# NETLINK 编程方法说明与IPSET 源码流程解析

部　　门: 网络安全线

拟　　制: 2016 年 11 月 14 日

作　　者: 薛萌

# 0 版本记录

| 序号 | 版本号 | 生成日期 | 主 要 修 改 记 录 | 第一作者 | 审核人 | 备注 |
|---|---|---|---|---|---|---|
| 1 | 1.0 | 2016.11.14 | 首次生成。 | 薛萌 | | 监控网口状态程序由陈阳欣提供 |
| 2 | 2.0 | 2017.1.6 | 增加 ipset 代码解析 | 薛萌 | | |
| 3 | 2.1 | 2017.1.9 | 增加 ipset 对 netlink 的扩展及 ipset 在 netfilter 中 match 规则的说明 | 薛萌 | | |
| 4 | 2.2 | 2017.2.27 | 增加 ipset 超时机制的说明 | 薛萌 | | |
| 5 | 3.0 | 2017.3.3 | 增加 ipset 创建，增加，匹配的流程及 hash 的简单使用说明 | 薛萌 | | |
| 6 | 3.1 | 2017.3.14 | 增加了对 ipset 配置出错的一些简单说明 | 薛萌 | | |
| 7 | 3.2 | 2017.3.16 | 对超时机制的解释做了一些增加和修改 | 薛萌 | | |

# 1 引言

## 1.1 相关文档

**表 1. 相关设计文档列表**

| 序号 | 文档名称 | 版本号 | 作者 | 备注 |
|---|---|---|---|---|
| 1 | Netlink3 netlink7 rtnetlink3 rtnetlink7 | | Linux man 手册 | |
| 2 | /include/uapi/linux/rtnetlink.h | | Linux 头文件 | |

# 2 引用的技术标准及规范

**表 2. 引用的技术标准及规范**

| 序号 | 文档名称 | 版本号 | 作者 | 发表日期 | 出版单位/来源 | 备注 |
|---|---|---|---|---|---|---|
| 1 | RFC3549 | | | 2003 | | |
| 2 | | | | | | |

## 2.1 术语

**表 3. 术语及解释**

| 序号 | 术语 | 英文 | 说明 |
|---|---|---|---|
| 1 | CM | Cache management | |
| 2 | | | |
| | | | |
| | | | |

## 2.2 适用范围

本文档用于解释 BD 产品控制平面中，CM 通过内核同步数据所用的 NETLINK 编程方法，其中通过 ROUTE 及 LINK 的范例解释了基本的使用方法。

# 3 总体概述

## 3.1 测试环境

### 3.1.1 硬件环境

目前仅在虚拟机上进行测试

### 3.1.2 软件环境

操作系统：Linux (ubuntu 16.04)

程序设计语言：C

## 3.2 背景介绍

目前路由转发方案为软转发，所有的数据包需要在用户态进行路由判决。但是使用 iptables 配置的转发规则，在内核态的 netfilter 中。若再将路由数据复制进内核态进行判决则速度太慢，且认为统一的流程利于管理，因此，并不采用直接拦截 iptables 配置的方法，而是希望使用 netlink 将规则同步到用户态。

## 3.3 设计需求说明

目前仅采用 get 方法，或被动接收方法来获取内核态的路由或链接数据。

# 4 netlink 消息数据结构分析

## 4.1 netlink 消息介绍

netlink socekt 是一种用于在内核态和用户态进程之间进行数据传输的特殊的 IPC。它通过为内核模块提供一组特殊的 API，并为用户程序提供了一组标准的 socket 接口的方式，实现了一种全双工的通讯连接（相比/proc、ioctl）。类似于 TCP/IP 中使用 AF_INET 地址族一样，netlink socket 使用地址族 AF_NETLINK。netlink 具有以下特点：

① 支持全双工、异步通信(当然同步也支持)

② 用户空间可使用标准的 BSD socket 接口(但 netlink 需要使用者手动进行协议包的构造与解析过程，因此 netlink 手册推荐使用 libnl 等第三方库)

③ 在内核空间使用专用的内核 API 接口

④ 支持多播(因此支持“总线”式通信，可实现消息订阅)

⑤ 在内核端可用于进程上下文与中断上下文

## 4.2 netlink 消息结构

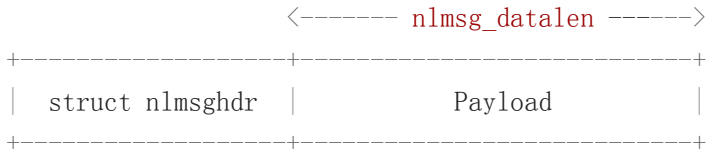
NETLINK 消息由一个定长的 NETLINK 消息头，和实际载荷组成。当 NETLINK 消息用在不用的用途时，载荷的种类和内容也不尽相同。其中，netlink 已有定义的协议族如下：

- 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

我们主要使用 NETLINK_ROUTE 协议，此时协议支持的消息类型为：

- 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

一个 netlink 消息的结构如下图：




payload 部分可能包含第二层级的消息头，及其他消息头所对应的实际有效载荷，具体见下文。

### 4.2.1 netlink 消息头数据结构

Netlink 的报文由消息头和消息体构成，struct nlmsghdr 即为消息头。消息头定义在文件里，由结构体 nlmsghdr 表示：

```
struct nlmsghdr
{
    __u32 nlmsg_len;
    __u16 nlmsg_type;
    __u16 nlmsg_flags;
    __u32 nlmsg_seq;
    __u32 nlmsg_pid;
};
```

消息头中各成员属性的解释及说明：

(1) nlmsg_len：整个 netlink 消息的长度，按字节计算。包括了 Netlink 消息头本身。

(2) nlmsg_type：消息的类型，即是数据还是控制消息。目前(内核版本 2.6.21)Netlink 仅支持四种类型的控制消息，如下：

a) NLMSG_NOOP-空消息，什么也不做；

b) NLMSG_ERROR-指明该消息中包含一个错误；

netlink 不可靠，可能会因内存用尽而丢包，可以通过设置在 NLM 消息头的 nlmsg_flags 位设置 NLM_F_ACK 标志向接收端要求 ACK 这个 ACK 实际就是一个 NLMSG_ERR 包，只不过其 error field 设置为 0，内核会尽力回复此确认消息，建议应用程序也这么回复

nlmsg_type 为 NLMSG_ERROR 时结构为 nlmsgerr 见 netlink7

c) NLMSG_DONE-一次请求可能会回复多个 netlink 消息，那么这一系列消息中，最后一条消息的类型为 NLMSG_DONE，其余所有消息的 nlmsg_flags 属性都被设置 NLM_F_MULTI 位，以说明后面还有后续的消息。

d) NLMSG_OVERRUN-暂时没用到。

(3) nlmsg_flags：附加在消息上的额外说明信息，如上面提到的 NLM_F_MULTI

```
#define NLM_F_REQUEST           1     /* It is request message.     */
#define NLM_F_MULTI         2     /* Multipart message, terminated by NLMSG_DONE */
#define NLM_F_ACK           4     /* Reply with ack, with zero or error code */
#define NLM_F_ECHO          8     /* Echo this request           */
#define NLM_F_DUMP_INTR         16    /* Dump was inconsistent due to sequence change */
/* GET 操作追加的宏 */
#define NLM_F_ROOT  0x100     /* specify tree root       */
#define NLM_F_MATCH 0x200     /* return all matching     */
#define NLM_F_ATOMIC        0x400     /* atomic GET          */
#define NLM_F_DUMP  (NLM_F_ROOT|NLM_F_MATCH)
/* NEW 操作追加的宏 */
#define NLM_F_REPLACE       0x100     /* Override existing           */
#define NLM_F_EXCL  0x200     /* Do not touch, if it exists      */
#define NLM_F_CREATE        0x400     /* Create, if it does not exist */
#define NLM_F_APPEND        0x800     /* Add to end of list          */
```

(4) nlmsg_seq: netlink 类似数据报,有丢包可能,可以使用序号来控制,发送时自己写入序号,内核收到后会提取序号然后在填入回复给用户的报文中，若一次请求产生了多个 netlink 消息回复，那么这多个回复的 seq 都是一样的，如果是内核主动向用户发消息,这里的 seq

一直都是 0

(5) nlmsg_pid： 如果当前的消息发送者，希望对端再进行交互式的回应,这里 pid 需要赋值，以标识发送方的身份，需要唯一。一般可以直接用 getpid()函数获取，但是此值并不和真实进程的 pid 有必然联系，举例，若一个进程有多个 netlink 消息连接时，此处必然不能单纯的填写进程的 pid。但是注意，在测试中发现，过小的数据可能会出错，例如使用 100，所以如果非要使用自定的立即数，请指定的大一些，否则这里会出现问题。

### 4.2.2 netlink 消息的一些操作宏

1.int NLMSG_ALIGN(size_t len);
#define NLMSG_ALIGNTO 4
#define NLMSG_ALIGN(len) ( ((len)+NLMSG_ALIGNTO-1) & ~(NLMSG_ALIGNTO-1) )
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)))
获取 netlink 消息的头部长度

3.int NLMSG_LENGTH(size_t len);
#define NLMSG_LENGTH(len) ((len)+NLMSG_ALIGN(NLMSG_HDRLEN))
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))
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)))
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。注意此时的“数据部分”是相对于 netlink 消息而言，此处返回结果一般指向子结构消息的消息头

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)))

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，即在 recv 后，一个 buffer 中有多个 netlink 消息时进行循环处理。

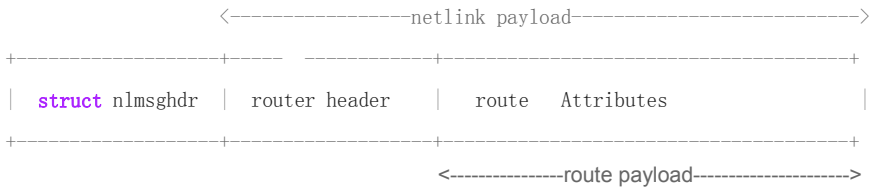
7.int NLMSG_OK(struct nlmsghdr *nlh, int len);
#define NLMSG_OK(nlh,len) ((len) >= (int)sizeof(struct nlmsghdr) && (nlh)->nlmsg_len >= sizeof(struct nlmsghdr) && (nlh)->nlmsg_len <= (len))
用于判断消息是有 len 的长度。常在解析数据前进行合法性判断。

8.int NLMSG_PAYLOAD(struct nlmsghdr *nlh, int len);
#define NLMSG_PAYLOAD(nlh,len) ((nlh)->nlmsg_len - NLMSG_SPACE((len)))

用于返回 payload 的长度。

## 4.3 netlink 消息的负载类型

前文描述过，netlink 消息的实际负载内容，是由协议和 netlink 消息类型决定的。此处根据 route 和 link 类型为例，说明实际的数据载荷格式。




如上图，一个路由类型的消息，首先是一个路由消息头，后面才是具体的数据。

### 4.3.1 route 消息的负载类型

对于路由操作(RTM_NEWROUTE, RTM_DELROUTE, RTM_GETROUTE)，包含 rtmsg 这么一个消息头:

```
struct rtmsg {
        unsigned char rtm_family;     /* Address family of route */
        unsigned char rtm_dst_len;    /* Length of destination */
        unsigned char rtm_src_len;    /* Length of source */
        unsigned char rtm_tos;        /* TOS filter */

        unsigned char rtm_table;      /* Routing table ID */
        unsigned char rtm_protocol; /* Routing protocol; see below */
        unsigned char rtm_scope;      /* See below */
        unsigned char rtm_type;       /* See below */

        unsigned int    rtm_flags;
    };
```

rtm_family：路由地址族，一般是 AF_INET

rtm_dst_len　rtm_src_len：是目的主机或者目的路由(源路由)的掩码位数　都 0 表示通配

rtm_type:

| | | |
|---|---|---|
| RTN_UNSPEC | 0 | 未知的路由 |
| RTN_UNICAST | 1 | 网管或者直接路由 |
| RTN_LOCAL | 2 | 本地接收 |
| RTN_BROADCAST | 3 | 广播式本地接收、发送 |
| RTN_ANYCAST | 4 | 本地广播接收，单播发送 |
| RTN_MULTICAST | 5 | 多播路由 |
| RTN_BLACKHOLE | 6 | 丢弃 |
| RTN_UNREACHABLE | 7 | 目标不可达 |
| RTN_PROHIBIT | 8 | 管理禁止 |

| | | |
|---|---|---|
| RTN_THROW | 9 | 不在此表中 |
| RTN_NAT | 10 | Translate this address |
| RTN_XRESOLVE | 11 | Useexternal resolver |

rtm_protocol

| | | |
|---|---|---|
| RTPROT_UNSPEC | 0 | 未知的路由来源 |
| RTPROT_REDIRECT | 1 | 通过 ICMP 转发建立路由（目前未使用） |
| RTPROT_KERNEL | 2 | 内核转发建立路由 |
| RTPROT_BOOT | 3 | 该路由在系统启动时建立 |
| RTPROT_STATIC | 4 | 该路由由系统管理员建立 |

大于 RTPROT_STATIC 的值不被内核识别(interpreted)，而是用作用户信息。

rtm_scope：

| | | |
|---|---|---|
| RT_SCOPE_UNIVERSE | 0 | 用户定义值 |
| RT_SCOPE_SITE | 200 | |
| RT_SCOPE_LINK | 253 | 目的地址 |
| RT_SCOPE_HOST | 254 | 本地地址 |
| RT_SCOPE_NOWHERE | 255 | 为不存在 destination 预留 |

取值在 RT_SCOPE_UNIVERSE 和 RT_SCOPE_SITE 之间的为用户可用。

rtm_flags：

| | | |
|---|---|---|
| RTM_F_NORIFY | 0x100 | 通知用户路由改变 |
| RTM_F_CLONED | 0x200 | 该路由为克隆的(cloned) |
| RTM_F_EQUALIZE | 0x400 | Multipath equalizer:NI |
| RTM_F_PREFIX | 0x800 | Prefix addresses |

rtm_table 用于区别路由表：

| | | |
|---|---|---|
| RT_TABLE_UNSPEC | 0 | 未指定的路由表 |
| RT_TABLE_DEFAULT | 253 | 默认的表（default table） |
| RT_TABLE_MAIN | 254 | 主表（main table） |
| RT_TABLE_LOCAL | 255 | 本地表（local table） |

路由消息头后是实际载荷，实际载荷内容是若干个 rtattr 结构和其相应载荷

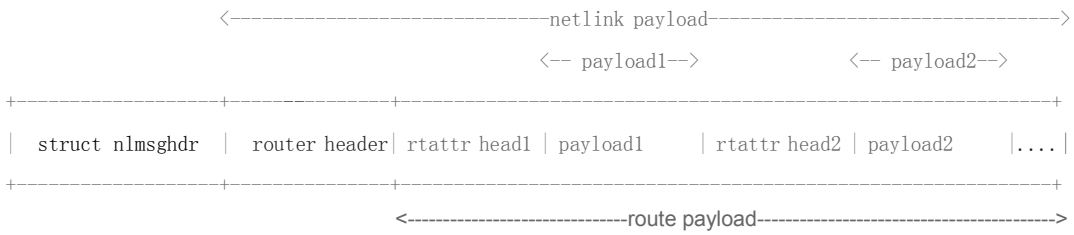
```
struct rtattr
{
        unsigned short rta_len;     /* Length of option */
        unsigned short rta_type;    /* Type of option */
        /*数据部分在此结构体后*/
};
```

rta_type：

| | | |
|---|---|---|
| RTA_UNSPEC | 0 | |
| RTA_DST | 1 | 路由目的地址 |
| RTA_SRC | 2 | 路由源地址 |

| | | |
|---|---|---|
| RTA_IIF | 3 | Input interface index |
| RTA_OIF | 4 | Output interface index |
| RTA_GATEWAY | 5 | 路由网关 |
| RTA_PRIORITY | 6 | 路由的优先级 |
| RTA_PREFSRC | 7 | |
| RTA_METRICS | 8 | Route metrics |
| RTA_MULTIPATH | 9 | |
| RTA_PROTOINFO | 10 | |
| RTA_FLOW | 11 | |
| RTA_CACHEINFO | 12 | |
| RTA_SESSION | 13 | |
| RTA_MP_ALGO | 14 | |

每个 rtattr 消息的实际载荷，就是本次 ROUTE 类型的 netlink 消息实际上最后得到的数据部分了，那么此时消息结构如下图：




那么如上图所示，实际的载荷即若干个 rtattr 及相应的 payload

### 4.3.2 link 消息的负载类型

同理，link 消息类似于上面的 route 消息

```
struct ifinfomsg {
/* struct ifinfomsg passes link level specific information, not dependent on network protocol.*/
unsigned char    ifi_family;    /* AF_UNSPEC */
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 */
};
```

ifi_index 为这个接口的接口索引

ifi_flags 的取值为：

```
/* Standard interface flags (netdevice->flags). */
#define IFF_UP          0x1     /* interface is up          */
#define IFF_BROADCAST   0x2     /* broadcast address valid  */
#define IFF_DEBUG       0x4     /* turn on debugging        */
#define IFF_LOOPBACK    0x8     /* is a loopback net        */
```

```
#define IFF_POINTOPOINT 0x10            /* interface is has p-p link       */
#define IFF_NOTRAILERS   0x20            /* avoid use of trailers       */
#define IFF_RUNNING 0x40            /* interface RFC2863 OPER_UP       */
#define IFF_NOARP     0x80            /* no ARP protocol          */
#define IFF_PROMISC 0x100           /* receive all packets          */
#define IFF_ALLMULTI      0x200          /* receive all multicast packets*/
#define IFF_MASTER   0x400           /* master of a load balancer       */
#define IFF_SLAVE     0x800          /* slave of a load balancer */
#define IFF_MULTICAST     0x1000          /* Supports multicast           */
#define IFF_PORTSEL 0x2000              /* can set media type           */
#define IFF_AUTOMEDIA     0x4000          /* auto media select active */
#define IFF_DYNAMIC 0x8000          /* dialup device with changing addresses*/
#define IFF_LOWER_UP      0x10000         /* driver signals L1 up         */
#define IFF_DORMANT 0x20000         /* driver signals dormant      */
#define IFF_ECHO      0x40000         /* echo sent packets             */
```

### 4.3.3 route 消息的一些操作宏

1. #define RTA_ALIGN(len)( ((len)+RTA_ALIGNTO-1) & ~(RTA_ALIGNTO-1) ) /*对齐*/

2.#define RTA_OK(rta,len) ((len) >= (int)sizeof(struct rtattr) && \
(rta)->rta_len >= sizeof(struct rtattr)&& \
(rta)->rta_len <= (len))

判断长度，在解析前做合法性判断

3. #define RTA_NEXT(rta,attrlen) ((attrlen) -= RTA_ALIGN((rta)->rta_len),\
(struct rtattr*)(((char*)(rta)) +RTA_ALIGN((rta)->rta_len)))

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

4. #define RTA_LENGTH(len) (RTA_ALIGN(sizeof(struct rtattr)) + (len))

计算对齐后的长度

5. #define RTA_SPACE(len) RTA_ALIGN(RTA_LENGTH(len))

返回数据的对齐的最小数值*

6. #define RTA_DATA(rta) ((void*)(((char*)(rta)) + RTA_LENGTH(0)))

取得数据部分首地址

7. #define RTA_PAYLOAD(rta)((int)((rta)->rta_len) - RTA_LENGTH(0))

计算载荷总长度

### 4.3.4 link 消息的一些操作宏

1、#define IFLA_RTA(r) ((struct rtattr*)(((char*)(r)) + NLMSG_ALIGN(sizeof(struct ifinfomsg))))

获取 rta 载荷部分地址

2、#define IFLA_PAYLOAD(n)　　NLMSG_PAYLOAD(n,sizeof(struct ifinfomsg))
计算载荷长度

实际上，这些操作宏除了名字头不一样，其他差别并不大，作用，名字，用法，参数，都是相似的，因此并不需要去一个一个强记，而且，相关的 define 在头文件中都有，需要时在头文件中查找即可。

## 5　范例代码描述

下文中,通过对路由信息的获取和解析，以及对网卡状态的监视，来演示如何使用 netlink 方法编程。

### 5.1　获取路由信息的范例程序

编写一小段代码，从内核中获取路由消息。

#### 5.1.1　定义相应的结构体

```
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<net/if.h>
#include<asm/types.h>
#include<linux/netlink.h>
#include<linux/rtnetlink.h>
#include<linux/socket.h>
#include<errno.h>

    #define REQUEST_BUFF 20
    #define RECV_BUFF 4096
    #define IF_MAX 32

  //设置消息序列号，以此确定收到的回应是否来自我们发出的请求
  int msgSeq = 0;

  //用于解析ROUTE类型的消息中，rtmsg后跟的，不定数量的实际载消息载荷
  struct rtattr *rtAttr;
  int rtlen;

  //每次socket读取的长度
  int readlen = 0;

  //定义socket消息，netlink消息头,及ROUTE类型netlink消息的负载结构
  struct sockaddr_nl nl;
  struct nlmsghdr *netlink_head = NULL;
  struct rtmsg *route_msg = NULL;

  //本例中用于将若干个netlink消息放入接收buffer后，用于循环处理数据用的指针
  struct nlmsghdr *netlink_temp = NULL;

  //存放解析后的路由信息
  struct route_info *rtInfo = NULL;

  //用于收取数据
  char recvbuff[RECV_BUFF] = {0};
  struct iovec iov = {recvbuff, sizeof(recvbuff)};
  struct sockaddr_nl recv_nl;
  struct msghdr msg = {(void *)&recv_nl, sizeof(recv_nl), &iov, 1, NULL, 0, 0};

  //保存解析后的数据
  struct rtattr *tb[IF_MAX + 1];
```

注意这里定义的 iov 是给下面的 recvmsg 函数使用，用法类似于 readv/writev 函数，可以在

一次 I/O 操作中写入到多个地址，也可以从多个地址读取。类型如下，注意其中数据区的大小并不代表实际长度，而是代表我们目的操作的地址，有多少个。

```
struct iovec
{
    void    *iov_base    /* 数据区的起始地址 */
    size_t  iov_len      /* 数据区的大小 */
}
```

当然此处直接使用 recv 也可以。

### 5.1.2 创建 netlink 消息的 socket

```
//创建socket
sock = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
if(sock < 0)
{
    printf("socket error %s\n",strerror(errno));
    return -1;
}

memset(&nl, 0, sizeof(nl));
nl.nl_family = AF_NETLINK;
//暂时用不到广播  这里填0就行
nl.nl_groups = 0;

ret = bind(sock, (struct sockaddr*)&nl, sizeof(nl));
if(ret < 0)
{
    printf("error socket bind %s\n",strerror(errno));
    close(sock);a
    return -1;
}

//给消息申请内存
netlink_head = (struct nlmsghdr*)malloc(NLMSG_SPACE(REQUEST_BUFF));
if(netlink_head == NULL)
{
    printf("error get netlink_head mem %s\n",strerror(errno));
    close(sock);
    return -1;
}
```

这里的创建方式和普通的 socket 并没有很大差别，注意 socket 的 domain 和协议即可。此处是一个发送请求，所以申请的内存很小就可以(REQUEST_BUFF 定为大于 16 的值)。多播组的数据填 0，因为此处我们只处理单播消息。

```
//消息类型为路由的GET请求
netlink_head->nlmsg_type = RTM_GETROUTE;
//这个长度是整个netlink消息(包括负载)的长度
netlink_head->nlmsg_len = NLMSG_SPACE(REQUEST_BUFF);

//以下是get请求的追加标志
netlink_head->nlmsg_flags = NLM_F_REQUEST|NLM_F_DUMP;

//以下两个字段用来追踪消息
netlink_head->nlmsg_seq = ++msgSeq;
netlink_head->nlmsg_pid = getpid();

//发送请求
ret = send(sock, netlink_head, netlink_head->nlmsg_len, 0);
if(ret < 0)
{a
    printf("error send %s\n",strerror(errno));
    close(sock);
    return -1;
}

free(netlink_head);
```

此处 netlink 消息头中字段的填写方法可以参照 4.2 节中的相应说明，以上的填法表明我们这是一个想要获取路由的请求消息，并填写了本进程的进程 ID，和序列号。实际上，序列号也可以用与时间相关的随机数生成器得出。

### 5.1.3 接收 netlink 消息

```
while(1)
{
    readlen = recvmsg(sock, &msg, 0);
    if(readlen < 0)
    {
        printf("error socket recv %s\n",strerror(errno));
        close(sock);
        return -1;
    }

    //说明已经接收不到数据了
    if(readlen == 0)
    {
        break;
    }

    //分析数据
    for(netlink_head = (struct nlmsghdr*)recvbuff; NLMSG_OK(netlink_head,readlen); netlink_head = NLMSG_NEXT(netlink_head,readle
    {
```

这里在一个 while 循环中接收数据，此处有几个地方需要说明：

1、一次请求后，可能会生成多个 netlink 消息，这些消息内核会做多次的 send 操作。实际上，并不会因为我们接收缓冲区长度定的很大，就能使一次 recv 接收到所有的数据。在一次实际测试中，我们将接收缓冲区大小定为 8196，并使用了 recv 函数代替 recvmsg 函数，依然收到了 3 次内核回应的消息，分别为 656，608，20。这些数据都远远低于我们预设的缓冲区大小及读 socket 的长度。

2、一次 recv 读取的数据会包含若干 netlink 消息，每个 netlink 消息都有独立的消息头。因此想要在 recv 操作之后直接对 netlink 消息进行合法性判断的做法，是不正确的。同样，把全部的数据 recv 到一个 buffer 中再解析，也可能会出现问题，即一个长度错误的消息影响了后面的数据。因此建议 recv 到一次数据后，马上对其进行解析。

3、注意 netlink 也是不可靠的协议，因此不能假设收到的数据包是按顺序的。

4、一般不会出现我们 recv 多次时，其中某一次 recv 的数据中，掺杂了非 netlink 消息回应的情况，因为前面 socket 创建时指明了协议类型。

5、此处只是范例，while(1)循环是比较消耗资源的操作。可以用 epoll 来监控此 socket，或者使用 Libevent 库也可

### 5.1.4 消息的合法性判断

```
//分析数据
for(netlink_head = (struct nlmsghdr*)recvbuff; NLMSG_OK(netlink_head,readlen); netlink_head = NLMSG_NEXT(netlink_head,readlen))
{
    //消息出错检测
    if(!NLMSG_OK(netlink_head, readlen) || (netlink_head->nlmsg_type == NLMSG_ERROR))
    {
        printf("error netlink message %s\n",strerror(errno));
        break;
    }

    //最后一条消息，可以结束了
    if(netlink_head->nlmsg_type == NLMSG_DONE)
    {
        goto END_LABEL;
    }

    //判断此netlink消息是否是我们发出请求的回应
    if((netlink_head->nlmsg_seq != msgSeq) || (netlink_head->nlmsg_pid != getpid()))
    {
        printf("this netlink message is not we need\n");
        continue;
    }
```

由于每个消息都有自己独立的头，因此在 5.1.3 中解释了为什么要在解析数据时才进行合法性判断。此处在 for 循环中，可以独立的检测每一条消息，对于长度不对的消息，是跳过还是退出，由编码者自行决定。

同时，需要检测每一条消息的 seq 和 pid，这相当于是消息的身份证，确定当前的回应是由我们之前的请求触发而来。

type 为 NLMSG_DONE 标志着这一系列的消息回复结束，可以退出本次接收了。通常这一次的 netlink 消息是一个只含有包头的空消息。

### 5.1.5 ROUTE 类型荷载的解析

```
//开始正常解析
rtInfo = (struct route_info *)malloc(sizeof(struct route_info));
memset(rtInfo, 0, sizeof(struct route_info));

//先找到路由信息的结构rtmsg,作为netlink_msg的载荷
route_msg = (struct rtmsg*)NLMSG_DATA(netlink_head);
rtlen  = RTM_PAYLOAD(netlink_head);


rtAttr = (struct rtattr *)RTM_RTA(route_msg);

//清空tb buffer
memset(tb, 0, sizeof(tb));

//循环处理消息到结束为止
for( ; RTA_OK(rtAttr,rtlen); rtAttr = RTA_NEXT(rtAttr,rtlen))
{
    if(rtAttr->rta_type < IF_MAX)
    {
        tb[rtAttr->rta_type] = rtAttr;

    }
}
```

在 4.3 节中提到过，不同类型的消息会携带不同的载荷。对于 route 类型，则是多了一个 rtmsg 类型的消息头，并在后面跟了数量不定的 rtattr 消息。这些 rtattr 消息就是我们最终需要获得的数据了。

此处使用 tb 数组保存，具体可以查看头文件 /include/uapi/linux/if_link.h 和 include/uapi/linux/if_addr.h 中的枚举类型。这样的保存方法是因为，一般一个 netlink 消息只表达一类的消息，即一个接口的 UP 和 DOWN 即便是切换的很快，也不会合并到一个 netlink 消息里面发送。

### 5.1.6 多播消息的接收

上面的代码是通过主动发消息让内核响应并回复数据，这是一种主动获取的方式。同样，我们可以通过主动加入多播组，并进行监控，在某些事件发生，例如连接建立删除时，收到内核主动发出的消息。

每一个 netlink 协议都支持最多 32 个的多播组，加入多播组就能收到相应的消息。

可以注意到 5.1.2 节中，我们对 nl_group 填了 0。现在就需要对这个位进行相应的赋值，让当前的 socket 可以监听到 LINK 消息。以监听接口为例，需要加入的多播组在 /include/uapi/linux/rtnetlink.h 中

```
#ifndef __KERNEL__
/* RTnetlink multicast groups - backwards compatibility for userspace */
```

```
#define RTMGRP_LINK             1
#define RTMGRP_NOTIFY           2
#define RTMGRP_NEIGH            4
#define RTMGRP_TC           8
#define RTMGRP_IPV4_IFADDR    0x10
#define RTMGRP_IPV4_MROUTE    0x20
#define RTMGRP_IPV4_ROUTE     0x40
#define RTMGRP_IPV4_RULE      0x80
#define RTMGRP_IPV6_IFADDR    0x100
#define RTMGRP_IPV6_MROUTE    0x200
#define RTMGRP_IPV6_ROUTE     0x400
#define RTMGRP_IPV6_IFINFO    0x800
#define RTMGRP_DECnet_IFADDR      0x1000
#define RTMGRP_DECnet_ROUTE       0x4000
#define RTMGRP_IPV6_PREFIX    0x20000
#endif
```

部分解释：

| | |
|---|---|
| RTMGRP_LINK | 网络接口的创建/删除/UP/DOWN 事件 |
| RTMGRP_IPV4_ROUTE | IPV4 地址的增加/删除事件 |
| RTMGRP_IPV4_IFADDR | 主机 IP 地址发生变化的事件 |
| RTMGRP_NEIGH | 用于用于通知 L3 到 L2 的地址映射的改变 |

此处文档极少，网上也没有找到解释。目前确定的用法并不多，希望以后在实际使用中有同事可以补充。

### 5.1.7 LINK 消息的接收

相对于以上代码，需要修改的地方，一是要加入相应的多播组，二是针对 LINK 类型的消息进行解析。

```
struct sockaddr_nl addr;
memset(&addr,0,sizeof(addr));
addr.nl_family=AF_NETLINK;
addr.nl_groups=RTMGRP_LINK|RTMGRP_IPV4_IFADDR;
addr.nl_pid=getpid();
```

注意这里 nl_group 的取值，我们想要监控主机 IP 地址和接口的事件变化。

```
//判断此netlink消息是否是我们发出请求的回应
if((netlink_head->nlmsg_seq != msgSeq) || (netlink_head->nlmsg_pid != getpid()))
{
    printf("this netlink message is not we need\n");
    continue;
}

if(netlink_head->nlmsg_type == RTM_NEWADDR)
{
    ......
}
else if(netlink_head->nlmsg_type==RTM_DELADDR)
{
    ......
}
else if(netlink_head->nlmsg_type==RTM_NEWLINK)
{
    struct ifinfomsg *ifi=(struct ifinfomsg*)NLMSG_DATA(netlink_head);
    struct rtattr *rta = (struct rtattr *)IFLA_PAYLOAD(netlink_head);

    if((ifi->ifi_flags & IFF_LOOPBACK)!=0)
    {
        exit(0);
    }

    while(RTA_OK(rta,len)){
    if(rta->rta_type == IFLA_IFNAME){
        char ifname[IFNAMSIZ];
        char *action;
        snprintf(ifname,sizeof(ifname),"%s",(char*)RTA_DATA(rta));
        action=(ifi->ifi_flags & IFF_RUNNING) ? "UP":"DOWN";
        printf("%s link %s\n",ifname,action);
    }
    rta=RTA_NEXT(rta,len);
}
```

则解析部分代码如上即可。注意 ifi_flags 的标志。

### 5.1.8 测试及结果

由于目前对 IP 协议不是很熟悉，规范的测试结果待后续补充（补充为相关的一个测试报告）。

## 6 IPSET 代码流程及相应的 netlink 封装

ipset 也是 netfilter 的一部分。在规则较多的时候，特别是针对 ip 地址的规则较多时，由于 iptables 线性存储和过滤的特性，会影响整体性能，且 iptables 的配置规则对顺序敏感。而 ipset 可以创建对整个地址集合的规则，这时对于性能和配置复杂度的改善都比较大。

### 6.1.1 ipset 配置方法

举一个简单的配置例子：

ipset -q create myhash hash:net
ipset add myhash 1.1.1.1
ipset add myhash 1.1.2.0/24
iptables -A FORWARD -m set --match-set myhash src -j DROP

以上规则：

1、创建了一个叫 myhash 的 ipset 地址集，使用静默模式(-q,强制输出到标准输出和标准错误)，保存类型为 hash:net

2、增加了一个地址为 1.1.1.1 的 ip 到 myhash 集内。

3、增加了一个 1.1.2.0/24 的网段到 myhash 集内。

4、在 Iptables 中增加一条规则，在 FOWARD 链上，使源地址在此集内的包都做 drop 处理。

这样就是一条包含命令，可选选项，扩展选项，地址类型的 ipset 配置了。如此配置可以极大的简化 iptables 配置规则的数量，其中 hash:net 为保存类型和数据类型，数据类型可为 ip，port 等，也可以使用保存类型 list 来创建“集合的集合”。

### 6.1.2 ipset 配置流程代码解析

```
/* Initialize session */
session = ipset_session_init(printf);
if (session == NULL)
    return exit_error(OTHER_PROBLEM,
        "Cannot initialize ipset session, aborting.");

ret = parse_commandline(argc, argv);
```

图 1

拿 6.1.1 中的配置为例，首先初始化 session 结构，这个结构贯穿了整个 ipset 配置流程。第一步初始化最重要的是注册相关的回调函数至 ipset_mnl_transport 上，注意这里回调函数都使用了 libmnl 库，是 netlink 方法的一种封装，提供了 4 个函数，很简单，分别是初始化，删除，填充头部及发送/接收消息(收发消息在一个函数中，即发完马上接收并保存)。

```
/* The single transport method yet */
session->transport = &ipset_mnl_transport;
```

图 2

```
const struct ipset_transport ipset_mnl_transport = {
    .init   = ipset_mnl_init,
    .fini   = ipset_mnl_fini,
    .fill_hdr = ipset_mnl_fill_hdr,
    .query  = ipset_mnl_query,
};
```

图 3

往下进入 parse_commandline 函数，在这里是命令行的真正解析过程。此处特殊的是，ipset 是为了将一系列数据包的特征做到一个集合里，真正的匹配还是需要靠 iptables，所以其功能和目的都相对单一，配置选项比较少，因此并没有使用过多的运行时初始化和外部加载等方法。对于 6.1.1 中我们可能配置的一些命令，ipset 采取的方法是直接创建一个涵盖所有配置选项的结构体，使用循环的方式查找关键字并赋值。

对于命令 ipset -q create myhash hash:net，处理步骤如下：

1、首先在 ipset_envopts 中查找我们配置了哪个选项参数，

```
const struct ipset_envopts ipset_envopts[] = {
    { .name = { "-o", "-output" },
      .has_arg = IPSET_MANDATORY_ARG,   .flag = IPSET_OPT_MAX,
      .parse = ipset_parse_output,
      .help = "plain|save|xml\n"
          "       Specify output mode for listing sets.\n"
          "       Default value for \"list\" command is mode \"
          "       and for \"save\" command is mode \"save\".",
    },
    { .name = { "-s", "-sorted" },
      .parse = ipset_envopt_parse,
      .has_arg = IPSET_NO_ARG,  .flag = IPSET_ENV_SORTED,
      .help = "\n"
          "        Print elements sorted (if supported by the s
    },
    { .name = { "-q", "-quiet" },
      .parse = ipset_envopt_parse,
```

图 4

点进此结构体，可以看出已经以列表的方式固化了每个可能选项的名称(以及别名，这是查找匹配的依据)，是否还要跟后续参数，后续参数有几个，help 信息，解析回调函数等。后续对于创建命令也是用类似的方式。

```
/* First: parse core options */
for (opt = ipset_envopts; opt->flag; opt++) {
    for (i = 1; i < argc; ) {
        if (!ipset_match_envopt(argv[i], opt->name)) {
            i++;
            continue;
        }
        /* Shift off matched option */
        ipset_shift_argv(&argc, argv, i);
        switch (opt->has_arg) {
        case IPSET_MANDATORY_ARG:
```

图 5

在匹配到相应参数后，调用 ipset_shift_argv，将命令行整体前移，这样现在的命令行就变成了 create myhash hash:net.  ipset 采用的方式是按顺序，处理一个删除一个，找到选项先查看后面参数与选项规定的参数数量是否匹配，存储需要的选项，继续执行。

这里 ipset 有一个互动模式，我们暂时先不理会。继续往下走，匹配命令参数，这里匹配到了 command，相同的方法，查到该命令有两个参数，移动两次，保存了 myhash 这个名称和 hash:net 的类型参数，并保存起来。走到这里应该注意到了，贯穿整个处理过程的，是一个 session 结构体。这个结构体保留了本次配置会话的所有内容。

到这里数据解析基本完成，开始调用 ipset_cmd 发送数据。第一步是初始化 netlink 的一些参数。注意，在这里，已经将未来解析 netlink 消息的函数注册到了 session 的 handle 上(handle 用来管理一切与内核通信相关的参数)。

```
handle->cb_ctl = cb_ctl;
```

```
static mnl_cb_t cb_ctl[] = {
    [NLMSG_NOOP] = callback_noop,
    [NLMSG_ERROR] = callback_error,
    [NLMSG_DONE]  = callback_done,
    [NLMSG_OVERRUN] = callback_noop,
    [NLMSG_MIN_TYPE] = callback_data,
};
```

图 6 7

这里左边的宏就是 netlink 消息的一些返回类型，可以在上面 4.2.1 节中看到。根据不同的消息返回类型，这里定义了不同的回调函数。NLMSG_MIN_TYPE 即正常的消息。

再下面有一个 aggregate 变量，这里意为当 add 或者 del 操作时，可能由于意外(不明确)导致发送未成功，再次发送的时候，是可以合并发送的。那么对于这次的 create 请求，虽然调用了 commit 函数，但是由于消息头未初始化，因此实际上此次的调用无效。

继续向下，build_msg 才真正准备开始发送消息。这里提一下，netlink 是一类方法，是一种用户和内核交互的手段，在此基础上，约定不同的协议(荷载格式等)，就有了不同的 netlink 规定，如 rtnetlink 处理路由信息，nfnetlink 处理 netfilter 相关信息等。同时，这里可以看到一个属性：nested。正常情况下，nlmsg 头后面跟的负载可能是多个，且是同一种，已经约定好的类型的载荷，但是有的情况下，一个属性又包含了其他类型的属性，即多级属性嵌套。这里就用到了 nest 属性，嵌套层数使用 nest_id 来标识。具体的使用方法，在下面再解析。

简单直白点说，之前在 4.3 节中讲过 netlink 消息的负载类型，就是一个头，带若干的扩展。但是实际开发的需求，我们想带的数据类型和数据格式，总是会比已经定下来的协议要多(或者说多得多)。这时候就需要使用拓展属性携带消息。具体的做法，其实也很简单，就是发端(user)和收端(kernel)约定一系列消息类型，然后顺便告诉对方，哪种类型的怎么处

理，就 OK 了。可以看到 IPSET 这里就是这样实现的。首先可以看到，为 ipset 定义的子属性类型:

```
/* Attributes at command level */
enum {
    IPSET_ATTR_UNSPEC,
    IPSET_ATTR_PROTOCOL,    /* 1: Protocol version */
    IPSET_ATTR_SETNAME, /* 2: Name of the set */
    IPSET_ATTR_TYPENAME,    /* 3: Typename */
    IPSET_ATTR_SETNAME2 = IPSET_ATTR_TYPENAME, /* Setname at rename/swap */
    IPSET_ATTR_REVISION,    /* 4: Settype revision */
    IPSET_ATTR_FAMILY,  /* 5: Settype family */
    IPSET_ATTR_FLAGS,   /* 6: Flags at command level */
    IPSET_ATTR_DATA,    /* 7: Nested attributes */
    IPSET_ATTR_ADT,     /* 8: Multiple data containers */
    IPSET_ATTR_LINENO,  /* 9: Restore lineno */
    IPSET_ATTR_PROTOCOL_MIN, /* 10: Minimal supported version number */
    IPSET_ATTR_REVISION_MIN = IPSET_ATTR_PROTOCOL_MIN, /* type rev min */
    __IPSET_ATTR_CMD_MAX,
};
```

图 8

再说回来，为什么要用 nest 这种方式呢？先看图 8 和下面的图 9。可以看到，前面对于 IPSET_ATTR_TYPENAME, IPSET_ATTR_REVISION 等消息类型都是在图 8 中约定好的，那么这些内容就可以直接依照类型--数据的方式填充。但是很明显，这里只规定了 9 种格式，实际使用中绝对是不够的。注意到第 7 种格式类型，叫 IPSET_ATTR_DATA.实际上，除了控制消息之外，nfnetlink 实际携带的消息都是在这个类型中填充的。但是填充的实际类型又不属于图 8 中所示的这些基本协议类型，那么就要用到 nest 属性，即允许在 IPSET_ATTR_DATA 类的消息中扩展子一级的消息类型，存放需要传送的数据。同时，也支持再往下一级的扩展，即最多 4 层的扩展，这样允许协议的子系统(扩展)实现和维护自己的属性模式，甚至是比较复杂的树形数据结构。解析的时候，接收者使用实现约定的不同的方法去接收消息的子部分并解析就可以了。这也是 netlink 高扩展性的一个体现。

下面继续看一下 create 操作填充数据的代码。

```
type = ipset_data_get(data, IPSET_OPT_TYPE);
/* Core attributes:
 * setname, typename, revision, family, flags (optional) */
ADDATTR_SETNAME(session, nlh, data);
ADDATTR(session, nlh, data, IPSET_ATTR_TYPENAME,
    NFPROTO_IPV4, cmd_attrs);
ADDATTR_RAW(session, nlh, &type->revision,
        IPSET_ATTR_REVISION, cmd_attrs);
D("family: %u, type family %u",
  ipset_data_family(data), type->family);
if (ipset_data_test(data, IPSET_OPT_FAMILY))
    ADDATTR(session, nlh, data, IPSET_ATTR_FAMILY,
        NFPROTO_IPV4, cmd_attrs);
else
    /* bitmap:port and list:set types */
    mnl_attr_put_u8(nlh, IPSET_ATTR_FAMILY, NFPROTO_UNSPEC);

/* Type-specific create attributes */
D("call open_nested");
open_nested(session, nlh, IPSET_ATTR_DATA);
addattr_create(session, nlh, data, type->family);
D("call close_nested");
close_nested(session, nlh);
break;
```

图 9

可以看到，真正填充数据的时候有一个 cmd_attrs 字段。这个字段是我们上面属性的 policy。这个 policy 就是用来给内核(或者说接收端)做指导，即告诉内核，当前来了一个非标准的属性，怎么解析呢？就按照这里给不同子属性类型定义的方法来就行了。

```
/* Attribute policies and mapping to options */
static const struct ipset_attr_policy cmd_attrs[] = {
    [IPSET_ATTR_PROTOCOL] = {
        .type = MNL_TYPE_U8,
    },
    [IPSET_ATTR_SETNAME] = {
        .type = MNL_TYPE_NUL_STRING,
        .opt  = IPSET_SETNAME,
        .len  = IPSET_MAXNAMELEN,
    },
    [IPSET_ATTR_TYPENAME] = {
        .type = MNL_TYPE_NUL_STRING,
        .opt = IPSET_OPT_TYPENAME,
        .len  = IPSET_MAXNAMELEN,
    },
```

图 10

上图，就是 IPSET 给内核定义的 ATTR 类型和解析方法的一部分。再回过头来，代码使用 open_nested 增加嵌套，再用 close_nested 关闭嵌套，这一段中间，使用 addattr_create，增加所有的属性到嵌套内。待会内核解析的时候，做一个相反的过程即可。

最后由 ipset_commit 发送消息。这里发送和接收回复都在 query 函数中，实际调用的依然是图 7 中的函数。如果配置不合法，内核会返回一个 NLMSG_ERROR 消息，并带有相应错误码：

```
static int
callback_error(const struct nlmsghdr *nlh, void *cbdata)
{
    struct ipset_session *session = cbdata;
    struct ipset_data *data = session->data;
    const struct nlmsgerr *err = mnl_nlmsg_get_payload(nlh);
    int ret = MNL_CB_ERROR;

    D(" called, cmd %s", cmd2name[session->cmd]);
    if (nlh->nlmsg_len < mnl_nlmsg_size(sizeof(struct nlmsgerr)))
        FAILURE("Broken error message received.");

    if (err->error == 0) {
```

图 11

err->error 即实际的错误码，在下面的代码中，查表 core_errcode_table 找到相应的错误，写入到 session->report 中，最后返回给界面，就可以看到相应的错误提示消息了：

```
/* Core kernel error codes */
static const struct ipset_errcode_table core_errcode_table[] = {
    /* Generic error codes */
    { ENOENT, 0,
      "The set with the given name does not exist" },
    { EMSGSIZE, 0,
      "Kernel error received: message could not be created" },
    { IPSET_ERR_PROTOCOL,  0,
      "Kernel error received: ipset protocol error" },

    /* CREATE specific error codes */
    { EEXIST, IPSET_CMD_CREATE,
      "Set cannot be created: set with the same name already exists" },
    { IPSET_ERR_FIND_TYPE, 0,
      "Kernel error received: set type not supported" },
```

图 12

在最后如果返回错误，程序会清空之前的一些数据，最后释放内存。这里错误与否，只要看 ipset_commit 的返回即可，ret == 0 的时候是成功的配置。由于 ipset 用户态程序本身并不存储数据，因此，需要再将这些配置发往 CM 或者其他模块时，内核就帮我们保证了配置是合法的。

### 6.1.3 内核对于 ipset netlink 消息的处理

现在转到内核，首先内核使用 nfnetlink_rcv_msg 接收 netlink 消息，前文提到过，内核对于不同协议族的管理方法是首先建立不同的子系统，根据收到消息的类型查询相应的子系统(ss = nfnetlink_get_subsys(type))，对于子系统，再根据类型字段，按照事先约定的操作

类型，调用不同操作注册的回调函数(nc = nfnetlink_find_client(type, ss))，进行后续处理。

首先是解析 netlink 消息，将一条消息的内容放到 cda 数组中，这样的模式，在 5.1.5 中有解释，这里不赘述了。

```
err = nla_parse(cda, ss->cb[cb_id].attr_count,
        attr, attrlen, ss->cb[cb_id].policy);
```

图 13

如下图，是 nfnetlink 调用 ipset 子系统注册的回调函数，及一些消息类型的相应回调注册。

```
else if (nc->call)
    err = nc->call(net->nfnl, skb, nlh,
                (const struct nlattr **)cda);
```

```
static const struct nfnl_callback ip_set_netlink_subsys_cb[IPSET_MSG_MAX]
    [IPSET_CMD_NONE]    = {
        .call       = ip_set_none,
        .attr_count = IPSET_ATTR_CMD_MAX,
    },
    [IPSET_CMD_CREATE]  = {
        .call       = ip_set_create,
        .attr_count = IPSET_ATTR_CMD_MAX,
        .policy     = ip_set_create_policy,
    },
    [IPSET_CMD_DESTROY] = {
        .call       = ip_set_destroy,
        .attr_count = IPSET_ATTR_CMD_MAX,
        .policy     = ip_set_setname_policy,
    },
```

图 14 15

这里调用 ip_set_create,这一层的意思是要创建一个 ipset 集。create 要做的事情也比较清晰，从 cda 数组中取出相应"set"类型名(如 hash:net)，实体化一个 set，并从内核查找相应的类型，关联到此 set 上。注意这个时候创建的是名为"myhash"的一个管理结构。

继续往下走，解析嵌套的数据，并调用此"类"的 create 函数，创建相应的记录表项。

```
if (attr[IPSET_ATTR_DATA] &&
    nla_parse_nested(tb, IPSET_ATTR_CREATE_MAX, attr[IPSET_ATTR_DATA],
             set->type->create_policy)) {
    ret = -IPSET_ERR_PROTOCOL;
    goto put_out;
}

ret = set->type->create(set, tb, flags);
```

图 16

这一层的 create，是指要创建一个 hash:net 这种类型的集。

```
static struct ip_set_type hash_net_type __read_mostly
    .name       = "hash:net",
    .protocol   = IPSET_PROTOCOL,
    .features   = IPSET_TYPE_IP | IPSET_TYPE_NOMATCH,
    .dimension  = IPSET_DIM_ONE,
    .family     = NFPROTO_UNSPEC,
    .revision_min   = REVISION_MIN,
    .revision_max   = REVISION_MAX,
    .create     = hash_net_create,
    .create_policy  = {
        [IPSET_ATTR_HASHSIZE]   = { .type = NLA_U32 },
        [IPSET_ATTR_MAXELEM]    = { .type = NLA_U32 },
        [IPSET_ATTR_PROBES] = { .type = NLA_U8 },
        [IPSET_ATTR_RESIZE] = { .type = NLA_U8  },
        [IPSET_ATTR_TIMEOUT]    = { .type = NLA_U32 },
        [IPSET_ATTR_CADT_FLAGS] = { .type = NLA_U32 },
    },
```

图 17

可以看到，实际和 6.1.2 中解释的所谓扩展规则是一样的，hash:net 也好，hash:ip 也

好，都有自己的 policy 和操作函数。这里，调用 create 函数以 hash:net 为类型创建了一个 hash 表项。到这里，就创建好了一个名为 myhash 的 hash 表项，现在需要将其放置在相应位置。内核对此的做法是对每一个 ipset 创建的 hash 表项，维护了一个唯一的 index，后面在真正使用 match 规则的时候，直接使用 index 就可以找到刚才建立的 hash 表了。

### 6.1.4 创建一个 ipset 地址集

由于 ipset 支持相当多的数据类型(net/ip/port/ipport 等)和存放类型(hash/bitmap/list)，因此在内核部分，为了减少代码冗余度，使用了大量的拼接函数，即 A##B 的形式，如上图的 hash_net_create 函数，实际是由以下方式拼接而成的：

```
static int
IPSET_TOKEN(HTYPE, _create)(struct net *net, struct ip_set *set,
                struct nlattr *tb[], u32 flags)
```

图 18

这时 HTYPE 是 hash_net，与 create 拼接即变成了 hash_net_create，后面还会有很多这样的写法。在 sourceinsight 里是无法直接找到定义的。

继续往下看，注意以下的内存申请：

```
    hsize += sizeof(struct net_prefixes) *
        (set->family == NFPROTO_IPV4 ? 32 : 128);
```

图 19

为什么 IPV4 要申请 32 个大小的 struct net_prefixes，而 IPV6 要申请 128 个呢？这是因为 ipset 特殊的 match 规则来决定的。事先为所有可能的掩码类型分配空间，到匹配时，可以按当前配置实际情况，从小网段到大网段进行匹配。后续还有一些为超时和统计申请内存及初始化的代码，在后面再详细说明。

### 6.1.5 向 ipset 中增加元素及内核 hash、匹配的说明

增加元素靠以下两个函数完成：

```
static int
hash_net4_uadt(struct ip_set *set, struct nlattr *tb[],
          enum ipset_adt adt, u32 *lineno, u32 flags, bool retried)
{
```

```
/* Add an element to a hash and update the interna
 * otherwise report the proper error code. */
static int
mtype_add(struct ip_set *set, void *value, co
      struct ip_set_ext *mext, u32 flags)
```

图 20 21

以上两个是拼接+回调函数，第一个函数完成对数据的解析，第二个函数真正将数据插入 hash。具体的 hash 插入看以下代码：

```
00616:  rcu_read_lock_bh();
00617:  t = rcu_dereference_bh(h->table);
00618:  key = HKEY(value, h->initval, t->htable_bits);
00619:  n = hbucket(t, key);
00620:  for (i = 0; i < n->pos; i++) {
00621:      data = ahash_data(n, i, h->dsize);
00622:      if (mtype_data_equal(data, d, &multi)) {
00623:          if (flag_exist ||
00624:              (SET_WITH_TIMEOUT(set) &&
00625:               ip_set_timeout_expired(ext_timeout(data, h))))
```

图 22

头两行是内核的锁机制，保证在使用当前 hash 表时，数据的安全。t 现在指向现在需要操作的 hash 数据结构。这里首先使用 HKEY 根据 value 的前 32 位(即 ip 地址)计算一个 key 值，并使用 key 值在 hash 表里寻找相应桶。n->pos 指当前桶中第一个空闲位置位于第几个元素。由于第一次创建，n->pos 为 0，不会执行 for 语句。注意，ipset 配置在同一个集里是不允许重复的，因为一条语句就代表一个 IP/IP 段，重复毫无意义。因此，这里即便是在 hash 里找到了相应 IP，如果不是使用-exist 命令强制下发，或者该条命令已经超时(参见 623 行之后的判断)，否则这里依然不会执行下去。

```
00649:          /* Use/create a new slot */
00650:          TUNE_AHASH_MAX(h, multi);
00651:          ret = hbucket_elem_add(n, AHASH_MAX(h), h->dsize);
00652:          if (ret != 0) {
00653:              if (ret == -EAGAIN)
00654:                  mtype_data_next(&h->next, d);
00655:              goto out;
00656:          }
00657:          data = ahash_data(n, n->pos++, h->dsize);
00658: #ifdef IP_SET_HASH_WITH_NETS
00659:          mtype_add_cidr(h, CIDR(d->cidr), NETS_LENGTH(set->f
00660: #endif
00661:          h->elements++;
00662:      }
00663:      memcpy(data, d, sizeof(struct mtype_elem));
```

图 23

继续向下看，由 hbucket_elem_add 函数向该 hash 桶中插入一个元素。内核的 hash 在创建时除了初始化桶外，并不管理真正待 hash 的数据。而是在 hash 后准备插入元素时，为该元素分配相应大小的数据(这里由 h->dsize 决定)。举例，若桶中原有 2 个元素，则插入函数先申请 3 个元素大小的内存，将那 2 个元素的值复制进来，释放原有 2 个元素的内存空间，并将新申请的内存挂回到 hash 表上。这样的好处就是一旦一段长 buffer 存储的数据被 hash 完成，这段 buffer 无论是被释放或者被改写都无所谓了，因为真实数据已经被安全的存放起来了。有了 pos，再使用 ahash_data 函数获取到刚才新插入的那个元素的地址，将 elem 拷进去，数据就算是插入了。

注意这里 659 行的 mtype_add_cidr 函数。这里以 IPV4 为例，6.1.4 节中提到直接为 32 个可能的掩码全部分配了空间。这是由于 ipset 比较灵活的 nomatch 机制，为了以后匹配的时候能够进行更少的比对。例如现在若有配置(假设内核也是这样的顺序)：

```
ipset add myhash 2.3.4.0/24                  1
ipset add myhash 4.3.0.0/16                  2
ipset add myhash 4.6.2.0/24                  3
ipset add myhash 1.0.0.0/8      nomatch      4
```

ipset add myhash 1.2.0.0/16 5
ipset add myhash 1.2.3.0/24 nomatch 6
ipset add myhash 1.2.3.4 7

后四条配置可以认为是允许不允许整栋楼的人上网，但是允许 6 楼例外，但是 6 楼的某个办公区域又都不能上网，只有该办公区域 6 楼的领导例外....这种需求场景是非常普遍的。

那么当前若有来自网段 1.2.3.0/24 的 IP，如果顺序匹配下去，那么前三条肯定不符合，到第四条的时候，按理是匹配了。但是如果这时候按 nomatch 处理掉，那么后面想要的场景就不符合了。 如果全都匹配下去，那么计算量实在太大，如果该 set 有 10000 条配置，如果直到匹配到第 10000 条才发现不匹配，那就太浪费时间了。

首先可以确定的原则是，越精确的 IP，优先级越高，参见上面配置的例子。那么这种情况下，为了优化匹配速率及满足上面提到的场景，ipset 使用了最长掩码匹配的算法。即一次为所有可能掩码分配空间，并按掩码长度由长到短(优先级从高到低)排序。那么对上面的配置举例来说，cidr 是这样分布的：

nets[0].cidr = 32, nets[0].nets = 1 (对应配置 7)
nets[1].cidr = 24, nets[0].nets = 3 (对应配置 1，3，6)
nets[2].cidr = 16, nets[0].nets = 2 (对应配置 2，5)
nets[3].cidr = 8, nets[0].nets = 1 (对应配置 4)

可以看到，cidr 代表当前掩码位数，nets 代表当前掩码位数下有多少条规则。如果此时再插入一条配置：

ipset add myhash 1.2.3.4/30 8

那么 cidr 分部就变为

nets[0].cidr = 32, nets[0].nets = 1 (对应配置 7)
nets[1].cidr = 30, nets[1].nets = 1 (对应配置 8)
nets[2].cidr = 24, nets[2].nets = 3 (对应配置 1，3，6)
nets[3].cidr = 16, nets[3].nets = 2 (对应配置 2，5)
nets[4].cidr = 8, nets[4].nets = 1 (对应配置 4)

这里，mtype_add_cidr 就是用来管理和排序集中所有的 ipset 配置，保证从大到小，优先级从高到低。具体代码有兴趣可以点进去看看，这里不再赘述了。具体匹配的例子在下一节讲。

### 6.1.6 set 作为 match 规则的使用

可以把 ipset 看作是一个大的 match 规则，一样有 match 函数和相应的 checkentry 函数(实际上 ipset 也可以作为 target 使用，这里限于时间和篇幅暂不解释，留至以后开发研究时补充)，可以在 xt_set.c 中看到相关的函数。

```
{
    .name       = "set",
    .family     = NFPROTO_IPV4,
    .revision   = 3,
    .match      = set_match_v3,
    .matchsize  = sizeof(struct xt_set_info_match_v3),
    .checkentry = set_match_v3_checkentry,
    .destroy    = set_match_v3_destroy,
    .me     = THIS_MODULE
},
```

图 24

这里先直接把整个函数代码贴上来：

```
const struct xt_set_info_match_v3 *info = par->matchinfo;
ADT_OPT(opt, par->family, info->match_set.dim,
    info->match_set.flags, info->flags, UINT_MAX);
int ret;

if (info->packets.op != IPSET_COUNTER_NONE ||
    info->bytes.op != IPSET_COUNTER_NONE)
    opt.cmdflags |= IPSET_FLAG_MATCH_COUNTERS;

ret = match_set(info->match_set.index, skb, par, &opt,
        info->match_set.flags & IPSET_INV_MATCH);

if (!(ret && opt.cmdflags & IPSET_FLAG_MATCH_COUNTERS))
    return ret;

if (!match_counter(opt.ext.packets, &info->packets))
    return 0;
return match_counter(opt.ext.bytes, &info->bytes);
```

图 25

就先拿这一层来说，match 函数做的事情无非就是先获取当前数据流 buffer 的一些参数和匹配参数的一些值，调用 match_set 函数匹配。匹配完成后，检查当前的配置是否有流量限制选项(按 packet 或按 byte)，如果有，再叠加流量限制匹配的结果，最后返回。整个函数如果匹配成功返回 0，匹配失败返回 1，里面的逻辑下面细讲。

match_set 函数的处理核心就是以下的函数调用：

ret = set->variant->kadt(set, skb, par, IPSET_TEST, opt);

其中 kadt 是当前集类型(hash:net)的回调函数，可以在 ip_set_hash_gen.h 中看到。

```
static const struct ip_set_type_variant mtype_variant = {
    .kadt    = mtype_kadt,
    .uadt    = mtype_uadt,
    .adt     = {
        [IPSET_ADD] = mtype_add,
        [IPSET_DEL] = mtype_del,
        [IPSET_TEST] = mtype_test,
    },
    .destroy = mtype_destroy,
    .flush   = mtype_flush,
    .head    = mtype_head,
    .list    = mtype_list,
    .resize  = mtype_resize,
    .same_set = mtype_same_set,
};
```

图 26

实际匹配使用了 mtype_test，这也是一个拼接函数。由于我们配置的有网段，因此使用以下 test_cidr 函数来进行匹配。

```
#ifdef IP_SET_HASH_WITH_NETS
    /* If we test an IP address and not a network address,
     * try all possible network sizes */
    if (CIDR(d->cidr) == SET_HOST_MASK(set->family))
        return mtype_test_cidrs(set, d, ext, mext, flags);
#endif
```

```
00756: for (; j < nets_length && h->nets[j].nets && !multi; j++) {
00757:     mtype_data_netmask(d, h->nets[j].cidr);
00758:     key = HKEY(d, h->initval, t->htable_bits);
00759:     n = hbucket(t, key);
00760:     for (i = 0; i < n->pos; i++) {
00761:         data = ahash_data(n, i, h->dsize);
00762:         if (!mtype_data_equal(data, d, &multi))
00763:             continue;
00764:         if (SET_WITH_TIMEOUT(set)) {
00765:             if (!ip_set_timeout_expired(ext_timeout(data, h)))
00766:                 return mtype_data_match(data, ext,
00767:                                         mext, set,
00768:                                         flags);
```

图 27 28

这里可以看到有两层循环，最外层是 6.1.5 中的 cidr 的遍历，内层才是真正查找 hash 和匹配的过程。那么针对于 6.1.5 节中的配置，如果当前来包 IP 为 1.2.3.4，则直接从 cidr32 开始寻找

nets[0].cidr = 32, nets[0].nets = 1　(对应配置 7)

ipset add myhash 1.2.3.4　　7

那么很明显，OK，第一次就命中了，可以返回结果了。如果再来包 1.2.3.50，那么第一次查找未能查到，开始查找下一个 cidr，即 24

nets[1].cidr = 24, nets[0].nets = 3　(对应配置 1，3，6)

那么也能直接匹配到相应语句 6，返回 nomatch 的结果。可以看出，这样的方法会避免大量不必要的匹配，直接找到优先级最高的结果并返回。接下来根据位置查找 hash 表的过程不再赘述了。最后综合 time_out 的结果，返回给整个 match 函数，这样，ipset 的 match 过程就结束了。

### 6.1.7　ipset 超时的设置原理

在 Linux 内核里，有一个名为 jiffies 的全局变量，它代表从机器启动时算起的时间滴答数。这个变量最初被初始化为 0，每次系统定时器发生时钟中断时，都会加 1。jiffies 的全局定义和相关函数在 系统中（/linux/include/linux/jiffies.h　unsigned long volatile jiffies;）提供。而系统定时器中断的频率（也可以称为节拍率，不要与 CPU 运行的 Hz 混为一谈）可以通过定义变量 HZ 的值（include/asm-x86/param.h　#define　HZ 1000）来调整,但是最好保持该 HZ 的默认值。

这样理解，jiffies 就是连续两次时钟中断的间隔时间，该值等于 1/HZ。内核就是使用这个节拍来计算系统运行时间的。即系统运行时间为 jiffies/HZ. 可以使用 get_jiffies_64 函数来读取 jiffies 的值，然后使用 jiffies_to_msecs 将其换算成毫秒或使用 jiffies_to_usecs 将其换算成微秒。ipset 就是利用这个 jiffies 来计算时间。

要使用超时属性，则首先在创建一个 ipset 集的时候对整个集设置超时。例如配置：

```
ipset create test hash:net timeout 300
ipset add test 1.2.3.4 timeout 200
ipset add test 2.3.4.5 timeout
ipset add test 3.4.5.6 timeout 0
```

表明对于 hash:net 类型的集 test，默认超时时间为 300 秒，即 2.3.4.5 这一条未说明时长的语句，其超时时间为 300 秒，而超时时间为 0，则表明该条语句永远生效。计时在配置下发后开始，超时的语句会自动被该集忽略。注意，若该集未设置超时属性，则下面的语句都不支持配置超时。从下面创建集的代码中可以看到原因。

下面针对代码来讲，ipset 针对不同的集类型（不支持超时和统计/支持超时/支持统计/既支持超时又支持统计），分别设置了不同的扩展结构。这里仅看支持超时：

```
? end if cadt_flags&IPSET_FLAG... ?  else if (tb[IPSET_ATTR_TIMEOUT]) {
  h->timeout = ip_set_timeout_uget(tb[IPSET_ATTR_TIMEOUT]);
  set->extensions |= IPSET_EXT_TIMEOUT;
  if (set->family == NFPROTO_IPV4) {
      h->dsize = sizeof(struct TOKEN(HTYPE, 4t_elem));
      h->offset[IPSET_OFFSET_TIMEOUT] =
          offsetof(struct TOKEN(HTYPE, 4t_elem),
                timeout);
      TOKEN(HTYPE, 4_gc_init)(set, TOKEN(HTYPE, 4_gc));
  } else {
```

图 29

首先在 extensions 位上增加对 timeout 的支持，同时可以看到这里使用的结构体是 hash_net4t_elem，将相应结构的大小及位置写入到 ipset 集内，并进行初始化操作。gc 是指超时后清除配置的机制，即平时条目超时后，我们只是在做 match 匹配和 list/save 操作时直接忽略该条记录，但是实际并未直接删掉。注意这里对于时长的偏移存放在 h->offset[IPSET_OFFSET_TIMEOUT]里。

下面再来看添加一条记录：

```
static int
hash_net4_uadt(struct ip_set *set, struct nlattr *tb[],
          enum ipset_adt adt, u32 *lineno, u32 flags, bool retried)
{
    const struct hash_net *h = set->data;
    ipset_adtfn adtfn = set->variant->adt[adt];
    struct hash_net4_elem e = { .cidr = HOST_MASK };
    struct ip_set_ext ext = IP_SET_INIT_UEXT(h);
```

图 30

首先取出这个集的 extension 字段，若是未配置超时，则就使用这个集默认的超时时间作为该条目的超时时间，范例配置中，此处应该是 300 秒。

```
ret = ip_set_get_hostipaddr4(tb[IPSET_ATTR_IP], &ip) ||
      ip_set_get_extensions(set, tb, &ext);
```

图 31

在 ip_set_get_extensions 里，如果当前条目配置了超时，那么就将条目的超时时间写入，不论是配置为 0 还是 200。如果未配置超时，那此处 ext 中的超时时间不改变，还是 300 秒。

```
if (SET_WITH_TIMEOUT(set) && h->elements >= h->maxelem)
    /* FIXME: when set is full, we slow down here */
    mtype_expire(h, NETS_LENGTH(set->family), h->dsize);
```

图 32

ipset 会定时清理超时的配置记录，从途中也可以看到，当集已经满了但又有配置下来的时候，也会调用 mtype_expire 函数主动清理表中超时的配置腾空间。

```
   if (SET_WITH_TIMEOUT(set))
       ip_set_timeout_set(ext_timeout(data, h), ext->timeout);
```

图 33

配置语句超时的地方可以分两部分看。首先看以下宏的定义：

```
#define ext_timeout(e, h)   \
(unsigned long *)(((void *)(e)) + (h)->offset[IPSET_OFFSET_TIMEOUT])
```

图 34

这里呼应上面创建偏移的内容，ext_timeout()函数用来根据偏移确定超时的具体时间。下面看这个时间如何计算，如果现在有配置

ipset add test 1.2.3.4 timeout 100

由于这是第一次配置，因此 ext_timeout(data, h)的结果为 0.但是注意这个函数返回的是一个地址，后面会由 ip_set_timeout_set 函数计算超时的一个滴答值并写入这个地址：

```
static inline void
ip_set_timeout_set(unsigned long *timeout, u32 t)
{
    if (!t) {
        *timeout = IPSET_ELEM_PERMANENT;
        return;
    }

    *timeout = msecs_to_jiffies(t * 1000) + jiffies;
    if (*timeout == IPSET_ELEM_PERMANENT)
        /* Bingo! :-) */
        (*timeout)--;
}
```

图 35

计算过程比较明确，就是将 t(本范例中为 10 秒)换算成 timeout(滴答值，单位 jiffies)传递出去。这里可以看到，除非是永不超值，则将超时的时间换算成毫秒，并转为 jiffies 单位，加上当前的 jiffies，即算出：本条目在系统的 jiffies 达到多少时会超时。那么经过计算后，这里再打印 ext_timeout(data, h)结果就是 100 秒了。

继续往下看，这个超时时间最直观的作用就是在 list 和 match 时作为一个衡量标准，查看当前配置是否应该生效。这里以 list 为例：

```
for (i = 0; i < n->pos; i++) {
    e = ahash_data(n, i, h->dsize);
    if (SET_WITH_TIMEOUT(set) &&
        ip_set_timeout_expired(ext_timeout(e, h)))
        continue;
```

图 36

调用了 ip_set_timeout_expired 查看当前条目是否有效，其实现原理也比较简单，就是拿当前的 jiffies 和上次存的超时时间比较即可。

再往下看 list 的函数

```
if (SET_WITH_TIMEOUT(set) &&
    nla_put_net32(skb, IPSET_ATTR_TIMEOUT,
          htonl(ip_set_timeout_get(
            ext_timeout(e, h)))))
    goto nla_put_failure;
```

图 37

ip_set_timeout_get()函数将超时时间减去现在的 jiffies 值，转换成秒。这样就得到了我们使用 ipset list 命令看到的时间了。

```
[root@localhost src]# ./ipset list
Name: test
Type: hash:net
Revision: 3
Header: family inet hashsize 1024 maxe
Size in memory: 16976
References: 0
Members:
1.2.3.8 timeout 3
[root@localhost src]#
[root@localhost src]#
[root@localhost src]# ./ipset list
Name: test
Type: hash:net
Revision: 3
Header: family inet hashsize 1024 maxe
Size in memory: 16976
References: 0
Members:
1.2.3.8 timeout 1
[root@localhost src]#
```

图 38

每次使用 list，都可以看到超时时间在变化。