#define TCP_SLOW_INTERVAL         (2*TCP_TMR_INTERVAL) //  慢速定时器

#define TCP_FIN_WAIT_TIMEOUT     20000   // FIN_WAIT_2 状态超时时间
#define TCP_SYN_RCVD_TIMEOUT    20000   // SYN_RCVD 状态超时时间
```

#define TCP_OOSEQ_TIMEOUT 6U //ooseq 队列中数据等待的 rto 周期数
#define TCP_MSL 60000U // MSL

到这里，我们的 PCB 控制块中的各个字段基本都已涉及到了，除了 polltmr 和 pollinterval。这两个字段用于周期性调用函数 tcp_output，用以发送控制块上残留的未发送数据段。

27 TCP 终结与小结

TCP 还有最后一点东西需要扫尾。现在我们要跳出 tcp_process 函数，继续回到 tcp_input 中，前面说过，tcp_input 接收 IP 层递交上来的数据包，并根据数据包查找相应 TCP 控制块，并根据相关控制块所处的状态调用函数 tcp_timewait_input、tcp_listen_input 或 tcp_process 进行处理。如果是调用的前两个函数，则 tcp_input 在这两个函数返回后就结束了，但若调用的是 tcp_process 函数，则函数返回后，tcp_input 还要进行许多相应的处理。

要继续往下讲就得看看一个很重要的全局变量 recv_flags，前面说 TCP 全局变量的时候也简单说到过。这个变量与控制块中的 flags 字段相似，都是用来描述当前 TCP 控制块的所处状态的。flags 字段可以设置的各个标志位及其意义如下宏定义所示：

```
#define TF_ACK_DELAY        (u8_t)0x01U     // 延迟回复 ACK 包
#define TF_ACK_NOW          (u8_t)0x02U     // 立即发送 ACK 包
#define TF_INFR             (u8_t)0x04U     // 处于快速重传状态
#define TF_FIN              (u8_t)0x20U     // 本地上层应用关闭连接
#define TF_NODELAY          (u8_t)0x40U     // 禁止 Nagle 算法禁止
#define TF_NAGLEMEMERR      (u8_t)0x80U     // 发送缓存空间不足
```

上面的各个字段基本都已经涉及过了，再来看看全局变量 recv_flags 可以设置的各个标志位及其意义，如下宏定义所示：

```
#define TF_RESET        (u8_t)0x08U     // 接收到 RESET 包
#define TF_CLOSED       (u8_t)0x10U     // 在 LAST_ACK 状态收到 ACK 包，连接成功关闭
#define TF_GOT_FIN      (u8_t)0x20U     // 接收到 FIN 包
```

为什么要用两个字段来描述相应 TCP 控制块的状态呢，不是很明了？个人理解有两个原因：一是由于控制块中的 flags 字段本身是 8 位的，若以每位描述一种状态，则不足以描述上面的 9 种状态；二是上面的 9 种描述状态很明显可以分为两类，第一类的 6 种与 TCP 的数据包处理密切相关，第二类的 3 种与 TCP 的状态转换密切相关。

在 tcp_input 每次调用 tcp_process 之前，recv_flags 都会被初始化为 0，在 tcp_process 的处理中，相关控制块在完成状态转换后，该全局变量与状态转换相关的位则会被置位，在函数返回到 tcp_input 后，tcp_input 还会根据相应设置好的 recv_flags 值对控制块做后续处理。

```
    if (recv_flags & TF_RESET) {
      // TF_RESET 标志表示接收到了对端的 RESET 包
      TCP_EVENT_ERR(pcb->errf, pcb->callback_arg, ERR_RST); // 若注册了回调函数
                                                            // 则调用该函数通知上层
      tcp_pcb_remove(&tcp_active_pcbs, pcb); // 将控制块从链表中删除
      memp_free(MEMP_TCP_PCB, pcb);   // 释放控制块内存空间
    } else if (recv_flags & TF_CLOSED) {     // TF_CLOSED 表示服务器成功关闭连接
      tcp_pcb_remove(&tcp_active_pcbs, pcb); // 将控制块从链表中删除
      memp_free(MEMP_TCP_PCB, pcb); // 释放控制块内存空间
    } else {
      err = ERR_OK;
      if (pcb->acked > 0) { // 如果收到的数据包确认了 unacked 队列中的数据
        TCP_EVENT_SENT(pcb, pcb->acked, err); //则可调用自定义的函数发送数据包
```

```
        }
        if (recv_data != NULL) { // 若成功的接收了数据包中的数据
          if(flags & TCP_PSH) { //全局变量 flags 保存的是 TCP 头部中的标志字段
            recv_data->flags |= PBUF_FLAG_PUSH; // 将数据包 pbuf 字段
          }                                                    // 设置 PUSH 标志
         TCP_EVENT_RECV(pcb, recv_data, ERR_OK, err); // 调用自定义的函数接收数据
         if (err != ERR_OK) {    //若上层接收数据失败
            pcb->refused_data = recv_data;   //用控制块 refused_data 字段暂存数据
          }
        } // if (recv_data != NULL)

        if (recv_flags & TF_GOT_FIN) {    // 如果接收到 FIN 包
          TCP_EVENT_RECV(pcb, NULL, ERR_OK, err); // 调用自定义的函数接收数据
        }
        if (err == ERR_OK) { //若处理正常则
          tcp_output(pcb);   // 试图往外发数据包
        }
      }   //else
```

有两个地方需要说一下，首先是函数 tcp_pcb_remove，源码如下所示，代码里面比较重要的两个函数是 TCP_RMV 和 tcp_pcb_purge，这里就不再仔细说明了。

```
void tcp_pcb_remove(struct tcp_pcb **pcblist, struct tcp_pcb *pcb)
{
  TCP_RMV(pcblist, pcb);   // 从某链表上移除 PCB 控制块
  tcp_pcb_purge(pcb);        // 清空控制块的数据缓冲队列，释放内存空间

  if (pcb->state != TIME_WAIT &&   //如果该控制块上有被延迟的 ACK，则立即发送
     pcb->state != LISTEN &&
     pcb->flags & TF_ACK_DELAY) {
    pcb->flags |= TF_ACK_NOW;
    tcp_output(pcb);
   }
  pcb->state = CLOSED;   // 置状态
}
```

还有个需要注意的地方是回调函数的调用：如上面 TCP_EVENT_XXX 所示。在实际应用程序中，我们可以通过回调函数的方式与 LWIP 内核交互，在初始化一个 PCB 控制块的时候，可以设定控制块中相应函数指针字段的初始值，包括 sent、recv、connected、accept、poll、errf 等。在内核处理中，会在 TCP_EVENT_XXX 处调用我们预先注册的函数，从而完成应用程序与协议栈之间的交互。关于应用程序与协议栈间接口的问题，又是一个庞大的工程，也是我们以后会继续讨论的重点。

到这里，TCP 部分就基本讲完了，当然 TCP 层中还有一些东西没有讲到，如 tcp_write 等函数。TCP 层学习的关键是了解整个 TCP 层运行的机制，在这个基础上去阅读源代码，应该不会存在什么的问题的。从《随笔 14》到《随笔 27》，可以看出 TCP 的篇幅实在是太多了，实际源代码也是如此，从代码量上看，TCP 部分占了整个协议栈代码量的一半左右。注意，一般讲 TCP 协议时都会谈到 UDP，但在这里我还不想涉及 UDP 协议，原因是在一

般嵌入式产品中，都需要提供有效可靠的网络服务，而 UDP 的本质特点让其无法满足这一要求。所以，如果真是要将 LWIP 用于我们的产品中，则使用的基本是 TCP 协议，在后续的讲解中，我们会看到应用程序怎样利用 LWIP 建立一个 Web 服务器，这使得我们可以远程的通过 http 访问我们的设备了。接下来要说的就是协议栈与应用程序间的接口问题了。

28 API 实现及相关数据结构

LwIP 的内部机制讲解完了，接下来是应用程序与 LwIP 之间的结构问题，即应用程序如何使用协议栈内部提供的各种服务。应用程序可以通过两种方式与协议栈交互：一是直接调用协议栈各个模块函数，二是利用 LwIP 提供的 API 函数。

直接调用协议栈内部各模块函数，即要创建一个 TCP 控制块时则调用函数 tcp_new，进行 TCP 连接时则调用 tcp_connect，要接收数据则要向控制块中注册相关的回调函数，等等。通过这种方式的交互最大的缺点就是代码编写难度大，且代码常常是晦涩难懂，代码的调试也会异常艰辛。

使用得更多的也是我更关注的，就是第二种方式：LwIP 的 API 函数。用户程序通过简单的 API 函数调用就可以享用协议栈提供的服务了，这使得应用程序的编写更加容易。LwIP 的 API 相比 BSD 实现的 API 有个明显的特点，即数据在协议栈与应用程序中交互时不用拷贝，这点可以为我们节省大量的时间和内存开销，也说明 LwIP 更适用于嵌入式这种系统资源受限的系统。但有个关键是，我们必须有个好的内存管理机制，因为协议栈和应用程序都会对同一片数据进行操作。

LwIP 的 API 的实现主要有两部分组成：一部分驻留在用户进程中，一部分驻留在 TCP/IP 协议栈进程中。这两个部分间通过操作系统模拟层提供的进程通信机制(IPC)进行通信，从而完成用户进程与协议栈间的通信，IPC 包括共享内存、消息传递和信号量。通常尽可能多的工作在用户进程内的 API 部分实现，例如运算量及时间开销大的工作；TCP/IP 协议栈进程中的 API 部分只完成少量的工作，主要是完成进程间的通讯工作。两部分 API 之间通过共享内存传递数据，对于共享内存区的描述是采用和 pbuf 类似的结构来实现。综上，可以用简单的一段话来描述这种 API 实现的机制：API 函数库中处理网络连接的函数驻留在 TCP/IP 进程中。位于应用程序进程中的 API 函数使用邮箱这种通讯协议向驻留在 TCP/IP 进程中的 API 函数传递消息。这个消息包括需要协议栈执行的操作类型及相关参数。驻留在 TCP/IP 进程中的 API 函数执行这个操作并通过消息传递向应用程序返回操作结果。

很早以前描述过结构 pbuf，它是协议栈内部用来描述数据包的一种方式。这里介绍数据结构 netbuf，它是 API 用来描述数据的一种方式。netbuf 是基于 pbuf 来实现的，其结构如下所示，很明显，它就是包含了几个典型结构的指针。

```
struct netbuf {
  struct pbuf *p, *ptr;
  struct ip_addr *addr;
  u16_t port;
};
```

注意，这里的 netbuf 只是相当于一个数据头，而真正保存数据的还是字段 p 指向的 pbuf 链表。字段 ptr 也是指向该 netbuf 的 pbuf 链表，但与 p 的区别在于：p 一直指向 pbuf 链表中的第一个 pbuf 结构，而 ptr 则不然，它可能指向链表中的其他位置，源文档里面把它描述为 fragment pionter，与该指针调整密切相关的函数是 netbuf_next 和 netbuf_first。还有两个字段 addr 和 port 分别表示发出 netbuf 数据端的 IP 地址和端口号，这两个字段实际用处似乎不大，API 定义了宏 netbuf_fromaddr 和 netbuf_fromport 分别用于返回某个 netbuf 结构中这两个字段的值。

与 netbuf 相关的处理函数很多，在源文档 15.2 节中对每个函数也有了详细的说明，这

里说说比较重要的几个。netbuf_new 用于分配一个新的 netbuf 结构，注意这里只是一个头部结构，而真正需要的存储数据区域是在函数 netbuf_alloc 中分配的，同理函数 netbuf_delete 用于删除一个 netbuf 结构，同时函数 pbuf_free 会被调用，用以删除数据区域的空间。以上这几个函数使用的简单例子如下：

```
struct netbuf    *buf;     // 申明指针
buf = netbuf_new();              // 申请新的 netbuf
netbuf_alloc(buf, 200);          // 为 netbuf 分配 200 字节的空间
。。。。                          // 使用 buf 做相关事情
netbuf_delete(buf);              // 删除 buf
```

讲过了 API 的内存管理，再来讲具体的 API 函数。与前面的对应，API 函数由两部分组成，分别在文件 api_lib.c 和 api_msg.c 中。前者包括了用户程序直接调用的 API 接口函数，后者包括了与协议栈进程通信的 API 函数，用户程序不可直接调用。这两部分 API 函数之间通过邮箱传递的消息进行通信。这里又要涉及到两个重要的数据结构：API 应用接口部分提供给上层应用以描述一个网络连接的数据结构 netconn，以及描述两部分 API 函数间传递的消息结构的 api_msg。

应用程序要使用 API 函数建立一个连接连接，首先应该建立一个 netconn 的结构来描述这个连接的各种属性。netconn 结构如下，其中去掉了不必要的编译选项和注释。

```
struct netconn {
  enum netconn_type type;   // 连接的类型，包括 TCP, UDP 等
  enum netconn_state state;   // 连接的状态
  union {                        // 共用体，内核用来描述某个连接
    struct ip_pcb    *ip;
    struct tcp_pcb *tcp;
    struct udp_pcb *udp;
    struct raw_pcb *raw;
  } pcb;
  err_t err;       // 该连接最近一次发生错误的编码
  sys_sem_t op_completed;   // 用于两部分 API 间同步的信号量
  sys_mbox_t recvmbox;       // 接收数据的邮箱
  sys_mbox_t acceptmbox;     // 服务器用来接受外部连接的邮箱
  int socket;   // 该字段只在 socket 实现中使用
  u16_t recv_avail; //
  struct api_msg_msg *write_msg; // 对数据不能正常处理时，保存信息
  int write_offset;     // 同上，表示已经处理数据的多少
  netconn_callback callback; // 回调函数，在发生与该 netconn 相关的事件时可以调用
};
```

write_msg 是 api_msg_msg 类型的指针，该结构后续讲到；netconn_callback 是 API 定义的一种函数指针类型，其定义为：

```
typedef void (* netconn_callback)(struct netconn *, enum netconn_evt, u16_t len);
```

关键字 typedef 的功能是定义新的类型。这里它用于定义一种 netconn_callback 的类型，这种类型为指向某种函数的指针，且这种函数有三个类型的输入参数，返回类型为 void。在这定义以后就可以像使用 int,char 一样使用 netconn_callback，也即它也表示一种数据类型了。所以上面的 callback 字段表示一个函数指针，当把某个函数名赋给该字段后，该字段就记录了这个函数的起始地址。

netconn 的其他字段后面用到时再详解。接下来看看两部分 API 函数间传递的消息结构的 api_msg：

```
struct api_msg {
    void (* function)(struct api_msg_msg *msg);     // 函数指针
    struct api_msg_msg msg;          // 函数执行时需要的参数
}
```

字段 function 是一个函数指针，通常当消息被构造时，该字段被填充为 api_msg.c 中的某个与协议栈接口的 API 函数，然后该消息被投递到协议栈进程中进行处理，协议栈进程解析出并执行该函数，这样应用程序与协议栈之间的通信就完成了。字段 msg 中包含了这个函数执行时需要的所有参数，api_msg_msg 是个枚举类型，所以对于不同的 function，msg 有着不同的结构。api_msg_msg 结构如下所示：

```
struct api_msg_msg {
  struct netconn *conn;   // 与消息相关的某个连接
  union {
    struct netbuf *b;        // 函数 do_send 的参数
    struct {                 // 函数 do_newconn 的参数
      u8_t proto;
    } n;
    struct {                 // 函数 do_bind 和 do_connect 的参数
      struct ip_addr *ipaddr;
      u16_t port;
    } bc;
    struct {                 // 函数 do_getaddr 的参数
      struct ip_addr *ipaddr;
      u16_t *port;
      u8_t local;
    } ad;
    struct {    // 函数 do_write 的参数
      const void *dataptr;
      int len;
      u8_t apiflags;
    } w;
    struct {     // 函数 do_recv 的参数
      u16_t len;
    } r;
  } msg;
};
```

这个结构体只包含了两个字段：描述连接信息的 conn 和枚举类型 msg。在 api_msg_msg 中保存 conn 字段是必须的，因为 conn 结构中包含了与该连接相关的信箱和信号量等信息，协议栈进程要用这些信息来完成与应用进程间的同步与通信。枚举类型 msg 的各个成员与调用它的函数密切相关。因此在构建一个消息时，API 应首先填充消息的 function 字段，并针对特定的 function 种类往 api_msg_msg 的枚举字段写入参数，函数 function 被协议栈解析执行时，直接按照自己已定的格式取参数。这意味着，对于构造消息的 API 函数和解析消息的 function 函数，它们对参数个数、结构的认识是预先约定的，不能有其他变数。这一点

体现出 LwIP 呆板僵硬的一面。

29 API 消息机制

现在有必要来看看前面一直提到的内核协议栈进程是什么样子的。这个函数叫 tcpip_thread，其源码如下，其中去掉了不相关的编译选项和非重点讨论部分。

```
static void tcpip_thread(void *arg)
{
    struct tcpip_msg *msg;
    sys_timeout(IP_TMR_INTERVAL, ip_reass_timer, NULL);//创建 IP 分片重装超时事件
    sys_timeout(ARP_TMR_INTERVAL, arp_timer, NULL);// 创建 ARP 超时事件
    while (1) {                                    // 进程循环
        sys_mbox_fetch(mbox, (void *)&msg);       // 阻塞在邮箱上接收要处理的消息
        switch (msg->type) {    // 判断消息类型
          case TCPIP_MSG_API:   // 若是 API 消息，调用消息内部的 function 函数
                msg->msg.apimsg->function(&(msg->msg.apimsg->msg));
                break;
          case TCPIP_MSG_INPKT:// 若是接收到 IP 层递交的数据包
              if (msg->msg.inp.netif->flags & NETIF_FLAG_ETHARP) // 支持 ARP
                  ethernet_input(msg->msg.inp.p, msg->msg.inp.netif);//先进行 ARP 处
                                                          //理，再判断是否递交 IP 层处理
              else   //否则直接递交给 IP 层
                  ip_input(msg->msg.inp.p, msg->msg.inp.netif);
              memp_free(MEMP_TCPIP_MSG_INPKT, msg);
              break;
          。。。。。。
          default:   break;
        }// switch
    } // while
}//
```

邮箱 mbox 是协议栈初始化时建立的用于 tcpip_thread 接收消息的邮箱，该函数能够识别的消息类型是 tcpip_msg 结构的，所以不管是 API 部分还是 IP 数据包输入部分，都必须将自己的信息封装成 tcpip_msg 结构。

```
struct tcpip_msg {
  enum tcpip_msg_type type;   // 枚举结构，消息类型
  sys_sem_t *sem;       // 信号量指针，该字段似乎没怎么被用到
  union {     // 共用体，不同消息类型使用不同的结构
    struct api_msg *apimsg;   // API 消息指针
    struct netifapi_msg *netifapimsg;   // 不讨论
    struct {                    // 接收到 IP 层数据包相关指针
      struct pbuf *p;
      struct netif *netif;
      } inp;
```

```
    struct {        // 不讨论
      void (*f)(void *ctx);
      void *ctx;
     } cb;
    struct {        // 不讨论
      u32_t msecs;
      sys_timeout_handler h;
      void *arg;
    } tmo;
  } msg;     // 枚举类型的名字
};
```

这个结构和上节讨论的 api_msg_msg 有着很相似，api_msg_msg 里面也包含了一个叫做的 msg 的结构体。枚举型 tcpip_msg_type 的内部成员就是在函数 tcpip_thread 中看到的 TCPIP_MSG_API 和 TCPIP_MSG_INPKT 等，这里只讨论这两种类型。

API 函数内部调用来向内核进程发送消息的函数叫 tcpip_apimsg，该函数填充一个 TCPIP_MSG_API 类型的 tcpip_msg 结构，并把该结构投递到协议栈阻塞的邮箱 mbox：

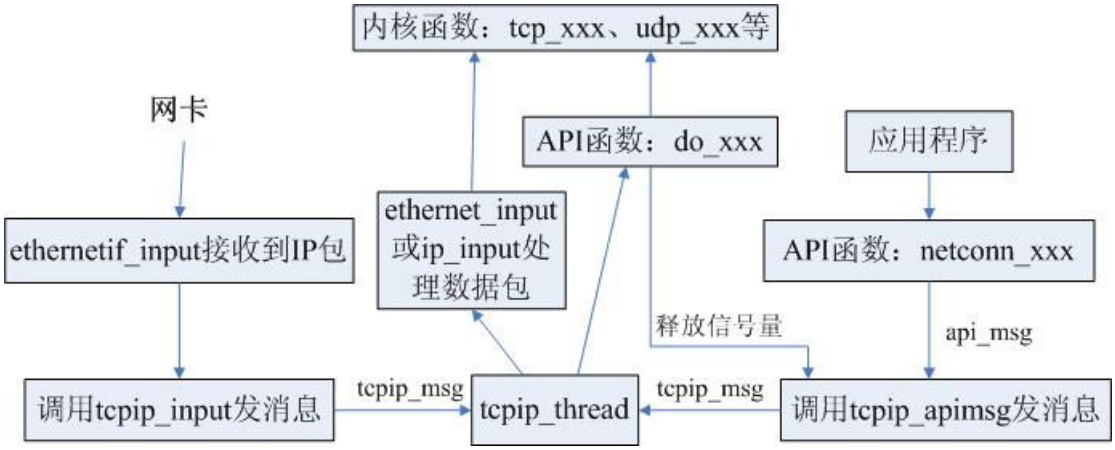
```
err_t tcpip_apimsg(struct api_msg *apimsg)
{
  struct tcpip_msg msg;         // 定义一个消息变量
  if (mbox != SYS_MBOX_NULL) {
    msg.type = TCPIP_MSG_API;     // 消息类型
    msg.msg.apimsg = apimsg;    // 使用 tcpip_msg 中的 msg.apimsg 字段记录相关信息
    sys_mbox_post(mbox, &msg); // 投递消息
    sys_arch_sem_wait(apimsg->msg.conn->op_completed, 0); //阻塞，等待内核处理完毕
    return ERR_OK;
  }
  return ERR_VAL;
}
```

底层向内核进程递交接收到的 IP 数据包是通过调用网络接口结构 netif 中指针 input 指向的函数来实现的（参见前面 netif 描述的部分）。通常这个函数是 tcpip_input。

```
err_t tcpip_input(struct pbuf *p, struct netif *inp)
{
  struct tcpip_msg *msg; // 定义了一个消息指针
  if (mbox != SYS_MBOX_NULL) {
    msg = memp_malloc(MEMP_TCPIP_MSG_INPKT);   // 为新的消息申请空间
    if (msg == NULL) {
      return ERR_MEM;
    }
    msg->type = TCPIP_MSG_INPKT; // 消息类型
    msg->msg.inp.p = p;   // 使用 tcpip_msg 中的 msg.inp 字段记录相关信息
    msg->msg.inp.netif = inp;
    if (sys_mbox_trypost(mbox, msg) != ERR_OK) { // 投递一次消息
      memp_free(MEMP_TCPIP_MSG_INPKT, msg);//投递不成功则删除消息
      return ERR_MEM;
```

```
        }
        return ERR_OK;
    }
    return ERR_VAL;
}
```

哎，这个过程是十分的复杂纠结啊，下面这个图可能更容易理解。这里函数 tcpip_thread 还是只处理 TCPIP_MSG_API 和 TCPIP_MSG_INPKT 这两种类型的消息。



从上面的图中可以清晰的看出两部分 API 函数之间的交互过程，以及应用程序和内核函数之间的交互过程。API 函数 netconn_xxx 在文件 api_lib.c 中，而 API 实现的另一部分函数 do_xxx 在 api_msg.c 中。

接下来的一节我们将精力集中在 api_lib.c 中的各个函数，而不去过多关心 api_msg.c 中的各个 do_xxx 是怎样与内核函数交互完成相关工作的。netconn_xxx 函数在 LwIP 说明文档 16 小节也有相关的描述，这里打算再重复的描述下各个函数的功能以及它们的实现过程。

30 API 函数及编程实例

函数 netconn_new 用来创建一个新的连接结构。连接结构的类型可以选择为 TCP 或 UDP 等。函数结构原型如下所示，参数 type 描述了连接的类型，可以为 NETCONN_TCP 或 NETCONN_UDP 等，这里都以 TCP 作为讨论的对象。

```
struct netconn* netconn_new(enum netconn_type type)
```

该函数首先调用 netconn_alloc 函数分配并初始化一个 netconn 结构。初始化的过程包括设置 netconn 结构类型字段，同时为该结构的 op_completed 创建一个信号量、recvmbox 字段创建一个接收邮箱。奇怪的是 netconn_alloc 函数并不是在文件 api_lib.c 文件中，而是在 api_msg.c 中,凌乱！接下来函数 netconn_new 会构建一个 api_msg 消息，该消息要求内核执行函数 do_newconn，最后函数 tcpip_apimsg 用来将消息包装成 tcpip_msg 结构并发送出去。tcpip_thread 函数解析该消息并调用函数 do_newconn，do_newconn 根据参数的类型最终调用函数 tcp_new 创建一个 TCP 控制块。tcpip_apimsg 会阻塞在一个信号量上，直至 do_newconn 释放该信号量。

函数 netconn_delete 用来删除一个连接结构 netconn。与前面的流程相同，它通过消息告诉内核执行 do_delconn，调用 tcp_close 函数关闭 TCP 连接。而后 netconn_delete 调用 netconn_free 函数释放 netconn 结构的空间。注意这里的 netconn_free 函数 netconn_alloc 函数一样，也不是在文件 api_lib.c 文件中，而是在 api_msg.c 中，尽管他们都是 netconn_xxx 结构。

netconn_bind 用于将一个 IP 地址及端口号与结构 netconn 绑定。事实上，内核是通过函数 do_bind 调用 tcp_bind 完成相应 TCP 控制块得绑定工作的。

netconn_connect 函数一般在客户端调用，用以将服务器端的 IP 地址和端口号与本地的 netconn 结构绑定。该函数与内核 tcp_connect 函数对应。

netconn_listen 函数一般在服务器端调用，用于将某个 TCP 控制块置为 LISTEN 状态。类似的函数 do_listen 会被调用，该函数有两个重要的工作：为结构 netconn 字段 acceptmbox 创建邮箱，该邮箱用于接受外部连接；向相应 TCP 的 PCB 控制块中 accept 字段注册一个回调函数 accept_function，当该 PCB 上有新连接出现时，回调函数会被调用，以向上面的 acceptmbox 邮箱中发送消息，告诉应用程序有新的连接到来，新连接的信息以 netconn 结构形式被保存在了邮箱中。

netconn_accept 函数在服务器上使用，用于接收远端的连接，该函数主要在阻塞在上面所述的 acceptmbox 邮箱上，当接收到新的连接后，在该邮箱上取下连接的 netconn 结构并返回。

netconn_recv 函数用于接收数据，接收到得数据被封装为 netbuf 结构。这里内核函数 tcp_recved 会被协议栈调用，以通知内核数据被正常接收，内核因此调整发送窗结构，返回 ACK 确认等。

函数 netconn_write 用于向相应的 TCP 连接上发送数据，主要这个函数只用于发送 TCP 数据，用于发送 UDP 数据的函数叫 netconn_send，这里先不讨论。netconn_write 函数原型如下，它用于将 dataptr 指向的 size 个数据放到连接 conn 的发送队列上，apiflags 用于描述

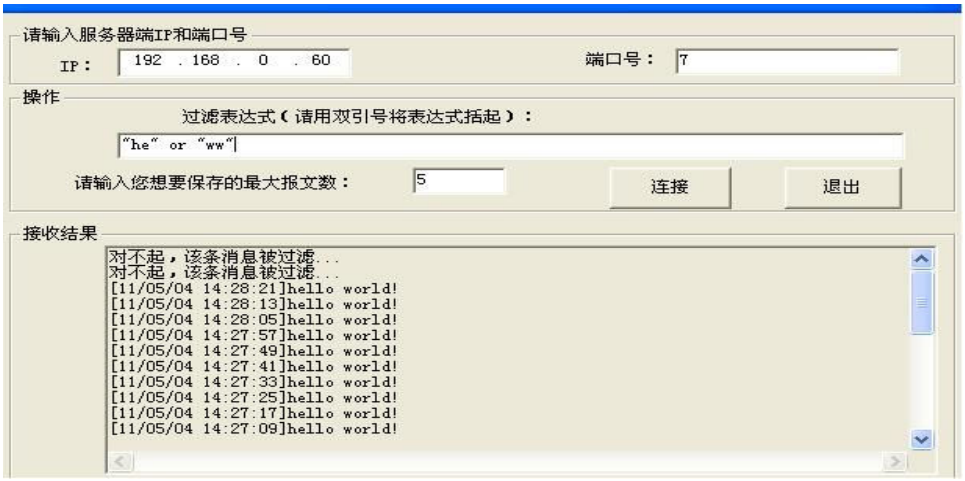
```
err_t netconn_write(struct netconn *conn, const void *dataptr, int size, u8_t apiflags)
```

对该数据的操作，包括是否拷贝，是否立即发送两种选择。最后 netconn_close 函数用于主动关闭连接。

API 函数就说这么一点点了。下面我们用这些 API 函数构造一个服务器程序。这个服务器程序很简单，它能响应一个客户端的连接和数据请求，并向客户端发送一个固定字符串。任务代码如下：

```
const uint8 data[]= "hello world!";     // 待发送字符串
void mytcp_thread(void *arg)
{
  struct netconn * conn, *newconn;   // API 描述的连接结构
  struct netbuf * buf;                    // API 数据缓冲

  conn = netconn_new(NETCONN_TCP); // 创建新的 TCP 连接结构
  netconn_bind(conn, NULL, 7);       // 该连接与端口 7 绑定
  netconn_listen(conn);          // 将结构置为侦听状态
  newconn = netconn_accept(conn);   // 接收到一个新的连接

  while(1)
  {
    buf = netconn_recv(newconn);   // 在新连接上接收到一个数据
    netbuf_delete(buf);                 // 删除接收到的数据
    netconn_write(newconn,data,sizeof(data), NETCONN_COPY); // 将字符串发送的客户端
  }
}
```

服务器程序之所以要这样设计是为了测试的方便，因为手上恰好有个小程序可以用来测试这个服务器程序以及我们的 LwIP 协议栈运转是否正常。这个小程序是当年参加中兴编程大赛的时候写的，名字叫报文监视器。它能接收某个 TCP 连接上的数据并能按照用户要求对这些数据进行过滤，去除用户不关心的数据。大嘴东哥和寝室的鹏鹏。。O(∩_∩)O~看到这个程序就想到了你们，大功臣啊。。。测试结果如下：



过滤表达式编辑框内的内容是用户输入的过滤条件，当接收的数据串满足过滤条件时，该字符串不会在接收结果中显示出来。过滤条件是一系列的引号括起来的字符串，它们可以用 or,and,not,括号等连接起来，组成很复杂的过滤条件。。不讲了。

首先，将过滤条件置为空，此时显示了从服务器接收到得所有数据“hello word!”,如图下方所示。然后将过滤条件设置为“he”or “ww”..即字符串中含有“he”或者“ww”字样的数据串将被滤除掉不以显示。。这正如接收结果中的前两行所示。OK…测试结果一切正常，

我们的 LwIP 稳定的跑起来了！不过，这里还可以用其他的测试方法，更常用的方法是构建一个 http 服务器，然后用我们的浏览器来连接服务器，这些在 LwIP 移植手册中有了很多例程以及详细的说明，不罗嗦了。

可见，使用 LwIP API 已经可以轻松完成所有 TCP 通信的相关任务了。除此之外，LwIP 还用自身的 API 函数实现了 BSD Socket API 函数。因为很多的软件编写是基于 BSD 套接字的，BSD 套接字更简单易懂，使用广泛，可见实现 Socket API 还是有必要的。但是 LwIP 说明文档中这样写道：

这一节提供使用 LwIP API 对 BSD Socket API 的一个简单实现。这个实现只能作为一个参考，不能用于实际编程中，因为它并不完善，比如它没有容错机制等。同时，这个简单实现也不支持 BSD Socket API 中的 select()与 poll()函数，因为 LwIP API 没有任何函数可以用于这两个函数的实现。要实现这两个函数，BSD socket 实现需要直接与 LwIP 协议栈通讯，而不是使用其 API。

所以这里不对 BSD Socket API 做详细讨论了，使用 LwIP API 完全可以完成相关的工作，且编程工作也很简单。

到这里，我们已经从头到尾的将 LwIP 协议走完了一遍，从网络接口层到 ARP 层,再到 IP 层，然后到 TCP 层，最后到 API 层。通常实际应用中，TCP 数据包也是按照这个次序依次被处理的。LwIP 还有很多其他内容还没有讨论到，首先是 UDP，接下来是 PPP，SILP，DNS，IGMP，DHCP，SNMP，IPV6 等等。这些都是在某些特殊的场合才会使用到的，不具有什么共性，所以这里先不涉及这些了。

全剧终。。。