TCP/IP 协议栈 LwIP 的设计与实现

Design and Implementation of the LwIP TCP/IP Stack

[瑞典]Adam Dunkels著

adam@sics.se

翻译　焦海波

marsstory99@hotmail.com

Swedish Institute of Computer Science
February 20, 2001

摘要

LwIP 是 TCP/IP 协议栈的一个实现。它的目的是减少内存使用率和代码大小，使 LwIP 适用于资源受限系统比如嵌入式系统。为了减少处理和内存需求，LwIP 使用不需要任何数据复制的经过裁剪的 API。

本文描述了 LwIP 的设计与实现。描述了在协议栈实现中以及像内存与缓冲管理这样的子系统中使用的算法和数据结构。本文还包括 LwIP 的参考手册以及使用 LwIP 的代码例子。

1 简介

最近几年，人们对计算机互联以及计算机无线网络支撑设备的兴趣一直不断的增长。计算机逐渐与日常使用的设备无缝集成在了一起，并且价格一直在下降。与此同时，无线网络技术比如蓝牙（Bluetooth）[HNI+98]及 IEEE 802.11b WLAN[BIG+97]正逐渐的出现在人们的视野中。这些新技术的出现，在许多诸如卫生保健、安全保密、运输及工业处理等领域提供了一个非常诱人的应用前景。一些像传感器一类的轻便设备可以连入互联网，以便随时随地进行监控。

在过去的近十年的时间里，互联网技术被证明拥有足够的灵活性以适应不断变化的网络环境。从原始的 ARPNET 一类的低速网络发展起来的互联网，发展到今天，在带宽和误码率方面拥有巨大差异的光纤连接技术已经使互联网实现了巨大的跨越。相当多的以互联网为基础的应用技术被开发出来。因此，未来的无线网络——使用已经存在的互联网技术成为人们的首选。同样，互联网在全球范围内的连通性也成为了人们选择它的动机之一。

一些轻便设备，比如在身体上使用的传感器，体积小而且便宜，内部的运算及存储资源有限，因此就必须在资源受限的情况下实现及处理 Internet 协议。本文讲述的就是在这样的条件下如何占用尽量少的资源实现一个轻型的 TCP/IP 协议栈，我们把该协议栈叫做 LwIP。

本文的章节安排是这样的：第 2、3、4 节对 LwIP 做一个总体上的描述，第 5 节是关于操作系统模拟层的内容，第 6 节是内存和缓冲区管理，第 7 节介绍 LwIP 网络接口抽象层，第 8、9、10 介绍 IP、UDP、TCP 协议的实现，第 11、12 节介绍如何与 LwIP 协议栈接口及 LwIP 提供的 API，第 13、14 节将分析协议栈的实现，第 15、16 节提供 LwIP API 的参考手册，17、18 节提供例子代码。

2 协议层

TCP/IP 协议族以分层的方式设计，每一层分别解决通讯问题的一部分。设计实现协议族——层可以提供指引，因为每一种协议可以被独立的实现。然而严格的按照分层的方式实现协议族，会因为协议层之间的通讯造成总体性能下降。要解决这个问题，协议的某些内部方面对其它协议来说应该可知，不过要注意的是，只有重要的信息在各层之间共享。

大部分的 TCP/IP 实现在应用层和底层协议层之间进行了严格的划分，而底层协议之间却可以进行或多或少的交叉存取。在大部分的操作系统中，底层协议族作为拥有应用层进程通讯入口的操作系统内核的一部分被实现。应用程序是 TCP/IP 实现的抽象表示，网络通讯与进程间通讯和文件 I/O 没多少差别。这意味着，因为应用层不知道底层协议使用的缓冲机制，那它就不能利用这些信息去做一些事情，比如，重新使用常用数据缓冲区。同样，当应用层发送数据，在被网络代码处理之前，这些数据必须由应用层进程内存空间复制到内部缓冲区。

像 LwIP 的目标系统这样的最小限度系统所使用的操作系统，通常不能在内核与应用层进程之间维持一个严格的保护屏障。这就允许使用一种比较松散的通讯机制，通过共享内存

的方式实现应用层与底层协议族之间的通讯。特别的，应用层能够了解底层协议使用的缓冲处理机制将使应用层可以更加有效的重复使用缓冲区。同样，既然应用层与网络代码可以使用相同的内存区，那么应用层就可以直接读写内部缓冲区，从而避免了内存复制产生的性能损失。

3 概览

与许多其它的 TCP/IP 实现一样，LwIP 也是以分层的协议为参照——设计实现 TCP/IP。每一个协议作为一个模块被实现，同时还提供了几个函数作为协议的入口点。尽管这些协议是被独立实现的，但是有些层却不是这样，就像上面讨论的，这样做的目的是为了在处理速度与内存占用率方面提升性能。比如，当验证一个到达的 TCP 段的校验和并且分解这个 TCP 段时，TCP 模块必须知道该 TCP 段的源及目的 IP 地址。因为 TCP 模块知道 IP 头的结构，因此它就可以自己提取这个信息，从而取代了通过函数调用传递 IP 地址信息的方式。

LwIP 由几个模块组成，除 TCP/IP 协议的实现模块外（IP，ICMP，UDP，TCP），还有包括许多相关支持模块。这些支持模块包括：操作系统模拟层（见第 5 节）、缓冲与内存管理子系统（见第 6 节）、网络接口函数（见第 7 节）及一组 Internet 校验和计算函数，LwIP 还包括一个 API 概要说明，详见第 12 节.

4 进程模型

TCP/IP 协议栈的进程模型指的是采用何种方法把系统分成不同的进程。首先要说的一种进程模型是 TCP/IP 协议族的每一个协议作为一个独立的进程存在。这种模型，必须符合协议的每一层，同时必须指定协议之间的通讯点。虽然，这种实现方法有它的优势，比如每一种协议可以随时参与到系统运行中，代码比较容易理解，调试方便，但是它的缺点也很明显。像前文描述过的，这种进程模型并不是最好的 TCP/IP 协议实现方法。同样更重要的是，数据跨层传递时将不得不产生进程切换（context switch）。对于接收一个 TCP 段来说，将会引起三次进程切换，从网络设备驱动层进程到 IP 进程，从 IP 进程到 TCP 进程，最终到应用层进程。对于大部分操作系统来说，进程切换得代价可是相当昂贵的。

另外一种比较普遍的方法是协议栈驻留在操作系统内核中，应用进程通过系统调用（system calls）与协议栈通讯。各层协议不必被严格的区分，但可以使用交叉协议分层技术。

LwIP 则采取将所有协议驻留在同一个进程的方式，以便独立于操作系统内核之外。应用程序既可以驻留在 LwIP 的进程中，也可以使用一个单独的进程。应用程序与 TCP/IP 协议栈通讯可以采用两种方法：一种是函数调用，这适用于应用程序与 LwIP 使用同一个进程的情况；另一种是使用更抽象的 API。

LwIP 在用户空间而不是操作系统内核实现，既有优点也有缺点。把 LwIP 作为一个进程的主要优点是可以轻易的移植到不同的操作系统中。由于 LwIP 被设计运行在小系统里，通常它既不支持进程换出（swapping out processes，这里译者将其翻译为进程换出，指得是操作系统将不具备运行条件的进程从内存换出到外存，以腾出内存空间，译者注）也不支持虚拟内存。因此就不会产生因 LwIP 进程的一部分被交换或分页到磁盘上（paged out to disk，即用到了虚拟内存，译者注），进程因等待磁盘激活而造成延时的问题。不过在获取一个偶然发生的服务请求之前因任务调度产生的等待延时依然是一个问题，不过在 LwIP 的设计中，这并没有妨碍它以后在操作系统内核实现。

5 操作系统模拟层

为了方便 LwIP 移植，属于操作系统的函数调用及数据结构并没有在代码中直接使用，而是用操作系统模拟层来代替对这些函数的使用。操作系统模拟层使用统一的接口提供定时

器、进程同步及消息传递机制等诸如此类的系统服务。原则上，移植 LwIP，只需针对目标操作系统修改模拟层实现即可。

TCP 用到的定时器功能由操作系统模拟层提供。这个定时器是一个时间间隔至少为 200ms 的单脉冲定时器（one-shot timer，单脉冲定时器，指的是当时钟启动时，它把存储寄存器的值复制到计数器中，然后晶体的每一个脉冲使计数器减 1。减至 0 时，产生一个中断，并停止工作，直至软件重新启动它，译者注），当时间溢出发生时就会调用一个已注册的函数。

进程同步机制仅提供了信号量。即使信号量不被底层的操作系统支持也可以使用其它基本的同步方式来模拟，比如条件变量或者加锁。

消息传递通过一个简单机制来实现，它使用一个被称作邮箱的抽象方法。邮箱有两种操作：邮递（post）与提取（fetch），邮递操作不会阻塞进程；相反，投递到邮箱的消息被操作系统模拟层放到队列中直至其它进程将它们取出。即使底层的操作系统本身并不支持邮箱机制，采用信号量的方式也是很容易实现的。

6 缓冲与内存管理

通讯系统里的内存与缓冲管理模块首要考虑的是如何适应不同大小的内存需求，一个 TCP 段可能有几百个字节，而一个 ICMP 回显数据却仅有几个字节。还有，为了避免复制，应该尽可能的让缓冲区中的数据内容驻留在不能被网络子系统管理的存储区中，比如应用程序存储区或者 ROM。

6．1 包缓冲区 pbufs

pbuf 是 LwIP 信息包的内部表示，为最小限度协议栈的特殊需求而设计。pbufs 与 BSD 实现中使用的 mbufs 相似。pbuf 结构即支持动态内存分配保存信息包内容，也支持让信息包数据驻留在静态存储区。pbufs 可以在一个链表中链接在一起，被称作一个 pbuf 链，这样一个信息包可以穿越几个 pbufs。

pbufs 有三种类型：PBUF RAM、PBUF ROM、PBUF POOL。图 1 所示的 pbuf 为 PBUF RAM 类型，包数据存储在由 pbuf 子系统管理的存储区中：

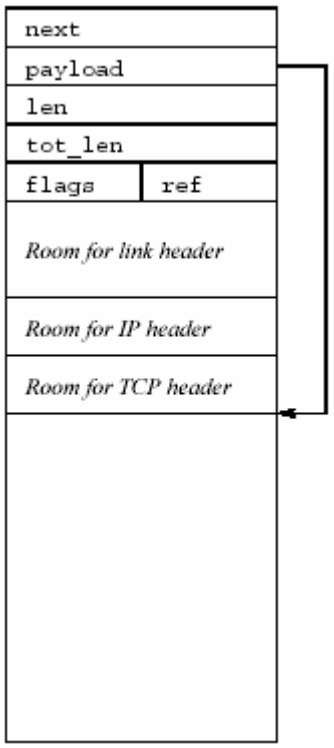

图 1 一个 PBUF RAM 类型的 pbuf，其数据保存在由 pbuf 子系统管理的存储区中

图 2 所示的 pbuf 是一个被链接的 pbuf 例子，在这个 pbuf 链中第一个 pbuf 是 PBUF RAM 类型，第二个是 PBUF ROM 类型，这意味着它所拥有的数据存储在 pbuf 子系统不能管理的存储区：

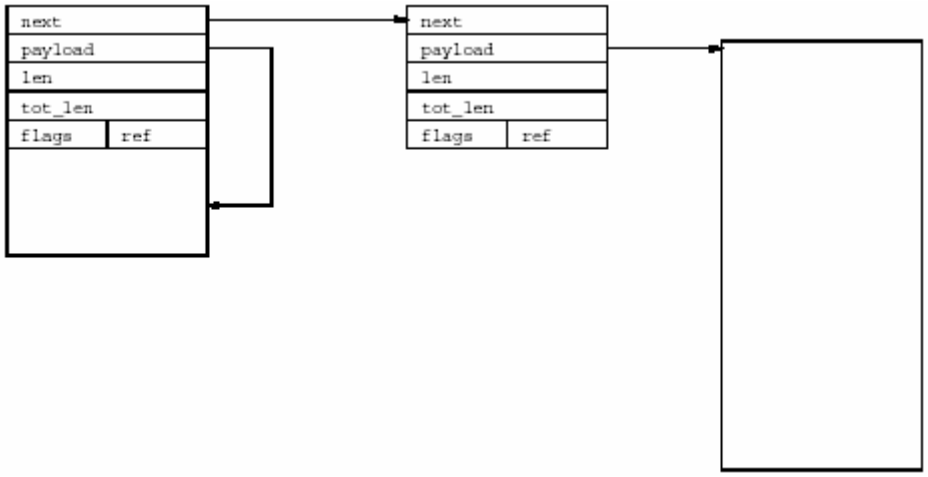

图 2 一个 PBUF RAM 类型的 pbuf 链接了一个数据存储在外部存储区的 PBUF ROM 类型的 pbuf

第三种 pbuf 类型，PBUF POOL，图 3 所示，它由分配自固定大小的 pbufs 池里的固定大小的 pbufs 组成。一个 pbuf 链可以由 pbufs 的不同类型组成。

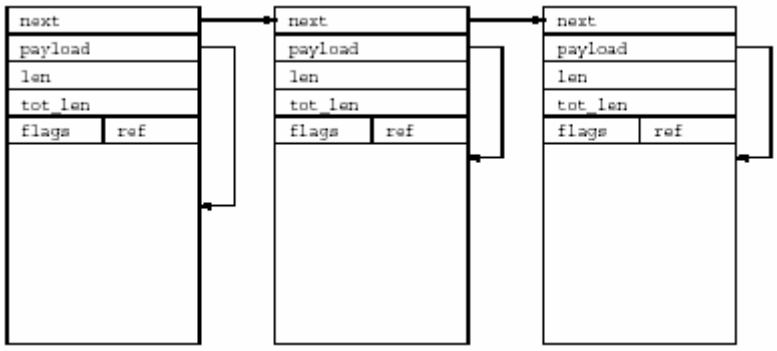

图 3 一个来自于 pbuf 池中的被链接的 PBUF POOL pbuf

这三种类型拥有不同的使用目的。PBUF POOL 主要用于网络设备驱动层，因为分配一个 pbuf 的操作可以快速完成，所以非常适合用于中断处理。PBUF ROM 类型的 pbufs 用于应用程序要发送的数据放置在应用程序管理的存储区的情况。在 pbuf 已经移交给 TCP/IP 协议栈后，这些数据是不能被编辑修改的，因此这种 pbuf 类型主要用于数据被放置在 ROM 中的情况（因此名字是 PBUF ROM）。为 PBUF ROM 类型的 pbuf 数据预置的包头存储在一个 PBUF RAM 类型的 pbuf 中，这个 pbuf 被链接到这个 PBUF ROM 类型的 pbuf 前面，如图 2 所示。

PBUF RAM 类型的 pbuf 还用于应用程序发送的数据被动态生成的情况。在这种情况下。pbuf 系统不仅为应用数据分配内存，还要给为这些数据预置的包头分配内存，见图 1。pbuf 系统不可能预先知道为这些数据预置什么样的包头，因而考虑最坏的情况。包头大小在编译时是可配置的。

其实，收到的 pbufs 是 PBUF POOL 类型，发送出的 pbufs 是 PBUF ROM 或 PBUF RAM 类型。

pbuf 的内部结构参见图 1 到图 3。pbuf 结构包括两个指针，两个长度字段，一个标志字段和一个引用计数（reference count）。next 字段是一个指向 pbuf 链中下一个 pbuf 的指针。payload 指针指向 pbuf 中数据的开始位置。len 字段包含 pbuf 中数据内容的长度。tot_len 字段包含当前 pbuf 的长度与在这个 pbuf 链中随后的所有 pbufs 的 len 字段之和。换句话说，tot_len 字段是 len 字段与 pbuf 链中随后一个 pbuf 的 tot_len 字段的和。flags 字段标识 pbuf 的类型，ref 字段包含一个引用计数。next 和 payload 字段是本地指针，它们占用的字节数与所使用的处理器架构有关。两个长度字段为 16 位无符号整形，flags 和 ref 字段是 4 位宽。pbuf 结构的实际大小与所使用的处理器架构下的指针大小及最小对齐方式有关。在 32 位指针及 4 字节对齐的架构里，pbuf 的大小为 16 个字节长，在 16 位指针及 1 字节对齐的架构里，pbuf 为 9 个字节长。

pbuf 模块提供了操作 pbufs 的函数。分配一个 pbuf 使用 pbuf_alloc()函数，该函数能够分配上面描述的三种类型中的任一类型 pbufs。pbuf_ref()函数增加引用计数。回收 pbuf 使用 pbuf_free()函数，该函数首先要减少 pbuf 索引计数（reference count）。如果引用计数已经减为 0，这个 pbuf 被回收。pbuf_realloc()函数可以收缩 pbuf 大小，以恰好够用的内存封装数据。pbuf_header()函数调整 payload 指针和长度字段以便为 pbuf 中的数据预置包头。pbuf_chain()与 pbuf_dechain()函数用于链接 pbufs。

6．2 内存管理

内存管理模块支撑的 pbuf 机制很简单。它负责处理内存连续区域的分配和回收以及收缩已分配内存块的大小。内存管理模块使用系统内存的一部分作为自己的专用区域，这确保

了网络系统不会使用系统中所有可用内存，即使网络系统使用了所有自己的内存，也不会扰乱其它程序的操作。

在内部，内存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。这个结构体拥有三个成员变量，两个指针一个标志，见图 4。next 与 prev 分别指向内存的下一个和上一个分配块，used 标志标示该内存块是否已被分配。

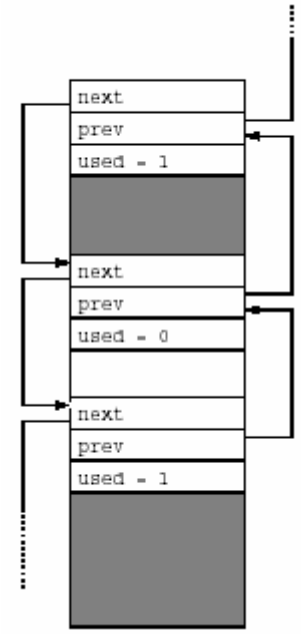

图 4 存储区分配结构

内存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。已经分配的内存块被回收后，使用标志 used 清零。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。

7 网络接口

在 LwIP，物理网络硬件的设备驱动通过一个与 BSD 中相似的网络接口结构来表示。见图 5。网络接口保存在一个全局链表中，它们通过结构体中的 next 指针连接。

每一个网络网络接口都拥有一个名字，保存在图 5 中的 name 字段。两个字符的名字标识网络接口使用的设备驱动的种类并且只用于这个接口在运行时由人工操作进行配置的情况。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。既然网络接口的名字不必具有唯一性，因此 num 字段被用来区分相同类别的不同网络接口。

三个 IP 地址 ip_addr，netmask 与 gw 用于 IP 层发送和接收信息包，有关它们的具体使用说明见下一节。一个网络接口只能拥有一个 IP 地址，每一个 IP 地址应当创建一个网络接口。

当收到一个信息包时，设备驱动程序调用 input 指针指向的函数。

网络接口通过 output 指针连接到设备驱动。这个指针指向设备驱动中一个向物理网络发送信息包的函数，当信息包被发送时由 IP 层调用。这个字段由设备驱动的初始设置函数填充。output 函数的第三个参数 ipaddr 是应该接收实际的链路层帧的主机的 IP 地址。它不

必与 IP 信息包的目的地址相同。特别地，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。

最后，state 指针指向网络接口的设备驱动特定状态，它由设备驱动设置。

```
struct netif {
    struct netif *next;
    char name[2];
    int num;
    struct ip_addr ip_addr;
    struct ip_addr netmask;
    struct ip_addr gw;
    void (* input)(struct pbuf *p, struct netif *inp);
    int (* output)(struct netif *netif, struct pbuf *p,
    struct ip_addr *ipaddr);
    void *state;
};
```

图 5 netif 结构

8 IP 处理

LwIP 仅实现了 IP 层大部分的基本功能，能够发送、接收以及转发信息包，但是不能接收和发送 IP 分片包，也不能处理携带 IP 参数选项的信息包。不过对大多数的应用来说，这不会引起任何问题。

8．1 接收信息包

对到达的 IP 信息包，由网络设备驱动调用 ip_input()函数开始处理。在这里完成对 IP 版本字段及包头长度的初始完整性检查，同时还要计算和验证包头校验和。协议栈假定代理会重新组合 IP 分片包为一个完整的包，因此它会把收到的 IP 分片包直接丢掉。同样，信息包携带的 IP 参数选项也被认为已经由代理处理过，这些内容会被删掉。

接下来，函数检查目的地址是否与网络接口的 IP 地址相符以确定信息包是否到达预定主机。网络接口在链表中被排序并且采用了线性检索。因为预计接口的数量比较少，所以没有实现比线性检索更好的检索策略。

如果一个到达的信息包被发现已经到达了目的主机，则由协议字段来决定信息包应该传送到哪一个上层协议。

8．2 发送信息包

外发的信息包由 ip_output()函数处理，该函数使用 ip_route()函数查找适当的网络接口来传送信息包。当外发的网络接口确定后，信息包传给以外发网络接口为参数的 ip_output_if()函数。在这里，所有的 IP 包头字段被填充，并且计算 IP 包头校验和。IP 信息包的源及目标地址作为参数被传递给 ip_output_if()函数。源 IP 地址可以被忽略，不过在这种情况下外发网络接口的 IP 地址会作为 IP 信息包的源 IP 地址被使用。

ip_route()函数通过线性检索网络接口链表找到适当的网络接口。在检索期间，用网络接口的网络掩码对 IP 信息包的目标地址进行掩码运算。如果经掩码运算的目标地址等与同样经掩码运算的接口 IP 地址（即网络地址相等，同在一个子网中，译者注），则选择这个接口。如

果没有找到匹配的，则使用缺省网络接口。缺省网络接口在启动时或运行时由人工操作进行配置（注意运行期间人工配置需要一个能配置协议栈的应用程序，LwIP 不包含这样的程序）。如果缺省网络接口的地址不匹配目的 IP 地址，则网络接口结构体（图 5）的 gw 字段被选择作为链路层帧的目的 IP 地址（注意，在这种情况下，IP 信息包的目标地址与链路层帧的目标地址是不同的）。路由的基本形式掩盖这样一个事实：一个网络可能拥有很多路由器依附它。不过对于最基本的情况，一个本地网络只拥有一个路由器进行路由工作。

因为传输层协议 UDP 与 TCP 在计算传输层校验和的时候需要拥有目标 IP 地址，因此在有些情况下，信息包被传递给 IP 层之前必须确定外发网络接口。这可以让传输层函数直接调用 ip_route()函数做到，因为在信息包到达 IP 层时已经知道了外发网络接口，因此就没有必要再检索网络接口链表，而是让那些协议改为直接调用 ip_output_if()函数。因为这个函数将网络接口作为参数，从而避免了对外发接口的检索。

8．3 转发信息包

如果没有网络接口的地址与到达的信息包的目标地址相同，信息包应该被转发。这项工作由 ip_forward()函数完成。在这里，TTL 字段值被减少（Time To Live 的简写，生存时间的意思，译者注），当减为 0 的时候，将会给 IP 信息包的最初发送者发送 ICMP 错误信息，并抛弃该信息包。因为 IP 包头被改变，因此需要调整 IP 包头校验和。不过，不必重新计算完整的校验和，因为可以使用简单的算法调整原始 IP 校验和。最后，信息包被转发到适当的网络接口。查找适当的网络接口的算法与发送信息包使用的算法相同。

8．4 ICMP 处理

ICMP 处理相当简单。ip_input()函数收到的 ICMP 信息包被移交给 icmp_input()函数对 ICMP 包头解码，然后进行适当的动作。某些 ICMP 消息被传递给上层协议，由传输层的特定函数处理。ICMP 目标不可到达消息可以由传输层发送，特别是通过 UDP，使用 icmp_dest_unreach()函数完成这项工作。

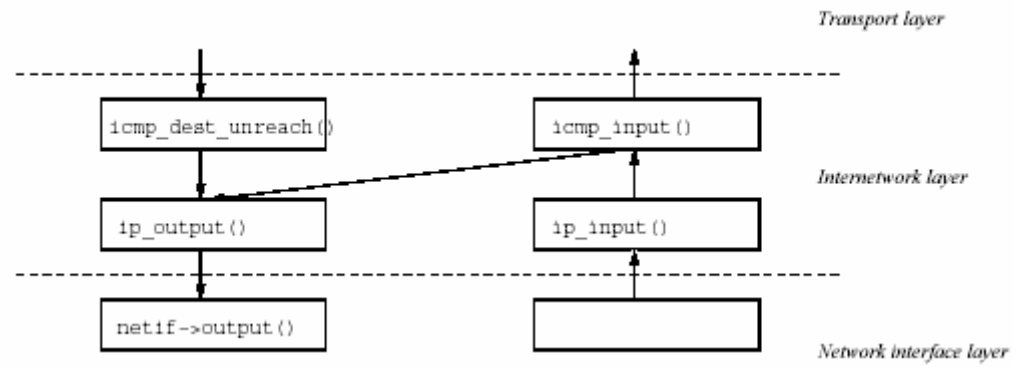

图 6 ICMP 处理

用 ICMP ECHO 消息来探测网络被广泛的使用，因而 ICMP 回显处理性能最优。实际处理被放置在 icmp_input()函数，由交换到达包的源与目的 IP 地址，改变 ICMP 类型为回显应答以及调整 ICMP 校验和组成。然后，信息包被回传给 IP 层传送。

9 UDP 处理

UDP 是被用来在不同的进程间分解信息包的一个简单协议。每一个 UDP 会话的状态保存在一个 PCB 结构体中，如图 7 所示。UDP PCBs 保存在一个链表中，当一个 UDP 数据包到达时对这个链表进行匹配检索。

```
struct udp_pcb {
    struct udp_pcb *next;
    struct ip_addr local_ip, dest_ip;
    u16_t local_port, dest_port;
    u8_t flags;
    u16_t chksum_len;
    void (* recv)(void *arg, struct udp_pcb *pcb, struct pbuf *p);
    void *recv_arg;
};
```

图 7 udp_pcb 结构

UDP PCB 结构体包含一个指向 UDP PCBs 全局链表中下一个 PCB 的指针。一个 UDP 会话由终端 IP 地址和端口号来定义，这些信息保存在 local_ip，dest_ip，local_port 以及 dest_port 字段中。flags 字段标识什么样的 UDP 校验和策略应该用于这个会话。或者可以完全关闭 UDP 校验和，或者使用 UDP 简化版（UDP Lite）[LDP99]校验和只覆盖数据包的一部分。如果使用 UDP Lite，chksum_len 字段指出应该进行校验和计算的数据段的长度。

最后两个参数 recv 与 recv_arg 是在由 PCB 指定的会话收到一个数据包时使用。在收到 UDP 数据包时调用 recv 指向的函数。由于 UDP 的简单性，输入输出处理比较简单，并且遵循图 8 所示的处理流程。要发送数据，由应用程序调用 udp_send()函数，然后再由该函数请求调用 udp_output()函数来完成。在此处进行必须的校验和计算以及填充 UDP 包头。因为校验和包括 IP 信息包的源地址，因此在某些情况下会调用 ip_route()函数以查找信息包将被传输到哪一个网络接口。网络接口的 IP 地址将作为信息包的源地址被使用。最后，信息包被移交给 ip_output_if()函数传送。

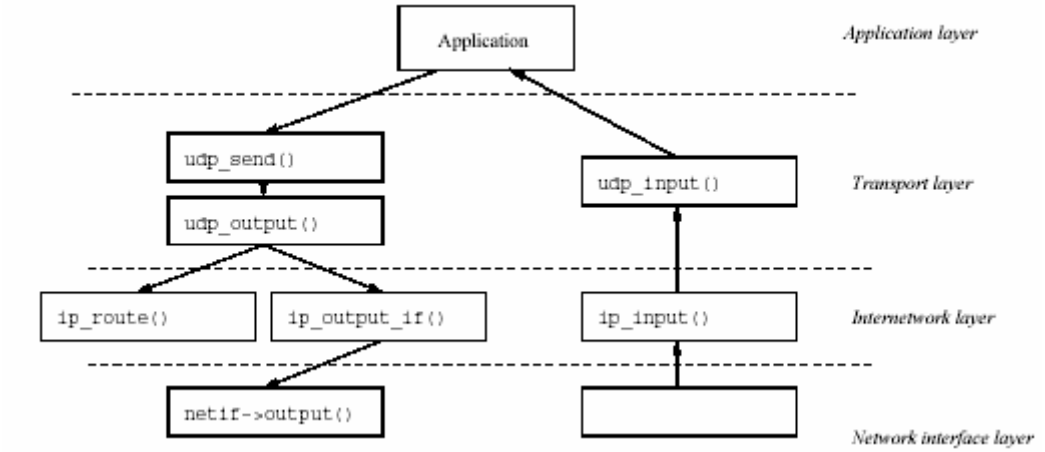

图 8 UDP 处理

当一个 UDP 数据包到达，IP 层调用 udp_input()函数。这里，如果校验和在这个会话中应该被使用，则 UDP 校验和被检查并分解数据包。当找到了相应的 UDP PCB，recv 函数被调用。

10 TCP 处理

TCP 属于传输层协议，它为应用层提供了可靠的字节流服务。对它的描述要比对其它协议的描述复杂的多，单从代码量来说，它就占了 LwIP 代码总量的 50%。

10．1 概览

基本的 TCP 处理过程被分割为六个功能函数来实现（如图 9 所示）：tcp_input()、tcp_process()及 tcp_receive()函数与 TCP 输入有关，tcp_write()、tcp_enqueue()及 tcp_output()则用于 TCP 输出。

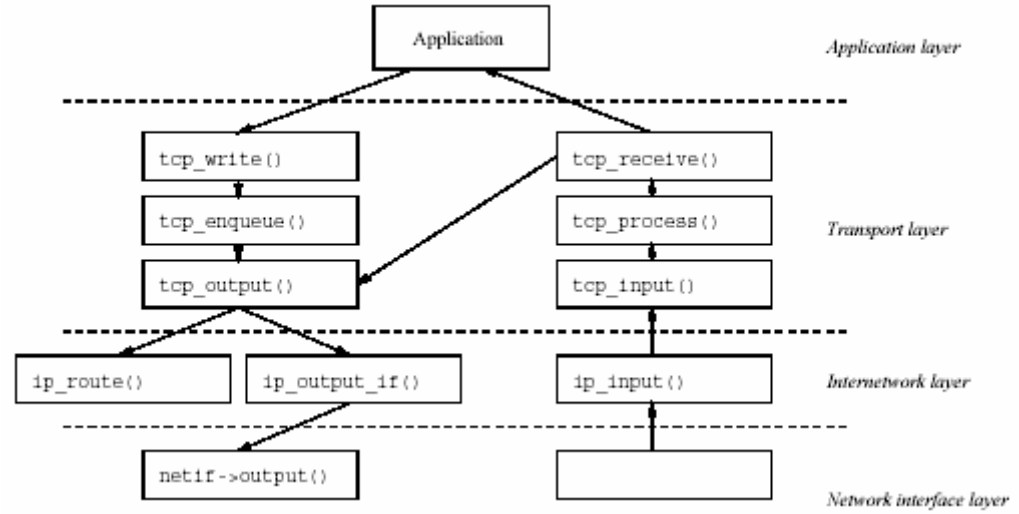

图 9 TCP 处理

现在，让我们先看看数据发送的过程是如何进行的。如上图所示，整个过程的发起者是应用层。应用层调用 ip_write()函数，接着 tcp_write()函数再将控制权交给 tcp_enqueue()函数，这个函数会在必要时将数据分割为适当大小的 TCP 段，然后再把这些 TCP 段放到所属连接的传输队列中。这时，tcp_output()函数会检查现在是不是能够发送数据，也就是判断接收器窗口是否拥有足够大的空间，阻塞窗口是否也足够大，如果条件满足，就使用 ip_route()及 ip_output_if()函数发送数据。

接下来，我们再看看数据的接收过程。图 9 所示，过程的发起者是网络接口层，这里不做描述。网络接口层将数据包传递给 ip_input()函数，该函数验证 IP 头后移交 TCP 段给 tcp_input()函数。tcp_input()函数完成两项工作：其一，初始完整性检查（也就是校验和验证与 TCP 选项解析）；其二，判定这个 TCP 段属于哪个 TCP 连接。接着，这个 TCP 段到达 tcp_process()函数，这个函数实现了 TCP 状态机，任何必要的状态转换在这里实现。当该 TCP 所属的连接正处于接受网络数据的状态，tcp_receive()函数将被调用。最终，tcp_receive()函数将数据传给上层的应用程序，完成接收过程。如果这个 TCP 段由一个不被承认的 ACK 应答数据构成，数据将会从缓冲区移走，它所占用的存储区被收回。同样，如果收到一个 ACK 应答确认数据，接收器同意接收更多的数据，如图 9 所示，tcp_output()函数将会被调用。

10．2 数据结构

由于 LwIP 被设计运行于内存受限的最小限度系统（即嵌入式系统，译者注），所以用于 TCP 实现的数据结构应该尽量较小。为此，我们必须在结构复杂性与使用这些结构的代码复杂性之间做出选择，为了能够让这些数据结构占用较少的内存单元，我们只能选择代码复杂性。

实际上，TCP PCB 结构还是相当大，见图 10。因为 TCP 连接在 LISTEN 和 TIME-WAIT 状态相对于其它状态的连接要保留的信息比较少，所以设计了一个较小的数据结构用于这种

状态下的连接。这个小数据结构包含在一个完整的 PCB 结构里，因此图 10 所示的结构成员列表显得有些笨拙。

```
struct tcp_pcb {
  struct tcp_pcb *next;
  enum tcp_state state;     /* TCP state */
  void (* accept)(void *arg, struct tcp_pcb *newpcb);
  void *accept_arg;
  struct ip_addr local_ip;
  u16_t local_port;
  struct ip_addr dest_ip;
  u16_t dest_port;
  u32_t rcv_nxt, rcv_wnd;   /* receiver variables */
  u16_t tmr;
  u32_t mss;                /* maximum segment size */
  u8_t flags;
  u16_t rttest;             /* rtt estimation */
  u32_t rtseq;              /* sequence no for rtt estimation */
  s32_t sa, sv;             /* rtt average and variance */
  u32_t rto;                /* retransmission time-out */
  u32_t lastack;            /* last ACK received */
  u8_t dupacks;             /* number of duplicate ACKs */
  u32_t cwnd, u32_t ssthresh;  /* congestion control variables */
  u32_t snd_ack, snd_nxt,   /* sender variables */
    snd_wnd, snd_wl1, snd_wl2, snd_lbb;
  void (* recv)(void *arg, struct tcp_pcb *pcb, struct pbuf *p);
  void *recv_arg;
  struct tcp_seg *unsent, *unacked, /* queues */
        *ooseq;
};
```

图 10 tcp_pcb 结构

这些 TCP PCB 结构保存在一个链表中，next 指针将它们链接在一块。state 变量包含网络连接的当前 TCP 状态。IP 地址和端口号存储着网络连接信息，而 mss 变量则包含网络连接所允许的段的最大容量。

rcv_nxt 与 rcv_wnd 字段在接收数据时使用。rcv_nxt 字段包含从远程终端期望得到的下一个包序号，因而该字段被用于向远程主机发送 ACK 包。接收器窗口由 rcv_wnd 字段保存并且字段值是在将要发送的 TCP 段中获取的。tmr 字段被作为一个定时器使用，当指定的计时结束后，连接应该被取消，比如处于 TIME-WAIT 状态的连接。连接允许的段的最大容量由 mss 字段指定。flags 字段包含了连接的附加状态信息，比如连接是快速恢复还是一个被延迟的 ACK 是否被发送。

rttest，rtseq，sa，sv 字段被用于 RTT（round-trip time：往返时间，指数据包在 TCP 链路上发送和收到确认的延时时间，译者注）评估。rtseq 保存段序号，rttest 保存段的发送时间。sa 及 sv 分别保存平均往返时间及时间差。这些字段被用于计算重发超时时间，得到的值保存在 rto 字段。

lastack 与 dupacks 字段用于快速重发及快速恢复的实现。lastack 字段包含收到的最后一个 ACK 包的序列号，dupacks 字段则对后续收到的与该序列号相等的 ACK 重复包计数。连接的当前阻塞窗口保存在 cwnd 字段，慢速启动阀值保存在 ssthresh 字段。

snd_ack、snd_nxt、snd_wnd、snd_wl1、snd_wl2、snd_lbb 这六个字段用于发送数据。接收器应答的最高顺序编号保存在 snd_ack 字段，下一个要发送的序号保存在 snd_nxt 字段。接收器公开窗口（advertised window）保存在 snd_wnd 字段。snd_wl1 与 snd_wl2 两个字段

用于更新 snd_wnd 字段。snd_lbb 字段保存传输队列最后一个字节的顺序编号。

函数指针 recv 及 recv_arg 用于向应用层传递收到的数据。三个队列字段 unsent、unacked 及 ooseq 用于发送和接收数据。从应用层接收但还未被发送的数据，被放置在 unsent 队列排队等待发送，已经发送但还未收到远程主机应答确认的数据保存在 unacked 队列。接收到序列以外的数据由 ooseq 缓冲。

```
struct tcp_seg {
  struct tcp_seg *next;
  u16_t len;
  struct pbuf *p;
  struct tcp_hdr *tcphdr;
  void *data;
  u16_t rtime;
};
```

图 11 tcp_seg 处理结构

图 11 所示的 tcp_seg 结构成员列表是 TCP 报文段的内部表示方法。结构的第一个成员是指向其自身的 next 指针，该指针用于将接收到的多个报文段链结在一起以形成一个等待队列。len 字段包含 TCP 报文段的长度。这意味着对于一个数据段来说，len 字段包含段中数据的长度，对于具备 SYN 或者 FIN 标志的空段来说，len 值为 1。pbuf 类型的指针 p 指向 TCP 报文段的存储缓冲区。tcphdr 及 data 指针分别指向 TCP 头和段中的数据。对于外发报文段，rtime 字段用于报文段的重发超时。而接收报文段是不需要重发的，因此，rtime 字段是不需要的，所以接收报文段该字段不分配内存。

10．3 顺序编号计算

用于对TCP字节流的每一个字节进行顺序编号的TCP序号是 32 位无符号整形，因此范围为[0，2^{32}-1]。如果TCP序号到到达 2^{32}-1，则从 0 开始重新编号，也就是说序号以 2^{32}为模进行计算。这意味着普通的比较操作符不能用于TCP序号。修改过的比较操作符叫做$<_{seq}$及$>_{seq}$，关系定义如下：

$s<_{seq}t \quad \Leftrightarrow \quad s-t<0$

$s>_{seq}t \quad \Leftrightarrow \quad s-t>0$

s 与 t 为 TCP 序号。比较操作符≤与≥同样被等效定义。这些操作符作为 C 的宏定义包含在头文件里。

10．4 队列与数据传输

tcp_enqueue()函数对要发送的数据按照适当大小进行分割，并对分割后的数据块进行顺序编号。在这里，数据被封装进 pbufs 并附加到 tcp_seg 结构。在 pbuf 内，TCP 头被创建，并且除应答数量，ackno、广播窗口，wnd 以外的其它所有字段都将被填充。tcp_output()函数可以在报文段排队时重新设置这些字段的值，这个函数完成实际的报文段传输。报文段被创建之后，它们被放入 PCB 内的 unsent 列表排队。tcp_enqueue()函数会尝试使用最大段大小的数据填充每一个报文段，当发现 unsent 队列的末尾存在一个低于最大容量的段时，函数会使用 pbuf 链表的功能将新数据附加到该段。

在 tcp_enqueue()函数格式化完成及排队报文段之后，tcp_output()函数被调用。它将检查当前窗口是否还有空间存放更多的数据。它通过获取阻塞窗口及发布的接收器窗口的最大数量来计算。接着，它会填充未被 tcp_enqueue()函数填充的 TCP 报头(TCP header)字段，然后使用 ip_route()与 ip_output_if()函数发送报文段。发送之后，报文段被放入 unacked

列表，并一直保留至收到相应的 ACK 应答包。

处于 unacked 列表的报文段，会像 10.8 节所描述的进行重发计时。当一个报文段被重发时，最初的 TCP 与 IP 报头被保留，TCP 报头需要作较小的改动。TCP 报头的 ackno 与 wnd 字段被设置为当前值，因为在报文段最初发送与重发期间，我们可能接收了数据。这仅改变报头的两个 16 位字，整体的 TCP 校验和不必被重新计算，因为简单算法[Rij94]可以用于更新校验和。报文段在最初发送时 IP 层已经加上了 IP 报头，并且没有理由去改变它。因而，重发报文段不需要重新计算 IP 报头校验和。

10．4．1 糊涂窗口的避免

糊涂窗口综合症[Cla82b](SWS，请参考《TCP-IP 详解卷 1：协议》第 22 章 22.3 节，译者注)是一种能够导致网络性能严重下降的 TCP 现象。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。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。

在 LwIP，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。

10．5 接收报文段

10．5．1 解析

当 TCP 报文段到达 tcp_input()函数，它们会在 TCP PCBs 之间被解析。解析的关键是源及目的 IP 地址和 TCP 端口号。解析报文段时，有两种 PCBs 类型必须被区分：相对于开放连接的 PCB 类型与相对于半开放连接的 PCB 类型。半开放连接指的是那些处于监听状态并且只有指定的本地 TCP 端口号及本地 IP 地址为任意值的连接；而开放连接则指拥有两个指定的 IP 地址及两个端口号的连接。许多 TCP 实现，像早期的 BSD 实现，使用具有单一入口缓存的 PCB 链表技术。在这之后的基本理论是：大部分的 TCP 连接组成代表性的显示一个大量位置[Mog92]的批量传输，这样就会导致一个高的缓冲区命中率。其它的缓冲方案包括保存两个单一的缓冲入口，一个用作已经被发送的最后一个包的 PCB，另一个用作已经收到的最后一个包的 PCB [PP93]。通过移动最近用过的 PCB 到链表的前端可以使用二者之中的任何一个方案。两种方案已经表明[MD92]通要胜过单一入口的方案。

对于 LwIP，只要在解析一个报文段时发现一个 PCB 匹配，这个 PCB 就要被移动到 PCB 链表的前端。不过用于监听状态连接的 PCB 不再被移动到前端，因为这种连接并不期望接收报文段。

10．5．2 接收数据

对到达报文段的实际处理在 tcp_receive()函数里进行。报文段应答序号与处在连接 unacked 队列里的报文段进行比较，如果应答序号比 unacked 队列里的报文段序号高，则这个报文段会被移出队列，并且为其分配的内存也被收回。

如果到达段的序号要比 PCB 中的 rcv_nxt 变量高，则该段脱离序列。脱离队列的报文段

会被放入 PCB 中的 ooseq 队列。如果到达段的序号等于 rcv_nxt，则通过调用 PCB 中的 recv 指向的函数将报文段转交到上层，并且通过到达段的长度来增加 rcv_nxt 值。因为序列中报文段的接收可能意味着先前收到的脱离序列的报文段是被期望接收的下一个段，ooseq 队列被检查。如果它包含一个序号等于 rcv_nxt 值的报文段，则通过调用 recv 指向的函数将该段转交给应用程序，并且 rcv_nxt 值被更新。这个过程会一直持续至 ooseq 队列为空或者 ooseq 队列中的下一个报文段脱离序列。

10．6 接受新的连接

处于监听状态也就是被动开放的连接，准备着接受远程主机新的连接请求。为了那些连接，必须建立新的 TCP PCB，并传递给打开初始监听 TCP 连接的应用程序。对于 LwIP，这个过程是通过让应用程序注册一个回调函数来实现的，这个回调函数在新的连接建立时调用。

当处于监听状态的连接收到一个 SYN 标志设置的 TCP 段时，一个新的连接被建立并且一个携带 SYN 与 ACK 标志的段被发送以响应收到的 SYN 段。连接进入 SYN-RCVD 状态并且等待发送的 SYN 段的应答。当应答到达，连接进入 ESTABLISHED 状态，并且 accept 指向的函数被调用（accept 函数指针请参考图 10 所示的 PCB 结构图）。

10．7 快速重发

LwIP 通过保存最后一个应答 ACK 来实现快速重发与恢复。这样，当收到相同序号的 ACK，TCP PCB 结构中的 dupacks 计数会加一。当 dupacks 值为 3，unacked 队列中的第一个报文段被重发且快速恢复被初始化。快速恢复按照[APS99]描述的过程实现。无论何时收到新数据的应答 ACK，dupacks 计数都将复位为 0。

10．8 定时器

与 BSD 中的 TCP 实现一样，LwIP 使用两个周期性定时器，周期分别为 200ms 和 500ms。这两个定时器又被用于实现更复杂的逻辑定时器，比如重发定时器，TIME-WAIT 定时器及延迟 ACK 定时器（delayed-ACK timer）。

细粒度定时器（fine grained timer）tcp_timer_fine()会遍历每一个 TCP PCB，检查是否存在应该被发送的被延迟的 ACKs，就像 tcp_pcb 结构里 flag 字段所指示的（图 10）。如果延迟 ACK 标志被设置，一个空的 TCP ACK 应答段被发送，并且标志被清除。

粗粒度定时器在 tcp_timer_coarse()函数里实现，同样扫描 PCB 列表。对每一个 PCB，将遍历未应答报文段列表(unacked 指针详细信息见图 10 及 11，译者注)，并且 rtime 变量值被增加。如果 rtime 值变得比 PCB 中 rto 变量给出的当前重发超时值大，报文段被重发并且重发超时加倍。只有在阻塞窗口与通告的接收器窗口的值允许的情况下报文段才被重发。重发之后，阻塞窗口被设置为最大段尺寸大小，慢启动阀值被设置为有效窗口大小的一半，并且慢启动在连接中被初始化。

对于 TIME-WAIT 状态的连接，粗粒度定时器也会增加 PCB 结构中的 tmr 字段值。当定时器到达 $2\times MSL$ 阀值，连接被取消。粗粒度定时器还会增加一个全局的 TCP 时钟值，tcp_ticks。这个时钟用于 RTT（往返时间：round-trip time）估算及重发超时（retransmission time-outs）。

10．9 往返时间估算

RTT 估算是 TCP 的主要部分，因为估算出的往返时间用于确定适当的重发超时。在 LwIP，往返时间算法与 BSD 实现类似，每一次往返时间就被测量一次并且使用[Jac88] 描述的 smoothing 函数计算适当的重发超时。

TCP PCB 中的 rttseq 变量保存着被测量过往返时间的报文段的序号。PCB 中的 rttest

变量保存报文段被第一次重发时的 tcp_ticks 值。当收到的 ACK 包的序号等于或者大于 rtseq，往返时间通过从 tcp_ticks 减去 rttest 来测量。如果重发发生在往返时间测量期间，测量值不被采纳。

10．10 阻塞控制

阻塞控制的实现令人惊讶的简单，其在输入与输出中仅有几行代码。当收到一个新数据的ACK应答，阻塞窗口cwnd值会加上最大段大小或$mss^2/cwnd$的大小，这取决于连接是慢速启动还是阻塞控制。当发送数据时，接收器通告的窗口与阻塞窗口的最小值被用于确定每个窗口能够发送多少数据。

11 协议栈接口

使用 TCP/IP 协议栈提供的服务有两种方法：1)直接调用 TCP 与 UDP 模块的函数；2)使用下一节将要介绍的 LwIP API 函数。

TCP 与 UDP 模块提供网络服务的一个基本接口，该接口基于函数回调技术，因此使用该接口的应用程序可以不用进行连续操作。不过，这会使应用程序编写难度加大且代码不易被理解。为了接收数据，应用程序会向协议栈注册一个回调函数。该回调函数与特定的连接相关联，当该关联的连接到达一个信息包，该回调函数就会被协议栈调用。

此外，与 TCP 和 UDP 模块直接接口的应用程序必须（至少部分地）驻留在像 TCP/IP 协议栈这样的进程中。这应归结于回调函数不能跨进程调用的事实（译者注：这里的意思是函数只能在进程内调用，不允许一个进程去调用另外一个进程的函数）。这即有优点也有缺点。优点是既然应用程序和 TCP/IP 协议栈驻留在同一个进程中，那么发送和接收数据就不再产生进程切换。主要缺点是应用程序不能使自己陷入长期的连续运算中，这样会导致通讯性能下降，原因是 TCP/IP 处理与连续运算是不能并行发生的。这个缺点可以通过把应用程序分为两部分来克服，一部分处理通讯，一部分处理运算。通讯部分驻留在 TCP/IP 进程，进行大量运算的部分放在一个单独的进程。将在下一节介绍的 LwIP API 提供了以这样一种方式分割应用程序的构造方法。

12 应用程序接口

由 BSD 提供的高级别的 Socket API（Socket：一个通过标准 UNIX 文件描述符和其他程序通讯的方法，可以认为 Socket 就是一个插座，用于网络的时候，Socket 提供一个网络插口，远程终端的连接请求可以当做是一条延伸的虚拟电缆需要找到一个可以连接的插座，当虚拟电缆插到这个插口上，通讯链路就建立了，译者注）不适合用于受限系统的 TCP/IP 实现，特别是 BSD Socket 需要将要发送的数据从应用程序复制到 TCP/IP 协议栈的内部缓冲区。复制数据的原因是应用程序与 TCP/IP 协议栈通常驻留在不同的受保护空间。大多数情况是应用程序是一个用户进程，而 TCP/IP 协议栈则驻留在操作系统内核。通过避免额外的复制操作，API 的性能可以大幅度提升。同样，为了复制数据，系统还需要为此分配额外的内存，这样每一个信息包都需要使用双倍的内存。

LwIP API 专为 LwIP 设计，所以它可以充分利用 LwIP 的内部结构以实现其设计目标。LwIP API 与 BSD API 类似，但操作相对低级。API 不需要在应用程序和协议栈之间复制数据，因为应用程序可以巧妙的直接处理内部缓冲区。

因为 BSD Socket API 易于理解，并且很多应用程序为它而写，所以 LwIP 保留一个 BSD Socket 兼容层是很有用的。在 17 节将介绍如何使用 LwIP API 重写 DSD Socket 函数。15

节是 LwIP API 的参考手册。

12．1 基本概念

从应用的角度来说，BSD Socket API 在连续的内存区域处理数据非常便于编写应用程序。因为应用程序内的数据处理通常是在这样的连续内存区域内进行的。但是，对于 LwIP，采用这种机制不具备任何优势。因为 LwIP 通常要处理的缓存中的数据是被分割放置在更小的内存块里的，所以当这些数据在被传递给应用程序之前必须先被复制到一块连续的内存区域内，这样即浪费处理时间又浪费内存。因此 LwIP 允许应用程序直接处理被分割的缓存内的数据以避免额外复制。

尽管 LwIP 与 BSD Socket API 非常相似，但是它们之间仍然存在着值得注意的区别，使用 BSD Socket API 的应用程序不必知道普通文件和网络连接的差别，而使用 LwIP API 的应用程序就必须知道正在使用的是一个网络连接。

接收到的网络数据暂存在分开的较小内存块内。因为许多应用程序需要在一块连续的内存区域内处理数据，因此这就需要一个专门的函数来负责从这些不连续的缓冲区复制数据到一个连续的内存区。发送数据采用不同的方式完成，这取决于数据是通过 TCP 连接还是通过 UDP 包发送。对 TCP，通过向 output 函数传递一个指向连续内存区域的指针发送数据。TCP/IP 协议栈会将这些数据分割成适当大小的信息包，然后放入传输队列。当发送 UDP 包时，应用程序必须明确的分配一个缓冲区并填上数据。TCP/IP 协议栈会在调用 output 函数时立即发送这个 UDP 包。

12．2 API 实现

由于 TCP/IP 协议栈处理模式的原因，API 被分成两部分实现。如图 12 所示，一部分作为应用程序的连接库实现，另一部分在 TCP/IP 进程内实现。这两部分之间采用由操作系统模拟层提供的进程间通讯机制（IPC）进行通讯。当前的实现是采用以下三种 IPC 方式：

- 共享内存
- 消息传递
- 信号量

虽然这些 IPC 类型被操作系统支持，但它们并不需要操作系统底层支持。因为操作系统本身并不支持它们，只是操作系统模拟层在模拟它们。

一般的设计原则是：在应用程序进程完成尽可能多的工作而不是在 TCP/IP 进程。这是很重要的一点，因为所有的需要通讯的进程都需要 TCP/IP 进程为它们提供通讯服务。控制与应用程序进行连接的 API 部分的代码覆盖量并不是很重要。这些代码可以在进程间共享，即使共享连接库并不被操作系统支持，这些代码仍然可以存储在 ROM 内。嵌入式系统尽管处理能力不足但通常都提供相当大的 ROM 空间。

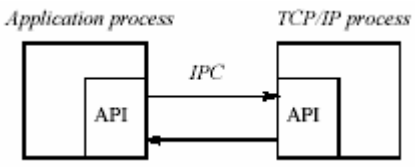

图 12 API 实现的分布

缓冲区管理位于 API 执行库，缓冲区的建立、复制于回收在应用程序内部完成。共享内存用于在应用程序和 TCP/IP 协议栈之间传递缓冲区。应用程序中用于通讯的缓冲区数据类型来源于 pbuf 类型（即从 pbuf 类型提取出的一种类型，译者注）。缓冲区传递用到的内存与分配的内存不同，使用共享内存也可以进行传递，因而可以在进程间共享用到的内存。运行 LwIP

的嵌入式操作系统一般有意不去实现任何形式的内存保护，所以这不是问题。

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 函数使用一个简单的通讯协议向驻留在 TCP/IP 进程中的 API 函数传递消息。这个消息包括应该执行的操作类型及操作参数。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。

13 代码统计分析

本节分析 LwIP 源码行数于编译成的目标代码大小的关系，代码在两种处理器架构下编译：

- Intel PentiumIII 处理器，即 Intel x86 处理器。代码在 FreeBSD4.1 下使用 gcc2.95.2 编译，编译器优化选项打开。
- 6502 处理器[Nab, Zak83]。代码使用 cc65 2.5.5 [vB] 编译，编译器优化选项打开。

Intel x86 处理器用于 32 位的寄存器，使用 32 位的指针。6502 处理器在今天主要用于嵌入式系统。它拥有一个 8 位累加器和两个 8 位索引寄存器，使用 16 位指针。

13．1 代码行

表 1 列举了 LwIP 源码中各部分的代码行数，图 13 比较直观的显示了各部分所占的比例。图中所示的支撑函数（Support functions）部分包括缓冲和内存管理函数，还包括 Internet 校验和计算函数。校验和函数使用普通的 C 算法实现，在实际配置时，应根据处理器的不同实现针对性的算法来取代普通的 C 算法实现。API 类包括两部分，一部分是应用程序 API，另一部分为 TCP/IP 协议栈 API。操作系统模拟层并没有包含在此次分析中，因为其大小完全依赖于所使用底层操作系统的不同而不同，所以没必要进行比较。

作为比较，这里忽略了所有的注视、空白行及头文件。我们看到，TCP 部分比其它协议的实现要大的多，API 部分和支撑函数（Support functions）部分两块加起来才和 TCP 部分差不多大。

表 1 代码行

Module	Lines of code	Relative size
TCP	1076	42%
Support functions	554	21%
API	523	20%
IP	189	7%
UDP	149	6%
ICMP	87	3%
Total	2578	100%

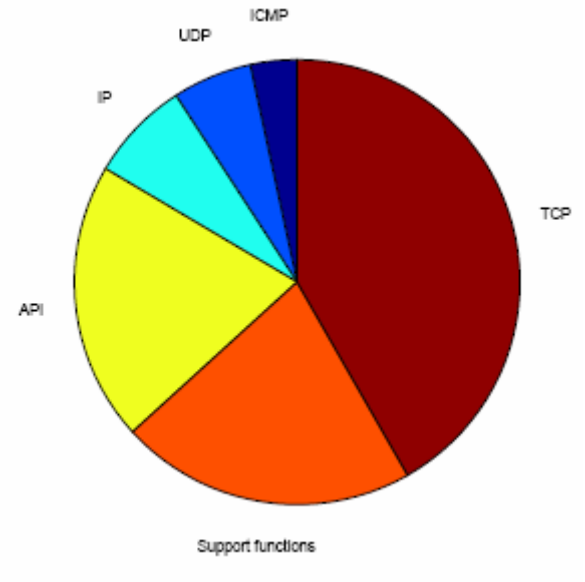

图 13 代码行

13．2 目标代码大小

表 2 x86 平台编译的目标代码大小

Module	Size (bytes)	Relative size
TCP	6584	48%
API	2556	18%
Support functions	2281	16%
IP	1173	8%
UDP	731	5%
ICMP	505	4%
Total	13830	100%

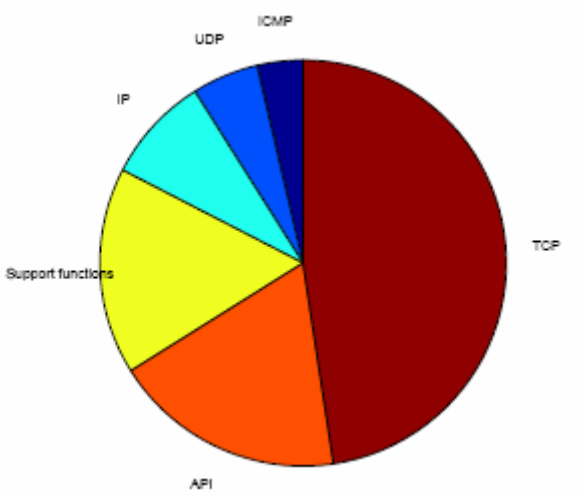

图 14 x86 平台编译的目标代码大小

表 2 列举了在 Intel x86 平台下编译后各部分目标代码的大小及所占比例。图 14 直观的显示了各部分在 LwIP 中所占的比例大小。我们看到各部分的顺序与表 1 稍微有些不同。在这里，API 要比支撑函数（Support functions）部分大，即使支撑函数部分的源码行比 API 多。我们还看到 TCP 部分编译后的代码大小占 48%，而其源码行则仅占 42%。检查 TCP 模块的汇编输出代码显示了造成这种情况的大致原因。TCP 模块包括大量的取指针指向内容（原文为 large amounts of pointer dereferencing，这里 pointer dereferencing 指的是 C 语言里使用*或者->操作符取指针指向的内容的意思，而 reference 则是指使用&操作符取地址的意思，译者注）的代码，这些代码被扩展成许多汇编代码行，因而这增加了目标代码的大小。因为在每个函数里，许多指针被取内容两三次，所以这可以通过编辑源码来优化，方法是只取指针指向的内容一次，然后存放在局部变量。当然减少了目标代码的大小，就需要给协议栈分配更多的 RAM 空间用于局部变量。

表 3 6502 平台编译的目标代码大小

Module	Size (bytes)	Relative size
TCP	11461	51%
Support functions	4149	18%
API	3847	17%
IP	1264	6%
UDP	1211	5%
ICMP	714	3%
Total	22646	100%

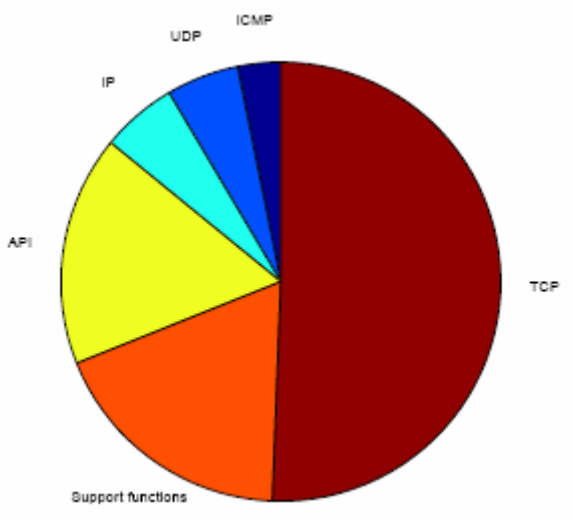

图 15 6502 平台编译的目标代码大小

表 3 列举了在 6502 处理器平台下编译后各部分目标代码的大小及所占的比例，图 14 直观的显示了各部分在 LwIP 中所占的比例关系。我们看到，TCP、API 与支撑函数的代码量几乎是 Intel x86 平台下编译后代码大小的两倍，而 IP、UDP、和 ICMP 的大小却没有多大变化。我们还看到，支撑函数部分比 API 部分的代码量要大，正好与表 2 相反。尽管如此，API 与支撑函数这两部分占代码总量的比例差别却变小了（对比表 2 和表 3，差别由 2%变成了 1%，译者注）。

TCP 模块代码量增加的主要原因是 6502 处理器本身并不支持 32 位整数。因此，每一个 32 位操作都会被编译器扩展成许多行汇编代码。而 TCP 序号就是 32 位整数，并且 TCP 模块需要执行无数的序号计算。

LwIP 中 TCP 模块的目标代码大小可以与其它 TCP/IP 协议栈实现中的 TCP 模块大小比较，像用于 FreeBSD4.1 的比较流行的 BSD TCP/IP 协议栈，还有来自于 Linux2.2.10 的 TCP/IP 协议栈。这三者同在 Intel x86 平台下使用 gcc 编译，并且编译器优化选项打开。编译后的结果是：LwIP 中 TCP 实现的目标代码大约 6600 个字节，FreeBSD4.1 下的 TCP 实现的目标代码为 27000 个字节，是 LwIP 的 4 倍多。Linu2.2.10 下 TCP 实现的目标代码更大，大约 39000 字节，几乎是 LwIP 的 6 倍。之所以会有这么大的差别，是因为相对于 LwIP，其它两个 TCP/IP 实现包括更多的 TCP 特性，像 SACK[MMFR96] （SACK 为一种 TCP 版本，它是针对一个发送窗口内有多个包丢失的情形提出的，详情请读者参阅相关资料，译者注），还包括 BSD Socket API 部分的实现。

我们在这里不再对 IP 实现的目标代码大小进行比较，原因是 FreeBSD 和 Linux 在它们各自的 IP 实现中包含更加大量的 IP 特性。例如，FreeBSD 和 Linux 即支持防火墙还支持 IP 隧道，同时，它们还支持动态路由表，这些 LwIP 都没有实现。

在 LwIP 中，API 部分大约占整个代码总量的六分之一。因为 LwIP 不包括 API 部分也可以使用，所以这部分代码可以在配置 LwIP 时省略以节约代码空间。

14 性能分析

LwIP 在内存使用率及代码效率方面的性能并没有在本论述中作过正式的测试，这一点请读者在以后的使用中注意。不过，我们做了一个简单测试，我们使用 LwIP 运行一个简单的 HTTP/1.0 服务器，在使用了不到 4K RAM 的情况下，服务器能够响应至少 10 个并发网页请求。测试中，不仅协议用到的内存，缓冲系统和应用程序用到的内存也都被计算在内，因

此，设备驱动使用的内存将增加以上数字。

15 API 参考手册

15．1 数据类型

LwIP API 使用两种数据类型：

- netbuf，描述网络缓存的数据类型
- netconn，描述网络连接的数据类型

每一种数据类型以 C 结构体的方式实现。应用程序不能直接使用结构体的内部构造，作为替代，API 提供了编辑和提取结构体内必要字段的函数。

15．1．1 网络缓存（Netbufs）

Netbufs 是用于发送和接收数据的缓冲区。从结构体内部来说，netbuf 与前面 6.1 节描述的 pbuf 相关联。Netbufs 就像 pbufs 一样即可以容纳分配的内存也可以容纳引用的内存（这里引用的内存译者理解为指针指向的内存，原文为 referenced memory，读者如果有更好的译法或者其它的理解请与译者联系探讨，本人的电邮地址及 MSN 为 marsstory99@hotmail.com）。直接分配的内存为 RAM，用于保存网络数据，而引用的内存则可能是应用程序管理的 RAM，也可能是外部 ROM。引用的内存在发送数据时有用，它不能被编辑修改，比如静态 WEB 页面或者图片。

netbuf 中的数据可以分割成不同大小的数据片，这就意味着应用程序必须准备接收这些数据片。从结构体内部来说，netbuf 有一个指针指向 netbuf 内的一个数据片，而 netbuf_next()和 netbuf_first()这两个函数就用于操作这个指针。

从网络接收到的 netbufs 还包含信息包发送方的 IP 地址和端口号，API 中存在提取这些值的函数。

15．2 缓冲区函数

15．2．1 netbuf_new()

原型声明

```
struct netbuf *netbuf_new(void)
```

描述

分配一个 netbuf 结构，该函数并不会分配实际的缓冲区空间，只创建顶层的结构。netbuf 用完后，必须使用 netbuf_delete()回收。

15．2．2 netbuf_delete()

原型声明

```
void netbuf_delete(struct netbuf*)
```

描述

回收先前通过调用 netbuf_new()函数创建的 netbuf 结构，任何通过 netbuf_alloc()函数分配给 netbuf 的缓冲区内存同样也会被回收。

例子：这个例子显示了使用 netbufs 的基本代码结构

```
int main()
{
```

```
    struct netbuf *buf;
    buf = netbuf_new();                 /* 建立一个新的netbuf */
    netbuf_alloc(buf, 100);             /* 为这个buf分配100 bytes */
    /* 使用这个netbuf作任何事情 */
    /* [...] */
    netbuf_delete(buf);                 /* 删除buf */
}
```

15．2．3 netbuf_alloc()

原型声明

```
void *netbuf_alloc(struct netbuf *buf, int size)
```

描述

为 netbuf buf 分配指定字节（bytes）大小的缓冲区内存。这个函数返回一个指针指向已分配的内存，任何先前已分配给 netbuf buf 的内存会被回收。刚分配的内存可以在以后使用 netbuf_free()函数回收。因为协议头应该要先于数据被发送，所以这个函数即为协议头也为实际的数据分配内存。

15．2．4 netbuf_free()

原型声明

```
int netbuf_free(struct netbuf *buf)
```

描述

回收与 netbuf buf 相关联的缓冲区。如果还没有为 netbuf 分配缓冲区，这个函数不做任何事情。

15．2．5 netbuf_ref()

原型声明

```
int netbuf_ref(struct netbuf *buf, void *data, int size)
```

描述

使数据指针指向的外部存储区与 netbuf buf 关联起来。外部存储区大小由 size 参数给出。任何先前已分配给 netbuf 的存储区会被回收。使用 netbuf_alloc()函数为 netbuf 分配存储区与先分配存储区——比如使用 malloc()函数——然后再使用 netbuf_ref()函数引用这块存储区相比，不同的是前者还要为协议头分配空间这样会使处理和发送缓冲区速度更快。

例子：下面这个例子显示了 netbuf_ref()函数的简单用法

```
int main()
{
    struct netbuf *buf;
    char string[] = "A string";

    buf = netbuf_new();
    netbuf_ref(buf, string, sizeof(string));    /* 引用这个字符串 */
```

```
    /* [...] */

    netbuf_delete(buf);
}
```

15．2．6 netbuf_len()

原型声明

```
int netbuf_len(struct netbuf *buf)
```

描述

返回 netbuf buf 中的数据长度，即使 netbuf 被分割为数据片断。对数据片断状的 netbuf 来说，通过调用这个函数取得的长度值并不等于 netbuf 中的第一个数据片断的长度。

15．2．7 netbuf_data()

原型声明

```
int netbuf_data(struct netbuf *buf, void **data, int *len)
```

描述

这个函数用于获取一个指向 netbuf buf 中的数据的指针，同时还取得数据块的长度。参数 data 和 len 为结果参数，参数 data 用于接收指向数据的指针值，len 指针接收数据块长度。如果 netbuf 中的数据被分割为片断，则函数给出的指针指向 netbuf 中的第一个数据片断。应用程序必须使用片断处理函数 netbuf_first()和 netbuf_next()来取得 netbuf 中的完整数据。

关于如何使用 netbuf_data()函数请参阅下面 netbuf_next()函数说明中给出的例子。

15．2．8 netbuf_next()

原型声明

```
int netbuf_next(struct netbuf *buf)
```

描述

函数修改netbuf中数据片断的指针以便指向netbuf中的下一个数据片断。返回值为0表明netbuf中还有数据片断存在，大于0表明指针现在正指向最后一个数据片断，小于0表明已经到了最后一个数据片断的后面的位置，netbuf中已经没有数据片断了。

例子：这个例子显示了如何使用 netbuf_next()函数。我们假定这是一个函数的中间部分，并且其中的 buf 就是一个 netbuf 类型的变量。

```
    /* [...] */
    do {
        char *data;
        int len;
```

```
        netbuf_data(buf, &data, &len);      /* 获取一个指针指向数据片段中的数据*/

        do_something(data, len);            /* 利用这些数据做任何事情 */
    } while(netbuf_next(buf) >= 0);
    /* [...] */
```

15．2．9 netbuf_first()

原型声明

```
int netbuf_first(struct netbuf *buf)
```

描述

复位netbuf buf中的数据片断指针，使其指向netbuf中的第一个数据片断。

15．2．10 netbuf_copy()

原型声明

```
void netbuf_copy(struct netbuf *buf, void *data, int len)
```

描述

将netbuf buf中的所有数据复制到data指针指向的存储区，即使netbuf buf中的数据被分割为片断。len参数指定要复制数据的最大值。

例子：这个例子显示了 netbuf_copy()函数的简单用法。这里，协议栈分配 200 个字节的存储区用以保存数据，即使 netbuf buf 中的数据大于 200 个字节，也只会复制 200 个字节的数据。

```
void example_function(struct netbuf *buf)
{
    char data[200];
    netbuf_copy(buf, data, 200);
    /* 利用这些数据做任何事情 */
}
```

15．2．11 netbuf_chain()

原型声明

```
void netbuf_chain(struct netbuf *head, struct netbuf *tail)
```

描述

将两个netbufs的首尾链接在一起，以使首部netbuf的最后一个数据片断成为尾部netbuf的第一个数据片断。函数被调用后，尾部netbuf会被回收，不能再使用。

15．2．12 netbuf_fromaddr()

原型声明

```
struct ip_addr *netbuf_fromaddr(struct netbuf *buf)
```

描述

返回接收到的netbuf buf的主机IP地址。如果指定的netbuf还没有从网络收到，函数返回一个未定义值。netbuf_fromport()函数用于取得远程主机的端口号。

15．2．13 netbuf_fromport()

原型声明

```
unsigned short netbuf_fromport(struct netbuf *buf)
```

描述

返回接收到的netbuf buf的主机端口号。如果指定的netbuf还没有从网络收到，函数返回一个不确定值。netbuf_fromaddr()函数用于取得远程主机的IP地址。

16 网络连接函数

16．0．14 netconn_new()

原型声明

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

描述

建立一个新的连接数据结构，根据是要建立TCP还是UDP连接来选择参数值是NETCONN_TCP还是NETCONN_UCP。调用这个函数并不会建立连接并且没有数据被发送到网络中。

16．0．15 netconn_delete()

原型声明

```
void netconn_delete(struct netconn *conn)
```

描述

删除连接数据结构conn，如果连接已经打开，调用这个函数将会关闭这个连接。

16．0．16 netconn_type()

原型声明

```
enum netconn_type netconn_type(struct netconn *conn)
```

描述

返回指定的连接conn的连接类型。返回的类型值就是前面netconn_new()函数说明中提到的NETCONN_TCP或者NETCONN_UDP。

16．0．17 netconn_peer()

原型声明

```
int netconn_peer(struct netconn *conn, struct ip_addr *addr, unsigned short *port)
```

描述

这个函数用于获取连接的远程终端的IP地址和端口号。addr和port为结果参数，它们的值由函数设置。如果指定的连接conn并没有连接任何远程主机，则获得的结果值并不确定。

16．0．18 **netconn_addr()**

原型声明

```
int netconn_addr(struct netconn *conn, struct ip_addr **addr, unsigned short *port)
```

描述

这个函数用于获取由conn指定的连接的本地IP地址和端口号。

16．0．19 **netconn_bind()**

原型声明

```
int netconn_bind(struct netconn *conn, struct ip_addr *addr, unsigned short port)
```

描述

为参数conn指定的连接绑定本地IP地址和TCP或UDP端口号。如果addr参数为NULL则本地IP地址由网络系统确定。

16．0．20 **netconn_connect()**

原型声明

```
int netconn_connect(struct netconn *conn, struct ip_addr *addr, unsigned short port)
```

描述

对UDP连接，该函数通过addr和port参数设定发送的UDP消息要到达的远程主机的IP地址和端口号。对TCP，netconn_connect()函数打开与指定远程主机的连接。

16．0．21 **netconn_listen()**

原型声明

```
int netconn_listen(struct netconn *conn)
```

描述

使参数conn指定的连接进入TCP监听（TCP LISTEN）状态。

16．0．22 **netconn_accept()**

原型声明

```
struct netconn *netconn_accept(struct netconn *conn)
```

描述

阻塞进程直至从远程主机发出的连接请求到达参数conn指定的连接。这个连接必须处于监听（LISTEN）状态，因此在调用netconn_accept()函数之前必须调用netconn_listen()函数。与远程主机的连接建立后，函数返回新连接的结构。

例子：这个例子显示了如何在 2000 端口上打开一个 TCP 服务器。

```
int main()
{
    struct netconn *conn, *newconn;
```

```
    /* 建立一个连接结构 */
    conn = netconn_new(NETCONN_TCP);

    /* 将连接绑定到一个本地任意IP地址的2000端口上 */
    netconn_bind(conn, NULL, 2000);

    /* 告诉这个连接监听进入的连接请求 */
    netconn_listen(conn);

    /* 阻塞直至得到一个进入的连接 */
    newconn = netconn_accept(conn);

    /* 处理这个连接 */
    process_connection(newconn);

    /* 删除连接 */
    netconn_delete(newconn);
    netconn_delete(conn);
}
```

16.0.23 netconn_recv()

原型声明

```
struct netbuf *netconn_recv(struct netconn *conn)
```

描述

阻塞进程，等待数据到达参数conn指定的连接。如果连接已经被远程主机关闭，则返回NULL，其它情况，函数返回一个包含着接收到的数据的netbuf。

例子：这是一个小例子，显示了 netconn_recv()函数的假定用法。我们假定在调用这个例子函数 example_function()之前连接已经建立。

```
void example_function(struct netconn *conn)
{
    struct netbuf *buf;
    /* 接收数据直到其它主机关闭连接 */
    while((buf = netconn_recv(conn)) != NULL) {
        do_something(buf);
    }
    /* 连接现在已经被其它终端关闭，因此也关闭我们自己的连接 */
    netconn_close(conn);
}
```

16.0.24 netconn_write()

原型声明

int netconn_write(struct netconn *conn, void *data, int len, unsigned int flags)

描述

这个函数只用于TCP连接。它把data指针指向的数据放在属于conn连接的输出队列。Len参数指定数据的长度，这里对数据长度没有任何限制。这个函数不需要应用程序明确的分配缓冲区（buffers），因为这由协议栈来负责。Flags参数有两种可能的状态，如下所示：

```
#define    NETCONN_NOCOPY        0x00
#define    NETCONN_COPY          0x01
```

当flags值为NETCONN_COPY时，data指针指向的数据被复制到为这些数据分配的内部缓冲区。这就允许这些数据在函数调用后可以直接修改，但是这会在执行时间和内存使用率方面降低效率。如果flags值为NETCONN_NOCOPY，数据不会被复制而是直接使用data指针来引用。这些数据在函数调用后不能被修改，因为这些数据可能会被放在当前指定连接的重发队列，并且会在里面逗留一段不确定的时间。当要发送的数据在ROM中因而数据不可变时这很有用。如果需要更多的控制数据的修改，则可以联合使用复制和不复制数据，如下面的例子所示。

例子：这个例子显示了 netconn_write()函数的基本用法。这里假定程序里的 data 变量在后面编辑修改，因此它被复制到内部缓冲区，方法是前文所讲的在调用 netconn_write()函数时将 flags 参数值设未 NETCONN_COPY。text 变量包含了一个不能被编辑修改的字符串，因此它采用指针引用的方式以代替复制。

```
int main()
{
    struct netconn *conn;
    char data[10];
    char text[] = "Static text";
    int i;

    /* 设置连接conn */
    /* [...] */
    /* 建立一些任意的数据 */
    for(i = 0; i < 10; i++)
        data[i] = i;

    netconn_write(conn, data, 10, NETCONN_COPY);
    netconn_write(conn, text, sizeof(text), NETCONN_NOCOPY);

    /* 这些数据可以被修改 */
    for(i = 0; i < 10; i++)
        data[i] = 10 - i;

    /* 关闭连接 */
    netconn_close(conn);
}
```

16．0．25 netconn_send()

原型声明

```
int netconn_send(struct netconn *conn, struct netbuf *buf)
```

描述

使用参数conn指定的UDP连接发送参数buf中的数据。netbuf中的数据不能太大，因为没有使用IP分段。数据长度不能大于发送网络接口（outgoing network interface）的最大传输单元值（MTU）。因为目前还没有获取这个值的方法，这就需要采用其它的途径来避免超过MTU值，所以规定了一个上限，就是netbuf中包含的数据不能大于1000个字节。

函数对要发送的数据大小没有进行校验，无论是非常小还是非常大，因而函数的执行结果是不确定的。

例子：这个例子显示了如何向 IP 地址为 10.0.0.1，UDP 端口号为 7000 的远程主机发送 UDP 数据。

```
int main()
{
    struct netconn *conn;
    struct netbuf *buf;
    struct ip_addr addr;
    char *data;
    char text[] = "A static text";
    int i;

    /* 建立一个新的连接 */
    conn = netconn_new(NETCONN_UDP);


    /* 设置远程主机的IP地址，执行这个操作后，addr.addr的值为0x0100000a，译者注 */
    addr.addr = htonl(0x0a000001);

    /* 连接远程主机 */
    netconn_connect(conn, &addr, 7000);

    /* 建立一个新的netbuf  */
    buf = netbuf_new();
    data = netbuf_alloc(buf, 10);

    /* 建立一些任意数据 */
    for(i = 0; i < 10; i++)
        data[i]= i;

    /* 发送任意的数据 */
    netconn_send(conn, buf);
```

```
    /* 引用这个文本给netbuf */
    netbuf_ref(buf, text, sizeof(text));

    /* 发送文本 */
    netconn_send(conn, buf);

    /* 删除conn和buf */
    netconn_delete(conn);
    netconn_delete(buf);
}
```

16．0．20 netconn_close()

原型声明

```
int netconn_close(struct netconn *conn)
```

描述

关闭参数conn指定的连接。

17 BSD Socket库

这一节提供了使用LwIP API对BSD Socket API的一个简单实现。这个实现仅仅为读者提供了一个参考，并不完善，比如它没有任何容错机制，所以它并不能用于实际编程中。

同样，这个实现也不支持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。

为了实现BSD Socket API，BSD socket实现会直接与LwIP协议栈通讯，而不是使用API。

17．1 socket 表示方法

BSD Socket API中sockets作为普通的文件描述符来表示。文件描述符是唯一的标识文件或者网络连接的整型数（integers）。在这个BSD Socket API实现中，socket由一个netconn结构在内部被表示。因为BSD sockets由一个整数来标识，所以可以将netconn变量保存在sockets[]数组中，用BSD socket标识符作为数组下标对数组进行访问。

17．2 分配socket

17．2．1 socket()函数调用

调用socket()函数分配一个socket。socket()函数的参数用于指定所需要的socket的类型。因为socket API实现仅涉及网络sockets，所以这里仅支持这一种socket类型。而且，只有UDP(SOCK_DGRAM)或者TCP(SOCK_STREAM)sockets可以被使用。

```
int socket(int domain, int type, int protocol)
{
    struct netconn *conn;
    int i;
```

```
    /* 建立一个连接 */
    switch(type) {
        case SOCK_DGRAM:
            conn = netconn_new(NETCONN_UDP);
            break;

        case SOCK_STREAM:
            conn = netconn_new(NETCONN_TCP);
            break;
    }

    /* 在sockets[]数组查找一个空位置 */
    for(i = 0; i < sizeof(sockets); i++) {
        if(sockets[i] == NULL) {
            sockets[i] = conn;
            return i;
        }
    }
    return -1;
}
```

17．3 连接设置

BSD Socket API对连接进行设置的函数与最小限度API的连接设置函数非常相似。这些函数的实现主要包括从socket的整型表示到——在最小限度API中使用的对连接进行抽象表示——的转变。

17．3．1 bind()函数调用

调用bind()函数为BSD socket绑定一个本地地址。调用时指定一个本地IP地址和端口号。这个函数与LwIP API中netconn_bind()函数非常相似。

```
int bind(int s, struct sockaddr *name, int namelen)
{
    struct netconn *conn;
    struct ip_addr *remote_addr;
    unsigned short remote_port;

    remote_addr = (struct ip_addr *)name->sin_addr;
    remote_port = name->sin_port;

    conn = sockets[s];

    netconn_bind(conn, remote_addr, remote_port);
    return 0;
}
```

17．3．2 connect()函数调用

connect()函数的实现与bind()函数的实现一样，很直接。

```
int connect(int s, struct sockaddr *name, int namelen)
{
    struct netconn *conn;
    struct ip_addr *remote_addr;
    unsigned short remote_port;

    remote_addr = (struct ip_addr *)name->sin_addr;
    remote_port = name->sin_port;

    conn = sockets[s];

    netconn_connect(conn, remote_addr, remote_port);

    return 0;
}
```

17．3．3 listen()函数调用

调用listen()函数相当于调用LwIP API函数netconn_listen()，且只能用于TCP连接。唯一的不同是，BSD Socket API允许应用程序指定等待连接队列的大小（backlog参数指定）。这对LwIP来说是不可能的，所以backlog参数被忽略。

```
int listen(int s, int backlog)
{
    netconn_listen(sockets[s]);
    return 0;
}
```

17．3．4 accept()函数调用

accept()函数等待一个连接请求到达指定的TCP socket，而这个socket先前已经通过调用listen()函数进入监听状态。对accept()函数的调用会阻塞进程直至与远程主机建立连接。这个监听变量（参数addr，译者注）是一个结果参数，它的值由accept()函数设置，这个值其实就是远程主机的地址。当新的连接已经建立，LwIP函数netconn_accept()将返回这个新连接的连接句柄（handle，句柄的意思，它能够唯一的标识某个对象，在这里指的就是指向新的连接结构netconn的指针，译者注）。将远程主机的IP地址和端口号保存到addr参数后，一个新的socket标识符被分配然后函数返回这个标识符。

```
int accept(int s, struct sockaddr *addr, int *addrlen)
{
    struct netconn *conn, *newconn;
    struct ip_addr *addr;
```

```
    unsigned short port;
    int i;

    conn = sockets[s];

    newconn = netconn_accept(conn);

    /* 获取远程主机的IP地址和端口号 */
    netconn_peer(conn, &addr, &port);
    addr->sin_addr = *addr;
    addr->sin_port = port;

    /* 分配一个新的socket标识符 */
    for(i = 0; i < sizeof(sockets); i++) {
        if(sockets[i] == NULL) {
            sockets[i] = newconn;
            return i;
        }
    }
    return -1;
}
```

17．4 发送和接收数据

17．4．1 send()函数调用

在BSD socket API中，send()函数用于UDP和TCP连接发送数据。在调用send()函数之前，必须使用connect()函数设置好数据接收器。对于UDP会话，send()函数从LwIP API中调用与之相似的netconn_send()函数，不过，因为LwIP API需要应用程序明确地分配缓冲区，所以分配和回收缓冲区就必须包含在send()函数的调用过程中，同样复制数据到分配的缓冲区也包含其中。

对TCP，就不能再调用netconn_send()函数了（如前文所述，netconn_send()函数专门用于UDP连接，译者注）。这时send()函数改为调用LwIP API中用于TCP连接的netconn_write()函数。在BSD socket API中，允许应用程序在调用send()函数之后仍然可以直接修改要发送的数据，因此，传递给netconn_write()函数的flags参数值为NETCONN_COPY，以便将数据复制到协议栈的内部缓冲区。

```
int send(int s, void *data, int size, unsigned int flags)
{
    struct netconn *conn;
    struct netbuf *buf;

    conn = sockets[s];

    switch(netconn_type(conn)) {
```

```
        case NETCONN_UDP:
            /* 建立一个缓冲区 */
            buf = netbuf_new();
            /* 让buf指向要发送的数据 */
            netbuf_ref(buf, data, size);
            /* 发送数据 */
            netconn_send(sock->conn.udp, buf);
            /* 删除buf */
            netbuf_delete(buf);
            break;

        case NETCONN_TCP:
            netconn_write(conn, data, size, NETCONN_COPY);
            break;
    }
    return size;
}
```

17.4.2 sendto()与sendmsg()函数调用

sendto()和sendmsg()函数调用与send()函数调用类似。不过，这两个函数允许应用程序在调用时指定数据接收器，并且它们只能用于UDP连接。这里使用netconn_connect()函数来设置数据包接收器，因此，如果指定的socket先前已经建立连接，那么在数据发送完之后，就必须恢复先前的连接（如以下代码所示，也就是说，我们为socket建立了指向新的IP地址和端口号的连接用于发送数据，发送完了，我们就必须恢复与先前指向的地址的连接，译者注）。这里不包括sendmsg()函数的实现。

```
int sendto(int s, void *data, int size, unsigned int flags, struct sockaddr *to, int tolen)
{
    struct netconn *conn;
    struct ip_addr *remote_addr, *addr;
    unsigned short remote_port, port;
    int ret;

    conn = sockets[s];

    /* 如果当前已经连接，则获取连接方的地址 */
    netconn_peer(conn, &addr, &port);

    remote_addr = (struct ip_addr *)to->sin_addr;
    remote_port = to->sin_port;
    netconn_connect(conn, remote_addr, remote_port);

    ret = send(s, data, size, flags);
```

```
    /* 重新设置连接的远程地址与端口号 */
    netconn_connect(conn, addr, port);
}
```

17.4.3 write()函数调用

在BSD socket API中，write()函数用于TCP或UDP连接发送数据。对TCP连接，这个函数直接映射到LwIP API函数netconn_write()；对UDP，这个函数send()函数等效。

```
int write(int s, void *data, int size)
{
    struct netconn *conn;

    conn = sockets[s];

    switch(netconn_type(conn)) {
        case NETCONN_UDP:
            send(s, data, size, 0);
            break;

        case NETCONN_TCP:
            netconn_write(conn, data, size, NETCONN_COPY);
            break;
    }
    return size;
}
```

17.4.4 recv()与read()函数调用

在BSD socket API中，recv()与read()函数用于一个已连接的socket接收数据。它们可以用于TCP和UDP连接。recv()函数包含一个flags参数，不过在我们的实现中并没有使用这个参数，所以被忽略。如果接收得数据大于提供的内存区，多余的数据会被直接丢弃。

```
int recv(int s, void *mem, int len, unsigned int flags)
{
    struct netconn *conn;
    struct netbuf *buf;
    int buflen;

    conn = sockets[s];
    buf = netconn_recv(conn);
    buflen = netbuf_len(buf);

    /* 复制收到的缓冲区内容到mem指针指向的存储区 */
    netbuf_copy(buf, mem, len);
    netbuf_delete(buf);
```

```
    /* 如果收到的数据长度大于len，则这些数据被丢弃并且
       返回len，否则返回实际接收的数据的长度 */
    if(len > buflen) {
        return buflen;
    } else {
        return len;
    }
}

int read(int s, void *mem, int len)
{
    return recv(s, mem, len, 0);
}
```

17．4．5 recvfrom()与recvmsg()函数调用

recvfrom()和recvmsg()两个函数与recv()函数类似，不同的地方是这两个函数可以获取数据发送方的IP地址和端口号。这里不包含recvmsg()函数的实现。

```
int recvfrom(int s, void *mem, int len, unsigned int flags, struct sockaddr *from, int *fromlen)
{
    struct netconn *conn;
    struct netbuf *buf;
    struct ip_addr *addr;
    unsigned short port;
    int buflen;

    conn = sockets[s];
    buf = netconn_recv(conn);
    buflen = netbuf_len(conn);

    /* 复制收到的缓冲区内容到mem指针指向的存储区 */
    netbuf_copy(buf, mem, len);

    addr = netbuf_fromaddr(buf);
    port = netbuf_fromport(buf);
    from->sin_addr = *addr;
    from->sin_port = port;
    *fromlen = sizeof(struct sockaddr);
    netbuf_delete(buf);

    /* 如果收到的数据长度大于len，则这些数据被丢弃并且
       返回len，否则返回实际接收的数据的长度 */
    if(len > buflen) {
        return buflen;
```

```
    } else {
        return len;
    }
}
```

18 例子代码

18．1 使用API

这一节介绍了使用LwIP API编写的一个简单的web服务器，代码将在后面给出。这个简单的web服务器仅实现了HTTP/1.0协议的基本功能，显示了如何使用LwIP API实现一个实际地应用。

这个应用由单一进程组成，它负责接受来自网络的连接，响应HTTP请求，以及关闭连接。在这个应用中有两个函数：main()函数负责必要的初始化及连接设置工作；process_connection()函数实现HTTP/1.0的一个小的子集。连接设置过程是一个相当简单的例子，显示了如何使用最小限度API初始化连接。使用netconn_new()函数建立一个连接后，这个连接被绑定在TCP 80端口并且进入监听（LISTEN）状态，等待连接。一旦一个远程主机连接进来，netconn_accept()函数将返回连接的netconn结构。当这个连接已经被process_connection()函数处理后，必须使用netconn_delete()函数删除这个netconn。

在process_connection()函数，调用netconn_recv()函数接收一个netbuf，然后通过netbuf_data()函数获取一个指向实际地请求数据的指针。这个指针指向netbuf中的第一个数据片断，并且我们希望它包含这个请求。这并不是一个不合实际的想法，因为我们只读取这个请求的前七个字节。如果我们想读取更多的数据，简单的方法是使用netbuf_copy()函数复制这个请求到一个连续的内存区然后在那里处理它。

这个简单的web服务器只响应HTTP GET对文件“/”的请求，并且检测到请求就会发出响应。这里，我们即需要发送针对HTML数据的HTTP头，还要发送HTML数据，所以对netconn_write()函数调用了两次。因为我们不需要修改HTTP头和HTML数据，所以我们将netconn_write()函数的flags参数值设为NETCONN_NOCOPY以避免复制。

最后，连接被关闭并且process_connection()函数返回。连接结构也会在这个调用后被删除。

下面就是这个应用的C源码。

```
/*使用最小限度API实现的一个简单的HTTP/1.0服务器*/

#include "api.h"

/* 这是实际的web页面数据。大部分的编译器会将这些数据放在ROM里 */
const static char indexdata[] =
"<html> \
<head><title>A test page</title></head> \
<body> \
This is a small test page. \
</body> \
</html>";
```

```
const static char http_html_hdr[] = "Content-type: text/html\r\n\r\n";

/* 这个函数处理进入的连接 */
static void process_connection(struct netconn *conn)
{
    struct netbuf *inbuf;
    char *rq;
    int len;

    /* 从这个连接读取数据到inbuf，我们假定在这个netbuf中包含完整的请求 */
    inbuf = netconn_recv(conn);

    /* 获取指向netbuf中第一个数据片断的指针，在这个数据片段里我们希望包含这个请求 */
    netbuf_data(inbuf, &rq, &len);

    /* 检查这个请求是不是HTTP "GET /\r\n" */
    if(rq[0] == 'G' && rq[1] == 'E' &&
       rq[2] == 'T' && rq[3] == ' ' &&
       rq[4] == '/' && rq[5] == '\r' &&
       rq[6] == '\n') {
          /* 发送头部数据 */
          netconn_write(conn, http_html_hdr, sizeof(http_html_hdr),
                  NETCONN_NOCOPY);

          /* 发送实际的web页面 */
          netconn_write(conn, indexdata, sizeof(indexdata),
                  NETCONN_NOCOPY);

          /* 关闭连接 */
          netconn_close(conn);
    }
}

/* main()函数 */
int main()
{
    struct netconn *conn, *newconn;

    /* 建立一个新的TCP连接句柄 */
    conn = netconn_new(NETCONN_TCP);

    /* 将连接绑定在任意的本地IP地址的80端口上 */
    netconn_bind(conn, NULL, 80);
```

```
    /* 连接进入监听状态 */
    netconn_listen(conn);

    /* 一直循环 */
    while(1) {
        /* 接受新的连接请求 */
        newconn = netconn_accept(conn);

        /* 处理进入的连接 */
        process_connection(newconn);

        /* 删除连接句柄 */
        netconn_delete(newconn);
    }

    return 0;
}
```

18．2 协议栈直接接口

因为这个基本的web服务器功能非常简单，它只能接受一个请求并响应这个请求，向远程主机发送一个文件，所以这比较适合使用基于协议栈接口的内部机制来实现它。同样，因为它并没有陷入大量的运算中，TCP/IP处理没有被延时。接下来的例子显示了如何实现这样的一个应用。这个例子与上一个例子非常相似。

```
/* 使用协议栈直接接口实现的一个简单的HTTP/1.0服务器 */

#include "tcp.h"

/* 实际的web页面数据 */
static char indexdata[] =
"HTTP/1.0 200 OK\r\n\
Content-type: text/html\r\n\
\r\n\
<html> \
<head><title>A test page</title></head> \
<body> \
This is a small test page. \
</body> \
</html>";

/* 这是一个回调函数，当一个TCP段到达这个连接时会被调用 */
static void http_recv(void *arg, struct tcp_pcb *pcb, struct pbuf *p)
{
```

```
    char *rq;

    /* 如果参数p值为NULL，表明远程终端已经关闭了连接。 */
    if(p != NULL) {
        /* pbuf中的payload指针指向了TCP段中的数据 */
        rq = p->payload;

        /* 检查这个请求是不是HTTP "GET /\r\n" */
        if(rq[0] == 'G' && rq[1] == 'E' &&
           rq[2] == 'T' && rq[3] == ' ' &&
           rq[4] == '/' && rq[5] == '\r' &&
           rq[6] == '\n') {

            /* 向远程主机发送web页面。最后一个参数值为0表明要发送的数
               据不必复制到内部缓冲区 */
           tcp_write(pcb, indexdata, sizeof(indexdata), 0);
        }

        /* 释放pbuf */
        pbuf_free(p);
    }
    /* 关闭连接 */
    tcp_close(pcb);
}
/* 这是一个回调函数，当一个连接已经被接受时会被调用 */
static void http_accept(void *arg, struct tcp_pcb *pcb)
{
    /* 设置数据到达时要调用的函数为http_recv() */
    tcp_recv(pcb, http_recv, NULL);
}

/* 初始化函数 */
void http_init(void)
{
    struct tcp_pcb *pcb;

    /* 建立一个新的TCP PCB */
    pcb = tcp_pcb_new();

    /* 绑定上面新建立的pcb到TCP 80端口上 */
    tcp_bind(pcb, NULL, 80);

    /* TCP进入监听状态 */
    tcp_listen(pcb);
```

```
    /* 设置新的连接到达时要调用的函数为http_accept().*/
    tcp_accept(pcb, http_accept, NULL);
}
```