

[首页](#) [资讯](#) [精华](#) [论坛](#) [问答](#) [博客](#) [专栏](#) [群组](#) [更多](#) ▼
[您还未登录!](#) [登录](#) [注册](#)

指点江山

- [博客](#)
- [微博](#)
- [相册](#)
- [收藏](#)
- [留言](#)
- [关于我](#)

Linux——Netlink

博客分类：

- [Linux网络开发](#)

转载：http://blog.csdn.net/firo_baidu/article/details/6145231

放假回家的第一天，呵呵。

缅怀Stevens大师。

最好的参考资料：

- 1.师从互联网。
- 2.Linux man 命令：man netlink，man rtnetlink。
- 3.UNP v1第18章。
- 4.<http://blog.csdn.net/unbutun/archive/2010/01/10/5170059.aspx>

<http://en.wikipedia.org/wiki/Netlink>

<http://yongqig.onlyblog.com/blog2/enchen/8605.html>

<http://linux.chinaunix.net/techdoc/develop/2006/10/15/942169.shtml>

<http://yangelc.blog.sohu.com/68245920.html>

<http://www.chinaunix.net/jh/4/822500.html>

<http://www.ibm.com/developerworks/cn/linux/l-kerns-usrs/>

第一条：概述。

简单说来，linux通过Netlink机制与内核中相应的模块进行通信来掌控设备（网络方面的居多）。

man手册如是说：

Netlink is used to transfer information between kernel and userspace processes. It consists of a standard sockets-based interface for userspace processes and an internal kernel API for kernel modules. The internal kernel interface is not documented in this manual page. There is also an obsolete netlink interface via netlink character devices; this interface is not documented here and is only provided for backwards compatibility.

我翻译的：

Netlink用于在内核和用户空间的进程之间通信。Netlink机制由在用户空间的SocketAPI和内核模块的系统调用组成。手册不提及内核的系统调用接口。至于，Netlink字符设备的接口API也没说，Netlink字符设备的接口API是为了向后兼容。

另外：UNPv1第18章的路由Socket不适用于Linux，linux使用netlink机制全面代替BSD的路由套接口机制。

```
#define PF_ROUTE PF_NETLINK /* Alias to emulate 4.4BSD. */
```

题外话：什么是套接字？即IP地址和端口的组合。

第二条：Netlink套接字描述符

```
int sockfd = socket(AF_NETLINK, sock_type, netlink_faimly);
```

1.sock_type：man手册中说：Netlink is a datagram-oriented service。netlink机制是面向数据报的服务，故可以使用SOCK_RAW 和SOCK_DGRAM，不能使用SOCK_STREAM。手册中也提到：netlink机制不区分数据报（datagram）套接字和原始（raw）套接字。

2.netlink_faimly：指定与哪个内核模块进行通信的协议，如下：

NETLINK_ROUTE：路由 daemon

Receives routing and link updates and may be used to modify the routing tables (both IPv4 and IPv6), IP addresses, link parameters, neighbor setups, queueing disciplines, traffic classes and packet classifiers (see rtnetlink(7)).

NETLINK_W1：1-wire 子系统

Messages from 1-wire subsystem.

NETLINK_USERSOCK：用户态 socket 协议

Reserved for user-mode socket protocols.

NETLINK_FIREWALL: 防火墙

Transport IPv4 packets from netfilter to userspace. Used by ip_queue kernel module.

NETLINK_INET_DIAG: socket 监视

INET socket monitoring.

NETLINK_NFLOG: netfilter 日志

Netfilter/iptables ULOG.

NETLINK_XFRM: ipsec 安全策略

IPsec.

NETLINK_SELINUX: SELinux 事件通知

SELinux event notifications.

NETLINK_ISCSI: iSCSI 子系统

Open-iSCSI.

NETLINK_AUDIT: 进程审计

Auditing.

NETLINK_FIB_LOOKUP: 转发信息表查询

Access to FIB lookup from userspace.

NETLINK_CONNECTOR: netlink connector

Kernel connector. See Documentation/connector/* in the kernel source for further information.

NETLINK_NETFILTER: netfilter 子系统

Netfilter subsystem.

NETLINK_IP6_FW: IPv6 防火墙

Transport IPv6 packets from netfilter to userspace. Used by ip6_queue kernel module.

NETLINK_DNRTMSG: DECnet 路由信息

DECnet routing messages.

NETLINK_KOBJECT_UEVENT: 内核事件向用户态通知

Kernel messages to userspace.

NETLINK_GENERIC: 通用 netlink

Generic netlink family for simplified netlink usage.

函数close用于关闭打开的netlink socket。

第三条：Netlink 套接字地址结构

```
#include<linux/netlink.h>
```

```
struct sockaddr_nl { //几乎和TCP里的sockaddr一样。
```

```
    sa_family_t nl_family; /* 必须为AF_NETLINK或PF_NETLINK */
```

```
    unsigned short nl_pad; /*保留未用,初始为0 */
```

```
    __u32 nl_pid; /* port ID,*/
```

```
    __u32 nl_groups; /* multicast groups mask 多播 组掩码 */
```

```
};
```

nl_pid:

1.当作为bind函数的参数时，就是给没有名字的socketfd赋上一个名字，只有一个要求在有多个Netlink socketfd时要保证唯一性，方式一：由用户保证唯一性：一个进程只有一个Netlink socketfd时可以指定nl_pid为任意整数，getpid（）是个不错的选择。但是一个进程有多个Netlinksocketfd时 就不能都指定为getpid（），必须加以区别。

方式二：man手册指出当把nl_pid赋为0，无论一个进程内有几个 Netlink socketfd，内核将保证他们唯一性。

2.作为sendto等函数的参数：是用来指定发送数据目的地，当目的地是其他的进程，就赋上那个进程的pid就可，这个几乎用不到。当发送到内核，直接赋为0。

nl_groups:

对于Netlink 的每个协议，都有一个容纳32个多播组的集合。nl_groups的一个二进制位代表一个组，共有32个。

1.作为bind 函数的参数，用于把调用进程加入到该nl_groups指定的多播组(是否可以同时被添加进多个组，就是nl_groups多位为1，未验证)，如果设置为0，表示调用者不加入任何多播组。

2.作为sendto等函数的参数时。若nl_groups为0，配合nl_pid发送单播数据，当nl_groups不为0，配合nl_pid发送多播。

第四条：Netlink消息。

Netlink与内核通信的消息有两部分：首部和数据。

首先，Netlink socket 和TCP协议中接收放送数据一样，也需要首部。主要用于多路复用和多路分解，以及其它的一些控制。

```
struct nlmsghdr { //这个结构就是用于表示首部（头部）。
```

```
    __u32 nlmsg_len; /* Length of message including header. 整个数据的大小，包括这个首部和要接收/发送的数据*/
```

```
    __u16 nlmsg_type; /* Type of message content. 接收/发送数据的用途 */
```

```
    __u16 nlmsg_flags; /* Additional flags. 附加标志*/
```

```
    __u32 nlmsg_seq; /* Sequence number. 序列号，就是这个消息是第几个*/
```

```
    __u32 nlmsg_pid; /* PID of the sending process. */
```

```
};
```

nlmsg_seq 和 nlmsg_pid 用于应用追踪消息，前者表示顺序号，后者为消息来源进程 ID。

如果一个消息是由多个数据报组成，也就是说这个消息有多个首部，当然每个首部后面跟着数据部分。那么除了最后数据报，每个部分的首部都要在 nlmsg_flags设置NLM_F_MULTI，最后数据报的首部nlmsg_type设置为NLMSG_DONE。这种情况多是由内核向用户空间造成的，所以这些标志一般都是由内核赋值的我们不需要赋值，只在接收消息检测这些标志位。

nlmsg_type: 取值如下:

以下四个值一般由内核设置，用于我们接收数据后，检测。

1.NLMSG_NOOP : message is to be ignored; 这个消息类型表示数据内容为空，应用可以忽略该报文

2.NLMSG_ERROR: message signals an error and the payload contains an nlmsgerr structure 这个消息类型表示数据部分是一个错误信息，数据部分的结构如下

```
struct nlmsgerr {
    int error;          /* Negative errno or 0 for acknowledgements; 负数表示的出错号 errno 或为 0 要求确认 acks*/

    struct nlmsghdr msg; /* Message header that caused the error: 造成出错的消息报头*/
};
```

3.NLMSG_DONE :

message terminates a multipart message.在我们接收或者发送消息给内核的时候，可能一次发送多个报文，这个消息类型标志最后一个报文。

4.NLMSG_OVERRUN : Data lost.

以下是由程序员设置的，这里的是NETLINK_ROUTE协议支持的类型，至于其他协议的有待研究，每种类型对应着后面数据部分的不同承载结构。使用NETLINK_ROUTE协议时候支持的类型如下，他们都定义在linux/rtnetlink.h中:

1.Link Layer:创建，删除、获取、设置网络设备的信息：RTM_NEWLINK, RTM_DELLINK, RTM_GETLINK, RTM_SETLINK

对应数据部分数据结构：在linux/rtnetlink.h中

```
struct ifinfomsg { /* struct ifinfomsg passes link level specific information, not dependent on network protocol.*/
    unsigned char ifi_family;
    unsigned char __ifi_pad;
    unsigned short ifi_type; /* ARPHRD_* */
    int ifi_index; /* Link index */
    unsigned ifi_flags; /* IFF_* flags */
    unsigned ifi_change; /* IFF_* change mask */
};
```

2.Address Settings:创建，删除、获取网络设备的IP信息：RTM_NEWADDR, RTM_DELADDR, RTM_GETADDR

对应数据部分数据结构：在linux/if_addr.h中

```
struct ifaddrmsg {
    __u8 ifa_family;
```

```
__u8 ifa_prefixlen; /* The prefix length */  
  
__u8 ifa_flags; /* Flags */  
  
__u8 ifa_scope; /* Address scope */  
  
__u32 ifa_index; /* Link index */  
  
};
```

3.Routing Tables: 创建、删除、获取网络设备的路由信息：RTM_NEWROUTE, RTM_DELROUTE, RTM_GETROUTE

对应数据部分数据结构：在linux/rtnetlink.h中

```
struct rtmsg { //Definitions used in routing table administration.
```

```
    unsigned char rtm_family; /* 路由表地址族 */
```

```
    unsigned char rtm_dst_len; /* 目的长度 */
```

```
    unsigned char rtm_src_len; /* 源长度 */
```

```
    unsigned char rtm_tos; /* TOS */
```

```
    unsigned char rtm_table; /* Routing table id */ /* 路由表选取 */
```

```
    unsigned char rtm_protocol; /* Routing protocol; see below */ /* 路由协议 */
```

```
    unsigned char rtm_scope; /* See below */
```

```
    unsigned char rtm_type; /* See below */
```

```
    unsigned rtm_flags;
```

```
};
```

4.Neighbor Cache:创建、删除、获取网络设备的相邻信息：RTM_NEWNEIGH, RTM_DELNEIGH, RTM_GETNEIGH

对应数据部分数据结构：在linux/neighbour.h中

```
struct ndmsg {  
    __u8 ndm_family;  
    __u8 ndm_pad1;  
    __u16 ndm_pad2;  
    __s32 ndm_ifindex;  
    __u16 ndm_state;  
    __u8 ndm_flags;  
    __u8 ndm_type;  
};
```

```
struct nda_cacheinfo {  
    __u32 ndm_confirmed;  
    __u32 ndm_used;  
    __u32 ndm_updated;  
    __u32 ndm_refcnt;  
};
```

5.Routing Rules:创建，删除、获取路由规则信息：RTM_NEWRULE, RTM_DELRULE, RTM_GETRULE

对应数据部分数据结构：在linux/rtnetlink.h中struct rtmsg

6.Queueing Discipline Settings:创建，删除、获取队列的原则：RTM_NEWQDISC, RTM_DELQDISC, RTM_GETQDISC

对应数据部分数据结构：在linux/rtnetlink.h中

```
struct tcmsg { //Traffic control messages.  
    unsigned char tcm_family;  
    unsigned char tcm__pad1;
```

```

unsigned short tcm__pad2;

int tcm_ifindex;

__u32 tcm_handle;

__u32 tcm_parent;

__u32 tcm_info;

};

```

7.Traffic Classes used with Queues:创建，删除、获取流量的类别：RTM_NEWTCCLASS, RTM_DELTCLASS, RTM_GETTCCLASS

对应数据部分数据结构：linux/rtnetlink.h中struct tcmsg

8.Traffic filters:创建，删除、获取流量的过滤：RTM_NEWTFILTER, RTM_DELTFILTER, RTM_GETTFILTER

对应数据部分数据结构：linux/rtnetlink.h中struct tcmsg

9.Others: RTM_NEWACTION , RTM_DELACTION , RTM_GETACTION , RTM_NEWPREFIX , RTM_GETPREFIX , RTM_GETMULTICAST ,

RTM_GETANYCAST ,RTM_NEWNEIGHTBL ,RTM_GETNEIGHTBL , RTM_SETNEIGHTBL

nlmsg_flags：这个成员用于控制和表示消息， 值如下：

1.Standard flag bits in nlmsg_flags

NLM_F_REQUEST Must be set on all request messages.表示消息是一个请求，所有应用首先发起的消息都应设置该标志。，这个标志可以和以下的一个标志组合

- NLM_F_ROOT 被许多 netlink 协议的各种数据获取操作使用，该标志指示被请求的数据表应当整体返回用户应用，而不是一个条目一个条目地返回。有该标志的请求通常导致响应消息设置 NLM_F_MULTI标志。注意，当设置了该标志时，请求是协议特定的，因此，需要在字段 nlmsg_type 中指定协议类型。
- NLM_F_MATCH 表示该协议特定的请求只需要一个数据子集，数据子集由指定的协议特定的过滤器来匹配。
- NLM_F_ATOMIC 返回对象表的快照
- NLM_F_DUMP 被定义为NLM_F_ROOT|NLM_F_MATCH
- NLM_F_REPLACE 用于取代在数据表中的现有条目。
- NLM_F_EXCL 用于和 CREATE 和 APPEND 配合使用，如果条目已经存在，将失败。
- NLM_F_CREAT 指示应当在指定的表中创建一个条目。
- NLM_F_APPEND 指示在表末尾添加新的条目。

NLM_F_MULTI The message is part of a multipart message terminated by **NLMSG_DONE**. 用于指示该消息是一个多部分消息的一部分，后续的消息可以通过宏**NLMSG_NEXT**来获得。

NLM_F_ACK Request for an acknowledgment on success.表示该消息是前一个请求消息的响应，顺序号与进程ID可以把请求与响应关联起来。

NLM_F_ECHO Echo this request.表示该消息是相关的一个包的回传。

2.Additional flag bits for GET requests

NLM_F_ROOT Return the complete table instead of a single entry.

NLM_F_MATCH Return all entries matching criteria (标准, 要求) passed in message content. Not implemented yet.

NLM_F_ATOMIC Return an atomic snapshot of the table.

NLM_F_DUMP Convenience macro; equivalent to (**NLM_F_ROOT**|**NLM_F_MATCH**).

Note that **NLM_F_ATOMIC** requires the **CAP_NET_ADMIN** capability or an effective UID of 0.

3.Additional flag bits for NEW requests

NLM_F_REPLACE Replace existing matching object.

NLM_F_EXCL Don't replace if the object already exists.

NLM_F_CREATE Create object if it doesn't already exist.

NLM_F_APPEND Add to the end of the object list.

第五条Netlink与内核通信

Linux定义了多个宏，来辅助我们发送和接收Netlink消息，与内核进行通信。

```
#include <asm/types.h>
```

```
#include <linux/netlink.h>
```

```
1.int NLMSG_ALIGN(size_t len);
```

```
#define NLMSG_ALIGNTO 4
```

```
#define NLMSG_ALIGN(len) ((len)+NLMSG_ALIGNTO-1) & ~(NLMSG_ALIGNTO-1)
```

```
//宏NLMSG_ALIGN(len)用于得到不小于len且字节对齐的最小数值。
```

```
2.#define NLMSG_HDRLEN ((int) NLMSG_ALIGN(sizeof(struct nlmsghdr)))
```

```
//头部长度
```

```
3.int NLMSG_LENGTH(size_t len);
```

```
#define NLMSG_LENGTH(len) ((len)+NLMSG_ALIGN(NLMSG_HDRLEN))
```

```
//宏NLMSG_LENGTH(len)用于计算数据部分长度为len时实际的消息长度。它一般用于分配消息缓存。
```

```
4.int NLMSG_SPACE(size_t len);
```

```
#define NLMSG_SPACE(len) NLMSG_ALIGN(NLMSG_LENGTH(len))
```

```
//宏NLMSG_SPACE(len)返回不小于NLMSG_LENGTH(len)且字节对齐的最小数值，它也用于分配消息缓存。
```

```
5.void *NLMSG_DATA(struct nlmsghdr *nlh);
```

```
#define NLMSG_DATA(nlh) ((void*)((char*)nlh) + NLMSG_LENGTH(0))
```

```
//宏NLMSG_DATA(nlh)用于取得消息的数据部分的首地址，设置和读取消息数据部分时需要使用该宏。
```

```
6.struct nlmsghdr *NLMSG_NEXT(struct nlmsghdr *nlh, int len);
```

```
#define NLMSG_NEXT(nlh,len) ((len) -= NLMSG_ALIGN((nlh)->nlmsg_len), (struct nlmsghdr*)((char*)(nlh)) + NLMSG_ALIGN((nlh)->nlmsg_len)))
```

//宏NLMSG_NEXT(nlh,len)用于得到下一个消息的首地址,同时len也减少为剩余消息的总长度,该宏一般在一个消息被分成几个部分发送或接收时使用。

7.int NLMSG_OK(struct nlmsghdr *nlh, int len);

#define NLMSG_OK(nlh,len) ((len) >= (int)sizeof(struct nlmsghdr) && (nlh)->nlmsg_len >= sizeof(struct nlmsghdr) && (nlh)->nlmsg_len <= (len))



//宏NLMSG_OK(nlh,len)用于判断消息是否有len这么长。

8.int NLMSG_PAYLOAD(struct nlmsghdr *nlh, int len);

#define NLMSG_PAYLOAD(nlh,len) ((nlh)->nlmsg_len - NLMSG_SPACE((len)))

//宏NLMSG_PAYLOAD(nlh,len)用于返回payload的长度。

设置好上面消息后,我们就可以用sendto和recv分别发送和接收数据了。

分享到:  

[netlink和rtnetlink \(一\)](#) | [Fedora15,Fedora16 root登录和自动登录](#)

- 2011-09-19 01:24
- 浏览 777
- [评论\(0\)](#)
- 分类:[编程语言](#)
- [相关推荐](#)

评论



发表评论



[您还没有登录,请您登录后再发表评论](#)



haohetao

- 浏览: 393207 次
- 性别: 
- 来自: 郑州
-  我现在离线

最近访客 [更多访客>>](#)



[dylinshi126](#)



[dongbiying](#)



[wwwgui](#)



[sunburg](#)

文章分类

- [全部博客 \(474\)](#)
- [数据库 \(25\)](#)
- [操作系统 \(127\)](#)

- [PHP \(12\)](#)
- [前端 \(13\)](#)
- [手机 \(6\)](#)
- [电脑应用 \(30\)](#)
- [Linux网络开发 \(27\)](#)
- [C/C++ \(15\)](#)

社区版块

- [我的资讯](#) (0)
- [我的论坛](#) (6)
- [我的问答](#) (0)

存档分类

- [2014-08](#) (1)
- [2014-03](#) (3)
- [2014-01](#) (1)
- [更多存档...](#)

最新评论

- [exception01](#): 不错, 可以基本了解 概念 +1
[什么是动态语言和静态语言?](#)
- [nature_XD](#): 作者前两句解释的很清晰, 赞!
[SMI接口, SMI帧结构, MDC/MDIO](#)
- [chumignze](#): 楼主能给我发个实例吗, 现在急需解决这个问题, 谢谢3071081 ...
[highcharts,highstock用ajax延迟动态加载数据](#)
- [yaguang_li](#): 果然是有道划词搜索, 谢谢。
[SecureCRT鼠标双击或拖成变成Ctrl+C的解决方法](#)
- [pennyxi](#): ...
[DreamWeaver CS5 中文版可激活注册码](#)

声明: ITeye文章版权属于作者, 受法律保护。没有作者书面许可不得转载。若作者同意转载, 必须以超链接形式标明文章原始出处和作者。

© 2003-2014 ITeye.com. All rights reserved. [京ICP证110151号 京公网安备110105010620]