# 2

# LwIP 协议栈初体验

作为正文开始，本章将为大家揭开 LwIP 的神秘面纱，其代码结构将首先呈现在读者面前，本章会分析 LwIP 的组织架构，并详细说明其中每个模块及文件的功能。作为一款功能完善的网络协议栈，LwIP 的代码相对于其他嵌入式网络协议栈来说更为复杂，对代码的阅读和学习也为读者增加了不少困难，因此本章也会从代码阅读工具入手，介绍给读者一个简单方便的代码阅读工具。最后，笔者还将从实现原理与实现细节上对 LwIP 做简要描述，让读者对 LwIP 的框架有个大体的认识，这些内容都是后续使用协议栈进行移植和实战编程关键。综上所述，本章主要将设计三个方面的内容：

- LwIP 源代码的结构组成、各文件功能简介；
- 使用源码阅读工具 SI；
- LwIP 实现原理介绍。

## 2.1 庐山真面目之 LwIP 代码结构

### 2.1.1 LwIP 简介

LwIP 是 TCP/IP 协议中一种独立、简单的实现，其设计目的在于：在保证嵌入式产品拥有完整 TCP/IP 功能的同时，又能保证协议栈对处理器资源的有限消耗，其运行一般仅需要几十 kB 的 RAM 和 40kB 左右的 ROM。目前 LwIP 存在的版本较多，读者可以到地址 http://download.savannah.gnu.org/releases/lwip/上下载合适的版本。《嵌入式网络那些事：LwIP 协议深度详解与实战演练》一书中使用的是 1.3.2 版本，而作为第二版的本书使用的是最新的版本 1.4.1，值得指出的是，LwIP 的各个高级版本之间功能并没有太大的变化，只是高版本会修复低版本许多 bug，因此程序稳定性会更好。移植过程中各版本的移植内容也无明显差别，1.4.1 版本中实现

的主要功能有：

- ARP 协议，以太网地址解析协议；
- IP 协议，包括 IPv4 和 IPv6，支持 IP 分片与重装，支持多网络接口下数据包转发；
- ICMP 协议，用于网络调试与维护；
- IGMP 协议，用于网络组管理，可以实现多播数据的接收；
- UDP 协议，用户数据报协议；
- TCP 协议，支持 TCP 拥塞控制、RTT 估计、快速恢复与重传等；
- 提供三种用户编程接口方式：Raw/Callback API、Sequential API、BSD-style Socket API；
- DNS，域名解析；
- SNMP，简单网络管理协议；
- DHCP，动态主机配置协议；
- AUTOIP，IP 地址自动配置；
- PPP，点对点协议，支持 PPPoE。

对于各个版本之间的差别，LwIP 并没有官方文档进行描述，读者只能从代码上去看其中的变化，这里推荐一个文件比较工具 Beyond Compare 给读者，它是一款文件夹/文件对比工具，可以简单比较出两个同名文件/文件夹中数据的差别，并用特殊的字体和颜色标识出来，对代码对比/代码管理很有帮助，也可以快速地定位出各个文件的差别所在。如图 2-1 所示，笔者比较了 1.3.2 版本和 1.4.1 版本内核文件夹 core 的代码修改情况，从上面可以看出，1.4.1 内核（右侧）中的每个文件相对于 1.3.2（左侧）都有更新，其中 1.4.1 还新增了两个文件 def.c 和 timers.c，后续将对这两个文件进行介绍。双击下图中的某个文件，则 Beyond Compare 会显示出两侧文件代码的具体差别所在。

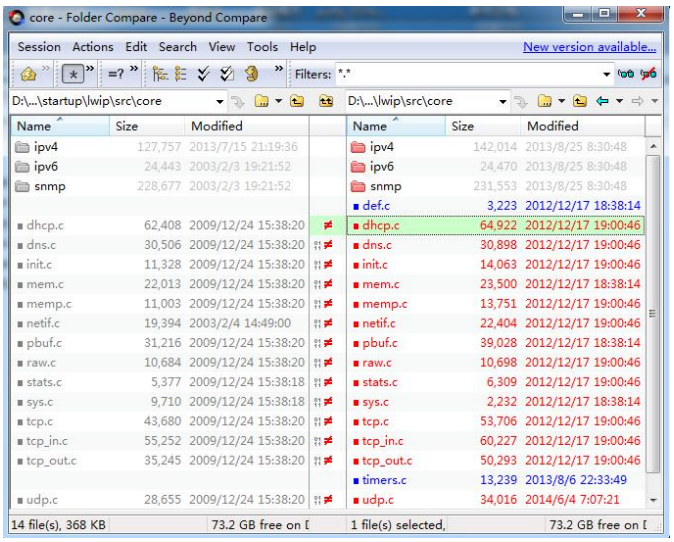


图 2-1　LwIP 版本变化比较

### 2.1.2 源代码结构

本小节详细地讲解 LwIP 的源代码结构。在网上下载源代码并解压，源代码目录下一共有三个文件夹：doc、src 和 test。其中 doc 文件夹下包含了几个与协议栈使用相关的文本文档，比较重要的文件有两个：rawapi.txt 告诉读者怎样使用协议栈的 Raw/Callback API 进行编程，因为 Raw/Callback API 是协议栈提供的三种编程接口中最复杂的一种，它通过直接与协议栈内核函数交互以实现编程，所以整个过程比较复杂，LwIP 用一个专门的文档来告诉使用者应该怎样做；sys_arch.txt 在移植的时候会被使用到，包含了移植说明，规定了移植者需要实现的函数、宏定义等。test 文件夹下是 LwIP 提供的一些协议栈内核测试程序，我们这里暂时不会使用到。最后的重头戏是 src 文件夹，它包含了协议栈内核的所有源代码，进入该文件夹，可以看到 LwIP 代码的组织结构，如图 2-2 所示。

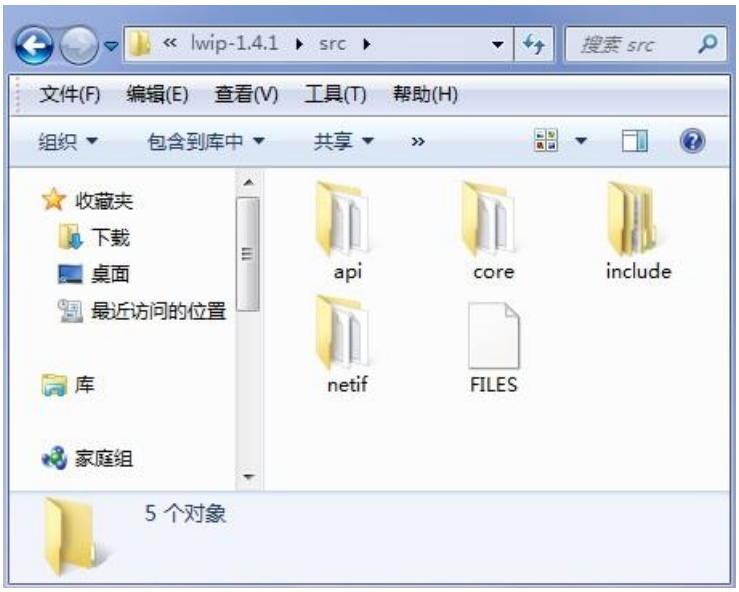


图 2-2 源代码 src 文件夹结构

从上图可以看出，内核源代码由四部分组成，其中 api 文件夹包含了 LwIP 的 Sequential API 和 Socket API 两类接口函数及实现相关的源代码，要使用这两种类型的 API，需要底层操作系统的支持；core 是 LwIP 内核源代码，内核源代码可以单独运行，且不需要操作系统的支持；include 主要包含了整个协议栈使用的头文件；netif 主要包含了与底层网络接口相关的文件。

现在依次看看各个文件夹中的内容，笔者尽量对各个文件功能进行描述，让读者对协议栈有宏观的把握，先从最底层看起，netif 文件夹内容如图 2-3 所示。

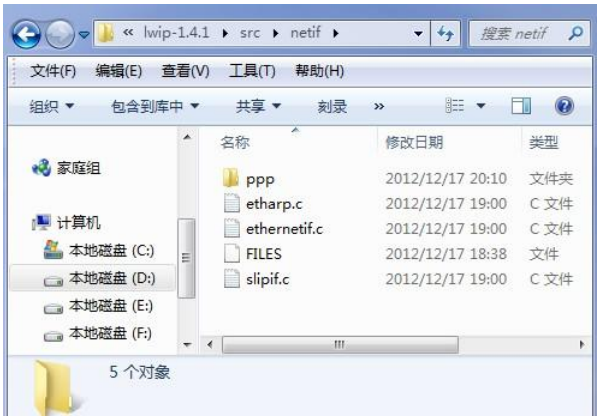


图 2-3　源代码 netif 文件夹结构

上图中的 ppp 文件夹包含了 PPP 协议实现的源代码。PPP 协议即点对点协议，它提供了一种在点对点线路上传输多协议数据包的标准格式，PPP 协议为链路的建立、控制与认证提供了标准。起初 PPP 主要用来替代 SLIP 这种简单的串行链路数据传输协议，但是由于其完整的认证机制，后来在以太网上也引入了 PPP 机制，即 PPPoE，它已成为近年来小区宽带拨号上网的主要方式。使用 PPPoE，为用户的上网计费、配置、接入等提供了方便。LwIP 提供了对 PPPoE 的支持，在 ppp 文件夹下的 PPPoE.c 文件中有相关的函数实现。

etharp.c 文件包含了 ARP 协议实现的相关函数，ARP 协议是以太网通信中的重要部分，主要用来实现主机以太网物理地址到 IP 地址的映射。这点是非常必要的，以太网中底层数据包的发送是基于网卡物理地址的，而不是主机的 IP 地址。通过 ARP 协议，主机可以发送请求，得到邻居节点的 IP 地址与物理地址等信息，为以太网数据包交互提供保证。

ethernetif.c 文件包含了与以太网网卡密切相关的初始化、发送、接收等函数的实现。注意这个文件夹中的函数并不能使用，它们都是一个框架性的结构，移植者需要根据自己使用的网卡特性来完成这些函数。

slipif.c 文件和 ethernetif.c 很相似，SLIP 即串行链路 IP，它提供了一种在串行链路上传送 IP 数据包的函数定义，移植者需要根据自己使用的串行线路特性（如串口）来实现这些函数。SLIP 协议比较简单，它只是定义了一系列的字符，以实现对链路上的 IP 数据包封装和发送，除此之外，它不提供任何寻址、错误检测、包类型识别机制，因此相关驱动程序的实现也比较简单。

接下来 core 文件夹是整个协议栈的重点，它包含了 IP、ICMP、IGMP、TCP、UDP 等核心协议以及建立在它们基础上的 DNS、DHCP、SNMP 等上层应用协议。该文件夹结构如图 2-4 所示。

其中 ipv4 文件夹包括了 IPv4 标准中与 IP 层数据包处理相关的所有代码；ipv6 文件夹包括了 IPv6 标准中和 IP 层数据包处理相关的所有代码，该文件夹内的文件结构与 ipv4 文件夹中的相似。IP 协议为数据包的点到点传输提供了支持，它是整个 TCP/IP 协议簇中最重要的协议，因为 ICMP、IGMP、TCP、UDP 等都要使用 IP 层提供的服务。

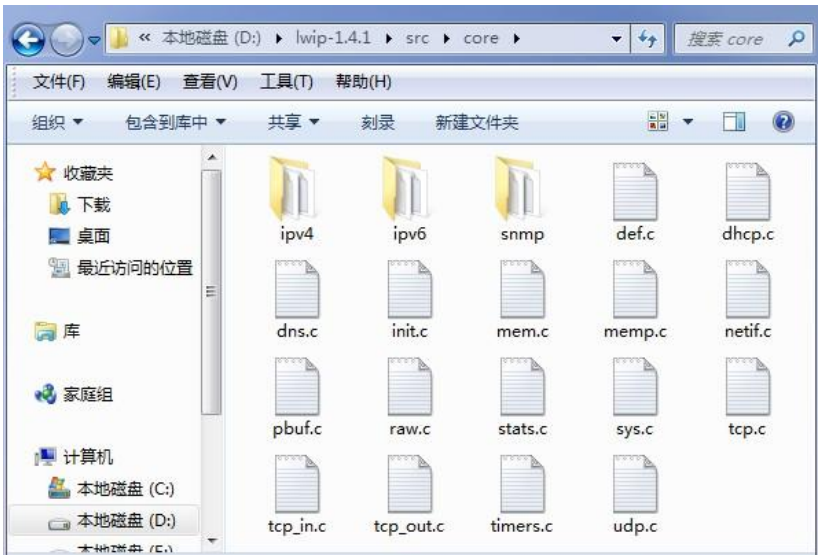


图 2-4　源代码 core 文件夹结构

snmp 文件夹包含了和 SNMP 协议实现相关的所有代码，SNMP 称为简单网络管理协议，它是一个基于 TCP/IP 协议栈的上层应用程序，在 LwIP 中，它使用 UDP 来实现。SNMP 为互联网上设备的管理提供了框架，通过这个框架，可以使管理器实现对不同厂商生产的网络设备（如路由器）的监控、管理。

def.c 文件包含了 IP 层使用到的一些功能函数的定义，如 IP 地址的转换、网络字节序与主机字节序转换等。

接下来，dhcp.c 文件实现了 DHCP 客户端的所有代码，DHCP 称为动态主机配置协议，DHCP 可以使计算机使用者不必为主机的 IP 地址的分配问题而烦恼。DHCP 也是一个上层应用程序，通常 DHCP 客户端通过使用 UDP 提供的功能来实现与 DHCP 服务器的通信，从 DHCP 服务器处获得一个有效的 IP 地址。

dns.c 文件实现了 DNS 客户端的所有代码，DNS 称为域名系统，通常用户要访问某个外部的主机时，可能只知道该主机的名字，而不知道该主机的 IP 地址。常理也是这样，用户可以简单地记得某个主机的名字，如 www.baidu.com，却很难记得这个主机的 IP 地址。通过使用 DNS，可以解决这个问题，通过访问 DNS 服务器，我们可以得到与主机名对应的 IP 地址，这为用户的使用带来了方便。值得指出的是，DNS 也是一个上层应用程序，它通常基于 UDP 来传输数据。

init.c 文件比较简单，主要包含了一个与 LwIP 协议栈初始化密切相关的函数，以及一些协议栈配置信息的检查与输出。

mem.c 包括了协议栈内存堆管理函数的实现；memp.c 包含了协议栈内存池管理函数的实现。LwIP 使用了上述这两种特有的内存管理方式来实现对协议栈内存的有效管理。

netif.c 文件包含了协议栈网络接口管理的相关函数，协议栈支持多个网络接口，例如前面所说的以太网接口、SLIP 接口等，协议栈内部对每个接口用一个对应的 netif 数据结构进行描述，并通过使用 netif.c 中的函数进行统一管理。同时，netif.c 还包含了环回接口管理和数据处理的相关函数，使用环回接口可以实现同一主机上两个应用程序之间的数据交换。

pbuf.c 文件包含了协议栈内核使用的数据包管理函数，数据包 pbuf 管理的实现是整个协议栈中很有特色的地方，采用特殊的数据包 pbuf 结构，可以避免数据在各个层次之间递交时的拷贝，这既提高了数据递交效率，也节省了内存空间。



raw.c 文件为应用层提供了一种直接和 IP 数据包交互的方式，这类似于 Socket 编程中原始套接字的概念，它同 TCP、UDP 处于同一等级，享受 IP 层提供的服务。使用原始套接字，可以直接读取 IP 层接收到的数据包，例如 ICMP 包、UDP 包等，也可以自行构造 ping 包等，为用户的程序编写提供了很大的方便。

stats.c 文件中包含了协议栈内部数据统计与显示的相关函数，如内存使用状况、邮箱、信号量等信息。

tcp.c 文件中包含了对 TCP 控制块操作的函数，也包括了 TCP 定时处理函数；tcp_in.c 包含了 TCP 协议中数据接收、处理相关的函数，最重要的 TCP 状态机函数也在这个文件中；tcp_out.c 包含了 TCP 中数据发送相关的函数，例如数据包发送函数、超时重传函数等。

udp.c 包含了实现 UDP 协议的相关函数，包括 UDP 控制块管理、UDP 数据包发送函数、UDP 数据包接收函数等。

sys.c 文件实现了一个简单的函数 sys_msleep，它借助操作系统模拟层的信号量机制完成睡眠一定时间的功能，该函数主要在 PPP 中使用。前面曾提到过，若需要使用协议栈的 Sequential API 和 Socket API，则必须使用底层操作系统提供的邮箱与信号量机制，这时内核要求移植者提供一个称为 sys_arch.c 的操作系统模拟层文件，该文件主要完成对操作系统中邮箱与信号量函数的封装。值得指出的是，只有在提供文件 sys_arch.c 的基础上，文件 sys.c 才有效，换句话说，在无操作系统环境下运行 LwIP 时，sys.c 文件都不会被编译。

timers.c 统一完成了对内核各个协议定时事件处理函数的封装，同时对各个注册的定时事件进行处理。在有无操作系统模拟层的环境下，timers.c 采用不同的方法来实现定时：在无操作系统模拟层时，timers.c 使用系统时钟函数 sys_now（不同的平台的都需要移植该函数）来获得当前系统时间，从而可以判断出各个事件是否超时；在有操作系统模拟层时，timers.c 实现了对操作系统模拟层邮箱等待函数的再次封装，得到一个具有协议栈特色的邮箱操作函数。所谓特色，就是在邮箱等待函数中加入一种机制，在邮箱上等待消息的同时，可以同时实现协议栈中各个定时事件的正确处理。

下面来看看 core 文件夹下 ipv4 文件夹中的内容，如图 2-5 所示。文件 autoip.c 包含了 IP 地址自动配置相关的函数，若主机从 DHCP 服务器处获取 IP 地址失败，则此时主机可以选择启动 AUTOIP 功能来配置自身的 IP 地址。AUTOIP 将主机配置为 169.254.0.0/16 中的某个地址值，并提供一套完整的机制来避免 IP 地址冲突。

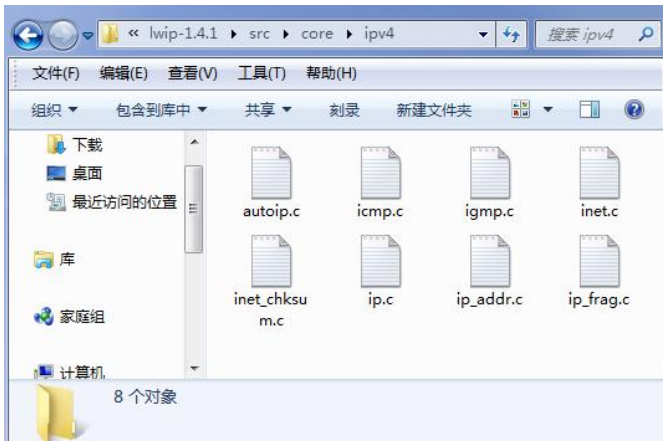


图 2-5　源代码 ipv4 文件夹结构

icmp.c 文件包含了 ICMP 协议实现的相关函数，ICMP 协议为 IP 数据包传递过程中的差错报告、差错纠正以及目的地址可达性测试提供了支持，常见的 Ping 命令就属于 ICMP 应用的一种。

igmp.c 文件包含了网络组管理协议 IGMP 的实现，IGMP 为网络中的多播数据传输提供了支持，主机加入某个多播组后，可以接收到该组的 UDP 多播数据。

inet.c 文件目前为空，在较早版本中，其完成了 IP 地址的转换、网络字节序与主机字节序转换等常用函数的定义；inet_chksum.c 文件包含了同 IP 数据包校验相关的函数；ip.c 包含了 IPv4 协议实现的相关函数，如数据包的接收、递交、发送等；ip_addr.c 实现了几个比较简单的 IP 地址处理函数，如判断一个 IP 地址是否为广播地址的函数，以及 32 位 IP 地址与点分十进制地址间的转换函数等；ip_frag.c 文件提供了 IP 层数据包分片与重组相关的函数。

回到图 2-2 中，看看其下 api 文件夹中的内容，如图 2-6 所示。

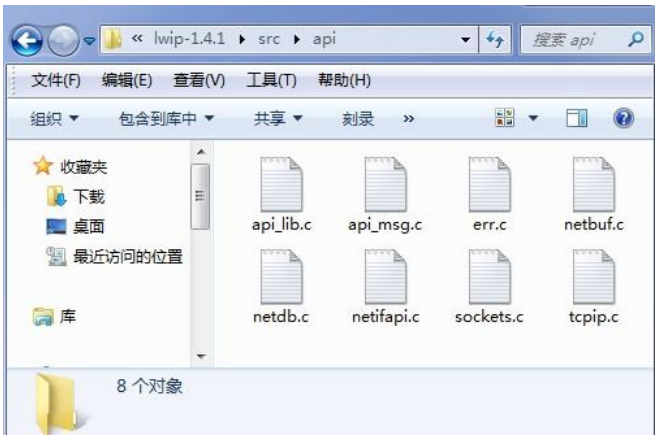


图 2-6　源代码 api 文件夹结构

为了方便用户的编程，LwIP 为用户提供了两种简单的上层 API 接口，一种是协议栈的 Sequential API，另一种是 Socket API。这两种 API 实现的原理都是通过引进邮箱和信号量等通信与同步机制，来实现对内核 core 中 Raw/Callback API 函数的封装和调用。因此，从用户编程的角度来看，这两种类型的 API 确实为程序的开发提供了很大的方便，但事实上，它们的使用也不可避免地会降低代码的执行效率和整个系统的稳定性。还得指出的是，要使用这两种 API，必须基于底层操作系统提供的邮箱和信号量机制，也就是要在板子上移植好操作系统。

文件 api_lib.c 和 api_msg.c 包含了所有 Sequential API 函数的实现，其中前者主要包含预留给用户的编程接口，后者主要包含 API 消息的封装与处理函数；netbuf.c 文件包含了上层数据包管理函数的实现；netdb.c 文件包含与主机名字转换相关的函数，主要在 socket 中被使用到；netifapi.c 文件包含了上层网络接口管理函数的实现；sockets.c 文件包含了 Socket API 函数的实现；tcpip.c 文件中提供了上层 API 与协议栈内核交互的函数，它是整个上层 API 功能得以实现的一个枢纽，其实现的功能可以简单理解为：从 API 函数处接收消息，然后将消息递交给内核函数，内核函数根据消息做出相应的处理。

最后，图 2-2 中还有一个 include 文件夹，它包含了与上述所有 C 文件对应的头文件声明，其下包含了五个子文件夹，如图 2-7 所示。读者可以自行了解下各文件夹中的内容。其中最重要的是 lwip 文件夹下的 opt.h 文件，它包含了所有 LwIP 内核参数的默认配置值；还有 lwip 文件夹下的 init.h 文件，它包含了与当前 LwIP 源代码信息相关的宏定义，如协议版本号、是否为官方版等。

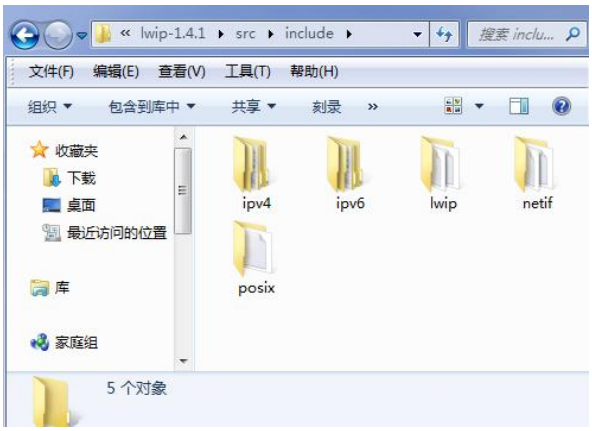


图 2-7　源代码 include 文件夹结构

## 2.2　怎样用 SI 阅读源代码

整个协议栈的学习过程与源代码的阅读是密切相关的，通过阅读代码，读者可以加深对协议、原理实现细节的认识，为应用程序代码的编写打下基础。LwIP 如此大的代码量，其间

的相互关系也十分复杂，这里笔者建议使用 Source Insight 3.5（SI）来浏览、编辑协议栈源代码。SI 提供了对包括 C 在内的多种程序开发语言的支持，使用 SI 可以方便查看函数、变量的上下文信息，也可以快速查找定位符号信息。

- 安装 Source Insight 3.5

从网上下载一个 SI 安装包，注意 SI 是个收费软件，只能使用其 30 天的适用版本。第一次开启软件时，会提示注册用户名及公司信息，随便填写即可。双击 SI 安装工具可执行文件，安装路径及所有配置选择默认即可，单击“下一步”按钮直至安装完成。

- 使用 Source Insight 3.5

在用 SI 创建新工程浏览代码前，先在 lwip-1.4.1 目录下创建一个名为 SI 的文件夹，SI 在创建一个新工程时，会创建数个与工程相关的文件，将这些文件统一放在 SI 文件夹下，方便管理。

打开 SI，在“项目（Project）”选项卡下，选择“新项目（New Project）”开始创建新工程。在弹出的编辑框中选择将要保存工程的文件夹，这里选择刚创建的文件夹 SI，并将工程名字取为 MY_LwIP_SI，如图 2-8 所示。

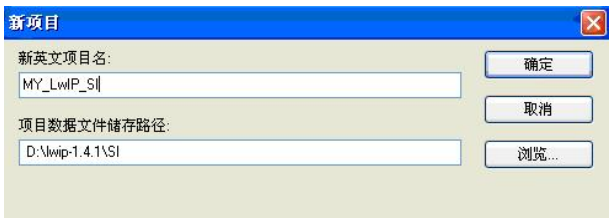


图 2-8　创建新工程

单击“确定”按钮后，进入图 2-9 所示的对话框，将项目源目录选择为 D:\lwip-1.4.1\src。

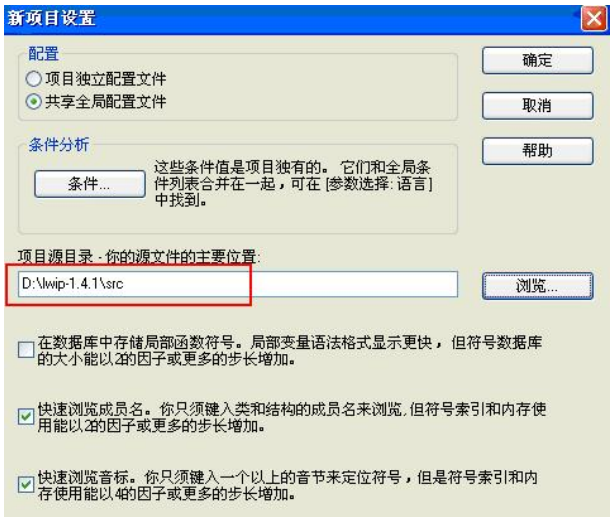


图 2-9　设置项目源目录

单击“确定”按钮后，进入图 2-10 所示的添加源文件界面，单击“全加入（Add All）按钮”，并在弹出的对话框中勾选“递归加入下级子目录（Recursively add lower sub-directories）”复选框，再单击“确定”按钮，SI 便开始为我们查找并添加源文件了。

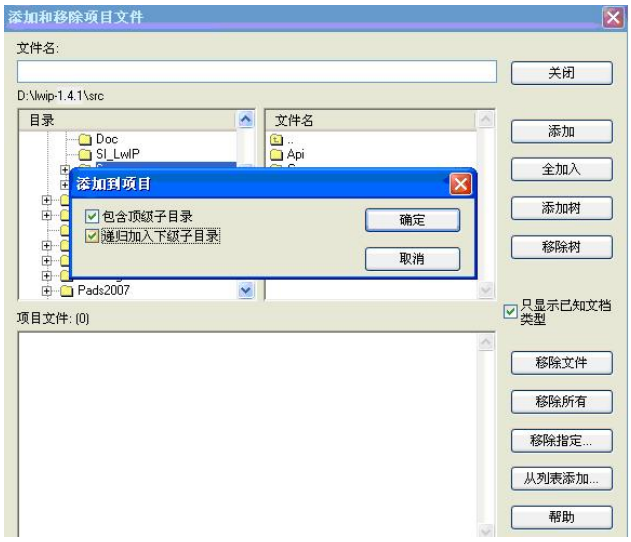


图 2-10　添加源文件

查找工程结束后，SI 会列出已经添加的文件总数及文件列表，如图 2-11 下方所示，在图中单击“关闭”按钮，完成文件的添加过程。

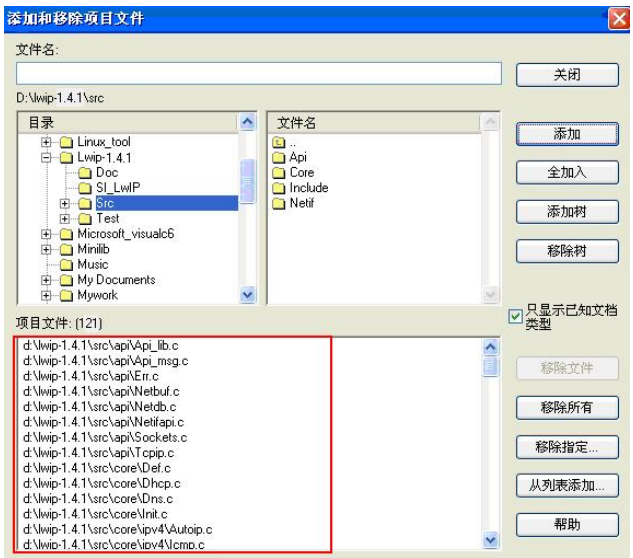


图 2-11　添加源文件完成

添加成功的文件会被列在工程的文件列表栏，如图 2-12 所示。接下来，刚被添加进入工程的文件需要做一个同步工作，通过同步工作，SI 可以扫描出工程中函数、变量的上下文关系，只有建立了正确的上下文环境，在浏览工程函数的时候才能快速地定位某个函数或变量的使用地点。如图 2-12 所示，在“项目（Project）”菜单下选择“同步文件（Synchronize Files）”选项，并按如图 2-13 所示在弹出的对话框上勾选相应复选框，单击“确定”按钮后，便开始了同步过程。

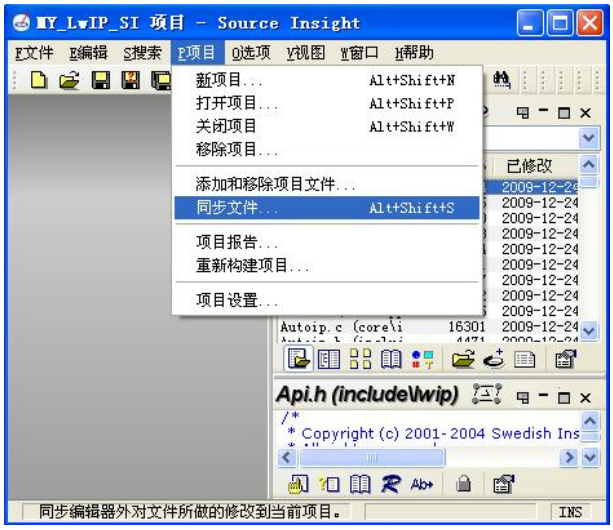


图 2-12　文件列表与文件同步

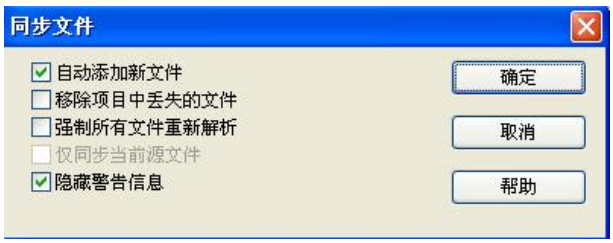


图 2-13　文件同步选项

当同步过程完成后，函数与变量的上下文关系就建立起来了，如图 2-14 所示，当我们将光标设置在代码区的 tcp_input 函数后面时，在右下侧的 Context 区域就显示出该函数的具体实现，SI 的这项功能使其在代码阅读上很方便。同时，可以在文件名栏（红线标记）上面的编辑框内输入文件名查找需要的某个文件；在“搜索”栏也可以通过多种方式查找文件信息，如在当前文件中查找某字符串、在整个工程中查找某字符串、在已有的搜索结果中查找某字符串等。

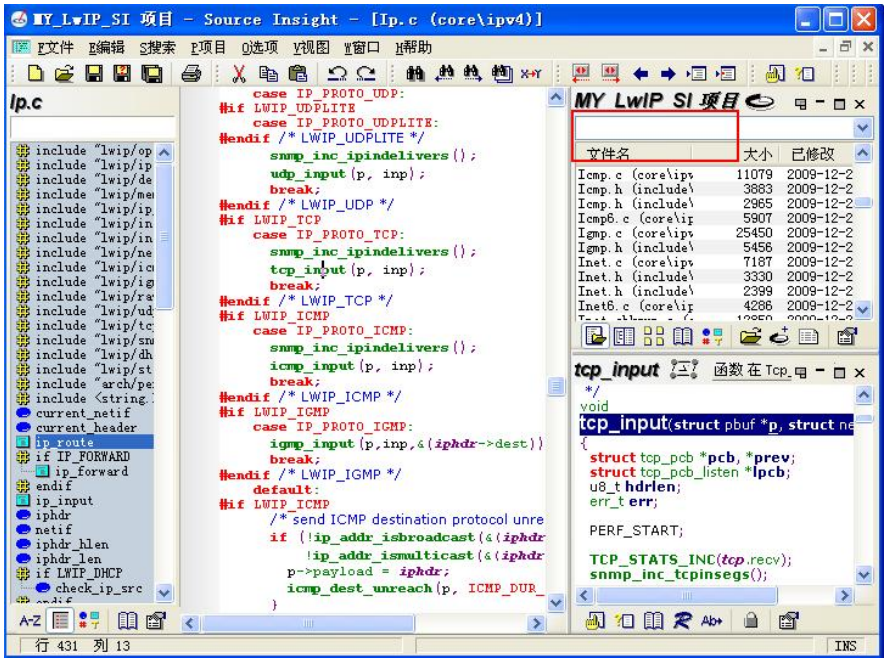


图 2-14　代码浏览

SI 出色的代码查找定位能力使得其目前在软件领域被广泛应用于代码的编辑、阅读与检视，作为一名软件开发人员，掌握其使用方法具有诸多好处。下面读者再介绍几个常用的 SI 快捷键，这些是在代码阅读可以经常使用的：

（1）Alt+<与 Alt+>，这两个快捷键分别用来定位到上一个光标处和下一个光标处，在代码阅读过程中可以快速实现跳转。

（2）Shift+F8，这是最有用的一组功能键了，它能使当前光标所在处的词条高亮，这在代码阅读时特别是观测某个变量出现的地方非常有用，如图 2-15 所示。

```
/* Move the payload pointer in the pbuf so that it points to the
  TCP data instead of the TCP header. */
hdrlen = TCPH_HDRLEN(tcphdr);
if(pbuf_header(p, -(hdrlen * 4))){
  /* drop short packets */
  LWIP_DEBUGF(TCP_INPUT_DEBUG, ("tcp_input: short packet\n"));
  TCP_STATS_INC(tcp.lenerr);
  goto dropped;
}

/* Convert fields in TCP header to host byte order. */
tcphdr->src = ntohs(tcphdr->src);
tcphdr->dest = ntohs(tcphdr->dest);
seqno = tcphdr->seqno = ntohl(tcphdr->seqno);
ackno = tcphdr->ackno = ntohl(tcphdr->ackno);
tcphdr->wnd = ntohs(tcphdr->wnd);

flags = TCPH_FLAGS(tcphdr);
tcplen = p->tot_len + ((flags & (TCP_FIN | TCP_SYN)) ? 1 : 0);
```

图 2-15　高亮词条

（3）Ctrl+/，用于查找光标所在处词条在整个工程中出现的地方，并将所有结果逐条显示出来，如图 2-16 所示，这对于定位每个函数在哪些地方被调用很方便，单击搜索结果中的某条，就可以跳转到相应的代码处。

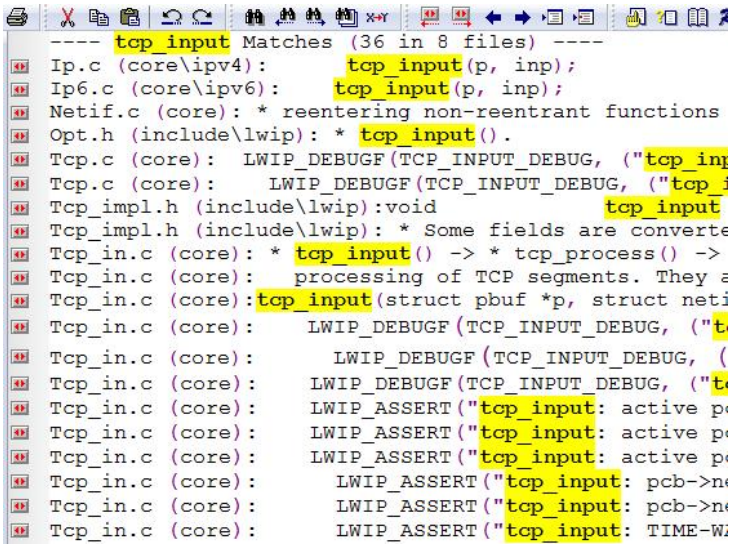


图 2-16 搜索词条

## 2.3 千里之行从 LwIP 框架做起

### 2.3.1 协议栈分层思想

TCP/IP 协议完整地包含了一系列构成互联网基础的网络协议。TCP/IP 协议的开发出现在 OSI 标准之前，因此它并没有按照 OSI 模型中描述的各种结构来实现，而是有着自己的协议层次划分特点。TCP/IP 协议模型可以分为四个层次，从下到上依次为：网络接口层（Network Interface Layer）、网络层（Internet Layer）、传输层（Transport Layer）和应用层（Application Layer），如图 2-17 所示。在所有层次结构中，每一层建立在低一层提供的服务上，低一层为高一层提供服务，最终完成数据在两台联网主机间的传递。在图 2-17 中还指出了各层中涉及的相关协议，这些协议也是 LwIP 中已经实现或支持的协议。

1. 网络接口层

网络接口层是 TCP/IP 协议模型的最底层，主要负责网络上数据帧的发送和接收，数据帧是底层网络传输的基本单元。网络接口有不同的实现方式，比如可以通过有线或无线的方式发送数据帧，不同的实现方式意味着不同的帧结构、发送速率等。网络接口层一方面将上层（网

络层）的数据组装成自己特定的数据帧并发送，另一方面接收网络中发给自己的数据帧，并解析出帧中的数据后递交上层（网络层）。

2. 网络层

网络层负责在主机之间的通信中选择数据报的传输路径，即路由。当网络层接收到来自于上层（传输层）的数据分组后，它会把分组封装在 IP 数据报中，填入数据报的首部，使用路由算法来确定是直接交付数据报，还是把它传递给路由器，然后把数据报交给适当的网络接口进行传输。

网络层还要负责处理传入的数据报，并检验其数据有效性，然后判断该数据报是否是给本机的，如果不是，则使用路由算法将数据报出去转发；如果是，网络层需要除去数据报中的首部得到数据分组，然后将数据分组递交给上层（传输层）。

3. 传输层

传输层主要提供应用程序之间的通信服务，这种通信又称为端到端通信。传输层协议把上层（应用层）要传输的数据流划分为分组，把每个分组连同目的地址交给网络层去发送。传输层要系统地管理两端数据的准确交互，要提供可靠的传输服务，以确保数据到达无差错、无乱序。为了达到这个目的，传输层协议可以采用协商、确认、重发等机制。

4. 应用层

应用层是分层模型的最高层，它最简单的解释就是利用传输层提供了数据传输功能发送自己的数据到对方。传输层协议类型有多种，不同的类型意味着不同的传输速度和可靠性，而往往这二者是不可兼得的。所以每个应用程序选择最合适的传输服务类型，以使双方之间的数据传输达到最佳效果。

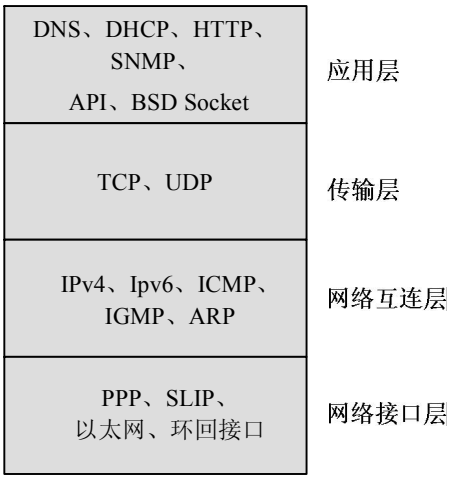


图 2-17　TCP/IP 分层结构

在标准 TCP/IP 协议结构中，各个层都被描述为一个独立的模块形式，每一层负责完成一

个独立的通信问题。例如，网络接口层完成 IP 数据包在物理线路传输时的封装、传递问题；网络层完成数据包选路和路由，负责将数据包从源主机发送到目标主机；传输层完成数据在两台主机间的端到端传递，IP 地址只能唯一地标识出一台主机，而端口号却可以标识出一台主机上的不同进程，传输层通过识别端口号来向不同的用户应用程序递交数据。TCP/IP 的这种分层方式为协议栈的设计提供了一种很好的解决方案，因为每个层次的协议相互独立，它们都可以被单独实现，只要保证它们之间的接口不变就可以了。

如果按照上述这种严格的分层模式来实现 TCP/IP 协议，会使数据包在各层间的递交变得非常慢，因为它涉及到一系列的内存拷贝问题，因此，系统总体性能也会受到影响。为了避免这种现象的发生，LwIP 内部并没有采用完整的分层结构，它会假设各层间的部分数据结构和实现原理在其他层是可见的。这样，在数据包递交过程中，各层协议可以直接对数据包中属于其他层次协议的字段进行操作，这将使整个协议栈对数据包的操作更加灵活。

LwIP 实现时，参考了 TCP/IP 协议的分层思想，即每层都在一个单独的模块中实现，并为其他层次模块提供一些输入/输出接口函数。LwIP 包含了许多的模块，每个模块都在一个单独的文件中实现，除了实现基本 TCP/IP 功能的（IP、ICMP、UDP、TCP）模块外，还有许多支持这些基本模块实现的附加模块，例如操作系统模拟层，数据包和内存管理机制，网络结构管理层，数据校验和计算模块及 API 模块。即使这样，LwIP 并没有遵循严格的分层机制，正如前面讨论的那样，为了节省时间和空间上的开销，各个层次之间可能存在交叉存取的现象。例如在 TCP 层计算 TCP 报文段的校验和 TCP 在重装无序报文段时，TCP 必须知道该报文段的源 IP 地址和目的 IP 地址。为了得到这些信息，TCP 层不是调用 IP 层的相关接口函数，而是直接访问数据包中的 IP 数据报首部，因为 TCP 已经知道该首部中各个字段的意义，所以它可以直接从中取出想要的信息。

另一方面，在许多其他 TCP/IP 协议的实现中，即使内核各层存在着一定的交错现象，但在用户应用程序和协议栈内核之间也会保持着很明显的分层结构。在操作系统中，TCP/IP 协议往往被设计为内核代码的一部分，用户可以使用的只是协议为用户提供的几个 API 函数，或者操作系统对网络通信函数进行了完全的封装，用户可以像操作简单文件那样来处理网络连接中的数据（例如常见的 BSD socket）。也正因如此，应用程序并不知道协议栈内部使用的数据包结构与机制，它也就不能充分利用协议栈中数据包的组织结构来避免数据的拷贝。在数据包发送时，用户数据必须全部从用户进程区域拷贝到协议栈内部缓冲区。

在小型嵌入式设备使用的操作系统中，操作系统代码和用户程序代码之间通常都没有出现明显分层的现象，这种方式允许用户程序和操作系统内核之间使用更多宽松的方式进行通信，例如共享内存等。LwIP 内核也基于这种机制来实现，它假设应用程序了解协议栈内部的数据处理机制。用户程序可以直接访问协议栈内部的数据包，也可以和协议栈共同使用一定的内存区域，对这些区域进行直接读取或改写，这就避免了数据在两者之间拷贝时的时间开销和内存开销。

应用程序和内核之间交叉访问，这种做法最直接的好处是网络程序的性能能够有很大的

提高，特别是在小型嵌入式这种处理器资源不是很丰富的领域。但是，这种做法也会导致一个明显的问题：它要求应用程序开发者对内核有足够的了解，应用程序的稳定性和可用性建立在开发者对内核的熟悉程度上。这一点对于广大高校学子来说，并不一定是坏事，因为这会迫使他们去学习内核更深层次的东西，对他们的成长有莫大的好处；但是对于广大工程开发人员来说，这一点可能是致命的，因为这可能导致项目开发周期大大延长，这正是目前 LwIP 在工程领域面临的最大问题，开发者更愿意花更多的时间去维护和调试自己的应用程序，而不是去研究内核代码。这也是笔者重新编写此书的最大出发点，希望通过此书，能够为读者提供更多学习 LwIP 的捷径。



### 2.3.2 进程模型

协议的进程模型可以理解为协议实现时被划分在几个进程中，即实现具体协议栈时需要几个进程。

单进程模型，即让协议栈中的每个模块都独立地成为一个进程，在这种模式下，各个模块之间有个明显的界限区分，各个模块间的接口定义也非常清晰。这种结构有很多优势：首先，协议栈各个模块能够随着程序的运行而自由添加，只有被使用到的模块才被添加，这使得协议的代码组织更加灵活；此外，这种模块化的结构也使协议栈的代码编写和调试显得更加轻松。但正如前面所述，严格的分层结构并不是实现协议栈的最好方式，最重要的，当数据包在各个层次间递交时，就会涉及到各个进程频繁的问题。例如，对于一个底层网卡收到的 TCP 报文段，这意味着必须进行最少三次的进程切换：底层数据包接收进程调用网卡驱动接收数据包；然后进程切换，进入 IP 层进程处理数据包并将数据报递交给 TCP 层；此后，系统进入 TCP 进程处理收到的报文；最后还需进入用户进程处理 TCP 层递交的数据，在许多操作系统中，进程的切换是一件开销很大的事情，所以单进程模型会使得协议栈的效率降低。

另一种实现方式是使协议栈驻留在操作系统中，成为操作系统的一部分，这样，用户进程与协议栈内核之间进行交互就是通过操作系统提供的系统函数来实现的。这种情况下，协议栈内部的各层之间不会有明显的分层界限，可以在各层之间采用一定的交叉编程技术来提高协议栈的执行效率。

LwIP 采用了这样一种进程模型，即协议栈内核同操作系统内核互相隔离，而同时整个协议栈作为操作系统中的一个单独进程而存在。用户应用程序可以驻留在协议栈内核的进程中，也可以实现为一个单独的进程。当采用第一种方式时，用户程序与协议栈之间的通信是通过回调函数（即前面所说的 Raw/Callback API）来实现的；当使用第二种方式时，需要使用协议栈中的操作系统模拟层提供的信号量与邮箱机制，实现用户进程和协议栈内核的数据交互，这时可以用其他两种 API 进行编程，即 Sequential API 和 Socket API。

将 LwIP 实现为系统中一个单独的进程，而不是作为系统内核的一部分，这样做既有优点也有缺点。最大的优点在于协议栈可以在任何操作系统上进行移植，缺点在于协议栈进程的运

行需要操作系统的调度，这使得有时候协议栈响应的实时性会受操作系统调度的影响。因此，在嵌入式操作系统中使用 LwIP 时，最好将 LwIP 进程设置为系统中具有最高优先级的进程，以提高协议栈的实时性。

### 2.3.3 协议栈编程接口

上面说到，LwIP 为用户应用程序的编写提供了三种编程接口，用户可以从自己的处理器特点及网络应用程序需求等多方面考虑，选择最佳的编程方式或组合。这里将简单介绍三种 API 各自的特点及其的使用场景。



1. Raw/Callback API

基于 Raw/Callback API 接口来实现 LwIP 网络编程时，协议栈与用户程序之间通信是通过回调函数来实现的，此时用户程序和协议栈内核运行于同一进程中。Raw/Callback API 是一种最直接的利用协议栈的方法，应用程序通过函数注册的方式与内核产生联系，在内核相关事件发生时，用户函数通过回调的方式被执行。在 Raw/Callback 方式下的应用程序与协议栈内核处于用一个进程中，用户程序通过回调执行的方式取得协议栈中的数据，基于回调机制的应用程序会使得整个代码的灵活性加大。在后面的实验过程中读者可以看到，Raw/Callback 可以方便地构造出一个服务器对多个客户端的 TCP 并发连接，而采用 Sequential API 时只能通过多线程的方式来实现并发连接。

另一方面，我们从编程模式上来看看 Raw/Callback API 的弊端，由于用户应用程序和协议栈内核处于同一进程中，用户程序通过回调的方式执行，这样，用户程序和协议栈内核出现了相互制约的关系，因为用户程序执行的时候，内核一直处于等待状态，等待用户函数返回一个处理结果再继续执行。如果用户程序的计算量很大，执行时间很长，则协议栈代码就一直得不到执行，协议栈接收、处理新数据包效率会受到直接的影响。最严重的后果，如果发送方的数据包发送速率很快，则协议栈会因为来不及处理而出现丢包的情况，这将导致不可原谅的错误。这些缺点使 Raw/Callback API 不适合在多任务、交互数据量大、数据处理时间开销长等场合下使用。

2. Sequential API

使用 Raw/Callback API 编程时，用户向内核注册回调函数，并通过直接调用内核 UDP 或 TCP 相关操作函数来完成应用程序的编写，这种方式没有明确规定用户编程的步骤及规则，使用这种 API 进行应用程序的开发，必须建立在对内核透彻理解的基础上，而这点往往会给许多刚接触 LwIP 的同学带来困扰。

使用 BSD Socket 函数进行程序开发是学习网络编程的最佳方式，因为它遵循了一套简单的编程步骤和规则，可以提高应用程序的开发效率。然而，由于 BSD Socket 函数实现上具有很高的抽象特性，它并不适合在小型嵌入式 TCP/IP 协议栈中使用。最主要的，BSD Socket 编程时，在应用程序进程和协议栈内核进程之间会出现数据拷贝的情况，因为操作系统认为它们

是两个互相独立的进程，应该使用相互独立的资源。

如果可以避免上述数据拷贝的出现，那么应用程序的效率将会有很大提高，因为拷贝消耗的时间和空间对系统来说都是非常宝贵的资源，尤其在嵌入式系统中。LwIP 协议栈 API（Sequential API）正是基于上述特点而设计，它的出发点是上层已经预知了协议栈内核的部分结构，API 可以使用这种预知来避免数据拷贝的出现。协议栈 API 同 BSD socket 具有很大相似性，它们遵循了极为相似的编程步骤，但前者工作在一个更低的层次，用户进程可以直接操作内核进程中的数据包数据。



另一方面，由于 BSD socket 简单、易理解，且目前已广泛应用于许多网络应用程序中，因此，为了让熟悉 socket 的用户使用 LwIP，协议栈内部包含了基于协议栈 API 实现的 BSD socket 编程接口（Socket API），下一小节将描述 Socket API 的相关知识。

在 BSD 中，应用程序处理的网络数据都处于一片连续的存储区域中，这可以使得用户对数据的处理变得方便。在 LwIP 中，若 API 使用上述这种数据存储机制可能导致很大的缺陷，因为内核的网络数据都存放在 pbuf 中，如果要为应用程序提供连续存储的数据，则内核必须将所有 pbuf 数据拷贝到连续区域中再递交给用户，这一点恰好违背了 API 设计的初衷。为了解决这个问题，协议栈为应用程序提供了一些直接操作内核 pbuf 数据的函数，应用程序也可以将 pbuf 数据拷贝到连续的存储区域，当然，这是用户自己的事，与协议栈内核无关。

与 BSD 相同，LwIP 协议栈 API 也对网络连接进行了抽象。但它们之间的抽象存在一定的差别：BSD 实现了更高级别的抽象，用户可以像操作一个普通文件那样来操作一个网络连接（打开、关闭、读、写等）；LwIP 中，API 只能实现较低级别的抽象，用户操作的仅仅是一个网络连接，不可能是文件。

协议栈 API 的实现由两部分组成：一部分作为用户编程接口函数提供给用户，这些函数在用户进程中执行；另一部分驻留在协议栈内核进程中。这两部分通过进程通信机制（IPC）实现通信和同步，共同为应用程序提供服务。使用到的进程通信机制包括以下三种：

- 邮箱，例如内核邮箱 mbox、连接上接收数据的邮箱 recvmbox；
- 信号量，例如 op_completed，用于两部分 API 的同步；
- 共享内存，例如内核消息结构 tcpip_msg、API 消息内容 api_msg 等。

两部分 API 间的关系如图 2-18 所示，API 设计的核心在于让用户进程负责尽可能多的工作，例如数据的计算、拷贝等；而协议栈进程只负责简单的通信工作，这点很关键，因为系统可能存在多个应用程序，它们都使用协议栈进程提供的通信服务，保证内核进程的高效性和实时性是提高系统性能的重要保障。

协议栈 API 实现时，也为用户提供了数据包管理函数，可以完成数据包内存申请、释放、数据拷贝等任务。应用程序使用 netbuf 结构来描述、组装数据包，该结构只是对内核 pbuf 的简单封装，通过共享一个 netbuf 结构，两部分 API 就能实现对数据包的共同处理，避免了数据的拷贝。

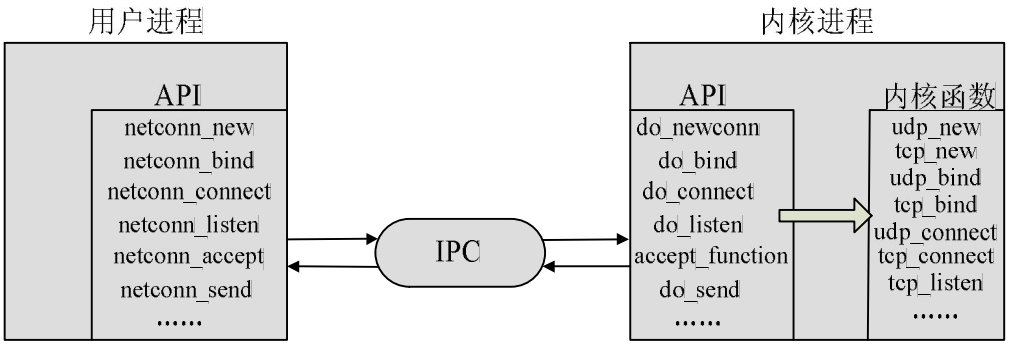


图 2-18　协议栈 API 实现

netbuf 是应用程序描述待发送数据和已接收数据的基本结构，引入 netbuf 结构看似会让应用程序更加繁杂，但实际上内核为用户提供了一系列的操作函数，通过这些函数可以直接对 netbuf 中的数据进行读取、填写等操作，为应用程序开发带来了方便。

无论是 UDP 连接还是 TCP 连接，当协议栈接收到数据包后，会将数据封装在一个 netbuf 中，并递交给应用程序。在数据发送时，不同类型的连接将导致不同的数据处理方式：对于 TCP 连接，用户只需要提供待发送数据的起始地址和长度，内核会根据实际情况将数据封装在合适大小的数据包中，并放入发送队列；对于 UDP 来说，用户需要自行将数据封装在 netbuf 结构中，当发送函数被调用时，内核直接将该数据包中的数据发送出去。

最后，有必要说说使用上层 API 进行编程的特点。首先从进程结构上看，协议栈内核运行于进程 tcpip_thread，而应用程序也是一个单独的进程。独立的进程结构可以使协议栈和应用程序的执行互不影响，而前面讲解的回调方式（Raw/Callback API）编程、用户应用程序和协议栈内核都驻留在同一个进程中，彼此间的执行都会互相制约。使用邮箱和信号量等机制，内核进程可以直接将数据递交到应用程序邮箱中然后继续执行，不必阻塞等待，邮箱对于应用程序来说就像一个输入队列。这在多应用程序环境下，使用上层 API 会带来很大的方便。

另一方面，使用上层 API 编程也存在一定的不足，首先，应用程序执行效率会有所降低，这点很好理解，因为这个过程涉及到了一系列的邮箱、信号量等交互过程。其次，应用程序编写时的函数调用必须遵守一定的顺序规则，正如 LwIP 手册上说的那样，必须遵守 open-read-write-close 的顺序，这也是典型的 Socket 编程的顺序。这样，用户程序的灵活性受到了限制。

总之，在系统响应时间要求较高的小型应用程序设计中，使用 Raw/Callback API 可以带来更高的优越性，而对于多任务环境、大数据量的大型应用程序编写，则采用上层 API 更合适。

3．Socket API

使用 LwIP 的最佳方式是运用 Raw/Callback API 和 Sequential API 进行灵活的编程，让这二者优势互补，应用程序会有很高的运行效率。但对初学者来说，由于对协议栈内部细节的生疏，可能无法准确使用这两种 API，他们可能更习惯使用套接字（Socket）进行编程；另一方面，目前很多网络应用程序都是基于套接字接口的，也为了增加这些程序的可移植性，LwIP

的设计者们对 Sequential API 函数进行了简单的封装，这样就得到了可供用户使用的套接字函数（Socket API）。但是值得注意的是，由于这种封装只是对 socket 函数功能的简单模拟，且标准 socket 库中的部分函数仍无法直接通过封装 Sequential API 来实现，因此 LwIP 中提供了 socket 函数并不完整，用户最好不要使用它进行实际应用程序开发。使用套接字函数编程还有个缺陷，因为它是基于 Sequential API 来实现的，所以程序的执行效率较基于后者实现的程序效率更低。

总之，虽然 socket 接口为用户使用 LwIP 协议栈提供了捷径，也为不同网络应用程序的移植提供了方便，但是 LwIP 中的 Socket API 并不适合用在实际项目开发中。首先，它是基于 Sequential API 来实现的，所以其运行效率较另外两种 API 来说更低；最重要的，Socket API 只是对 BSD socket 的简单模拟，其提供的函数功能并不完善，标准库中的部分函数在 LwIP 中也未实现。熟悉 LwIP 协议栈的用户应尽量使用前两种 API 进行应用程序的开发，特别是使用这两种 API 进行混合编程，让它们优势互补，这样，系统将达到很高的运行效率。