修改历史		
版本	日期	说明
1.0.0	2022-10-17	针对 onps 栈初始版本提供的 API 接口说明
1.1.0	2023-07-26	1. 增加了 ipv6 应用说明; 2. 增加了 NVT 应用说明; 3. 修改了个别函数原型; 4. 修正了一些描述错误;

Onps 栈用户使用手册

1. ping

协议栈提供 ping 工具，其头文件为“net_tools/ping.h”，将其 include 进你的目标系统中即可使用这个工具。

```
......
#include "onps.h"
#include "net_tools/ping.h"

//* 回调函数，收到目标地址的应答报文后 ping 工具会调用这个函数完成用户的特定处理逻辑
//* 针对这个测试，在这里就是简单地打印出了应答报文的内容以及 ping 的响应时间
static void  ping_recv_handler(USHORT usIdentifier,  //* ping 的标识 id，响应报文与探测报文这个 id 应该一致
                               void *pvFromAddr,     //* 响应报文的源地址
                               USHORT usSeqNum,      //* 响应报文序号，其与探测报文一致
                               UCHAR *pubEchoData,   //* 响应报文携带的响应数据，其与探测报文一致
                               UCHAR ubEchoDataLen,  //* 响应报文携带的数据长度
                               UCHAR ubTTL,          //* ttl 值
                               UCHAR ubElapsedMSecs, //* 响应时长，从发送探测报文开始计时到收到响应报文结束计时（毫秒）
                               void *pvParam)
{
    CHAR szSrcAddr[20];
    struct in_addr stInAddr;
    stInAddr.s_addr = *(in_addr_t *)pvFromAddr;
    printf("<Fr>%s, recv %d bytes, ID=%d, Sequence=%d, Data='%s', TTL=%d, time=%dms\r\n",
           inet_ntoa_safe(stInAddr, szSrcAddr), //* 这是一个线程安全的 ip 地址转 ascii 字符串函数
           (UINT)ubEchoDataLen,
           usIdentifier,
           usSeqNum,
           pubEchoData,
           (UINT)ubTTL,
           (UINT)ubElapsedMSecs);
}

int main(void)
{
```

```
    if(open_npstack_load(&enErr))
    {
        printf("The open source network protocol stack (ver %s) is loaded successfully. \r\n", ONPS_VER);

        //* 协议栈加载成功，在这里初始化 ethernet 网卡或等待 ppp 链路就绪
    #if 0
        emac_init(); //* ethernet 网卡初始化函数，并注册网卡到协议栈
    #else
        while(!netif_is_ready("ppp0")) //* 等待 ppp 链路建立成功
            os_sleep_secs(1);
    #endif
    }
    else
    {
        printf("The open source network protocol stack failed to load, %s\r\n", onps_error(enErr));
        return -1;
    }

    //* 启动 ping 测试
    USHORT usSeqNum = 0;
    UINT unErrCount = 0;
    INT nPing = ping_start(AF_INET, &enErr);
    if(nPing < 0)
    {
        //* 启动失败，输出一条日志信息
        printf("ping_start() failed, %s\r\n", onps_error(enErr));
        return -1;
    }

    while(TRUE && usSeqNum < 100)
    {
        //* ping 目标地址
        INT nRtnVal = ping(nPing, "192.168.0.2", usSeqNum++, 64,
                           GetElapsedMSecs, ping_recv_handler, NULL, 3, NULL, &enErr);
        if(nRtnVal <= 0) //* ping 返回一个错误
        {
            //* 累计 ping 错误数
            unErrCount++;

            //* 控制台打印当前错误数
            printf("no reply received, the current number of errors is %d, current error: %s\r\n", unErrCount,
                    nRtnVal ? onps_error(enErr) : "recv timeout");
        }
        os_sleep_secs(1);
    }

    //* 结束 ping 测试
    ping_end(nPing);
```

```
    return 0;
}
```

上述示例代码调用了 ping 测试工具提供的几个接口函数。ping_start()函数的调用非常简单，其功能就是开启 ping 测试。结束 ping 测试需要调用 ping_end()函数，否则 ping 测试会一直占用协议栈资源。这几个函数的说明如下：

函数原型

```
INT ping_start(INT family, EN_ONPSERR *penErr);
```

功能

启动 ping 测试。注意，启动后你可以随时更换目标地址，不必拘泥于一个固定的目标地址。

参数

family：地址族类型，IPv4 地址其值为 AF_INET，IPv6 地址为 AF_INET6

penErr：如果启动失败，该参数用于接收具体的错误码

返回值

成功，返回当前启动的 ping 测试任务的句柄；失败，返回值小于 0，具体错误信息参看 pennErr 保存的错误码。

-

函数原型

```
void ping_end(INT nPing);
```

功能

结束 ping 测试，释放占用的协议栈资源。

参数

nPing：ping_start()函数返回的 ping 测试句柄

返回值

无

-

函数原型

```
INT ping(INT nPing,
         const CHAR *pszDstAddr,
         USHORT usSeqNum,
         UCHAR ubTTL,
         UINT(*pfunGetCurMSecs)(void),
         PFUN_PINGRCVHANDLER pfunRcvHandler,
         PFUN_PINGERRHANDLER pfunErrHandler,
         UCHAR ubWaitSecs,
         void *pvParam,
         EN_ONPSERR *penErr);
```

功能

ping 目标地址并等待接收对端的响应报文。其功能与通用的 ping 测试工具完全相同。

参数

nPing：ping_start()函数返回的 ping 测试句柄
pszDstAddr：指向目标地址的指针，可以是点分 10 进制的 ipv4 地址串，也可以是冒号分隔 ipv6 地址串
usSeqNum：报文序号，用于唯一的标识发出的探测报文
ubTTL：ttl 值，探测报文到达目标地址之前的生存时间
pfunGetCurMSecs：函数指针，这个函数返回系统定时器自启动以来的工作时长，单位：毫秒，其精度取决于 os 定时器粒度，其用于统计 ping 的响应时间，必须提供这个函数，否则 ping 操作将无法完成
pfunRcvHandler：函数指针，收到响应报文后将调用这个函数完成用户指定的操作，ping_recv_handler()就是它的实际实现，其函数原型参见 net_tools/ping.h 文件中 PFUN_PINGRCVHANDLER 定义，注意这个参数值不能为空
pfunErrHandler：函数指针，当 ping 失败时调用，以完成用户自定义的处理逻辑，函数原型参见 PFUN_PINGERRHANDLER 定义
ubWaitSecs：响应报文的最长等待时间，单位：秒
pvParam：传递给 pfunRcvHandler 与 pfunErrHandler 函数的用户自定义参数
penErr：如果 ping 失败，该参数保存具体的错误码

返回值

大于 0，ping 成功；0，等待响应报文超时；小于 0，失败，具体的错误信息参看 penErr 保存的错误码。

2. dns 域名查询

dns 功能测试需要为网卡设定好能够访问互联网的网关、DNS 服务器地址等配置信息。当然如果采用 dhcp 动态地址申请的方式能够得到这些信息那就更省事了。dns 查询工具的头文件为"net_tools/dns.h"。

```
……
#include "onps.h"
#include "net_tools/ping.h"
int main(void)
{
    if(open_npstack_load(&enErr))
    {
        printf("The open source network protocol stack (ver %s) is loaded successfully. \r\n", ONPS_VER);

        //* 协议栈加载成功，在这里初始化 ethernet 网卡或等待 ppp 链路就绪
    #if 0
        emac_init(); //* ethernet 网卡初始化函数，并注册网卡到协议栈
    #else
        while(!netif_is_ready("ppp0")) //* 等待 ppp 链路建立成功
            os_sleep_secs(1);
    #endif
    }
    else
    {
        printf("The open source network protocol stack failed to load, %s\r\n", onps_error(enErr));
        return -1;
    }
```

```
    //* dns 查询测试
    in_addr_t unPrimaryDNS, unSecondaryDNS;
    INT nDnsClient = dns_client_start(&unPrimaryDNS, &unSecondaryDNS, 3, &enErr);
    if(nDnsClient < 0)
    {
        //* dns 客户端启动失败，输出一条错误日志
        printf("%s\r\n", onps_error(enErr));
    }
    else
    {
        //* 发送查询请求并等待 dns 服务器的应答
        in_addr_t unIp = dns_client_query(nDnsClient, unPrimaryDNS, unSecondaryDNS, "gitee.com", &enErr);
        if(unIp) //* 查询成功
        {
            CHAR szAddr[20];
            printf("The ip addr: %s\r\n", inet_ntoa_safe_ext(unIp, szAddr));
        }
        else
            printf("%s\r\n", onps_error(enErr)); //* 查询失败

        //* 结束 dns 查询，释放占用的协议栈资源
        dns_client_end(nDnsClient);
    }

    return 0;
}
```

与 ping 测试工具相同，dns 查询工具同样提供了一组简单的 api 函数用于实现域名查询。这一组函数包括 dns_client_start()、dns_client_end()以及 dns_client_query()，其使用说明如下：

函数原型

```
INT dns_client_start(in_addr_t *punPrimaryDNS, in_addr_t *punSecondaryDNS, CHAR bRcvTimeout, EN_ONPSERR *penErr);
```

功能

启动一个域名查询客户端。

参数

```
  punPrimaryDNS: 指针类型，用于接收主域名服务器地址。协议栈会选择缺省路由绑定的网卡，并得到网卡携带的 dns 服务器地址用
                 于接下来的域名查询，该参数即用于保存主 dns 服务器的地址
punSecondaryDNS: 指针类型，与上同，用于接收次域名服务器地址
    bRcvTimeout: 查询超时时间，单位：秒
         penErr: 如果启动失败，该参数用于接收具体的错误码
```

返回值

成功，返回当前启动的 dns 客户端的句柄；失败，返回值小于 0，具体错误信息参看 pennErr 保存的错误码。

_

函数原型

```
void dns_client_end(INT nClient);
```

功能

结束 dns 客户端，释放占用的协议栈资源。

参数

nClient：dns_client_start()函数返回的 dns 客户端句柄

返回值

无

-

函数原型

```
in_addr_t dns_client_query(INT nClient,
                           in_addr_t unPrimaryDNS,
                           in_addr_t unSecondaryDNS,
                           const CHAR *pszDomainName,
                           EN_ONPSERR *penErr);
```

功能

发送 dns 查询请求，并等待服务器的响应报文，功能与通用的 dns 客户端完全相同。

参数

nClient：dns_client_start()函数返回的 dns 客户端句柄
punPrimaryDNS：主域名服务器地址，其值为 dns_client_start()函数得到的主域名服务器地址
punSecondaryDNS：次域名服务器地址，其值为 dns_client_start()函数得到的次域名服务器地址
pszDomainName：指针类型，指向要查询的域名
penErr：如果查询失败，该参数用于接收具体的错误码

返回值

成功，返回域名对应的 ip 地址；失败，返回值为 0，具体错误信息参看 pennErr 保存的错误码。

3. sntp 网络校时

与 dns 功能测试的要求一样，要进行这个测试依然要确保你的开发板在物理层能够访问互联网，同时你的开发板支持 rtc，并提供一组 rtc 操作函数，包括读取、设置系统当前时间等 api。这里假设你的测试环境已经具备上述测试条件。sntp 网络校时工具的头文件为"net_tools/sntp.h"。

```
……
#include "onps.h"
#include "net_tools/sntp.h"
int main(void)
{
    if(open_npstack_load(&enErr))
    {
        printf("The open source network protocol stack (ver %s) is loaded successfully. \r\n", ONPS_VER);
```

```
        //* 协议栈加载成功，在这里初始化 ethernet 网卡或等待 ppp 链路就绪
    #if 0
        emac_init(); //* ethernet 网卡初始化函数，并注册网卡到协议栈
    #else
        while(!netif_is_ready("ppp0")) //* 等待 ppp 链路建立成功
            os_sleep_secs(1);
    #endif
    }
    else
    {
        printf("The open source network protocol stack failed to load, %s\r\n", onps_error(enErr));
        return -1;
    }

    //* 先设定个不合理的时间，以测试网络校时功能是否正常,由 rtc 驱动提供，负责修改系统当前时间
    //* RTC 前缀的函数为目标系统应提供的 rtc 时钟操作函数
    RTCSetSysTime(22, 9, 5, 17, 42, 30);

    //* 开启网络校时，sntp_update_by_ip()与 sntp_update_by_dns()均可使用
    ST_DATETIME stDateTime;
#if 1
    if(sntp_update_by_ip("52.231.114.183", NULL, RTCSetSystemUnixTimestamp, 8, &enErr)) //* ntp 服务器地址直接校时
#else
    if(sntp_update_by_dns("time.windows.com", Time, RTCSetSystemUnixTimestamp, 8, &enErr)) //* ntp 服务器域名方式校时
#endif
    {
        //* 获取系统时间，检查校时结果
        RTCGetSysTime(&stDateTime);

        //* 控制台输出当前系统时间
        printf("The time is %d-%02d-%02d %02d:%02d:%02d\r\n", stDateTime.usYear, stDateTime.ubMonth,
               stDateTime.ubDay, stDateTime.ubHour, stDateTime.ubMin, stDateTime.ubSec);
    }
    else
    {
        printf("%s\r\n", onps_error(enErr));
        return -1;
    }

    return 0;
}
```

测试代码首先把时间设定在了 2022 年 9 月 5 日 17 点 42 分 30 秒，目的是为了验证目标系统时间是否会被成功校正。测试代码用到了目标系统应提供的一组 rtc 时钟操作函数。其中 RTCSetSysTime()用于设置系统时间。RTCSetSystemUnixTimestamp()函数同样也是设置系统时间，只不过是通过 unix 时间戳进行设置。RTCGetSysTime()函数用于读取当前系统时间。相较于 ping 及 dns 工具，sntp 网络校时工具只提供了一个接口函数 sntp_update_by_xx()即可完成校时。我们可以通过 ntp 服务器地址也可以通过 ntp

服务器域名进行校时。该函数的详细使用说明如下：

函数原型

```
BOOL sntp_update_by_ip(const CHAR *pszNtpSrvIp,
                       time_t(*pfunTime)(void),
                       void(*pfunSetSysTime)(time_t),
                       CHAR bTimeZone,
                       EN_ONPSERR *penErr);
```

功能

发送一个校时请求到 pszNtpSrvIp 参数指定的 ntp 服务器，并等待服务器的响应报文，完成校时操作。

参数

pszNtpSrvIp：ntp 服务器 ip 地址
pfunTime：函数指针，与 c 库函数 time()功能及原型相同，返回自 1970 年 1 月 1 日 0 时 0 分 0 秒以来经过的秒数，可以为空
pfunSetSysTime：函数指针，通过 unix 时间戳设置系统当前时间，由 sntp_update_by_xx()内部调用，收到正确的响应报文后调用该函数设置系统时间
bTimeZone：时区，例如东 8 区，其值为 8；西 8 区其值为-8
penErr：如果校时失败，该参数用于接收具体的错误码

返回值

校时成功，返回 TRUE；失败，返回 FALSE，具体错误信息参看 pennErr 保存的错误码。

sntp_update_by_dns()函数与 sntp_update_by_ip()函数除了第一个入口参数变成了域名外，其它完全相同，不再赘述。

4. tcp 客户端

在协议栈源码工程下，存在一个用 vs2015 建立的 TcpServerForStackTesting 工程。其运行在 windows 平台下，模拟实际应用场景下的 tcp 服务器。当 tcp 客户端连接到服务器后，服务器会立即下发一个 1100 多字节长度的控制报文到客户端。之后在整个 tcp 链路存续期间，服务器会每隔一段随机的时间（90 秒到 120 秒之间）下发控制报文到客户端，模拟实际应用场景下服务器主动下发指令、数据到客户端的情形。客户端则连续上发数据报文到服务器，服务器回馈一个应答报文给客户端。客户端如果收不到该应答报文则会立即重发，直至收到应答报文或超过重试次数后重连服务器。总之，整个测试场景的设计目标就是完全契合常见的商业应用需求，以此来验证协议栈的核心功能指标是否完全达标。用 vs2015 打开这个工程，配置管理器指定目标平台为 x64。main.cpp 及 tcp_helper.h 文件的头部定义了服务器地址类型、端口号以及报文长度等信息：

```
#define SUPPORT_IPv6     1    //* 使能或禁止 tcp 服务器是否支持 ipv6 地址（tcp_helper.h 文件）
#define SRV_PORT         6410 //* 服务器端口
#define LISTEN_NUM       10   //* 最大监听数
#define RCV_BUF_SIZE     2048 //* 接收缓冲区容量
#define PKT_DATA_LEN_MAX 1200 //* 报文携带的数据最大长度，凡是超过这个长度的报文都将被丢弃
```

如果我们想测试 ipv6 通讯，将 SUPPORT_IPv6 宏置位即可。客户端将使用 ipv6 地址连接服务器。我们可以依据实际情形调整上述配置并利用这个模拟服务器测试 tcp 客户端的通讯功能。

```
……
#include "onps.h"
```

```
#define PKT_FLAG 0xEE //* 通讯报文的头部和尾部标志
typedef struct _ST_COMMUPKT_HDR_ { //* 数据及控制指令报文头部结构
    CHAR bFlag;         //* 报文头部标志，其值参看 PKT_FLAG 宏
    CHAR bCmd;          //* 指令，0 为数据报文，1 为控制指令报文
    CHAR bLinkIdx;      //* tcp 链路标识，当存在多个 tcp 链路时，该字段用于标识这是哪一个链路
    UINT unSeqNum;      //* 报文序号
    UINT unTimestamp;   //* 报文被发送时刻的 unix 时间戳
    USHORT usDataLen;   //* 携带的数据长度
    USHORT usChechsum;  //* 校验和（crc16），覆盖除头部和尾部标志字符串之外的所有字段
} PACKED ST_COMMUPKT_HDR, *PST_COMMUPKT_HDR;

typedef struct _ST_COMMUPKT_ACK_ { //* 数据即控制指令应答报文结构
    ST_COMMUPKT_HDR stHdr; //* 报文头
    UINT unTimestamp;      //* unix 时间戳，其值为被应答报文携带的时间戳
    CHAR bLinkIdx;         //* tcp 链路标识，其值为被应答报文携带的链路标识
    CHAR bTail;            //* 报文尾部标志，其值参看 PKT_FLAG 宏
} PACKED ST_COMMUPKT_ACK, *PST_COMMUPKT_ACK;

//* 提前申请一块静态存储时期的缓冲区用于 tcp 客户端的接收和发送，因为接收和发送的报文都比较大，所以不使用动态申请的方式
#define RCV_BUF_SIZE     1300          //* 接收缓冲区容量
#define PKT_DATA_LEN_MAX 1200          //* 报文携带的数据最大长度，凡是超过这个长度的报文都将被丢弃
static UCHAR l_ubaRcvBuf[RCV_BUF_SIZE]; //* 接收缓冲区
static UCHAR l_ubaSndBuf[sizeof(ST_COMMUPKT_HDR) + PKT_DATA_LEN_MAX]; //* 发送缓冲区，ST_COMMUPKT_HDR 为通讯报文头部结构体
int main(void)
{
    EN_ONPSERR enErr;
    SOCKET hSocket = INVALID_SOCKET;

    if(open_npstack_load(&enErr))
    {
        printf("The open source network protocol stack (ver %s) is loaded successfully. \r\n", ONPS_VER);

        //* 协议栈加载成功，在这里初始化 ethernet 网卡或等待 ppp 链路就绪
    #if 0
        emac_init(); //* ethernet 网卡初始化函数，并注册网卡到协议栈
    #else
        while(!netif_is_ready("ppp0")) //* 等待 ppp 链路建立成功
            os_sleep_secs(1);
    #endif
    }
    else
    {
        printf("The open source network protocol stack failed to load, %s\r\n", onps_error(enErr));
        return -1;
    }

    //* 分配一个 socket
    if(INVALID_SOCKET == (hSocket = socket(AF_INET, SOCK_STREAM, 0, &enErr)))
```

```
    {
        //* 返回了一个无效的 socket，打印错误日志
        printf("<1>socket() failed, %s\r\n", onps_error(enErr));
        return -1;
    }

    //* 连接成功则 connect()函数返回 0，非 0 值则连接失败
    if(connect(hSocket, "192.168.0.2", 6410, 10))
    {
        printf("connect 192.168.0.2:6410 failed, %s\r\n", onps_get_last_error(hSocket, NULL));
        close(hSocket);
        return -1;
    }

    //* 等待接收服务器应答或控制报文的时长（即 recv()函数的等待时长），单位：秒。0 不等待；大于 0 等待指定秒数；-1 一直
    //* 等待直至数据到达或报错。设置成功返回 TRUE，否则返回 FALSE。这里我们设置 recv()函数不等待
    //* 注意，只有连接成功后才可设置这个接收等待时长，在这里我们设置接收不等待，recv()函数立即返回，非阻塞型
    if(!socket_set_rcv_timeout(hSocket, 0, &enErr))
        printf("socket_set_rcv_timeout() failed, %s\r\n", onps_error(enErr));

    INT nThIdx = 0;
    while(TRUE && nThIdx < 1000)
    {
        //* 接收，前面已经设置 recv()函数不等待，有数据则读取数据后立即返回，无数据则立即返回
        INT nRcvBytes = recv(hSocket, ubaRcvBuf, sizeof(ubaRcvBuf));
        if(nRcvBytes > 0)
        {
            //* 收到报文，处理之，报文有两种：一种是应答报文；另一种是服务器主动下发的控制报文
            //* 在这里添加你的自定义代码
            ......
        }

        //* 发送数据报文到服务器，首先封装要发送的数据报文，PST_COMMUPKT_HDR 其类型为指向 ST_COMMUPKT_HDR 结构体的指
        //* 针，这个结构体是与 TcpServerForStackTesting 服务器通讯用的报文头部结构
        PST_COMMUPKT_HDR pstHdr = (PST_COMMUPKT_HDR)l_ubaSndBuf;
        pstHdr->bFlag = (CHAR)PKT_FLAG;
        pstHdr->bCmd = 0x00;
        pstHdr->bLinkIdx = (CHAR)nThIdx++;
        pstHdr->unSeqNum = unSeqNum;
        pstHdr->unTimestamp = time(NULL);
        pstHdr->usDataLen = 900; //* 填充随机数据，随机数据长度加 ST_COMMUPKT_HDR 结构体长度不超过 l_ubaSndBuf 的长度即可
        pstHdr->usChechsum = 0;
        pstHdr->usChechsum = crc16(l_ubaSndBuf + sizeof(CHAR), sizeof(ST_COMMUPKT_HDR) - sizeof(CHAR) + 900, 0xFFFF);
        l_ubaSndBuf[sizeof(ST_COMMUPKT_HDR) + 900] = PKT_FLAG;

        //* 发送上面已经封装好的数据报文
        INT nPacketLen = sizeof(ST_COMMUPKT_HDR) + pstHdr->usDataLen + 1;
#if SUPPORT_SACK //* 如果协议栈开启 tcp sack 支持
```

```
        INT nSndBytes, nHasSendBytes = 0;
        while(nHasSendBytes < nPacketLen)
        {
            nSndBytes = send(hSocket, &l_ubaSndBuf[nHasSendBytes], nPacketLen - nHasSendBytes, 0);
            if(nSndBytes < 0)
            {
                printf("<err>sent %d bytes failed, %s\r\n", nPacketLen, onps_get_last_error(hSocket, &enErr));

                //* 关闭 socket，断开当前 tcp 连接，释放占用的协议栈资源
                close(hSocket);
                return -1;
            }
            else
                nHasSendBytes += nSndBytes;
        }
#else
        INT nSndBytes = send(hSocket, l_ubaSndBuf, nPacketLen, 3);
        if(nSndBytes != nPacketLen) //* 与实际要发送的数据不相等的话就意味着发送失败了
        {
            printf("<err>sent %d bytes failed, %s\r\n", nPacketLen, onps_get_last_error(hSocket, &enErr));

            //* 关闭 socket，断开当前 tcp 连接，释放占用的协议栈资源
            close(hSocket);
            return -1;
        }
#endif
    }

    //* 关闭 socket，断开当前 tcp 连接，释放占用的协议栈资源
    close(hSocket);

    return 0;
}
```

编写 tcp 客户端的几个关键步骤：

1) 调用 socket 函数，申请一个数据流(tcp)类型的 socket；
2) connect()函数建立 tcp 连接；
3) recv()函数等待接收服务器下发的应答及控制报文；
4) send()函数将封装好的数据报文发送给服务器；
5) close()函数关闭 socket，断开当前 tcp 连接；

真实场景下，单个 tcp 报文携带的数据长度的上限基本在 1K 左右。所以，在上面给出的功能测试代码中，单个通讯报文的长度也设定在这个范围内。客户端循环上报服务器的数据报文的长度 900 多字节，服务器下发开发板的控制报文长度 1100 多字节。

与传统的 socket 编程相比，除了上述几个函数的原型与 Berkeley sockets 标准有细微的差别，在功能及使用方式上没有任何改变。之所以对函数原型进行调整，原因是传统的 socket 编程模型比较繁琐——特别是阻塞/非阻塞的设计很不简洁，需要一些看起来很“突兀”地额外编码，比如 select 操作。在设计协议栈的 socket 模型时，考虑到类似 select 之类的操作细节完全可以借助目标 OS 的信号量机制

将其封装到底层实现，从而达成简化用户编码，让 socket 编程更加简洁、优雅的目的。因此，最终呈现给用户的协议栈 socket 模型部分偏离了 Berkeley 标准。

5. tcp 服务器

常见的 tcp 服务器要完成的工作无外乎就是接受连接请求，接收客户端上发的数据，下发应答或控制报文，清除不活跃的客户端以释放其占用的系统资源。因此，tcp 服务器的功能测试代码分为两部分实现：一部分在主线程完成启动 tcp 服务器、等待接受连接请求这两项工作（为了突出主要步骤，清除不活跃客户端的工作在这里省略）；另一部分单独建立一个线程完成读取客户端数据并下发应答报文的工作。

```
……
#include "onps.h"
#define LTCPSRV_PORT        6411 //* tcp 测试服务器端口
#define LTCPSRV_BACKLOG_NUM 5    //* 排队等待接受连接请求的客户端数量
static SOCKET l_hSockSrv;        //* tcp 服务器 socket，这是一个静态存储时期的变量，因为服务器数据接收线程也要使用这个变量

//* 启动 tcp 服务器，采用 poll 模型被动等待接收客户端数据
SOCKET tcp_server_start(USHORT usSrvPort, USHORT usBacklog)
{
    EN_ONPSERR enErr;
    SOCKET hSockSrv;
    do {
        //* 申请一个 socket
        hSockSrv = socket(AF_INET, SOCK_STREAM, 0, &enErr);
        if(INVALID_SOCKET == hSockSrv)
            break;

        //* 绑定地址和端口，功能与 Berkeley sockets 提供的 bind()函数相同
        if(bind(hSockSrv, NULL, usSrvPort))
            break;

        //* 启动监听，同样与 Berkeley sockets 提供的 listen()函数相同，
        if(listen(hSockSrv, usBacklog))
            break;
        return hSockSrv;
    } while(FALSE);

    //* 执行到这里意味着前面出现了错误，无法正常启动 tcp 服务器了
    if(INVALID_SOCKET != hSockSrv)
        close(hSockSrv);
    printf("%s\r\n", onps_error(enErr));

    //* tcp 服务器启动失败，返回一个无效的 socket 句柄
    return INVALID_SOCKET;
}
//* 完成 tcp 服务器的数据读取工作
static void THTcpSrvRead(void *pvData)
{
```

```
    SOCKET hSockClt;
    EN_ONPSERR enErr;
    INT nRcvBytes;
    UCHAR ubaRcvBuf[256];

    while(TRUE)
    {
        //* 等待客户端有新数据到达
        hSockClt = tcpsrv_recv_poll(l_hSockSrv, 1, &enErr);
        if(INVALID_SOCKET != hSockClt) //* 有效的 socket
        {
            //* 注意这里一定要尽量读取完毕该客户端的所有已到达的数据，因为每个客户端只有新数据到达时才会触发一个信号到用户
            //* 层，如果你没有读取完毕就只能等到该客户端送达下一组数据时再读取了，这可能会导致数据处理延迟问题
            while(TRUE)
            {
                //* 读取数据
                nRcvBytes = recv(hSockClt, ubaRcvBuf, 256);
                if(nRcvBytes > 0)
                {
                    //* 原封不动的回送给客户端，利用回显来模拟服务器回馈应答报文的场景
                    send(hSockClt, ubaRcvBuf, nRcvBytes, 1);
                }
                else //* 已经读取完毕
                {
                    if(nRcvBytes < 0)
                    {
                        //* 协议栈底层报错，这里需要增加你的容错代码处理这个错误并打印错误信息
                        printf("%s\r\n", onps_get_last_error(hSocket, NULL));
                    }
                    break;
                }
            }
        }
        else //* 无效的 socket
        {
            //* 返回一个无效的 socket 时需要判断是否存在错误，如果不存在则意味着 1 秒内没有任何数据到达，否则打印这个错误
            if(ERRNO != enErr)
            {
                printf("tcpsrv_recv_poll() failed, %s\r\n", onps_error(enErr));
                break;
            }
        }
    }
}

int main(void)
{
    EN_ONPSERR enErr;
```

```
    if(open_npstack_load(&enErr))
    {
        printf("The open source network protocol stack (ver %s) is loaded successfully. \r\n", ONPS_VER);

        //* 协议栈加载成功，在这里初始化 ethernet 网卡,并注册网卡到协议栈
        emac_init();
    }
    else
    {
        printf("The open source network protocol stack failed to load, %s\r\n", onps_error(enErr));
        return -1;
    }

    //* 启动 tcp 服务器
    l_hSockSrv = tcp_server_start(LTCPSRV_PORT, LTCPSRV_BACKLOG_NUM);
    if(INVALID_SOCKET != l_hSockSrv)
    {
        //* 在这里添加工作线程启动代码，启动 tcp 服务器数据读取线程 THTcpSrvRead
        ......
    }

    //* 进入主线程的主逻辑处理循环，等待 tcp 客户端连接请求到来
    while(TRUE)
    {
        //* 接受连接请求
        in_addr_t unCltIP;
        USHORT usCltPort;
        SOCKET hSockClt = accept(l_hSockSrv, &unCltIP, &usCltPort, 1, &enErr);
        if(INVALID_SOCKET != hSockClt)
        {
            //* 在这里你自己的代码处理新到达的客户端
            ......
        }
        else
        {
            printf("accept() failed, %s\r\n", onps_error(enErr));
            break;
        }
    }

    //* 关闭 socket，释放占用的协议栈资源
    close(l_hSockSrv);

    return 0;
}
```

编写 tcp 服务器的几个主要步骤：

1) 调用 socket 函数，申请一个数据流(tcp)类型的 socket;
2) bind()函数绑定一个 ip 地址和端口号;
3) listen()函数启动监听;
4) accept()函数接受一个 tcp 连接请求;
5) 调用 tcpsrv_recv_poll()函数利用协议栈提供的 poll 模型（非传统的 select 模型）等待客户端数据到达;
6) 调用 recv()函数读取客户端数据并处理之，直至所有数据读取完毕返回第 5 步，获取下一个已送达数据的客户端 socket;
7) 定期检查不活跃的客户端，调用 close()函数关闭 tcp 链路，释放客户端占用的协议栈资源;

与传统的 tcp 服务器编程并没有两样。

协议栈实现了一个 poll 模型用于服务器的数据读取。poll 模型利用了目标 OS 的信号量机制。当某个 tcp 服务器端口有一个或多个客户端有新的数据到达时,协议栈会立即投递一个或多个信号到用户层。注意,协议栈投递信号的数量取决于新数据到达的次数(tcp 层每收到一个携带数据的 tcp 报文记一次),与客户端数量无关。用户通过 tcpsrv_recv_poll()函数得到这个信号，并得到最先送达数据的客户端 socket，然后读取该客户端送达的数据。注意这里一定要把所有数据读取出来。因为信号被投递的唯一条件就是有新的数据到达。没有信号， tcpsrv_recv_poll()函数无法得到一个有效的客户端 socket，那么剩余数据就只能等到该客户端再次送达新数据时再读了。

其实，poll 模型的运作机制非常简单。tcp 服务器每收到一组新的数据，就会将该数据所属的客户端 socket 放入接收队列尾部，然后投信号。所以，数据到达、获取 socket 与投递信号是一系列的连锁反应，且一一对应。tcpsrv_recv_poll()函数则在用户层接着完成连锁反应的后续动作：等信号、摘取接收队列首部节点、取出首部节点保存的 socket、返回该 socket 以告知用户立即读取数据。非常简单明了，没有任何拖泥带水。从这个运作机制我们可以看出:

1) poll 模型的运转效率取决于 rtos 的信号量处理效率;
2) tcpsrv_recv_poll()函数每次返回的 socket 有可能是同一个客户端的，也可能是不同客户端;
3) 单个客户端已送达的数据长度与信号并不一一对应,一一对应的是该客户端新数据到达的次数与信号投递的次数，所以当数据读取次数小于信号数时，存在读取数据长度为 0 的情形;
4) tcpsrv_recv_poll()函数返回有效的 sokcet 后，尽量读取全部数据到用户层进行处理，否则会出现剩余数据无法读取的情形，如果客户端不再上发新的数据的话;

协议栈还为用户提供了另外一种数据读取模式：主动读取模式。在某些应用场景中，用户需要主动读取客户端到达的数据而不是被动等待 poll 模型的通知,此时就需要通过调用 tcpsrv_set_recv_mode()函数显式地设置服务器为主动读取模式。有关这个函数的详情请参阅《onps 栈 API 接口手册》“Berkeley sockets”一节。这里唯一需要交待的地方是,对这个函数的调用一定要在 listen()函数之后,因为 listen()函数会将读取模式缺省设置为 poll 被动等待模型。

6. udp 通讯

相比 tcp，udp 通讯功能的实现相对简单很多。为 udp 绑定一个固定端口其就可以作为服务器使用，反之则作为一个客户端使用。

```
……
#include "onps.h"

#define RUDPSRV_IP   "192.168.0.2" //* 远端 udp 服务器的地址
#define RUDPSRV_PORT 6416          //* 远端 udp 服务器的端口
#define LUDPSRV_PORT 6415          //* 本地 udp 服务器的端口
```

```
//* udp 通讯用缓冲区（接收和发送均使用）
static UCHAR l_ubaUdpBuf[256];

int main(void)
{
    EN_ONPSERR enErr;
    SOCKET hSocket = INVALID_SOCKET;

    if(open_npstack_load(&enErr))
    {
        printf("The open source network protocol stack (ver %s) is loaded successfully. \r\n", ONPS_VER);

        //* 协议栈加载成功，在这里初始化 ethernet 网卡或等待 ppp 链路就绪
    #if 0
        emac_init(); //* ethernet 网卡初始化函数，并注册网卡到协议栈
    #else
        while(!netif_is_ready("ppp0")) //* 等待 ppp 链路建立成功
            os_sleep_secs(1);
    #endif
    }
    else
    {
        printf("The open source network protocol stack failed to load, %s\r\n", onps_error(enErr));
        return -1;
    }

    //* 分配一个 socket
    if(INVALID_SOCKET == (hSocket = socket(AF_INET, SOCK_DGRAM, 0, &enErr)))
    {
        //* 返回了一个无效的 socket，打印错误日志
        printf("<1>socket() failed, %s\r\n", onps_error(enErr));
        return -1;
    }

#if 0
    //* 如果是想建立一个 udp 服务器，这里需要调用 bind()函数绑定地址和端口
    if(bind(hSocket, NULL, LUDPSRV_PORT))
    {
        printf("bind() failed,  %s\r\n", onps_get_last_error(hSocket, NULL));

        //* 关闭 socket 释放占用的协议栈资源
        close(hSocket);
        return -1;
    }
#else
    //* 建立一个 udp 客户端，在这里可以调用 connect()函数绑定一个固定的目标服务器，接下来就可以直接使用 send()函数发送
    //* 数据，当然在这里你也可以什么都不做（不调用 connect()），但接下来你需要使用 sendto()函数指定要发送的目标地址
    if(connect(hSocket, RUDPSRV_IP, RUDPSRV_PORT, 0))
```

```
    {
        printf("connect %s:%d failed, %s\r\n", RUDPSRV_IP, RUDPSRV_PORT, onps_get_last_error(hSocket, NULL));

        //* 关闭 socket 释放占用的协议栈资源
        close(hSocket);
        return -1;
    }
#endif

    //* 与 tcp 客户端测试一样，接收数据之前要设定 udp 链路的接收等待的时间，单位：秒，这里设定 recv()函数等待 1 秒
    if(!socket_set_rcv_timeout(hSocket, 1, &enErr))
        printf("socket_set_rcv_timeout() failed, %s\r\n", szNowTime, onps_error(enErr));

    INT nCount = 0;
    while(TRUE && nCount < 1000)
    {
        //* 发缓冲区填充一段字符串然后得到其填充长度
        sprintf((char *)l_ubaUdpBuf, "U#%d#%d#>1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ", time(NULL), nCount++);
        INT nSendDataLen = strlen((const char *)l_ubaUdpBuf);

        //* 调用 send()函数发送数据，如果实际发送长度与字符串长度不相等则说明发送失败
        if(nSendDataLen != send(hSocket, l_ubaUdpBuf, nSendDataLen, 0))
            printf("send failed, %s\r\n", onps_get_last_error(hSocket, NULL));

        //* 接收对端数据之前清 0，以便本地能够正确输出收到的对端回馈的字符串
        memset(l_ubaUdpBuf, 0, sizeof(l_ubaUdpBuf));

        //* 调用 recv()函数接收数据，如果想知道对端地址调用 recvfrom()函数，在这里 recv()函数为阻塞模式，最长阻
        //* 塞 1 秒(如果未收到任何 udp 报文的话)
        INT nRcvBytes = recv(hSocket, l_ubaUdpBuf, sizeof(l_ubaUdpBuf));
        if(nRcvBytes > 0)
            printf("recv %d bytes, Data = <%s>\r\n", nRcvBytes, (const char *)l_ubaUdpBuf);
        else
        {
            //* 小于 0 则意味着 recv()函数报错
            if(nRcvBytes < 0)
            {
                printf("recv failed, %s\r\n", onps_get_last_error(hSocket, NULL));

                //* 关闭 socket 释放占用的协议栈资源
                close(hSocket);
                break;
            }
        }
    }

    //* 关闭 socket，断开当前 tcp 连接，释放占用的协议栈资源
    close(hSocket);
```

```
    return 0;
}
```

udp 通讯编程依然遵循了传统习惯，主要编程步骤还是那些：
1) 调用 socket 函数，申请一个 SOCK_DGRAM(udp)类型的 socket；
2) 如果想建立服务器，调用 bind()函数；想与单个目标地址通讯，调用 connect()函数；与任意目标地址通讯则什么都不用做；
3) 调用 send()或 sendto()函数发送 udp 报文；
4) 调用 recv()或 recvfrom()函数接收 udp 报文；
5) close()函数关闭 socket 释放当前占用的协议栈资源；

7. 扩展 NVT 命令

协议栈提供了一个虚拟网络终端 NVT 以方便用户能够通过 telnet 登录到目标系统进行调参、调试等工作。其实，凡是涉及网络通讯的应用场景，我们都会不可避免的遇到修改 ip 或 mac 地址以解决冲突问题或者 ping 一下看网络是否正常等需求。这也是协议栈提供 NVT 的原因所在。NVT 为此还提供了一组通用命令用于网络参数配置，以最大限度简化用户开发及部署难度：

- **netif**：打印所有网络接口信息，类似 ipconfig/ifconfig 命令
- **route**：打印路由表信息；增加、删除路由表条目
- **ifip**：修改、增加、删除 ip 地址；修改 MAC 地址；修改主从 DNS 地址；修改地址配置策略（动态或静态地址模式）
- **nslookup**：dhs 域名查询工具
- **ntpdate**：网络校时，使用公共 NTP 服务器校正本地时间
- **ping**：轻型 ping 探测工具，用于确定本机到目标地址的网络是否畅通
- **telnet**：telnet 客户端，用于登录其它提供 telnet 服务的主机系统，如 windows、linux 系统

上述命令主要是为了满足基本的网络配置需求。我们只需在 sys_config.h 文件中使能 telnet 服务及相应命令即可为 telent 登录用户提供 NVT 操作能力，详情请参阅《onps 栈移植手册》1.2 节。

当然，仅仅是上述网络配置命令是远远无法满足不同应用场景需求的。所以，协议栈允许用户增加自己的 NVT 命令。我们只需按照一定规则编码实现命令，然后注册到协议栈即可。NVT 支持两种类型命令：**阻塞型**和**非阻塞型**。顾名思义，**阻塞型命令**会阻塞当前程序的执行。NVT 控制台执行该命令时将无法继续其它操作，比如基本的网络通讯等。这将导致严重的 telnet 通讯故障，这是完全不被允许的。因此，阻塞型命令是不能被 NVT 直接调用执行的，其应当作为一个独立的线程/任务启动执行。实际上就是阻塞型命令需要提供两个入口函数：一个是供 NVT 调用的命令入口函数，另一个是作为线程/任务启动的执行入口函数。NVT 负责调用命令入口函数，然后再由命令入口函数调用目标 OS 提供的线程/任务创建函数来启动命令。命令由此开始独立执行。**非阻塞型命令**就相对比较简单，因为它不会阻塞 NVT，所以我们只需提供一个命令入口函数即可。NVT 会直接等待其执行完毕后再继续执行主处理逻辑。

NVT 在设计时充分考虑了传统控制台程序的编写习惯，命令入口函数的原型与 main()函数基本相同：

```
INT (*pfun_cmd_entry)(CHAR argc, CHAR* argv[], ULONGLONG ullNvtHandle);
```

参数 argc 与 argv 其实就是标准意义的控制台输入参数，与 pc 下控制台命令行程序的使用方法完全相同。多出来的第三个参数 ullNvtHandle 唯一的标识一个 NVT，其被用于访问 NVT 提供的输入/输出接口。函数的返回值 NVT 没有统一规定，当然也没有处理这个值，所以我们可以根据实际情况自行定义。对于执行入口函数（如果需要），由于其作为线程/任务启动，所以其原型就相对固定：

```
void (*pfun_cmd_exec_entry)(void *pvParam);
```

pvParam 为用户自定义入口参数。

7.1 编写阻塞型命令

前面提到的 ping 和 telnet 均为阻塞型命令。我们以 ping 为例来看看如何编写一个阻塞型命令。如前文所述，首先我们要实现一个供 NVT 直接调用的命令入口函数：

```
typedef struct _ST_PING_STARTARGS_ { //* ping 启动参数
    CHAR bIsCpyEnd;          //* 启动参数副本建立结束标志
    ULONGLONG ullNvtHandle; //* NVT 句柄
    INT nFamily;             //* 目标地址族：ipv4 或 ipv6
    CHAR szDstIp[40];        //* 目标地址
} ST_PING_STARTARGS, *PST_PING_STARTARGS;

//* ping 命令入口函数
INT nvt_cmd_ping(CHAR argc, CHAR* argv[], ULONGLONG ullNvtHandle)
{
    ST_PING_STARTARGS stArgs;

    //* 根据 argc 及 argv 读取用户输入参数，将地址族类型、目标地址赋值给变量 stArgs
    if ('4' == argv[1][0])
        stArgs.nFamily = AF_INET;
    else
        stArgs.nFamily = AF_INET6;
    snprintf(stArgs.szDstIp, sizeof(stArgs.szDstIp), "%s", argv[2]);
    stArgs.bIsCpyEnd = FALSE; //* 创建线程/任务之前，副本建立标志一定是 FALSE
    stArgs.ullNvtHandle = ullNvtHandle;

    //* 在这里添加线程/任务启动代码，执行实际的 ping 探测，其执行入口函数为 nvt_cmd_ping_entry()
    ……

    //* 等待 nvt_cmd_ping_entry()函数完成启动参数副本的建立
    while (!stArgs.bIsCpyEnd)
        os_sleep_ms(10);
    return 0;
}
```

接着实现 ping 命令的执行入口函数：

```
//* ping 命令执行入口函数
void nvt_cmd_ping_entry(void *pvParam)
{
    ST_PING_STARTARGS stArgs = *((PST_PING_STARTARGS)pvParam); //* 建立启动参数副本
    ((PST_PING_STARTARGS)pvParam)->bIsCpyEnd = TRUE; //* 通知命令入口函数副本建立完毕

    //* 调用 ping_start()启动 ping 探测
    ……
```

```
    //* 主循环开始连续发送 icmp/icmpv6 echo 报文
    UCHAR ubRcvBuf[4];
    INT nRcvBytes;
    while (nvt_cmd_exec_enable(stArgs.ullNvtHandle))
    {
        //* 读取 telnet 登录用户的终端输入
        nRcvBytes = nvt_input(stArgs.ullNvtHandle, ubRcvBuf, sizeof(ubRcvBuf));
        if (nRcvBytes)
        {
            //* 是否收到了用户输入的“ctrl+c”结束信号，收到了则退出循环结束 ping，其它输入不做任何处理，直接丢弃
            if (nRcvBytes == 1 && ubRcvBuf[0] == '\x03')
            {
                nvt_output(stArgs.ullNvtHandle, "\r\n", 2);
                break;
            }
        }

        //* ping 探测代码
        ……
    }

    //* 调用 ping_end()函数结束 ping
    ……

    //* 调用 nvt_output()/nvt_outputf()函数输出 ping 统计结果
    ……

    nvt_cmd_exec_end(stArgs.ullNvtHandle); //* 显式地通知 NVT——“我”以结束执行
    nvt_cmd_thread_end();  //* 显式地通知 NVT——“我”作为线程/任务的身份也已结束，也就是线程/任务已安全退出
}
```

上述代码中涉及到的几个以“nvt_”为前缀的函数中，最为核心的是 nvt_cmd_exec_enable()、nvt_cmd_exec_end()、nvt_cmd_thread_end()这三个（红色粗体标注部分）。它们负责控制命令的正常执行。其中 nvt_cmd_exec_enable()函数由 NVT 使用，其可以随时停止命令的执行，**所以它必须被用作主循环继续进行的必要条件**。另外两个则被用于显式地通知 NVT——“我”已结束运行，可以让终端用户输入下一个命令了。虽然都是通知结束，但在用途上还是有区别的：NVT 收到 nvt_cmd_exec_end()函数的通知后，会立即返回主处理逻辑，允许用户输入下一个命令；而收到 nvt_cmd_thread_end()通知则意味着可以执行下一个阻塞型命令了。有关它们的详细说明请参阅《onps 栈移植手册》5.3 节。

nvt_input()函数的作用是从远程终端读取用户输入，对于 ping 命令我们只需关注用户是否按下了“ctrl+c”组合键即可，一旦检测到该组合键被按下，则立即停止 ping，调用 nvt_output()/nvt_outputf()函数输出最终的统计结果。

7.2 编写非阻塞型命令

如前所述，非阻塞型命令只需编写一个命令入口函数即可。其编写规则也很简单，只需在入口函数的尾部调用 nvt_cmd_exec_end()函数后再返回即可：

```
//* 命令入口函数
```

```
INT my_nvt_cmd(CHAR argc, CHAR* argv[], ULONGLONG ullNvtHandle)
{
    //* 你的功能实现代码
    ……


    nvt_cmd_exec_end(ullNvtHandle);
    return 0;
}
```

7.3 注册命令

自定义的命令需要通过 nvt_cmd_add()函数注册到协议栈。该函数的详细使用说明请参阅《onps 栈 API 接口手册》第 4 节或《onps 栈移植手册》5.3 节，不再赘述。