# netlink 编程介绍(V0.2)

作者: Hoyt Luo <hoytluo@21cn.com>

Linux从2.2开始支持PF_NETLINK域的通讯方式，这个方式主要的用途是在Linux的内核空间和用户空间进行通讯。目前在网络上面关于netlink编程的中文资料很少，为了促进对netlink编程的理解我编写了这篇文章，由于我对netlink的了解不是很透彻，特别是对于内核部分不是很熟悉，所以文章中肯定有很多错误的地方还请大家指正。文章分下面几个部分进行讲述



# A. netlink基础知识

我们在使用socket(2)的man手册时候可以找到man手册中有下面一行说明

```
PF_NETLINK          Kernel user interface device     netlink(7)
```

在我们通过PF_NETLINK创建一个SOCKET以后表示我们期望同内核进行消息通讯。使用netlink(7)的手册可以看到关于PF_NETLINK的详细说明。

```
#include <asm/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>

netlink_socket = socket(PF_NETLINK, socket_type, netlink_family);
```

按照netlink的手册，socket_type可以取SOCK_RAW和SOCK_DGRAM，不过内核不区分这两个字段。netlink_family字段指定了我们期望的通讯协议，主要有：

- NETLINK_ROUTE
  用来获取，创建和修改设备的各种信息，详细参见 rtnetlink(7)
- NETLINK_SKIP
  Enskip 的保留选项
- NETLINK_USERSOCK
  为今后用户程序空间协议用保留选项
- NETLINK_FIREWALL

接收 IPv4 防火墙编码发送的数据包
- NETLINK_TCPDIAG
  TCP套接字监控
- NETLINK_NFLOG
  netfilter的用户空间日志
- NETLINK_ARPD
  用以维护用户地址空间里的 arp 表
- NETLINK_ROUTE6
  接收和发送 IPv6 路由表更新消息
- NETLINK_IP6_FW
  接收未通过 IPv6 防火墙检查的数据包(尚未实现)
- NETLINK_TAPBASE
  是 ethertap 设备实例

后面我们会对每一个协议进行解释和说明.

# B. nlmsghdr结构介绍

每一个发送给内核或者从内核介绍的报文都有一个相同的报文头，这个报文头的结构如下定义：

```
struct nlmsghdr
{
        __u32 nlmsg_len;     /* 包括报头在内的消息长度*/
        __u16 nlmsg_type;    /* 消息正文 */
        __u16 nlmsg_flags;   /* 附加标志*/
        __u32 nlmsg_seq;     /* 序列号*/
        __u32 nlmsg_pid;     /* 发送进程号 PID */
};
```

所有发送给内核或者内核的报文的第一部分都必须使用这个机构，后面跟随相应的内容。nlmsg_type为后面消息的内容个数，对于前面我们提到的不同通讯协议有着不同的消息类型。下面是三个通用的消息类型

- NLMSG_NOOP
  这个消息类型表示消息内容为空，应用可以忽略该报文
- NLMSG_ERROR
  这个消息类型表示后面的消息是一个错误信息，错误信息的机构为nlmsgerr

```
struct nlmsgerr
{
    int error;           /* 负数表示的出错号 errno 或为 0 要求确认 acks*/
    struct nlmsghdr msg; /* 造成出错的消息报头*/
};
```

- NLMSG_DONE
  在我们接收或者发送消息给内核的时候，我们有可能一次发送多个报文，这个消息类型表示是报文的最后一个，类似于在链表中我们将最后一个成员的next指针设置为NULL。

附加的标志用于控制或者表示消息的其它信息，一些比较通用的标志是
- NLM_F_REQUEST
  表示这个消息是一个请求消息，这个消息可以同以下一个标志组合
  - NLM_F_ROOT
    返回树的根
  - NLM_F_MATCH
    返回所有匹配的
  - NLM_F_ATOMIC
    返回对象表的单一快照
  - NLM_F_DUMP
    被定义为NLM_F_ROOT|NLM_F_MATCH
  - NLM_F_REPLACE
    表示替换现有的规则
  - NLM_F_EXCL
    如果现有规则存在则不修改
  - NLM_F_CREAT

创建一个规则
- NLM_F_APPEND
  追加一个规则

- NLM_F_MULTI
  表示这个消息是多个报文中的一个，报文的结尾通过NLMSG_DONE来表示
- NLM_F_ACK
  表示这个消息是一个应答消息
- NLM_F_ECHO
  表示这个消息是一个要求返回请求信息的消息

# C. 解析nlmsghdr数据

为了获取netlink报文中数据的方便，netlink提供了下面几个宏进行数据的获取和解包操作

```
#include <asm/types.h>
#include <linux/netlink.h>
int NLMSG_ALIGN(size_t len);
int NLMSG_LENGTH(size_t len);
int NLMSG_SPACE(size_t len);
void *NLMSG_DATA(struct nlmsghdr *nlh);
struct nlmsghdr *NLMSG_NEXT(struct nlmsghdr *nlh, int len);
int NLMSG_OK(struct nlmsghdr *nlh, int len);
int NLMSG_PAYLOAD(struct nlmsghdr *nlh, int len);
```

NLMSG_ALIGN:进行数据长度的对齐操作
NLMSG_DATA:获取通讯报文中的数据
NLMSG_NEXT:获取下一个报文
NLMSG_OK:判断是否数据可以继续获取
NLMSG_PAYLOAD:获取数据的长度
在我们后面的实例中会介绍如何使用这几个宏。

# D. sockaddr_nl结构介绍

在socket程序中，如果我们要求接收报文则要求调用bind，表示我们期望接收什么样的报文。对于netlink也一样，我们要求指定我们期望接收的地址信息，不过同传统的sockaddr不同，这个地方是一个sockaddr_nl的结构:

```
struct sockaddr_nl
{
        sa_family_t nl_family; /* AF_NETLINK */
        unsigned short nl_pad; /* 用来填充的字段，赋值为0 */
        pid_t nl_pid; /* 进程标识号pid */
        __u32 nl_groups; /* 多址广播组掩码*/
};
```

每一个 netlink 数据类都有一个32位广播分组，当对套接字调用 bind(2) 时， sockaddr_nl 中的 nl_groups 字段设置成所要侦听的广播组的位掩码。其默认值为 0，表示不接收任何广播,我们会在后面中看到如何使用这个广播组的例子。

# E. NETLINK_ROUTE协议介绍

netlink目前使用最广泛的是通过这个选项来获取网络设备或者网址的一些信息。在使用这个协议时候支持的类型有:

- RTM_NEWLINK, RTM_DELLINK, RTM_GETLINK
  创建，删除或者获取网络设备的信息
- RTM_NEWADDR, RTM_DELADDR, RTM_GETADDR
  创建，删除或者获取网络设备的IP信息
- RTM_NEWROUTE, RTM_DELROUTE, RTM_GETROUTE

创建，删除或者获取网络设备的路由信息
- RTM_NEWNEIGH, RTM_DELNEIGH, RTM_GETNEIGH
  创建，删除或者获取网络设备的相邻信息
- RTM_NEWRULE, RTM_DELRULE, RTM_GETRULE
  创建，删除或者获取路由规则信息
- RTM_NEWQDISC, RTM_DELQDISC, RTM_GETQDISC
  创建，删除或者获取队列的原则
- RTM_NEWTCLASS, RTM_DELTCLASS, RTM_GETTCLASS
  创建，删除或者获取流量的类别
- RTM_NEWTFILTER, RTM_DELTFILTER, RTM_GETTFILTER
  创建，删除或者获取流量的过虑

由于NETLINK_ROUTE支持的类型实在太多了，我们这个地方只重点介绍一下RTM_GETLINK这个类型，然后通过一个例子介绍如何同内核进行通讯。关于其它类型的用法大家可以参考rtnetlink(7)的man手册。

按照rtnetlink的说法，在获取设备信息的时候我们要求首先发送一个报文给内核表示我们的请求，这个报文的格式是1个nlmsghdr头部机构+1个ifinfomsg接口结构+多个rtattr属性机构。其中后面两个结构的定义是:

```
struct ifinfomsg
{
    unsigned char  ifi_family;  /* AF_UNSPEC */
    unsigned short ifi_type;    /* Device type */
    int            ifi_index;   /* Interface index   */
    unsigned int   ifi_flags;   /* Device flags  */
    unsigned int   ifi_change;  /* change mask */
};
struct rtattr
{
    unsigned short rta_len;     /* Length of option */
    unsigned short rta_type;    /* Type of option */
    /* Data follows */
};
```

ifi_type:这个字段包含了硬件的类型,可以参考<linux/if_arp.h>比较常见的是

```
ARPHRD_ETHER    10M以太网
ARPHRD_PPP      PPP拨号
ARPHRD_LOOPBACK 环路设备
```

ifi_flags:这个字段包含了设备的一些标志，相应的值为:

```
IFF_UP              接口正在运行.
IFF_BROADCAST       有效的广播地址集.
IFF_DEBUG           内部调试标志.
IFF_LOOPBACK        这是自环接口.
IFF_POINTOPOINT     这是点到点的链路接口.
IFF_RUNNING         资源已分配.
IFF_NOARP           无arp协议，没有设置第二层目的地址.
IFF_PROMISC         接口为杂凑(promiscuous)模式.
IFF_NOTRAILERS      避免使用trailer .
IFF_ALLMULTI        接收所有组播(multicast)报文.
IFF_MASTER          主负载平衡群(bundle).
IFF_SLAVE           从负载平衡群(bundle).
IFF_MULTICAST       支持组播(multicast).
IFF_PORTSEL         可以通过ifmap选择介质(media)类型.
IFF_AUTOMEDIA       自动选择介质.
IFF_DYNAMIC         接口关闭时丢弃地址.
```

rta_type:这个字段指定属性的类型，相应的值为：

```
IFLA_UNSPEC 后面的数据格式未指定
IFLA_ADDRESS 后面的数据是一个硬件地址
IFLA_BROADCAST 后面的数据是一个硬件广播地址
IFLA_IFNAME 后面的数据是一个char型的设备名称
IFLA_MTU unsigned int型的设备MTU值
```

IFLA_LINK int型的链路类型
IFLA_QDISC 字符串型的队列规则
IFLA_STATS struct net_device_stats型的设备信息

在我们接收和发送数据的时候为了我们获取属性的方便，rtnetlink提供了一个宏供我们取获取其中的结构，这些宏的定义为：

```
#include <asm/types.h>
#include <linux/netlink.h>
#include <linux/rtnetlink.h>
#include <sys/socket.h>

int RTA_OK(struct rtattr *rta, int rtabuflen);
void *RTA_DATA(struct rtattr *rta);
unsigned int RTA_PAYLOAD(struct rtattr *rta);
struct rtattr *RTA_NEXT(struct rtattr *rta, unsigned int rtabuflen);
unsigned int RTA_LENGTH(unsigned int length);
unsigned int RTA_SPACE(unsigned int length);
```

下面是一个用来获取所有网卡信息的例子.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <asm/types.h>
#include <linux/netlink.h>
#include <linux/rtnetlink.h>
#include <linux/netdevice.h>
#include <net/if_arp.h>
#include <netinet/if_ether.h>
#include <netinet/ether.h>

int main()
{
    int nSocket, nLen, nAttrLen;
    char szBuffer[4096];
    struct {
        struct nlmsghdr nh;
        struct ifinfomsg ifi;
    }struReq;
    struct sockaddr_nl struAddr;
    struct nlmsghdr *pstruNL;
    struct ifinfomsg *pstruIF;
    struct rtattr *pstruAttr;
    struct net_device_stats *pstruInfo;
    struct ether_addr *pstruEther;

    /*
     * 创建一个PF_NETLINK的SOCKET,使用NETLINK_ROUTE协议
     */
    nSocket = socket(PF_NETLINK, SOCK_DGRAM, NETLINK_ROUTE);
    if(nSocket < 0)
    {
        fprintf(stderr, "创建SOCKET错误:%s\n", strerror(errno));
        return -1;
    }

    /*
     * 绑定地址
     */
    memset(&struAddr, 0, sizeof(struAddr));
    struAddr.nl_family = AF_NETLINK;
```

```
    struAddr.nl_pid = getpid();
    struAddr.nl_groups = 0;
    if(bind(nSocket, (struct sockaddr *)&struAddr, sizeof(struAddr)) < 0)
    {
        fprintf(stderr, "绑定SOCKET错误:%s\n", strerror(errno));
        return -1;
    }

    /*
     * 发送一个请求
     */
    memset(&struReq, 0, sizeof(struReq));
    struReq.nh.nlmsg_len = NLMSG_LENGTH(sizeof(struReq));
    struReq.nh.nlmsg_type = RTM_GETLINK;
    struReq.nh.nlmsg_flags = NLM_F_REQUEST | NLM_F_DUMP;
    struReq.ifi.ifi_family = AF_UNSPEC;
    memset(&struAddr, 0, sizeof(struAddr));
    struAddr.nl_family = AF_NETLINK;
    struAddr.nl_pid = 0;
    struAddr.nl_groups = 0;
    if(sendto(nSocket, &struReq, struReq.nh.nlmsg_len, 0,
        (struct sockaddr *)&struAddr, sizeof(struAddr)) < 0)
    {
        fprintf(stderr, "发送数据错误:%s\n", strerror(errno));
        return -1;
    }

    /*
     * 循环接收数据，直到超时
     */
    alarm(30);
    memset(szBuffer, 0, sizeof(szBuffer));
    while((nLen = recv(nSocket, szBuffer, sizeof(szBuffer), 0)))
    {
        alarm(0);
        pstruNL = (struct nlmsghdr *)szBuffer;
        /*
          * 判断是否继续有数据
          */
        while(NLMSG_OK(pstruNL, nLen))
        {
            /*
              * 数据已经获取完成
              */
            if(pstruNL -> nlmsg_type == NLMSG_DONE)
                break;
            if(pstruNL -> nlmsg_type == NLMSG_ERROR)
            {
                /*
                  * 发生一个错误
                  */
                struct nlmsgerr *pstruError;

                pstruError = (struct nlmsgerr *)NLMSG_DATA(pstruNL);
                fprintf(stderr, "发生错误[%s]\n",
                    strerror(-pstruError -> error));
                break;
            }

            /*
              * 下面通过宏获取数据
              */
            pstruIF = NLMSG_DATA(pstruNL);
            fprintf(stderr, "获取到设备[%d]信息\n", pstruIF -> ifi_index);
            fprintf(stderr, "\t设备类型:");
```

```
switch(pstruIF -> ifi_type)
{
    case ARPHRD_ETHER:
        fprintf(stderr, "以太网\n");
        break;
    case ARPHRD_PPP:
        fprintf(stderr, "PPP拨号\n");
        break;
    case ARPHRD_LOOPBACK:
        fprintf(stderr, "环路设备\n");
        break;
    default:
        fprintf(stderr, "未知\n");
        break;
}
fprintf(stderr, "\t设备状态:");
if((pstruIF -> ifi_flags & IFF_UP )== IFF_UP)
    fprintf(stderr, " UP");
if((pstruIF -> ifi_flags & IFF_BROADCAST) == IFF_BROADCAST)
    fprintf(stderr, " BROADCAST");
if((pstruIF -> ifi_flags & IFF_DEBUG) == IFF_DEBUG)
    fprintf(stderr, " DEBUG");
if((pstruIF -> ifi_flags & IFF_LOOPBACK) == IFF_LOOPBACK)
    fprintf(stderr, " LOOPBACK");
if((pstruIF -> ifi_flags & IFF_POINTOPOINT) == IFF_POINTOPOINT)
    fprintf(stderr, " POINTOPOINT");
if((pstruIF -> ifi_flags & IFF_RUNNING) == IFF_RUNNING)
    fprintf(stderr, " RUNNING");
if((pstruIF -> ifi_flags & IFF_NOARP) == IFF_NOARP)
    fprintf(stderr, " NOARP");
if((pstruIF -> ifi_flags & IFF_PROMISC) == IFF_PROMISC)
    fprintf(stderr, " PROMISC");
if((pstruIF -> ifi_flags & IFF_NOTRAILERS) == IFF_NOTRAILERS)
    fprintf(stderr, " NOTRAILERS");
if((pstruIF -> ifi_flags & IFF_ALLMULTI) == IFF_ALLMULTI)
    fprintf(stderr, " ALLMULTI");
if((pstruIF -> ifi_flags & IFF_MASTER) == IFF_MASTER)
    fprintf(stderr, " MASTER");
if((pstruIF -> ifi_flags & IFF_SLAVE) == IFF_SLAVE)
    fprintf(stderr, " SLAVE");
if((pstruIF -> ifi_flags & IFF_MULTICAST) == IFF_MULTICAST)
    fprintf(stderr, " MULTICAST");
if((pstruIF -> ifi_flags & IFF_PORTSEL) == IFF_PORTSEL)
    fprintf(stderr, " SLAVE");
if((pstruIF -> ifi_flags & IFF_AUTOMEDIA) == IFF_AUTOMEDIA)
    fprintf(stderr, " AUTOMEDIA");
if((pstruIF -> ifi_flags & IFF_DYNAMIC) == IFF_DYNAMIC)
    fprintf(stderr, " DYNAMIC");
fprintf(stderr, "\n");

/*
   * 下面通过宏获取属性
   */
pstruAttr = IFLA_RTA(pstruIF);
nAttrLen = NLMSG_PAYLOAD(pstruNL, sizeof(struct ifinfomsg));
while(RTA_OK(pstruAttr, nAttrLen))
{
    switch(pstruAttr->rta_type)
    {
        case IFLA_IFNAME:
            fprintf(stderr, "\t设备名称:%s\n",
                (char *)RTA_DATA(pstruAttr));
            break;
        case IFLA_MTU:
            fprintf(stderr, "\t设备MTU:%d\n",
```

```
                    *(unsigned int *)RTA_DATA(pstruAttr));
                break;
            case IFLA_QDISC:
                fprintf(stderr, "\t设备队列:%s\n",
                    (char *)RTA_DATA(pstruAttr));
                break;
            case IFLA_ADDRESS:
                if(pstruIF -> ifi_type == ARPHRD_ETHER)
                {
                    pstruEther = (struct ether_addr *)
                        RTA_DATA(pstruAttr);
                    fprintf(stderr, "\tMAC地址:%s\n",
                        ether_ntoa(pstruEther));
                }
                break;
            case IFLA_BROADCAST:
                if(pstruIF -> ifi_type == ARPHRD_ETHER)
                {
                    pstruEther = (struct ether_addr *)
                        RTA_DATA(pstruAttr);
                    fprintf(stderr, "\t广播MAC地址:%s\n",
                        ether_ntoa(pstruEther));
                }
                break;
            case IFLA_STATS:
                pstruInfo = (struct net_device_stats *)
                    RTA_DATA(pstruAttr);
                fprintf(stderr, "\t接收信息:\n");
                fprintf(stderr, "\t\t接收报文:%lu 字节:%lu\n",
                    pstruInfo -> rx_packets, pstruInfo -> rx_bytes);
                fprintf(stderr, "\t\terrors:%lu dropped:%lu "
                        "multicast:%lu collisions:%lu\n",
                    pstruInfo -> rx_errors, pstruInfo -> rx_dropped,
                    pstruInfo -> multicast, pstruInfo -> collisions);
                fprintf(stderr, "\t\tlength:%lu over:%lu crc:%lu "
                        "frame:%lu fifo:%lu missed:%lu\n",
                    pstruInfo -> rx_length_errors,
                    pstruInfo -> rx_over_errors,
                    pstruInfo -> rx_crc_errors,
                    pstruInfo -> rx_frame_errors,
                    pstruInfo -> rx_fifo_errors,
                    pstruInfo -> rx_missed_errors);
                fprintf(stderr, "\t发送信息:\n");
                fprintf(stderr, "\t\t发送报文:%lu 字节:%lu\n",
                    pstruInfo -> tx_packets, pstruInfo -> tx_bytes);
                fprintf(stderr, "\t\terrors:%lu dropped:%lu\n",
                    pstruInfo -> tx_errors, pstruInfo -> tx_dropped);
                fprintf(stderr, "\t\taborted:%lu carrier:%lu fifo:%lu"
                        " heartbeat:%lu window:%lu\n",
                    pstruInfo -> tx_aborted_errors,
                    pstruInfo -> tx_carrier_errors,
                    pstruInfo -> tx_fifo_errors,
                    pstruInfo -> tx_heartbeat_errors,
                    pstruInfo -> tx_window_errors);
                break;
            default:
                break;

        }
        /*
            * 继续下一个属性
            */
        pstruAttr = RTA_NEXT(pstruAttr, nAttrLen);
    }
    /*
```

```
                * 继续下一个数据
                */
            pstruNL = NLMSG_NEXT(pstruNL, nLen);
        }
        memset(szBuffer, 0, sizeof(szBuffer));
        alarm(30);
    }
    return 0;
}
```

下面是我的机器的输出信息

```
获取到设备[1]信息
        设备类型:环路设备
        设备状态: UP LOOPBACK RUNNING
        设备名称:lo
        设备MTU:16436
        设备队列:noqueue
        接收信息:
                接收报文:73 字节:5105
                errors:0 dropped:0 multicast:0 collisions:0
                length:0 over:0 crc:0 frame:0 fifo:0 missed:0
        发送信息:
                发送报文:73 字节:5105
                errors:0 dropped:0
                aborted:0 carrier:0 fifo:0 heartbeat:0 window:0
获取到设备[2]信息
        设备类型:以太网
        设备状态: UP BROADCAST RUNNING MULTICAST
        设备名称:eth0
        MAC地址:0:e0:4c:68:17:e2
        广播MAC地址:ff:ff:ff:ff:ff:ff
        设备MTU:1500
        设备队列:pfifo_fast
        接收信息:
                接收报文:23942563 字节:2498812213
                errors:0 dropped:0 multicast:0 collisions:0
                length:0 over:0 crc:0 frame:0 fifo:0 missed:0
        发送信息:
                发送报文:26027259 字节:782054736
                errors:0 dropped:0
                aborted:0 carrier:0 fifo:0 heartbeat:0 window:0
获取到设备[3]信息
        设备类型:以太网
        设备状态: UP BROADCAST RUNNING MULTICAST
        设备名称:eth1
        MAC地址:0:11:5b:16:ac:52
        广播MAC地址:ff:ff:ff:ff:ff:ff
        设备MTU:1500
        设备队列:pfifo_fast
        接收信息:
                接收报文:28576652 字节:97399242
                errors:0 dropped:0 multicast:0 collisions:0
                length:0 over:0 crc:0 frame:0 fifo:0 missed:0
        发送信息:
                发送报文:22489921 字节:2255181112
                errors:0 dropped:0
                aborted:0 carrier:0 fifo:0 heartbeat:0 window:0
获取到设备[4]信息
        设备类型:PPP拨号
        设备状态: UP POINTOPOINT RUNNING NOARP MULTICAST
        设备名称:ppp0
        设备MTU:1492
        设备队列:pfifo_fast
        接收信息:
```

```
        接收报文:25021013 字节:3614587231
        errors:0 dropped:0 multicast:0 collisions:0
        length:0 over:0 crc:0 frame:0 fifo:0 missed:0
    发送信息:
        发送报文:22442439 字节:1757067695
        errors:0 dropped:0
        aborted:0 carrier:0 fifo:0 heartbeat:0 window:0
```

# F. NETLINK_SKIP 协议介绍

TODO:还没有找到资料

# G. NETLINK_USERSOCK协议介绍

TODO:还没有找到资料

# H. NETLINK_FIREWALL 协议介绍

FIREWALL协议一般主要是用于netfilter的包过虑机制中，我们通过使用iptables的-j QUEUE机制将报文进行队列排队，然后就从用户层来进行报文的接收或者拒绝操作。对于FIREWALL协议，支持的消息类型有三种分别是：

IPQM_MODE
设置内核模式
IPQM_VERDICT
控制内核报文下一步骤
IPQM_PACKET
报文为从内核接收到

当消息类型为IPQM_PACKET，表示报文是从内核到用户的。从内核到用户的报文格式为nlmsghdr头加上一个ipq_packet_msg数据报文。其中ipq_packet_msg的结构定义为：

```
1  /*
2   * 从内核发送到用户的消息格式
3   */
4  typedef struct ipq_packet_msg {
5      unsigned long packet_id;        /* 报文ID号          */
6      unsigned long mark;             /* MARK标记          */
7      long timestamp_sec;            /* 到达时间(秒)     */
8      long timestamp_usec;           /* 到达时间(微秒)   */
9      unsigned int hook;              /* 钩子位置          */
10     char indev_name[IFNAMSIZ];     /* 进入接口          */
11     char outdev_name[IFNAMSIZ];    /* 出去接口          */
12     unsigned short hw_protocol;     /* 硬件协议          */
13     unsigned short hw_type;         /* 硬件类型          */
14     unsigned char hw_addrlen;       /* 硬件地址长度      */
15     unsigned char hw_addr[8];       /* 硬件地址          */
16     size_t data_len;                /* 数据长度          */
17     unsigned char payload[0];       /* 数据起始位置      */
18 } ipq_packet_msg_t;
```

当消息类型为IPQM_MODE，表示报文是从用户到内核，用户请求内核完成某项操作。请求报文的格式为nlmsghdr头加上一个ipq_mode_msg数据报文。其中ipq_mode_msg的结构定义为：

```
21  * 从用户到内核的消息
22  */
23 typedef struct ipq_mode_msg {
```

```
24      unsigned char value;            /* 要求操作指令      */
25      size_t range;                   /* 数据范围          */
26  } ipq_mode_msg_t;
```

其中value的合法取值为：

- IPQ_COPY_NONE
  取消从内核传送报文到用户
- IPQ_COPY_META
  只从内核拷贝基本数据到用户
- IPQ_COPY_PACKET
  除了拷贝基本数据外还拷贝报文数据，这个时候从内核出来的报文中payload指向数据的地址。

range在value为IPQ_COPY_PACKET时候指定我们希望从内核拷贝多少的数据出来。

当消息类型为IPQM_VERDICT，表示报文是从用户到内核，用户指定内核完成某项操作。请求报文的格式为nlmsghdr头加上一个ipq_mode_verdict数据报文。其中ipq_mode_verdict的结构定义为：

```
28  /*
29   * 从用户到内核的消息
30   */
31  typedef struct ipq_verdict_msg {
32      unsigned int value;             /* 控制操作模式      */
33      unsigned long id;               /* 要求操作的报文ID  */
34      size_t data_len;                /* 数据长度          */
35      unsigned char payload[0];       /* 数据的长度        */
36  } ipq_verdict_msg_t;
```

其中value的取值为NF_ACCEPT表示要求内核接受这个报文，NF_DROP表示丢弃这个报文，其余我们不关心的取值有NF_STOLEN，NF_QUEUE，NF_REPEAT等。下面是一个实际的例子：

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <asm/types.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4/ip_queue.h>
#include <netinet/ether.h>
#include <linux/netfilter_ipv4.h>
#include <signal.h>

static volatile int gnContinue = 1;
static void in_catch_sig(int nSig)
{
    gnContinue = 0;
}

int main(int argc, char *argv[])
{
    int nSocket, nLen;
    struct sockaddr_nl struAddr;
    struct nlmsghdr *pstruNL;
    struct ipq_packet_msg *pstruPacketMsg;
    char szBuffer[4096];
    struct tm struTmNow;
    struct sigaction struAct;

    struct {
        struct nlmsghdr head;
        union{
```

```
        struct ipq_mode_msg mode;
        struct ipq_verdict_msg verdict;
    }body;
} struReq;

/*
 * 创建SOCKET
 */
nSocket = socket(AF_NETLINK, SOCK_RAW, NETLINK_FIREWALL);
if (nSocket < 0)
{
    fprintf(stderr, "创建SOCKET错误:%s\n", strerror(errno));
    return -1;
}

/*
 * 监听本地地址
 */
memset(&struAddr, 0, sizeof(struAddr));
struAddr.nl_family = AF_NETLINK;
struAddr.nl_pid = getpid();
struAddr.nl_groups = 0;
if(bind(nSocket, (struct sockaddr *)&struAddr, sizeof(struAddr)) < 0)
{
    fprintf(stderr, "绑定SOCKET错误:%s\n", strerror(errno));
    return -1;
}

memset(&struAct, 0, sizeof(struAct));
struAct.sa_handler = in_catch_sig;
sigfillset(&struAct.sa_mask);
if(sigaction(SIGINT, &struAct, NULL) < 0)
{
    fprintf(stderr, "设置信号捕捉错误:%s\n", strerror(errno));
    return -1;
}

/*
 * 设置报文内容
 */
memset(&struReq, 0, sizeof(struReq));
struReq.head.nlmsg_len = NLMSG_LENGTH(sizeof(struReq));
struReq.head.nlmsg_type = IPQM_MODE;
struReq.head.nlmsg_flags =  NLM_F_REQUEST | NLM_F_DUMP;
struReq.head.nlmsg_pid = getpid();
struReq.body.mode.value = IPQ_COPY_META;

/*
 * 发送报文到内核
 */
memset(&struAddr, 0, sizeof(struAddr));
struAddr.nl_family = AF_NETLINK;
struAddr.nl_pid = 0;
struAddr.nl_groups = 0;
if(sendto(nSocket, &struReq, struReq.head.nlmsg_len, 0,
    (struct sockaddr *)&struAddr, sizeof(struAddr)) < 0)
{
    fprintf(stderr, "设置 IPQM_MODE 错误:%s\n", strerror(errno));
    return -1;
}

memset(szBuffer, 0, sizeof(szBuffer));
fprintf(stderr, "Press CTRL+C to quit\n");
while(gnContinue)
{
```

```
nLen = recv(nSocket, szBuffer, sizeof(szBuffer), 0);
if(nLen < 0)
    break;
pstruNL =(struct nlmsghdr *)szBuffer;

/*
  * 判断是否继续有数据
  */
while(NLMSG_OK(pstruNL, nLen))
{
    /*
      * 数据已经获取完成
      */
    if(pstruNL -> nlmsg_type == NLMSG_DONE)
        break;
    if(pstruNL -> nlmsg_type == NLMSG_ERROR)
    {
        /*
          * 发生一个错误
          */
        struct nlmsgerr *pstruError;

        pstruError = (struct nlmsgerr *)NLMSG_DATA(pstruNL);
        fprintf(stderr, "发生错误[%s]\n",
            strerror(-pstruError -> error));
        break;
    }

    /*
      * 下面通过宏获取数据
      */
    if(pstruNL -> nlmsg_type == IPQM_PACKET)
    {
        pstruPacketMsg = NLMSG_DATA(pstruNL);
        fprintf(stderr, "获取到一个报文:[MARK-%lu]",
            pstruPacketMsg -> mark);
        switch(pstruPacketMsg -> hook)
        {
            case NF_IP_PRE_ROUTING:
                fprintf(stderr, "[PREROUTING]");
                break;
            case NF_IP_LOCAL_IN:
                fprintf(stderr, "[INPUT]");
                break;
            case NF_IP_FORWARD:
                fprintf(stderr, "[FORWARD]");
                break;
            case NF_IP_LOCAL_OUT:
                fprintf(stderr, "[OUTPUT]");
                break;
            case NF_IP_POST_ROUTING:
                fprintf(stderr, "[POSTROUTING]");
                break;
            default:
                fprintf(stderr, "[UNKNOWN]");
                break;
        }
        fprintf(stderr, "\n");
        struTmNow = *localtime(&pstruPacketMsg -> timestamp_sec);
        fprintf(stderr,
            "\t时间[%04d-%02d-%02d %02d:%02d:%02d] ID:%lu\n",
            struTmNow.tm_year + 1900, struTmNow.tm_mon + 1,
            struTmNow.tm_mday, struTmNow.tm_hour,
            struTmNow.tm_min, struTmNow.tm_sec,
            pstruPacketMsg -> packet_id);
```

```
                fprintf(stderr, "\t进入:%s 出去:%s\n",
                    pstruPacketMsg -> indev_name,
                    pstruPacketMsg -> outdev_name);
                if(pstruPacketMsg -> hw_type == ARPHRD_ETHER)
                {
                    fprintf(stderr, "\tMAC地址:%s\n",
                        ether_ntoa((struct ether_addr *)
                            pstruPacketMsg -> hw_addr));
                }

                /*
                    * 决定数据的下一步
                    */
                memset(&struReq, 0, sizeof(struReq));
                struReq.head.nlmsg_len = NLMSG_LENGTH(sizeof(struReq));
                struReq.head.nlmsg_type = IPQM_VERDICT;
                struReq.head.nlmsg_flags =  NLM_F_REQUEST | NLM_F_DUMP;
                struReq.head.nlmsg_pid = getpid();
                struReq.body.verdict.value = NF_ACCEPT;
                struReq.body.verdict.id = pstruPacketMsg -> packet_id;

                /*
                    * 发送报文到内核
                    */
                memset(&struAddr, 0, sizeof(struAddr));
                struAddr.nl_family = AF_NETLINK;
                struAddr.nl_pid = 0;
                struAddr.nl_groups = 0;
                if(sendto(nSocket, &struReq, struReq.head.nlmsg_len, 0,
                    (struct sockaddr *)&struAddr, sizeof(struAddr)) < 0)
                {
                    fprintf(stderr, "设置 IPQM_VERDICT 错误:%s\n",
                        strerror(errno));
                    return -1;
                }

            }
            /*
                * 继续下一个数据
                */
            pstruNL = NLMSG_NEXT(pstruNL, nLen);
        }
    }

    /*
      * 取消模式
      */
    memset(&struReq, 0, sizeof(struReq));
    struReq.head.nlmsg_len = NLMSG_LENGTH(sizeof(struReq));
    struReq.head.nlmsg_type = IPQM_MODE;
    struReq.head.nlmsg_flags =  NLM_F_REQUEST | NLM_F_DUMP;
    struReq.head.nlmsg_pid = getpid();
    struReq.body.mode.value = IPQ_COPY_NONE;

    /*
      * 发送报文到内核
      */
    memset(&struAddr, 0, sizeof(struAddr));
    struAddr.nl_family = AF_NETLINK;
    struAddr.nl_pid = 0;
    struAddr.nl_groups = 0;
    if(sendto(nSocket, &struReq, struReq.head.nlmsg_len, 0,
        (struct sockaddr *)&struAddr, sizeof(struAddr)) < 0)
    {
        fprintf(stderr, "设置 IPQM_MODE 错误:%s\n", strerror(errno));
```

```
        return -1;
    }
    return 0;
}
```

如果想上面的程序能够执行，我们必须以root身份执行下面几个命令（假设已经将上面的代码编译为了filter程序）

1. 设置报文规则
   iptables -A INPUT -p icmp --icmp-type echo-request -j QUEUE
   这个规则表示所有的ping请求报文我们都进行排队记录，这样我们的程序中就可以获取到这个报文了.
2. 执行监听程序
   ./filter
3. 进行测试
   使用ping命令发送请求报文ping -c 1 192.168.0.1
4. 删除报文规则
   iptables -D INPUT -p icmp --icmp-type echo-request -j QUEUE 删除规则

下面是我的执行结果

```
-- 这个是从本机ping过来的
获取到一个报文:0
[INPUT]
        时间[2005-01-24 15:03:05] ID:3332502528
        进入:lo 出去:
-- 这个是从另外一台ping过来的
获取到一个报文:0
[INPUT]
        时间[2005-01-24 15:03:34] ID:3332502784
        进入:eth0 出去:
        MAC地址:0:a:e4:24:24:86
```

# I. NETLINK_TCPDIAG 协议介绍

TCPDIAG用于TCP连接的诊断状态，由于网络上面没有找到任何关于这个协议的说明，我就简单的分析了Linux的tcp_diag.c源代码，由于我对内核不熟悉，所以只分析了其中的一个TCPDIAG_GETSOCK消息类型，TCPDIAG_REQ_BYTECODE这个消息类型还没有进行分析.使用TCPDIAG_GETSOCK时候表示我们期望获取一个socket的状态，这个时候请求消息的格式为1个nlmsghdr机构加上一个tcpdiagreq结构，应答消息结构格式为1个nlmsghdr结构加上一个tcpdiagmsg机构

```
/*
 * SOCKET ID结构
 */
struct tcpdiag_sockid
{
    __u16   tcpdiag_sport;  /* 源端口    */
    __u16   tcpdiag_dport;  /* 目的端口 */
    __u32   tcpdiag_src[4]; /* 源地址    */
    __u32   tcpdiag_dst[4]; /* 目的地址 */
    __u32   tcpdiag_if;     /* 接口设备 */
    __u32   tcpdiag_cookie[2];
#define TCPDIAG_NOCOOKIE (~0U)
};

/*
 * 请求结构
 */
struct tcpdiagreq
{
    __u8    tcpdiag_family;     /* Family of addresses. */
    __u8    tcpdiag_src_len;
    __u8    tcpdiag_dst_len;
```

```
    __u8    tcpdiag_ext;        /* Query extended information */

    struct tcpdiag_sockid id;

    __u32   tcpdiag_states;     /* States to dump */
    __u32   tcpdiag_dbs;        /* Tables to dump (NI) */
};

struct tcpdiagmsg
{
    __u8    tcpdiag_family;
    __u8    tcpdiag_state;
    __u8    tcpdiag_timer;
    __u8    tcpdiag_retrans;

    struct tcpdiag_sockid id;

    __u32   tcpdiag_expires;
    __u32   tcpdiag_rqueue;
    __u32   tcpdiag_wqueue;
    __u32   tcpdiag_uid;
    __u32   tcpdiag_inode;
};
```

**下面是一个简单的测试代码**

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <asm/types.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>
#include <linux/netfilter.h>
#include <netinet/ether.h>
#include <linux/netfilter_ipv4.h>
#include "/usr/src/linux/include/linux/tcp_diag.h"  /* 这个地方是头文件位置 */
#include <signal.h>
#include <sys/socket.h>
#include <netinet/tcp.h>
#include <netinet/in.h>

static volatile int gnContinue = 1;
static void in_catch_sig(int nSig)
{
    gnContinue = 0;
}

int main(int argc, char *argv[])
{
    int nSocket, nLen;
    struct sockaddr_nl struAddr;
    struct nlmsghdr *pstruNL;
    struct tcpdiagmsg *pstruPacketMsg;
    char szBuffer[4096];
    struct tm struTmNow;
    struct sigaction struAct;
    char szTemp[256];

    struct {
        struct nlmsghdr head;
        struct tcpdiagreq body;
```

```
} struReq;

/*
 * 创建SOCKET
 */
nSocket = socket(AF_NETLINK, SOCK_RAW, NETLINK_TCPDIAG);
if (nSocket < 0)
{
    fprintf(stderr, "创建SOCKET错误:%s\n", strerror(errno));
    return -1;
}

/*
 * 监听本地地址
 */
memset(&struAddr, 0, sizeof(struAddr));
struAddr.nl_family = AF_NETLINK;
struAddr.nl_pid = getpid();
struAddr.nl_groups = 0;
if(bind(nSocket, (struct sockaddr *)&struAddr, sizeof(struAddr)) < 0)
{
    fprintf(stderr, "绑定SOCKET错误:%s\n", strerror(errno));
    return -1;
}

memset(&struAct, 0, sizeof(struAct));
struAct.sa_handler = in_catch_sig;
sigfillset(&struAct.sa_mask);
if(sigaction(SIGINT, &struAct, NULL) < 0)
{
    fprintf(stderr, "设置信号捕捉错误:%s\n", strerror(errno));
    return -1;
}

/*
 * 设置报文内容
 */
memset(&struReq, 0, sizeof(struReq));
struReq.head.nlmsg_len = NLMSG_LENGTH(sizeof(struReq));
struReq.head.nlmsg_type = TCPDIAG_GETSOCK;
struReq.head.nlmsg_flags =  NLM_F_REQUEST;
struReq.head.nlmsg_pid = getpid();

struReq.body.tcpdiag_family = AF_INET;
struReq.body.id.tcpdiag_sport = htons(23);
/*
 * 下面三个数据你的机器可能不一致
 */
struReq.body.id.tcpdiag_dport = htons(3311);
inet_pton(AF_INET, "192.168.0.1", struReq.body.id.tcpdiag_src);
inet_pton(AF_INET, "192.168.0.81", struReq.body.id.tcpdiag_dst);
struReq.body.id.tcpdiag_cookie[0] = TCPDIAG_NOCOOKIE;
struReq.body.id.tcpdiag_cookie[1] = TCPDIAG_NOCOOKIE;

/*
 * 发送报文到内核
 */
memset(&struAddr, 0, sizeof(struAddr));
struAddr.nl_family = AF_NETLINK;
struAddr.nl_pid = 0;
struAddr.nl_groups = 0;
if(sendto(nSocket, &struReq, struReq.head.nlmsg_len, 0,
    (struct sockaddr *)&struAddr, sizeof(struAddr)) < 0)
{
    fprintf(stderr, "设置 IPQM_MODE 错误:%s\n", strerror(errno));
```

```
        return -1;
    }

    memset(szBuffer, 0, sizeof(szBuffer));
    fprintf(stderr, "Press CTRL+C to quit\n");
    while(gnContinue)
    {
        nLen = recv(nSocket, szBuffer, sizeof(szBuffer), 0);
        if(nLen < 0)
            break;
        pstruNL =(struct nlmsghdr *)szBuffer;

        /*
          * 判断是否继续有数据
          */
        while(NLMSG_OK(pstruNL, nLen))
        {
            /*
              * 数据已经获取完成
              */
            if(pstruNL -> nlmsg_type == NLMSG_DONE)
                break;
            if(pstruNL -> nlmsg_type == NLMSG_ERROR)
            {
                /*
                  * 发生一个错误
                  */
                struct nlmsgerr *pstruError;

                pstruError = (struct nlmsgerr *)NLMSG_DATA(pstruNL);
                fprintf(stderr, "发生错误[%s][%d]\n",
                    strerror(-pstruError -> error),
                    -pstruError -> error);
                break;
            }

            /*
              * 下面通过宏获取数据
              */
            pstruPacketMsg = NLMSG_DATA(pstruNL);
            fprintf(stderr, "获取到一个报文\n");
            fprintf(stderr, "\tFAMILY: %s",
                pstruPacketMsg -> tcpdiag_family == AF_INET ?
                    "AF_INET" : "AF_INET6");
            fprintf(stderr, "\tSTATE: ");
            switch(pstruPacketMsg -> tcpdiag_state)
            {
                case TCP_ESTABLISHED:
                    fprintf(stderr, "ESTABLISHED");
                    break;
                case TCP_SYN_SENT:
                    fprintf(stderr, "SYN_SENT");
                    break;
                case TCP_SYN_RECV:
                    fprintf(stderr, "SYN_RECV");
                    break;
                case TCP_FIN_WAIT1:
                    fprintf(stderr, "FIN_WAIT1");
                    break;
                case TCP_FIN_WAIT2:
                    fprintf(stderr, "FIN_WAIT2");
                    break;
                case TCP_TIME_WAIT:
                    fprintf(stderr, "TIME_WAIT");
                    break;
```

```
                case TCP_CLOSE:
                    fprintf(stderr, "CLOSE");
                    break;
                case TCP_CLOSE_WAIT:
                    fprintf(stderr, "CLOSE_WAIT");
                    break;
                case TCP_LAST_ACK:
                    fprintf(stderr, "LAST_ACK");
                    break;
                case TCP_LISTEN:
                    fprintf(stderr, "LISTEN");
                    break;
                case TCP_CLOSING:
                    fprintf(stderr, "CLOSING");
                    break;
                default:
                    fprintf(stderr, "UNKNOW");
                    break;
            }
            if_indextoname(pstruPacketMsg -> id.tcpdiag_if, szTemp);
            fprintf(stderr, " INTERFACE:%s\n",
                if_indextoname(pstruPacketMsg -> id.tcpdiag_if, szTemp));
            memset(szTemp, 0, sizeof(szTemp));
            inet_ntop(AF_INET, pstruPacketMsg -> id.tcpdiag_src,
                szTemp, sizeof(szTemp));
            fprintf(stderr, "\tFROM %s:%d ",
                szTemp, ntohs(pstruPacketMsg ->id.tcpdiag_sport));
            memset(szTemp, 0, sizeof(szTemp));
            inet_ntop(AF_INET, pstruPacketMsg -> id.tcpdiag_dst,
                szTemp, sizeof(szTemp));
            fprintf(stderr, "TO %s:%d ",
                szTemp, ntohs(pstruPacketMsg ->id.tcpdiag_dport));
            fprintf(stderr, "RQ: %lu SQ: %lu\n",
                pstruPacketMsg -> tcpdiag_rqueue,
                pstruPacketMsg -> tcpdiag_wqueue);

            /*
               * 继续下一个数据
               */
            pstruNL = NLMSG_NEXT(pstruNL, nLen);
        }
    }
    return 0;
}
```

# J. NETLINK_NFLOG 协议介绍

NETLINK_NFLOG和FIREWALL协议其实非常类似，不过NFLOG只负责接收从内核发送过来的报文，对这些报文进行记录操作，不进行任何其它的操作。从内核过来的报文格式为nlmsghdr+ulog_packet_msg结构。其中ulog_packet_msg的定义在<linux/netfilter_ipv4/ipt_ULOG.h>文件

```
31  /* Format of the ULOG packets passed through netlink */
32  typedef struct ulog_packet_msg {
33          unsigned long mark;
34          long timestamp_sec;
35          long timestamp_usec;
36          unsigned int hook;
37          char indev_name[IFNAMSIZ];
38          char outdev_name[IFNAMSIZ];
39          size_t data_len;
40          char prefix[ULOG_PREFIX_LEN];
41          unsigned char mac_len;
```

```
42          unsigned char mac[ULOG_MAC_LEN];
43          unsigned char payload[0];
44   } ulog_packet_msg_t;
```

大部分字段同我们前面介绍的FIREFOX协议中的字段含义相同，其中prefix是我们在使用iptables设定记录规则时候指定的--ulog-prefix参数。payload的数据指向一个ip包的起始地址。下面是一个实际的例子

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <asm/types.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>
#include <linux/netfilter.h>
#include <netinet/ether.h>
#include <linux/netfilter_ipv4.h>
#include <net/if.h>
#include <linux/netfilter_ipv4/ipt_ULOG.h>
#include <signal.h>

static volatile int gnContinue = 1;
static void in_catch_sig(int nSig)
{
    gnContinue = 0;
}

int main(int argc, char *argv[])
{

    int nSocket, nLen;

    struct sockaddr_nl struAddr;

    struct nlmsghdr *pstruNL;

    struct ulog_packet_msg *pstruPacketMsg;

    char szBuffer[4096];

    struct tm struTmNow;

    struct sigaction struAct;

    int i;



    /*
     * 创建SOCKET
       */
    nSocket = socket(AF_NETLINK, SOCK_RAW, NETLINK_NFLOG);

    if (nSocket < 0)
    {
        fprintf(stderr, "创建SOCKET错误:%s\n", strerror(errno));

        return -1;
    }
```

```
/*
  * 监听本地地址
     */
memset(&struAddr, 0, sizeof(struAddr));
struAddr.nl_family = AF_NETLINK;
struAddr.nl_pid = getpid();
struAddr.nl_groups = 1 << (5 - 1); /* 这个地方是我们指定的组号    */

if(bind(nSocket, (struct sockaddr *)&struAddr, sizeof(struAddr)) < 0)
{
    fprintf(stderr, "绑定SOCKET错误:%s\n", strerror(errno));

    return -1;
}

memset(&struAct, 0, sizeof(struAct));
struAct.sa_handler = in_catch_sig;
sigfillset(&struAct.sa_mask);

if(sigaction(SIGINT, &struAct, NULL) < 0)
{
    fprintf(stderr, "设置信号捕捉错误:%s\n", strerror(errno));

    return -1;
}

memset(szBuffer, 0, sizeof(szBuffer));
fprintf(stderr, "Press CTRL+C to quit\n");

while(gnContinue)
{
    nLen = recv(nSocket, szBuffer, sizeof(szBuffer), 0);

    if(nLen < 0)

        break;
    pstruNL =(struct nlmsghdr *)szBuffer;

    /*
      * 判断是否继续有数据
             */

    while(NLMSG_OK(pstruNL, nLen))
    {

        /*
           * 数据已经获取完成
                    */

        if(pstruNL -> nlmsg_type == NLMSG_DONE)

            break;

        if(pstruNL -> nlmsg_type == NLMSG_ERROR)
        {

            /*
               * 发生一个错误
                        */

            struct nlmsgerr *pstruError;

            pstruError = (struct nlmsgerr *)NLMSG_DATA(pstruNL);
```

```
                fprintf(stderr, "发生错误[%s]\n",
                    strerror(-pstruError -> error));

                break;
            }



            /*
               * 下面通过宏获取数据
                            */
            pstruPacketMsg = NLMSG_DATA(pstruNL);
            struTmNow = *localtime(&pstruPacketMsg -> timestamp_sec);
            fprintf(stderr,

                "\t时间[%04d-%02d-%02d %02d:%02d:%02d]\n",
                struTmNow.tm_year + 1900, struTmNow.tm_mon + 1,
                struTmNow.tm_mday, struTmNow.tm_hour,
                struTmNow.tm_min, struTmNow.tm_sec);
            fprintf(stderr, "\t进入:%s 出去:%s\n",
                    pstruPacketMsg -> indev_name,
                    pstruPacketMsg -> outdev_name);
            fprintf(stderr, "\tMAC地址:");

            for(i = 0 ; i < pstruPacketMsg -> mac_len ; i ++)
                fprintf(stderr, "%02X ", pstruPacketMsg -> mac[i]);
            fprintf(stderr, "\n");



            /*
               * 继续下一个数据
                            */
            pstruNL = NLMSG_NEXT(pstruNL, nLen);
        }
    }

    return 0;
}
```

如果想上面的程序能够执行，我们必须以root身份执行下面几个命令（假设已经将上面的代码编译为了ulog程序）

1. 设置报文规则
   iptables -A INPUT -p icmp --icmp-type echo-request -j ULOG --ulog-nlgroup 5
   这个规则表示所有的ping请求报文我们都进行记录，记录的组为5，这样我们的程序中就可以获取到这个报文了.
2. 执行监听程序
   ./ulog
3. 进行测试
   使用ping命令发送请求报文ping -c 1 192.168.0.1
4. 删除报文规则
   iptables -D INPUT -p icmp --icmp-type echo-request -j ULOG --ulog-nlgroup 5 删除规则

# K. NETLINK_ARPD 协议介绍

## TODO:还没有找到资料

# L. NETLINK_ROUTE6 协议介绍

TODO:还没有找到资料

## M. NETLINK_IP6_FW 协议介绍

TODO:还没有找到资料

## N. NETLINK_DNRTMSG 协议介绍

TODO:还没有找到资料

## O. NETLINK_TAPBASE 协议介绍

TODO:还没有找到资料

## P. 参考资料


1. Linux 系统内核空间与用户空间通信的实现与分析
   这篇文章是陈鑫在IBM中国开发网站上的一篇文章，讨论了在LINUX下如何进行内核空间和用户空间通讯。
2. Netlink Sockets Tour
   这是一篇英文的介绍Linux内核中是如何实现netlink通讯的文章。
3. Netlink Sockets - Overview
   这是一篇介绍netlink编程入门的英文资料，介绍了一些基础性的东西。
4. Understanding And Programming With Netlink Sockets
   一篇介绍netlink程序的英文文章，介绍的非常详细。
5. Linux Netlink as an IP Services Protocol
   这个可能是最详细介绍netlink之间通讯协议报文的一篇RFC文档，详细的描述了每一个协议的通讯报文格式。
6. NETLINK中文MAN手册
   CMPP已经将netlink(7)的man手册翻译成中文了，不过现在找不到这个的html版，只找到了一个doc版。
7. IPROUTE 路由工具
   IPROUTE路由工具的源代码中有大量的关于netlink的ROUTE代码可以参考，同时也提供了一个libnetlink库用来进行操作。
8. IPTABLES
   IPTABLES是一个Linux下强有力的防火墙工具。在IPTABLES的源代码中有大量的关于netlink编程的FIREWALL代码可以参考，同时也提供了一个libipq库来进行操作。
9. ULOG 用户层日志
   ULOG通过使用netlink的NFLOG功能，在用户层进行了通讯报文的日志记录功能，您可以参考ULOG的源代码来了解NFLOG的使用


## Q. 版权说明

# R. 修改记录

1. 2005-1-20
   创建文档，完成文档主体框架
2. 2005-1-25
   完成文档的ROUTE,FIREWALL,NFLOG协议说明
3. 2005-1-26
   完成文档的TCPDIAG初步介绍