实验二十二：LAN_UDP 实验——以太网数据传输

一、 实验目的与意义

1、 了解 LwIP 协议栈和 LAN8720 物理层

2、 了解 UCOSII 的使用方法

3、 掌握 UDP 的使用方法

4、 掌握 STM32 HAL 库中 UDP 属性的配置方法

5、 掌握 KEIL MDK 集成开发环境使用方法

二、 实验设备及平台

1、 iCore4 双核心板

2、 JLINK（或相同功能）仿真器

3、 Micro USB 线缆

4、 网线

5、 Keil MDK 开发平台

6、 装有 WIN XP（及更高版本）系统的计算机

三、 实验原理

1、LwIP 简介

LwIP 是 Light Weight (轻型)IP 协议，有无操作系统的支持都可以运行。LwIP 实现的重点是在保持 TCP 协议主要功能的基础上减少对 RAM 的占用，它只需十几 KB 的 RAM 和 40K 左右的 ROM 就可以运行，这使 LwIP 协议栈适合在低端的嵌入式系统中使用。

LwIP 协议栈主要关注的是怎么样减少内存的使用和代码的大小，这样就可以让 LwIP 适用于资源有限的小型平台例如嵌入式系统。为了简化处理过程和内存要求，LwIP 对 API 进行了裁减，可以不需要复制一些数据。

LwIP 提供三种 API：1)RAW API 2)LwIP API 3)BSD API。

RAW API 把协议栈和应用程序放到一个进程里边，该接口基于函数回调技术，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。这既有优点也有缺点。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。

LwIP API 把接收与处理放在一个线程里面。这样只要处理流程稍微被延迟，接收就会被阻塞，直接造成频繁丢包、响应不及时等严重问题。因此，接收与协议处理必须 分开。LwIP 的作者显然已经考虑到了这一点，他为我们提供了 tcpip_input() 函数来处理这个问题，虽然他并没有在 rawapi 一文中说明。讲到这里，读者应该知道 tcpip_input()函数投递的消息从哪里来的答案了吧，没错，它们来自于由底层网络驱动组成的接收线程。我们在编写网络驱动时， 其接收部分以任务的形式创建。 数据包到达后， 去掉以太网包头得到 IP 包，然后直接调用 tcpip_input()函数将其 投递到 mbox 邮箱。投递结束，接收任务继续下一个数据包的接收，而被投递得 IP 包将由 TCPIP 线程继续处理。这样，即使某个 IP 包的处理时间过长也不 会造成频繁丢包现象的发生。这就是 LwIP API。

BSD API 提供了基于 open-read-write-close 模型的 UNIX 标准 API，它的最大特点是使应用程序移植到其它系统时比较容易，但用在嵌入式系统中效率比较低，占用资源多。这对于我们的嵌入式应用有时是不能容忍的。

其主要特性如下：

(1) 支持多网络接口下的 IP 转发；

(2) 支持 ICMP 协议；

(3) 包括实验性扩展的 UDP(用户数据报协议)；

(4) 包括阻塞控制、RTT 估算、快速恢复和快速转发的 TCP(传输控制协议)；

(5) 提供专门的内部回调接口(Raw API)，用于提高应用程序性能；

(6) 可选择的 Berkeley 接口 API (在多线程情况下使用) ；

(7) 在最新的版本中支持 ppp；

(8) 新版本中增加了的 IP fragment 的支持；

(9) 支持 DHCP 协议,动态分配 ip 地址。

2、UDP 简介

UDP 是 User Datagram Protocol 的简称，中文名是用户数据报协议，是 OSI（Open System Interconnection，开放式系统互联） 参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务，IETF RFC 768 是 UDP 的正式规范。UDP 在 IP 报文的协议号是 17。

UDP 协议与 TCP 协议一样用于处理数据包，在 OSI 模型中，两者都位于传输层，处于 IP 协议的上一层。UDP 有不提供数据包分组、组装和不能对数据包进行排序的缺点，也就是说，当报文发送之后，是无法得知其是否安全完整到达的。UDP 用来支持那些需要在计算机之间传输数据的网络应用。包括网络视频会议系统在内的众多的客户/服务器模式的网络应用都需要使用 UDP 协议。UDP 协议从问世至今已经被使用了很多年，虽然其最初的光彩已经被一些类似协议所掩盖，但即使在今天 UDP 仍然不失为一项非常实用和可行的网络传输层协议。

许多应用只支持 UDP，如：多媒体数据流，不产生任何额外的数据，即使知道有破坏的包也不进行重发。当强调传输性能而不是传输的完整性时，如：音频和多媒体应用，UDP 是最好的选择。在数据传输时间很短，以至于此前的连接过程成为整个流量主体的情况下，UDP 也是一个好的选择。

UDP 是 OSI 参考模型中一种无连接的传输层协议，它主要用于不要求分组顺序到达的传输中，分组传输顺序的检查与排序由应用层完成，提供面向事务的简单不可靠信息传送服务。UDP 协议基本上是 IP 协议与上层协议的接口。UDP 协议适用端口分别运行在同一台设备上的多个应用程序。

UDP 提供了无连接通信，且不对传送数据包进行可靠性保证，适合于一次传输少量数据，UDP 传输的可靠性由应用层负责。常用的 UDP 端口号有：53（DNS）、69（TFTP）、161（SNMP），使用 UDP 协议包括：TFTP、SNMP、NFS、DNS、BOOTP。

UDP 报文没有可靠性保证、顺序保证和流量控制字段等，可靠性较差。但是正因为 UDP 协议的控制选项较少，在数据传输过程中延迟小、数据传输效率高，适合对可靠性要求不高的应用程序，或者可以保障可靠性的应用程序，如 DNS、TFTP、SNMP 等。

3、UDP 的主要特点

UDP 是一个无连接协议，传输数据之前源端和终端不建立连接，当它想传送时就简单地去抓取来自应用程序的数据，并尽可能快地把它扔到网络上。在发送端，UDP 传送数据的速度仅仅是受应用程序生成数据的速度、计算机的能力和传输带宽的限制；在接收端，

UDP 把每个消息段放在队列中，应用程序每次从队列中读一个消息段。

由于传输数据不建立连接，因此也就不需要维护连接状态，包括收发状态等，因此一台服务机可同时向多个客户机传输相同的消息。

UDP 信息包的标题很短，只有 8 个字节，相对于 TCP 的 20 个字节信息包而言 UDP 的额外开销很小。

吞吐量不受拥挤控制算法的调节，只受应用软件生成数据的速率、传输带宽、源端和终端主机性能的限制。

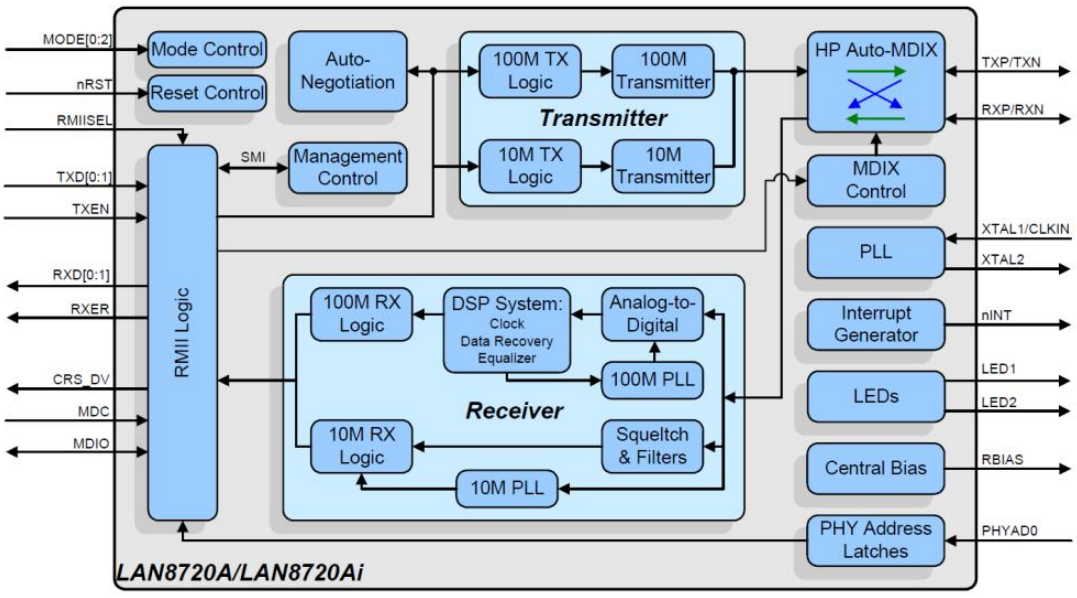
UDP 是面向报文的。发送方的 UDP 对应用程序交下来的报文，在添加首部后就向下交付给 IP 层。既不拆分，也不合并，而是保留这些报文的边界，因此，应用程序需要选择合适的报文大小。

虽然 UDP 是一个不可靠的协议，但它是分发信息的一个理想协议。例如，在屏幕上报告股票市场、显示航空信息等等。UDP 也用在路由信息协议 RIP（Routing Information Protocol）中修改路由表。在这些应用场合下，如果有一个消息丢失，在几秒之后另一个新的消息就会替换它。UDP 广泛用在多媒体应用中。

DMA(直接存储器访问)传输不需要占用 CPU，可以在存储器至存储器实现高速的数据传输。本实验采用 DMA2 控制器的数据流 0，选用通道 0 进行数据传输。通过 LED 的颜色来判断传输是否成功。

4、LAN8720A 简介

LAN8720A 功能框图如图所示：



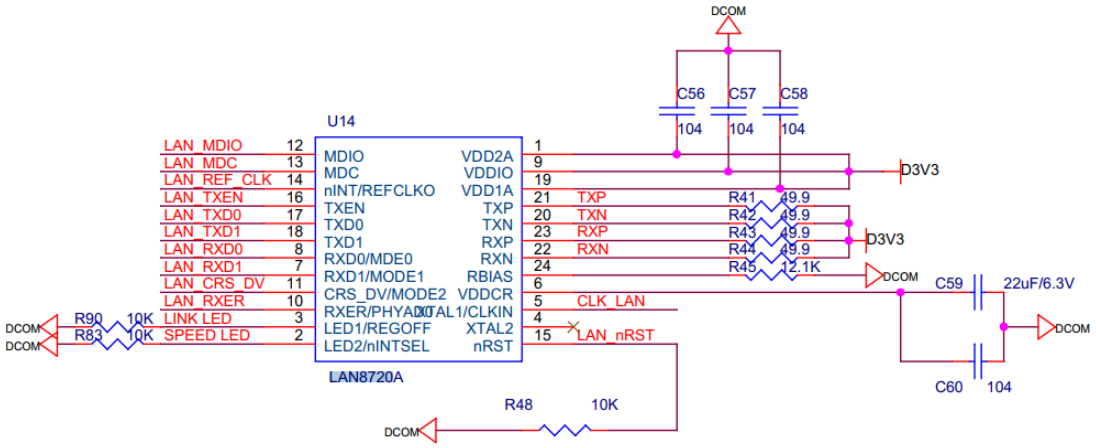
LAN8720A 是低功耗的 10/100M 以太网 PHY 层芯片，I/O 引脚电压符合 IEEE802.3-2005 标准，支持通过 RMII 接口与以太网 MAC 层通信，内置 10-BASE-T/100BASE-TX 全双工传输模块，支持 10Mbps 和 100Mbps。

LAN8720A 可以通过自协商的方式与目的主机最佳的连接方式(速度和双工模式)，支持 HPAuto-MDIX 自动翻转功能，无需更换网线即可将连接更改为直连或交叉连接。LAN8720A 的主要特点如下：

- 高性能的 10/100M 以太网传输模块
- 支持 RMII 接口以减少引脚数
- 支持全双工和半双工模式
- 两个状态 LED 输出
- 可以使用 25M 晶振以降低成本
- 支持自协商模式
- 支持 HPAuto-MDIX 自动翻转功能
- 支持 SMI 串行管理接口
- 支持 MAC 接口

5、原理图

iCore4 带有 LAN8720A 嵌入式以太网控制器，本实验实现 UDP 功能。PC 的 IP 地址 192.168.0.2， iCore4 的 IP 地址为 192.168.0.10，端口号为 60000。上电即可进行数据信息传输。原理图如下：



四、 实验程序

1、主函数

```
int main(void)
{
    system_clock.initialize();        //系统时钟初始化
    led.initialize();                 //LED 初始化
    adc.initialize();                 //ADC 初始化
    delay.initialize(216);            //延时初始化
    my_malloc.initialize(SRAMIN);      //动态内存初始化
    usart6.initialize(115200);        //串口波特设置
    usart6.printf("\033[1;32;40m");      //设置字体终端为绿色
    usart6.printf("\r\nHello, I am iCore4!\r\n\r\n");  //串口信息输出

    OSInit();         //UCOS 初始化

    while(lwip.initialize())            //lwip 初始化
    {
     LED_RED_ON;
        usart6.printf("\r\nETH initialize error!\r\n\r\n");//ETH 初始化失败
    }
    udp.initialize();       //modbus_tcp 初始化

    OSTaskCreate(start_task,(void*)0,(OS_STK*)&START_TASK_STK[START_STK_S
IZE-1],START_TASK_PRIO);
    OSStart();          //开启 UCOS
}
```

2、内存管理初始化

```
void my_mem_init(u8 memx)
{
    mymemset(mallco_dev.memmap[memx],0,memtblsize[memx]*4);
     //内存状态表数据清零
     mallco_dev.memrdy[memx]=1;      //内存管理初始化 OK
}
```

3、LwIP 初始化

```
//LWIP 初始化(LWIP 启动的时候使用)
//返回值:0,成功
//       1,内存错误
//       2,LAN8720 初始化失败
//       3,网卡添加失败.
u8 lwip_comm_init(void)
{
    OS_CPU_SR cpu_sr;
    struct netif *Netif_Init_Flag;    //调用 netif_add()函数时的返回值,用于判断网络初始化是否成功
    struct ip_addr ipaddr;          //ip 地址
    struct ip_addr netmask;        //子网掩码
    struct ip_addr gw;             //默认网关
    if(lan8720.memory_malloc())return 1;     //内存申请失败
    if(lwip_comm_mem_malloc())return 1;     //内存申请失败
    if(lan8720.initialize())return 2;       //初始化 LAN8720 失败
    tcpip_init(NULL,NULL);          //初始化 tcp ip 内核,该函数里面会创建 tcpip_thread 内核任务
    lwip_comm_default_ip_set(&lwipdev); //设置默认 IP 等信息
#if LWIP_DHCP     //使用动态 IP
    ipaddr.addr = 0;
   netmask.addr = 0;
   gw.addr = 0;
#else         //使用静态 IP
    IP4_ADDR(&ipaddr,lwipdev.ip[0],lwipdev.ip[1],lwipdev.ip[2],lwipdev.ip[3]);
    IP4_ADDR(&netmask,lwipdev.netmask[0],lwipdev.netmask[1] ,lwipdev.netmask[2],lwipdev.netmask[3]);
    IP4_ADDR(&gw,lwipdev.gateway[0],lwipdev.gateway[1],lwipdev.gateway[2],lwipdev.gateway[3]);
    usart6.printf("网卡 en 的 MAC 地址为:.....%d.%d.%d.%d.%d.%d\r\n",lwipdev.mac[0],lwipdev.mac[1],lwipdev.mac[2],lwipdev.mac[3],lwipdev.mac[4],lwipdev.mac[5]);
    usart6.printf("静态 IP 地址.....%d.%d.%d.%d\r\n",lwipdev.ip[0],lwipdev.ip[1],lwipdev.ip[2],lwipdev.ip[3]);
```

```
    usart6.printf("子网掩码.....................%d.%d.%d.%d\r\n",lwip
dev.netmask[0],lwipdev.netmask[1],lwipdev.netmask[2],lwipdev.netmask[3]);
    usart6.printf("默认网关.....................%d.%d.%d.%d\r\n",lwip
dev.gateway[0],lwipdev.gateway[1],lwipdev.gateway[2],lwipdev.gateway[3]);
#endif
    Netif_Init_Flag=netif_add(&lwip_netif,&ipaddr,&netmask,&gw,NULL,&ethe
rnetif_init,&tcpip_input);//向网卡列表中添加一个网口

#if LWIP_DNS
    dns_init();
#endif

    if(Netif_Init_Flag==NULL)return 3;//网卡添加失败
    else//网口添加成功后,设置 netif 为默认值,并且打开 netif 网口
    {
        netif_set_default(&lwip_netif); //设置 netif 为默认网口
        netif_set_up(&lwip_netif);     //打开 netif 网口
    }
    return 0;//操作 OK.
}
```

4、UDP 初始化

```
static INT8U udp_demo_init(void)//创建 UDP 线程
{
    INT8U res;
    OS_CPU_SR cpu_sr;

    OS_ENTER_CRITICAL();     //关中断
    res = OSTaskCreate(udp_thread,(void*)0,(OS_STK*)&UDP_TASK_STK[UDP_STK
_SIZE-1],UDP_PRIO); //创建 UDP 线程
    OS_EXIT_CRITICAL();     //开中断

    return res;//返回值:0 UDP 创建成功
}
```

5、UDP 任务函数

```
static void udp_thread(void *arg)//udp 任务函数
{
    err_t err,recv_err;
    struct netconn *udpconn;
    struct netbuf  *udprecvbuf;
    struct ip_addr udp_destipaddr;
```

```
    LWIP_UNUSED_ARG(arg);
    udpconn = netconn_new(NETCONN_UDP);  //创建一个 UDP 链接

    if(udpconn != NULL)  //创建 UDP 连接成功
    {
        err = netconn_bind(udpconn,IP_ADDR_ANY,UDP_REMOTE_PORT);
        IP4_ADDR(&udp_destipaddr,lwipdev.remoteip[0],lwipdev.remoteip[1],
 lwipdev.remoteip[2],lwipdev.remoteip[3]); //构造目的 IP 地址
        netconn_connect(udpconn,&udp_destipaddr,UDP_LOUCAL_PORT);  //连接
到远端电脑主机端口 : 60002
        if(err == ERR_OK)//绑定完成
        {
            udpconn->recv_timeout = 10;
          while(1)
          {
        recv_err = netconn_recv(udpconn,&udprecvbuf);
                if((recv_err == ERR_OK)||(udprecvbuf != NULL)) //接收到数据
                {
                  recv_err = netconn_send(udpconn,udprecvbuf);
                    OSTimeDlyHMSM(0,0,0,5);
                  netbuf_delete(udprecvbuf);
                }
                OSTimeDlyHMSM(0,0,0,5);
            }
        }else usart6.printf("UDP 绑定失败\r\n");
    }else usart6.printf("UDP 连接创建失败\r\n");
}
```

五、 实验步骤

1、 把仿真器与 iCore4 的 SWD 调试口相连（直接相连或者通过转接器相连）；

2、 将跳线冒插在 USB UART；

3、 把 iCore4（USB UART）通过 Micro USB 线与计算机相连，为 iCore4 供电；

4、 把 iCore4 网口通过网线与计算机网口相连；

5、 打开 Keil MDK 开发环境，并打开本实验工程；

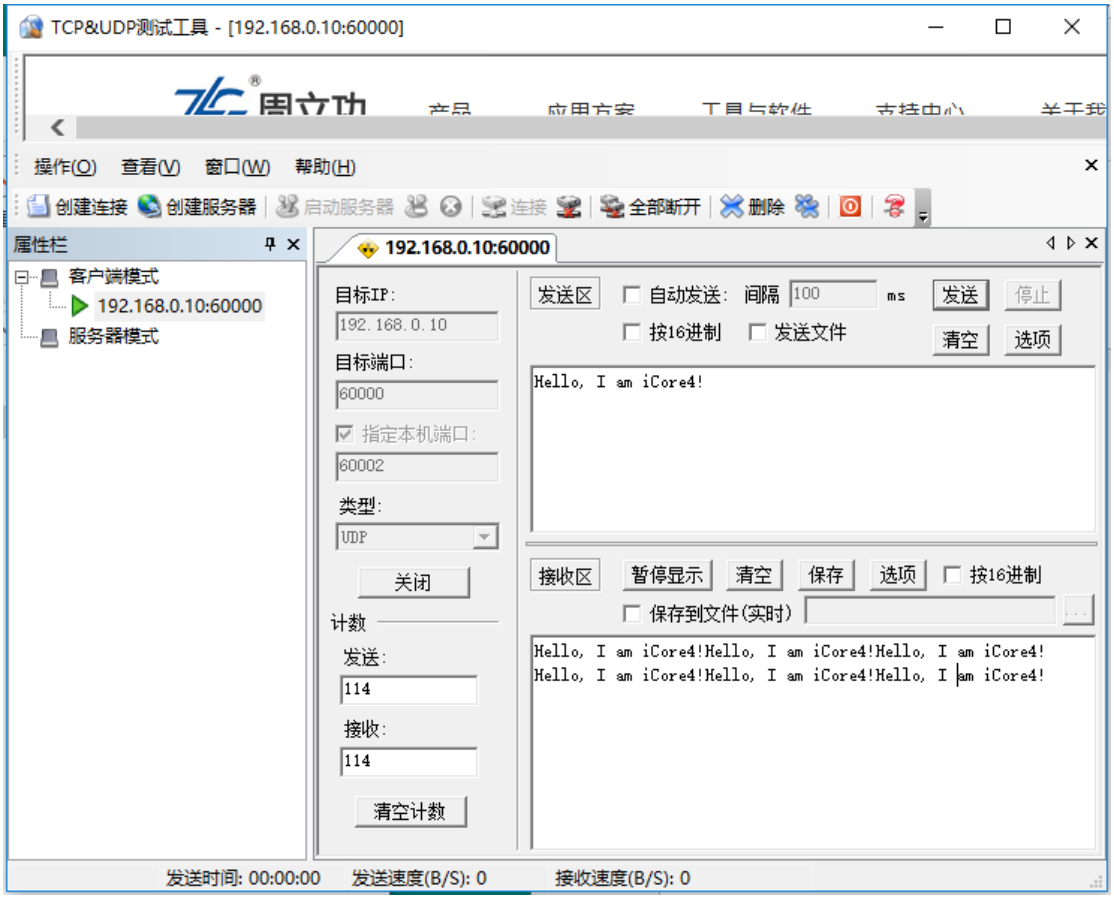
6、 打开 TCP&UDP 测试工具；（安装及使用方法见附录）；

7、 烧写程序到 iCore4 上；

8、 也可以进入 Debug 模式，单步运行或设置断点验证程序逻辑。

六、 实验现象

在发送区编辑完要发送的数据信息后，点击发送即可收到发送的数据包。如图所示：



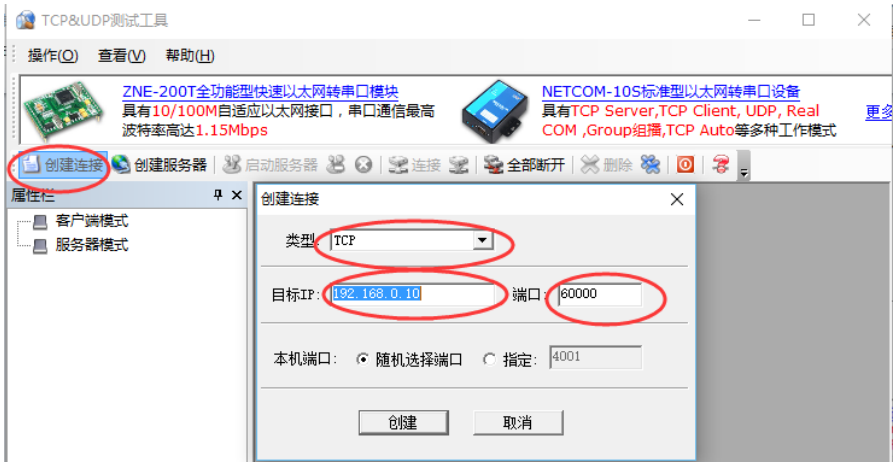
附录：

1、 TCP&UDP 测试工具安装

双击 TCPUDPDebug102_Setup.exe，点击下一步，在这里安装路径我们默认即可，点击安装，然后 Finish。

2、 TCP&UDP 测试工具的使用

（1）打开测试工具，界面如下。点击创建连接，弹出了设置端口的窗口，端口设置为 60000。



（2）连接已经创建完成（如下图），点击连接

（3）PC 客户端连接服务器后，即可进行通信。