# 原始套接字

对于标准的套接字，通常数据按照选定的传输层协议(例如TCP、UDP)自动封装，socket用户并不知道在网络介质上广播的数据包含了这种协议包头。

从原始套接字读取数据包含了传输层协议包头。用原始套接字发送数据，是否自动增加传输层协议包头是可选的。

原始套接字用于安全相关的应用程序，如nmap。原始套接字一种可能的用例是在用户空间实现新的传输层协议。[1] 原始套接字常在网络设备上用于路由协议，例如IGMPv4、开放式最短路径优先协议(OSPF)、互联网控制消息协议 (ICMP)。Ping就是发送一个ICMP响应请求包然后接收ICMP响应回复.

大部分套接字API都支持原始套接字功能。Winsock自2001年起在Windows XP上支持原始套接字。但由于安全原因，2004年微软限制了Winsock的原始套接字功能。

# 例子

Linux利用原始套接字上实现**ping**命令

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>
#include <netdb.h>

/* 校验和計算 */
u_int16_t checksum(unsigned short *buf, int size)
{
    unsigned long sum = 0;
    while (size > 1) {
        sum += *buf;
        buf++;
        size -= 2;
    }
    if (size == 1)
        sum += *(unsigned char *)buf;
    sum = (sum & 0xffff) + (sum >> 16);
    sum = (sum & 0xffff) + (sum >> 16);
    return ~sum;
}

/* protocol指定的raw socket创建 */
int make_raw_socket(int protocol)
{
    int s = socket(AF_INET, SOCK_RAW, protocol);
```

```
    if (s < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    return s;
}

/* ICMP头部的作成 */
void setup_icmphdr(u_int8_t type, u_int8_t code, u_int16_t id, u_int16_t seq,
struct icmphdr *icmphdr)
{
    memset(icmphdr, 0, sizeof(struct icmphdr));
    icmphdr->type = type;
    icmphdr->code = code;
    icmphdr->checksum = 0;
    icmphdr->un.echo.id = id;
    icmphdr->un.echo.sequence = seq;
    icmphdr->checksum = checksum((unsigned short *)icmphdr, sizeof(struct
icmphdr));
}

int main(int argc, char **argv)
{
    int n, soc;
    char buf[1500];
    struct sockaddr_in addr;
    struct in_addr insaddr;
    struct icmphdr icmphdr;
    struct iphdr *recv_iphdr;
    struct icmphdr *recv_icmphdr;

    if (argc < 2) {
        printf("Usage : %s IP_ADDRESS\n", argv[0]);
        return 1;
    }

    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr(argv[1]);
    soc = make_raw_socket(IPPROTO_ICMP);
    setup_icmphdr(ICMP_ECHO, 0, 0, 0, &icmphdr);

    /* ICMP包的送信 */
    n = sendto(soc, (char *)&icmphdr, sizeof(icmphdr), 0, (struct sockaddr
*)&addr, sizeof(addr));
    if (n < 1) {
        perror("sendto");
        return 1;
    }

    /* ICMP包的受信 */
    n = recv(soc, buf, sizeof(buf), 0);
    if (n < 1) {
        perror("recv");
        return 1;
    }

    recv_iphdr = (struct iphdr *)buf;
    /* 从IP包头获取IP包的长度，从而确定icmp包头的开始位置 */
```

```
    recv_icmphdr = (struct icmphdr *)(buf + (recv_iphdr->ihl << 2));
    insaddr.s_addr = recv_iphdr->saddr;
    /* 检查送信包的源地址匹配受信包的目的地址  */
    if (!strcmp(argv[1], inet_ntoa(insaddr)) && recv_icmphdr->type ==
ICMP_ECHOREPLY)
        printf("icmp echo reply from %s\n", argv[1]);
    close(soc);
    return 0;
}
```

# 同步和异步

> 同步：在发出一个同步调用时，在没有得到结果之前，该调用就不返回。
>
> 异步：在发出一个异步调用后，调用者不会立刻得到结果，该调用就返回了。

## 同步例子

```
int n = func();
next();
// func() 的结果没有返回，next() 就不会执行，直到 func() 运行完。
```

## 异步例子

```
func(callback);
next();
...

void callback(int n)      // func 结果回调
{
    int k = n;
}
// func() 执行后，还没得出结果就立即返回，然后执行 next() 了
// 等到结果出来，func() 回调 callback() 通知调用者结果。
```

> 同步阻塞调用：得不到结果不返回，线程进入阻塞态等待。
>
> 同步非阻塞调用：得不到结果不返回，线程不阻塞一直在CPU运行。
>
> 异步阻塞调用：去到别的线程，让别的线程阻塞起来等待结果，自己不阻塞。
>
> 异步非阻塞调用：去到别的线程，别的线程一直在运行，直到得出结果。