# Generic Netlink介绍

## 出现原因

netlink协议的一个重要确定就是协议族（protocol family）被限制在32个。为了增加family的个数，netlink逐渐演化出基于netlink 协议，并且能进行多路通信的netlink family（NETLINK\_GENERIC）

## 1.2 优势

1、没有family的数量限制

2、不需要修改内核文件

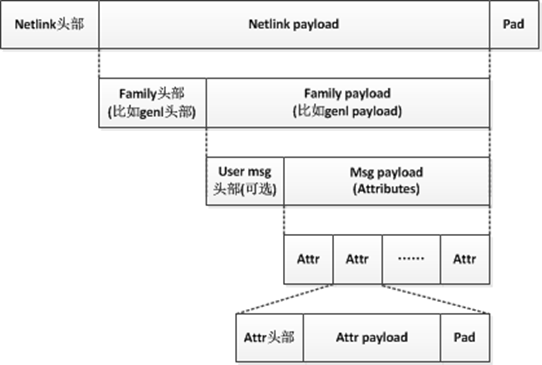
3、提供通用的内核空间与用户空间的信息交流通道（用户与用户之间也可以使用，但不常用）

**在generic netlink通信中，不需要像netlink一样，需要在include/linux/netlink.h中添加协议族（protocol family）。而只需要在特定的链表（family\_list）中加入自定义的family即可，当用户空间程序通过family.name获取family.id，然后向family.id发送netlink消息。内核空间对应id的family就可以获取该netlink消息。**

# Generic Netlink 数据结构

## 2.1 消息相关结构体

用户与内核发送的generic netlink 报文如下图所示：



### 2.1.1 netlink header

struct nlmsghdr {

\_\_u32 nlmsg\_len; /\* Length of message including header \*/

\_\_u16 nlmsg\_type; /\* Message content \*/

\_\_u16 nlmsg\_flags; /\* Additional flags \*/

\_\_u32 nlmsg\_seq; /\* Sequence number \*/

\_\_u32 nlmsg\_pid; /\* Sending process port ID \*/

};

nlmsg\_len：整个消息长度，包括netlink header

nlmsg\_type：应该为内核注册的family.id。可以通过family.name向genl控制器获取。

有以下两种方式：

1. int genl\_ctrl\_resolve(struct nl\_sock \*sk, const char \*name)（genl\_ctrl.c）
2. 自定义的函数

nlmsg\_flags：额外添加的，功能未知

nlmsg\_seq：发送序号

nlmsg\_pid：pid，可以是用户空间的进程pid，port id ，具体什么用，有待查证

### 2.1.2 generic netlink header

struct genlmsghdr {

\_\_u8 cmd;

\_\_u8 version;

\_\_u16 reserved;

};

cmd：命令，与内核注册 操作结构体genl\_ops中的cmd相同。

version：版本，一般使用0x01即可，不同版本能否通信，有待查证

reserved：保留，暂无用处

### 2.1.3 user header（可选）

以上的两个头就能完成信息发送，user header是用户定义的，相当一个属性（下一小节介绍）。完全可以不使用

### 2.1.4 属性

一个属性由header和payload组成。

header在内核使用枚举型，将所有的header列举定义。

在与generic netlink header中指定的cmd，对应的genl\_ops结构体中定义policy解析属性。

**注意：如果在policy中无法找到对应属性，依旧会触发回调函数。因为policy是属性验证策略**

属性的最大类型个数在内核family注册结构体genl\_family中设定

最好属性能与命令一一对应，则根据命令选择ops时，能够直接解析对应属性，如果包含多个属性，则需要保证属性必须包含在消息中，以免解析错误。

## 2.2 内核注册相关结构体

内核注册主要有以下几个结构体：

genl\_family：用于注册generic netlink family

genl\_ops：不同命令，选择不同的回调函数（可多个）

nla\_policy：用于解析属性的类型（与genl\_ops一一对应）

### 2.2.1 genl\_family

struct genl\_family {

unsigned int id;

unsigned int hdrsize;

char name[GENL\_NAMSIZ];

unsigned int version;

unsigned int maxattr;

bool netnsok;

bool parallel\_ops;

int (\*pre\_doit)(const struct genl\_ops \*ops,

struct sk\_buff \*skb,

struct genl\_info \*info);

void (\*post\_doit)(const struct genl\_ops \*ops,

struct sk\_buff \*skb,

struct genl\_info \*info);

struct nlattr \*\* attrbuf; /\* private \*/

const struct genl\_ops \* ops; /\* private \*/

const struct genl\_multicast\_group \*mcgrps; /\* private \*/

unsigned int n\_ops; /\* private \*/

unsigned int n\_mcgrps; /\* private \*/

unsigned int mcgrp\_offset; /\* private \*/

struct list\_head family\_list; /\* private \*/

struct module \*module;

};

id: family id。用于标识一个family，当新注册一个family的时候，应该用GENL\_ID\_GENERATE宏(0x0)，表示请控制器自动为family分配的一个id。0x00保留供genl控制器使用，其实就是GENL\_ID\_GENERATE。 （这个和我们Netlink头部中的pid不同， family id 其实就是netlink 头部的type字段）

 hdrsize: 用户自定义头部长度。即User 头部的长度。如果没有用户自定义头部，这个值被赋为0。

version: 版本号，一般填1即可。不同版本可以通信

name:  family name，要求不同的family使用不同的名字。以便控制器进行正确的查找。

maxattr：genl使用netlink标准的attr来传输数据。此字段定义了最大attr类型数。（注意：不是一次传输多少个attr，而是一共有多少种attr，因此，这个值可以被设为0，为0代表不区分所收到的数据的attr type）。在接收数据时，可以根据attr type，获得指定的attr type的数据在整体数据中的位置。

struct nlattr \*\*attrbuf

struct list\_head ops\_list

struct list\_head family\_list

以上的三个字段为私有字段，由系统自动配置，开发者不需要做配置。

其他字段后续补充

### 2.2.2 genl\_ops

struct genl\_ops {

const struct nla\_policy \*policy;

int (\*doit)(struct sk\_buff \*skb,

struct genl\_info \*info);

int (\*start)(struct netlink\_callback \*cb);

int (\*dumpit)(struct sk\_buff \*skb,

struct netlink\_callback \*cb);

int (\*done)(struct netlink\_callback \*cb);

u8 cmd;

u8 internal\_flags;

u8 flags;

};

cmd：消息cmd 与该cmd相同，调用doit

policy：属性校验策略

doit：回调函数，如果消息的cmd与属性policy符合该机构体，则调用doit函数

start、dumpit、done功能未知

 dumpit；这是一个回调函数，当genl\_ops的flag标志被添加了NLM\_F\_DUMP以后，每次收到genl消息即会回触发这个函数。dumpit与doit的区别是:dumpit的第一个参数skb不会携带从客户端发来的数据。相反地，开发者应该在skb中填入需要传给客户端的数据，然后，并skb的数据长度（可以用skb->len）return。skb中携带的数据会被自动送到客户端。只要dumpit的返回值大于0，dumpit函数就会再次被调用，并被要求在skb中填入数据。当服务端没有数据要传给客户端时，dumpit要返回0。如果函数中出错，要求返回一个负值。关于doit和dumpit的触发过程，可以查看源码中的genl\_rcv\_msg函数。

internal\_flags：family使用的标志。不需要配置

**flags：标志，（用处未知）**

### 2.2.3 nla\_policy

struct nla\_policy {

u16 type;

u16 len;

};

type字段表示attr中的数据类型，这里一定不要搞混，这里的type和attr中的type可不是一回事，attr中的type是用户自己定义的类型，只有用户自己能够理解。而这里的type是attr中的value中内容的类型。   可被配置为：

1) NLA\_UNSPEC--未定义 ；

2) NLA\_U8, NLA\_U16, NLA\_U32, NLA\_U64为8bits, 16bits, 32bits, 64bits的无符号整型 ；

3) NLA\_STRING--字符串 ；

4) NLA\_NUL\_STRING--空终止符字符串

5) NLA\_NESTED--attr流 ；

     len字段的意思是：如果在type字段配置的是字符串有关的值，要把len设置为字符串的最大长度（不包含结尾的'\0'）。如果type字段未设置或被设置为NLA\_UNSPEC，那么这里要设置为attr的payload部分的长度。

### 2.2.4 genl\_info

在genl\_ops的doit函数中，有一个参数genl\_info结构体。包含了用户层发送的消息

struct genl\_info {

u32 snd\_seq;

u32 snd\_portid;

struct nlmsghdr \* nlhdr;

struct genlmsghdr \* genlhdr;

void \* userhdr;

struct nlattr \*\* attrs;

possible\_net\_t \_net;

void \* user\_ptr[2];

struct netlink\_ext\_ack \*extack;

};

1) snd\_seq：发送序号；

2) snd\_portid：发送客户端的PID  ；

3) nlhdr：netlink header的指针；

4) genlmsghdr：genl头部的指针；

5) userhdr：用户自定义头部指针；

6) attrs：attrs，如果定义了genl\_ops->policy，这里的attrs是被policy过滤以后的结果。在完成了操作以后，如果执行正确，返回0；否则，返回一个负数。负数的返回值会触发NLMSG\_ERROR消息。当genl\_ops的flag标志被添加了NLMSG\_ERROR时，即使doit返回0，也会触发NLMSG\_ERROR消息

attrs是一个二级指针，指向的是二维数组（纵向是不同类型属性，横向是相同种类的属性）

## 2.3 用户程序注册相关结构体

### 2.3.1 sockaddr\_nl

struct sockaddr\_nl {

\_\_kernel\_sa\_family\_t nl\_family; /\* AF\_NETLINK \*/

unsigned short nl\_pad; /\* zero \*/

\_\_u32 nl\_pid; /\* port ID \*/

\_\_u32 nl\_groups; /\* multicast groups mask \*/

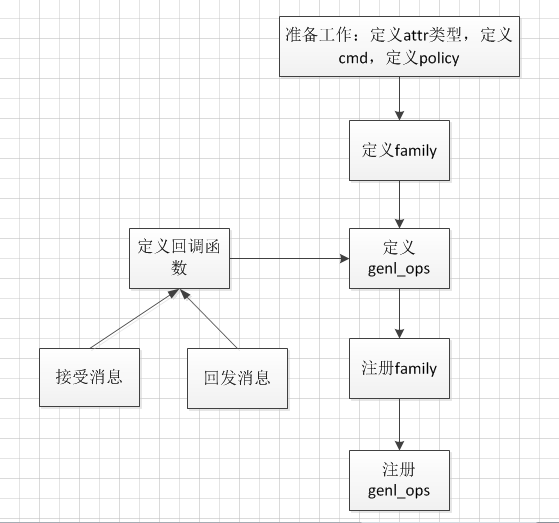
};

# 用户程序和内核模块设计流程

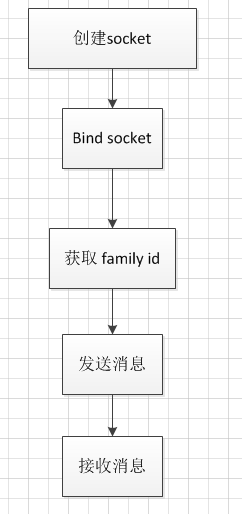
## 所有的API

http://www.infradead.org/~tgr/libnl/doc/api/index.html

## 3.1 内核模块设计流程



## 3.2 用户程序设计流程



# 四、问题

## 4.1、用户程序发送消息，内核无法触发doit函数

**现象：**

在ubuntu16.04 tls中创建demo，用户程序可以根据family name获取family id。但是发送消息却不成功。

**错误原因：**

将genl\_ops定义使用的是如下：

  struct genl\_ops exmpl\_gnl\_ops\_echo = {

        .cmd = EXMPL\_C\_ECHO,

        .flags = 0,

        .policy = exmpl\_genl\_policy,

        .doit = genl\_recv\_doit,

        .dumpit = NULL,

  };

因为exmpl\_gnl\_ops\_echo 中只有一个结构体长度，所有genl\_family中的n\_ops，应该赋值1，但是使用ARRAY\_SIZE，因为丈量时，因为exmpl\_gnl\_ops\_echo 不是数组，所以其值为0.

导致注册的family认为genl\_ops结构体为0（因为需要n\_ops值，去遍历genl\_ops数组，所以n\_ops值为零，那么就算genl\_ops数组挂在成功，也没用）。

**问题出现原因：**

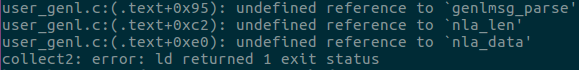
demo采用的是网上的模板，网上一般是基于linux 3.x版本设计的，但是运行环境是在ubuntu 16.04 tls，采用的是linux4.15.0-39-generic版本内核，对于绑定genl\_family和genl\_ops的函数genl\_register\_ops已经去除。而需要在genl\_family结构体定义时将genl\_ops直接赋值。

结论：

使用generic netlink虽然大致步骤从未改变，但是具体函数的使用随着linux 新版本的出现，已经发生了变化。建议根据具体的开发环境所使用的内核版本，中寻找使用generic netlink的模块作为demo。

## 4.2、collect2: error: ld returned 1 exit status

编译用户程序时，报错如下：



原因为上述函数为libnl库的函数。需要连接libnl库。

libnl库的安装方法：

libnl-3.2.25为例

1，tar zxvf libnl-3.2.25.tar.gz

2，cd libnl-3.2.25

3，./configure --prefix=/usr  --sysconfdir=/etc  --disable-static && make

4，make install  
  
5， make check.  
  
If you wish to install the API documentation, as the root user:

mkdir -vp /usr/share/doc/libnl-3.2.28 &&

tar -xf ../libnl-doc-3.2.28.tar.gz --strip-components=1 --no-same-owner \

-C /usr/share/doc/libnl-3.2.28  
  
链接libnl命令实例如下

gcc -Wall n1.c $(pkg-config --cflags --libs libnl-3.0 libnl-route-3.0)

注意：

在libnl-3.2.25中包含四个库

1. libnl：核心库，实现使用网络链接协议所需的基本原理，如套接字处理、消息构造和解析，以及数据的发送和接收。这个图书馆保持小型化和简约化。该套件的其他库依赖于此库。
2. libnl-route：NETLINK\_ROUTE系列的配置接口的API，包括网络接口、路由、地址、邻居和流量控制。
3. libnl-genl：generic netlink的API，是netlink协议的拓展版本
4. libnl-nf：与基于netfilter配置和监视接口的netlink交互的API

四个库名如下：

ibnl-3.0

libnl-route-3.0

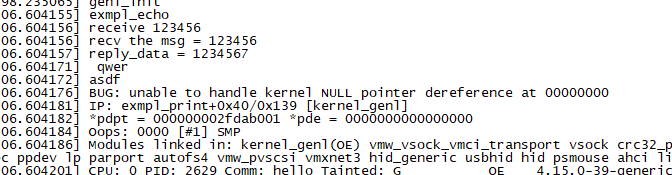
libnl-genl-3.0

libnl-nf-3.0

demo使用libnl和libnl-genl，所以编译如下

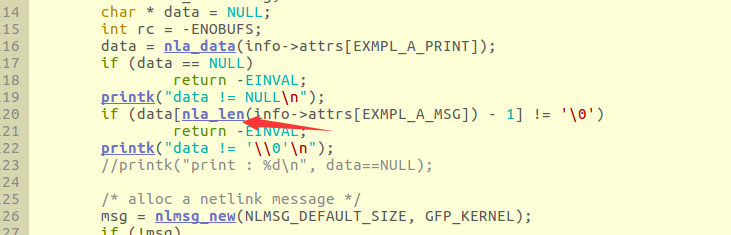
**gcc user\_genl.c -o hello $(pkg-config --cflags --libs libnl-3.0 libnl-genl-3.0)**

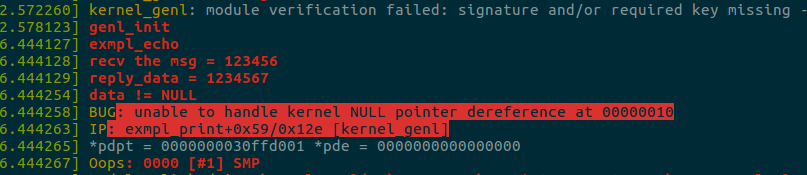
## 4.3、内核模块崩溃



用户程序发送消息后，内核模块调用函数进行解析，之后崩溃。

原因是：命令和属性不匹配，导致内核无法接受到数据，读取了空指针。应用程序发送的是EXMPL\_A\_MSG的属性，而内核需要解析的是EXMPL\_A\_PRINT的属性





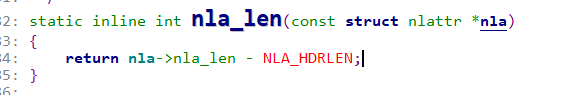
如上，应该不是读取空指针，而是段错误。

虽然print属性未发送，而发送的是msg，但是应该不是空指针，而是里面的数据为空

然后nla\_len判断的是msg的长度（必然不为0）。则data[nla\_len()-1]本身就不合法。

修改msg为print测试。

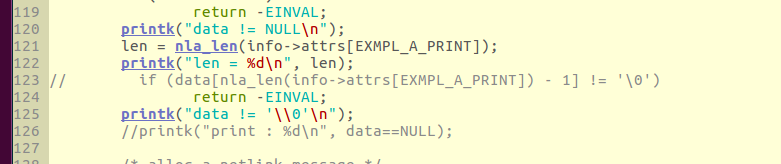
依旧如上图。



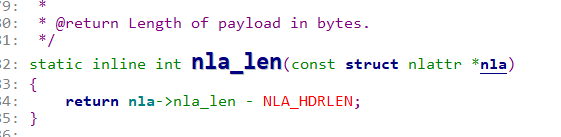
如果print为空，那么nla\_len返回的值应该为0；

导致访问data[-1]

测试

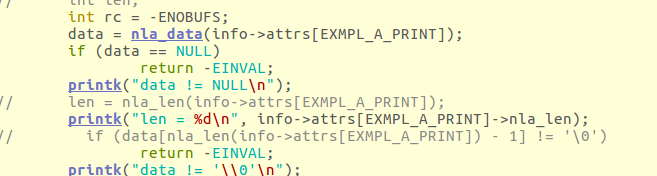


依旧错误



查看nla->nla\_len的值是多少？

因为没有参数传递，应该值为0，或者干脆没有值。

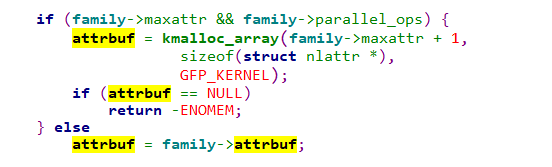


依旧是报错。

综上，struct nlattr \*\* attrs;是二级指针，每个成员指向的是单个的struct nlattr结构体。然后当接收到数据时，malloc数据，然后将对应属性类型的成员指针指向malloc数据。所以理论上，如果没有接受到数据时，该成员应该为NULL指针啊。

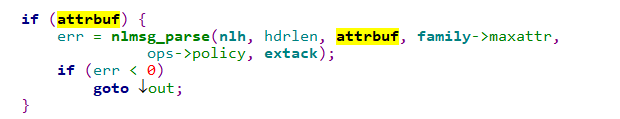
分析：

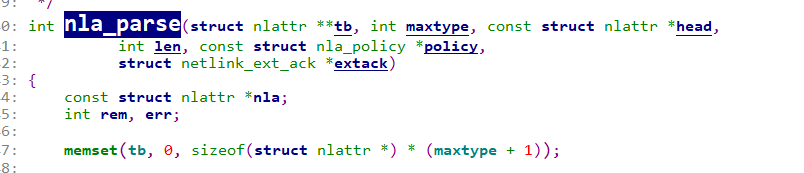
在内核的net/netlink/genetlink.c的函数genl\_family\_rcv\_msg中定义。



在此处为attr二级指针的每个指针成员分配空间。

然后在之后





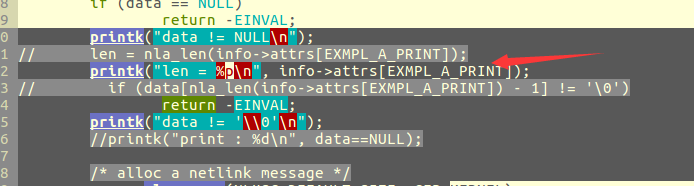
将attr二级指针所执行的空间清空，则attr二级指针指向的指针数组全部成为0，即所有指针为零指针。

1.4 零指针

零值指针，是值为0的指针，可以是任何一种指针类型，可以是通用变体类型void\*，也可以是char\*，int\*等等。

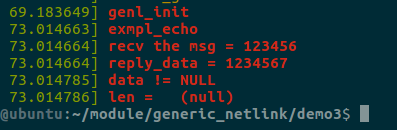
在C++里面，任何一个概念都要以一种语言内存公认的形式表现出来，例如std::vector会提供一个empty()子函数来返回容器是否为空，然而对于一个基本数值类型（或者说只是一个类似整数类型的类型）我们不可能将其抽象成一个类（当然除了auto\_ptr等只能指针）来提供其详细的状态说明，所以我们需要一个特殊值来做为这种状态的表现。

C++标准规定，当一个指针类型的数值是0时，认为这个指针是空的。（我们在其他的标准下或许可以使用其他的特殊值来定义我们需要的NULL实现，可以是1，可以是2，是随实现要求而定的，但是在标准C++下面我们用0来实现NULL指针）



打印成员的值，看是否为0；

程序正常执行。打印结果



？？？怎么办，不为NULL，也不为0？？？，如何规避

generic netlink详解

# 一、简介

**Generic Netlink 是内核专门为了扩展netlink协议簇而设计的“通用netlink协议簇”。**由于netlink协议最多支持32个协议簇，目前Linux4.1的内核中已经使用其中21个，对于用户需要定制特殊的协议类型略显不够，而且用户还需自行在include/linux/netlink.h中添加簇定义（略显不妥），为此Linux设计了这种通用Netlink协议簇，用户可在此之上定义更多类型的子协议。前两篇博文已经较为详细的分析了netlink的创建和通信流程，本文以Generic Netlink为例首先来深入分析一下netlink的消息结构及创建初始化流程。

Generic Netlink使用NETLINK\_GENERIC类型协议簇，同样基于netlink子系统。具体框架如下：

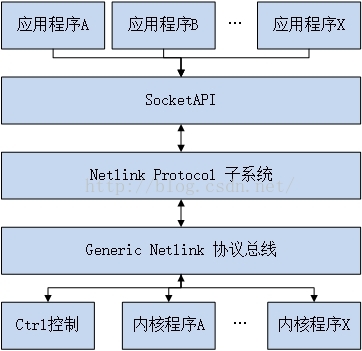


图1 Generic Netlink模型框架

其中Ctrl控制器是一种特殊类型的Genetlink协议族，它用于用户空间通过Genetlink簇名查找对应的ID号，后文中会详细分析。

Generic Netlink的消息结构基于netlink消息结构并在此基础上继续扩展，首先来看一下netlink类型的消息结构（见include/net/netlink.h）：

首先最上层，**一个netlink消息有netlink消息头和netlink消息载荷组成**，它们之间存在**内存对齐的pad**留空空间（这在《Netlink 内核实现分析（1）（2）》中已经看到了，但并未对消息载荷进行进一步分析）；然后往下一级消息的实际载荷又可分为**family头级具体的消息属性，其中family头针对不同协议种类的netlink定义各部相同；到最底层消息属性又分为消息属性头和实际的消息载荷。**Genetlink消息基于这个消息结构类型并定制化为如下结构：

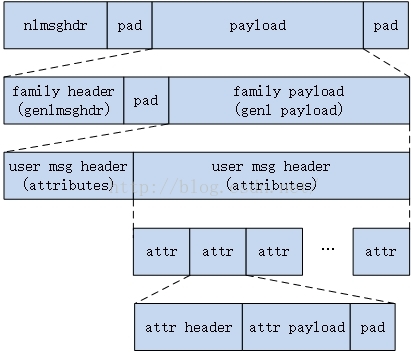


图2 Generic Netlink消息结构

**其中family头对于Genetlink来说就是Generic消息头genlmsghdr，**接下来是可选的用户特定消息头，最后才是可选的有效载荷，即一个个消息属性实例。

**Genetlink消息是命令驱动式的，即每一条消息的genlmsghdr中都指明了当前消息的cmd消息命令**，**这些消息cmd命令由用户自行定义。**内核在接收到用户的genl消息后，首先会对命令cmd做判断，找到对应的消息处理结构（可能会执行attr有效性检查），然后才会去调用消息处理回调函数从消息载荷区中读取并处理其所需要的的attr属性载荷。

# 二、Generic Netlink相关结构体

## 1、Generic Netlink消息头结构：struct genlmsghdr

struct genlmsghdr {

\_\_u8 cmd;

\_\_u8 version;

\_\_u16 reserved;

};

Generic Netlink消息头比较简单，仅包含了两个字段。其中cmd表示消息命令，对于用户自己定义的每个子协议类型都需要定义特定的消息命令集，这里该字段表示当前消息的消息命令；version字段表示版本控制（可以在在不破坏向后兼容性的情况下修改消息的格式），可以不使用该字段；最后的reserved字段保留。

## 2、Generic Netlink Family结构：struct genl\_family（内核中完成注册）

struct genl\_family {

unsigned int id;

unsigned int hdrsize;

char name[GENL\_NAMSIZ];

unsigned int version;

unsigned int maxattr;

bool netnsok;

bool parallel\_ops;

int (\*pre\_doit)(const struct genl\_ops \*ops,

struct sk\_buff \*skb,

struct genl\_info \*info);

void (\*post\_doit)(const struct genl\_ops \*ops,

struct sk\_buff \*skb,

struct genl\_info \*info);

int (\*mcast\_bind)(struct net \*net, int group);

void (\*mcast\_unbind)(struct net \*net, int group);

struct nlattr \*\* attrbuf; /\* private \*/

const struct genl\_ops \* ops; /\* private \*/

const struct genl\_multicast\_group \*mcgrps; /\* private \*/

unsigned int n\_ops; /\* private \*/

unsigned int n\_mcgrps; /\* private \*/

unsigned int mcgrp\_offset; /\* private \*/

struct list\_head family\_list; /\* private \*/

struct module \*module;

};

Generic Netlink按照family进行管理，用户需注册自己定义的genl\_family结构，同时内核使用一个**哈希表family\_ht对已经注册的genl family进行管理**。各字段的含义如下：

**id：**genl family的ID号，一般由内核进行分配，取值范围为GENL\_MIN\_ID~GENL\_MAX\_ID（16~1023），其中GENL\_ID\_CTRL为控制器的family ID，不可另行分配，该familyID全局唯一并且在family\_ht中的位置也由该值确定；

**hdrsize：**用户私有报头的长度，即可选的user msg header长度，若没有则为0；

**name：**genl family的名称，必须是独一无二且用户层已知的（用户通过它来向控制查找family id）；

**version：**版本号；

**maxattr：**消息属性attr最大的类型数（即该genl family所支持的最大attr属性类型的种类个数）；

**netnsok：**指示当前簇是否能够处理网络命名空间；

**pre\_doit：**调用genl\_ops结构中处理消息函数doit()前调用的钩子函数，一般用于执行一些前置的当前簇通用化处理，例如对临界区加锁等；

**post\_doit：**调用genl\_ops结构中处理消息函数doit()后调用的钩子函数，一般执行pre\_doit函数相反的操作；

**mcast\_bind/mcast\_unbind：**在绑定/解绑定socket到一个特定的genl netlink组播组中调用（目前内核中没有相关使用）；

**attrbuf：**保存拷贝的attr属性缓存；

**ops/n\_ops：**保存genl family命令处理结构即命令的个数，后面详细描述；

**family\_list：**链表结构，用于将当前当前簇链入全局family\_ht散列表中；

**mcgrps/n\_mcgrps：**保存当前簇使用的组播组及组播地址的个数；

## 3、Generic Netlink Family命令处理结构：struct genl\_ops（内核中完成注册）

struct genl\_ops {

const struct nla\_policy \*policy;

int (\*doit)(struct sk\_buff \*skb,

struct genl\_info \*info);

int (\*dumpit)(struct sk\_buff \*skb,

struct netlink\_callback \*cb);

int (\*done)(struct netlink\_callback \*cb);

u8 cmd;

u8 internal\_flags;

u8 flags;

};

该结构用于注册genl family的用户命令cmd处理函数（对于只向应用层发送消息的簇可以不用实现和注册该结构），各个字段的含义如下：

**cmd：**簇命令类型，由用户自行根据需要定义；

**internal\_flags：**簇私有标识，用于进行一些分支处理，可以不使用；

**flags：**操作标识，有以下四种类型（在genetlink.h中定义）：

#define GENL\_ADMIN\_PERM 0x01 /\* 当设置该标识时表示本命令操作需要具有CAP\_NET\_ADMIN权限 \*/

#define GENL\_CMD\_CAP\_DO 0x02 /\* 当genl\_ops结构中实现了doit()回调函数则设置该标识 \*/

#define GENL\_CMD\_CAP\_DUMP 0x04 /\* 当genl\_ops结构中实现了dumpit()回调函数则设置该标识 \*/

#define GENL\_CMD\_CAP\_HASPOL 0x08 /\* 当genl\_ops结构中定义了属性有效性策略（nla\_policy）则设置该标识 \*/

**policy：**属性attr有效性策略，该结构定义在《Netlink内核实现分析（一）》种已经见过了，若该字段不为空，在genl执行消息处理函数前会对消息中的attr属性进行校验，否则则不做校验；

**doit：**标准命令回调函数，在当前族中收到数据时触发调用，函数的第一个入参skb中保存了用户下发的消息内容；

**dumpit：**转储回调函数，当genl\_ops的flag标志被添加了NLM\_F\_DUMP以后会调用该回调函数，这里的第一个入参skb中不再有用户下发消息内容，而是要求函数能够在传入的skb中填入消息载荷并返回填入数据长度；

**done：**转储结束后执行的回调函数；

## 4、Generic Netlink Family内核接收消息结构：struct genl\_info

struct genl\_info {

u32 snd\_seq;

u32 snd\_portid;

struct nlmsghdr \* nlhdr;

struct genlmsghdr \* genlhdr;

void \* userhdr;

struct nlattr \*\* attrs;

possible\_net\_t \_net;

void \* user\_ptr[2];

struct sock \* dst\_sk;

};

内核在接收到用户的genetlink消息后，会对消息解析并封装成genl\_info结构，便于命令回校函数进行处理，其中各字段含义如下：

**snd\_seq：**消息的发送序号（不强制使用）；

**snd\_portid：**消息发送端socket所绑定的ID；

**nlhdr：**netlink消息头；

**genlhdr：**generic netlink消息头；

**userhdr：**用户私有报头；

**attrs：**netlink属性，包含了消息的实际载荷；

**dst\_sk：**目的socket；

# 三、net/genetlink.h（内核）

本章介绍genetlink.h头文件（基于linux 4.0.5.39-generic，不同的linux版本定义可能不同）

## 3.1、宏定义

在genetlink.h头文件

**#define GENLMSG\_DEFAULT\_SIZE (NLMSG\_DEFAULT\_SIZE - GENL\_HDRLEN)**

NLMSG\_DEFAULT\_SIZE：

（netlink.h）#define NLMSG\_DEFAULT\_SIZE (NLMSG\_GOODSIZE - NLMSG\_HDRLEN)

即默认的netlink长度减去genl的头长

**#define GENL\_SET\_ERR\_MSG(info, msg) NL\_SET\_ERR\_MSG((info)->extack, msg)**

## 3.2、结构体

### 3.2.1、struct genl\_multicast\_group

generic netlink广播组：

struct genl\_multicast\_group {

char name[GENL\_NAMSIZ]; //广播组的名称

};

同一个广播组必须有相同的family

### 3.2.2、struct genl\_info

接收信息结构体

struct genl\_info {

u32 snd\_seq; //发送序号

u32 snd\_portid; //发送者的netlink portid

struct nlmsghdr \* nlhdr; //netlink消息头

struct genlmsghdr \* genlhdr; //genl消息头

void \* userhdr; //用户头

struct nlattr \*\* attrs; //属性

possible\_net\_t \_net; //网络域名

void \* user\_ptr[2]; //用户指针

struct netlink\_ext\_ack \*extack; //拓展的ack报道结构体

};

### 3.2.3、struct genl\_family

generic netlink family结构体

struct genl\_family {

int id; // protocol family identifier (private)

unsigned int hdrsize; //用户头长度

char name[GENL\_NAMSIZ];//family 的名字

unsigned int version; //协议版本

unsigned int maxattr; //最多属性支持数

bool netnsok; //如果家族能够处理网络名称空间并且应该在所有名称空间中呈现，则设置为true ？？？

bool parallel\_ops; // 操作可以并行调用，并且不由核心genetlink代码同步

int (\*pre\_doit)(const struct genl\_ops \*ops,

struct sk\_buff \*skb,

struct genl\_info \*info); //回调之前调用的函数

void (\*post\_doit)(const struct genl\_ops \*ops,

struct sk\_buff \*skb,

struct genl\_info \*info); //回调之后调用的函数

int (\*mcast\_bind)(struct net \*net, int group);

void (\*mcast\_unbind)(struct net \*net, int group);

struct nlattr \*\* attrbuf; /\* 缓存属性 \*/

const struct genl\_ops \* ops; //该family支持的操作

const struct genl\_multicast\_group \*mcgrps; //这个family使用的广播组

unsigned int n\_ops; //这个family支持的操作的个数

unsigned int n\_mcgrps; //这个family使用的广播组的个数

unsigned int mcgrp\_offset; /\* private \*/开始广播的广播组ID

struct module \*module; //哪个模块，一般为THIS\_MODULE

};

### 3.2.4、struct genl\_ops

generic netlink的操作

struct genl\_ops {

const struct nla\_policy \*policy; //属性生效策略

int (\*doit)(struct sk\_buff \*skb, //doit回调（标准回调）

struct genl\_info \*info);

int (\*start)(struct netlink\_callback \*cb); //跳转的开始回调？

int (\*dumpit)(struct sk\_buff \*skb,

struct netlink\_callback \*cb); //跳转的回调函数？

int (\*done)(struct netlink\_callback \*cb); //dumps回调完成

u8 cmd; //命令标识符

u8 internal\_flags; //family使用的内部标识符

u8 flags; //标志

};

## 3.3 函数

### 3.3.1、genl\_info\_net

函数功能：获取接收消息的网络域名

static inline struct net \*genl\_info\_net(struct genl\_info \*info)

{

return read\_pnet(&info->\_net);

}

### 3.3.2、genl\_info\_net\_set

函数功能：设置消息的网络域名

static inline void genl\_info\_net\_set(struct genl\_info \*info, struct net \*net)

{

write\_pnet(&info->\_net, net);

}

### 3.3.3、？genl\_err\_attr

函数功能：为消息添加error属性

static inline int genl\_err\_attr(struct genl\_info \*info, int err,

struct nlattr \*attr)

{

info->extack->bad\_attr = attr;

return err;

}

### 3.3.4、genl\_register\_family

函数原型：

int genl\_register\_family(struct genl\_family \*family)

函数功能：

根据genl\_family结构体的信息配置一个generic netlink family

返回值：

成功 0；

失败 err；

一般在模块的init函数中设置。

### 3.3.5、genl\_unregister\_family

函数原型：

int genl\_unregister\_family(const struct genl\_family \*family)

函数功能：

根据family移除一个generic netlink family

返回值：

成功 0；

失败 err；

一般在模块的exit函数中设置

### 3.3.6、？genl\_notify

函数原型：

void genl\_notify(const struct genl\_family \*family, struct sk\_buff \*skb,

struct genl\_info \*info, u32 group, gfp\_t flags);

函数功能：

将消息通知给指定的group（偏移）

### 3.3.7、genl\_family\_attrbuf

struct nlattr \*\*genl\_family\_attrbuf(const struct genl\_family \*family);

获取genl\_family结构体中的缓存属性字段

### 3.3.8、genlmsg\_put

void \*genlmsg\_put(struct sk\_buff \*skb, u32 portid, u32 seq,

const struct genl\_family \*family, int flags, u8 cmd);

函数功能：

添加一个generic netlink 头到netlink消息

void \*genlmsg\_put(struct sk\_buff \*skb, u32 portid, u32 seq,

struct genl\_family \*family, int flags, u8 cmd)

{

struct nlmsghdr \*nlh;

struct genlmsghdr \*hdr;

nlh = nlmsg\_put(skb, portid, seq, family->id, GENL\_HDRLEN +

family->hdrsize, flags);//创建一个netlink message

if (nlh == NULL)

return NULL;

hdr = nlmsg\_data(nlh); //返回一个genl头

hdr->cmd = cmd;

hdr->version = family->version;

hdr->reserved = 0;

return (char \*) hdr + GENL\_HDRLEN; //返回一个user头

}

### 3.3.9、genlmsg\_nlhdr

static inline struct nlmsghdr \*genlmsg\_nlhdr(void \*user\_hdr)

{

return (struct nlmsghdr \*)((char \*)user\_hdr -

GENL\_HDRLEN -

NLMSG\_HDRLEN);

}

函数功能：

使用user头获取netlink头

### 3.3.10、genlmsg\_parse

static inline int genlmsg\_parse(const struct nlmsghdr \*nlh,

const struct genl\_family \*family,

struct nlattr \*tb[], int maxtype,

const struct nla\_policy \*policy,

struct netlink\_ext\_ack \*extack)

{

return nlmsg\_parse(nlh, family->hdrsize + GENL\_HDRLEN, tb, maxtype,

policy, extack);//调用netlink的解析方法

}

函数功能：

解析一个genetlink消息属性

参数：

\* @nlh: netlink message header

\* @family: genetlink message family

\* @tb: destination array with maxtype+1 elements

\* @maxtype: maximum attribute type to be expected

\* @policy: validation policy

\* @extack: extended ACK report struct

### 3.3.11、genl\_dump\_check\_consistent

static inline void genl\_dump\_check\_consistent(struct netlink\_callback \*cb,

void \*user\_hdr)

{

nl\_dump\_check\_consistent(cb, genlmsg\_nlhdr(user\_hdr));

}

函数功能：

检查序号是否一致，是否通告（本质上还是调用netlink的函数）

\* @cb: netlink callback structure that stores the sequence number

\* @user\_hdr: user header as returned from genlmsg\_put()

### 3.3.12、genlmsg\_put\_reply

static inline void \*genlmsg\_put\_reply(struct sk\_buff \*skb,

struct genl\_info \*info,

const struct genl\_family \*family,

int flags, u8 cmd)

{

return genlmsg\_put(skb, info->snd\_portid, info->snd\_seq, family,

flags, cmd);

}

函数功能：

添加genl头到回复消息中

\* @skb: socket buffer holding the message

\* @info: receiver info

\* @family: generic netlink family

\* @flags: netlink message flags

\* @cmd: generic netlink command

### 3.3.13、genlmsg\_end

static inline void genlmsg\_end(struct sk\_buff \*skb, void \*hdr)

{

nlmsg\_end(skb, hdr - GENL\_HDRLEN - NLMSG\_HDRLEN);

}

函数功能：

结束一个generic netlink消息

### 3.3.14、genlmsg\_cancel

static inline void genlmsg\_cancel(struct sk\_buff \*skb, void \*hdr)

{

if (hdr)

nlmsg\_cancel(skb, hdr - GENL\_HDRLEN - NLMSG\_HDRLEN);

}

关闭generic netlink消息通道

### 3.3.15、genlmsg\_multicast\_netns

static inline int genlmsg\_multicast\_netns(const struct genl\_family \*family,

struct net \*net, struct sk\_buff \*skb,

u32 portid, unsigned int group, gfp\_t flags)

{

if (WARN\_ON\_ONCE(group >= family->n\_mcgrps))

return -EINVAL;

group = family->mcgrp\_offset + group;

return nlmsg\_multicast(net->genl\_sock, skb, portid, group, flags);

}

组播消息到指定的命名空间

### 3.3.16、genlmsg\_multicast

static inline int genlmsg\_multicast(const struct genl\_family \*family,

struct sk\_buff \*skb, u32 portid,

unsigned int group, gfp\_t flags)

{

return genlmsg\_multicast\_netns(family, &init\_net, skb,

portid, group, flags);

}

组播消息到默认的命名空间

### 3.3.17、genlmsg\_multicast\_allns

int genlmsg\_multicast\_allns(const struct genl\_family \*family,

struct sk\_buff \*skb, u32 portid,

unsigned int group, gfp\_t flags);

组播消息到所有命名空间

### 3.3.18、genlmsg\_unicast

static inline int genlmsg\_unicast(struct net \*net, struct sk\_buff \*skb, u32 portid)

{

return nlmsg\_unicast(net->genl\_sock, skb, portid);

}

广播netlink消息

# 三、Generic Netlink初始化

Generic Netlink只是中特殊类型的Netlink，它本质上还是依赖于netlink的内核机制，相关的函数在genetlink.c中，由genl\_init()启动初始化流程：

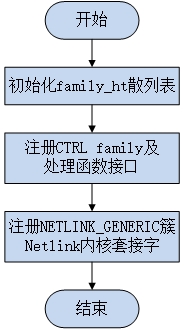


图3 Genetic Netlink初始化流程

**static int \_\_init genl\_init(void)**

**{**

**int i, err;**

**for (i = 0; i < GENL\_FAM\_TAB\_SIZE; i++)**

**INIT\_LIST\_HEAD(&family\_ht[i]);**

**err = genl\_register\_family\_with\_ops\_groups(&genl\_ctrl, genl\_ctrl\_ops,**

**genl\_ctrl\_groups);**

**if (err < 0)**

**goto problem;**

**err = register\_pernet\_subsys(&genl\_pernet\_ops);**

**if (err)**

**goto problem;**

**return 0;**

**problem:**

**panic("GENL: Cannot register controller: %d\n", err);**

**}**

首先初始化用于保存和维护Generic netlink family的散列表family\_ht数组，然后调用**genl\_register\_family\_with\_ops\_groups**向内核Generic netlink子系统注册控制器簇类型的Genetlink Family，首先来看一下genl\_ctrl的定义：

**static struct genl\_family genl\_ctrl = {**

**.id = GENL\_ID\_CTRL,**

**.name = "nlctrl",**

**.version = 0x2,**

**.maxattr = CTRL\_ATTR\_MAX,**

**.netnsok = true,**

**};**

这里的ID为GENL\_ID\_CTRL(16)，即分配区间的最小值，maxattr定义为支持的attr属性最大个数CTRL\_ATTR\_MAX，该值定义如下：

**enum {**

**CTRL\_ATTR\_UNSPEC,**

**CTRL\_ATTR\_FAMILY\_ID,**

**CTRL\_ATTR\_FAMILY\_NAME,**

**CTRL\_ATTR\_VERSION,**

**CTRL\_ATTR\_HDRSIZE,**

**CTRL\_ATTR\_MAXATTR,**

**CTRL\_ATTR\_OPS,**

**CTRL\_ATTR\_MCAST\_GROUPS,**

**\_\_CTRL\_ATTR\_MAX,**

**};**

**#define CTRL\_ATTR\_MAX(\_\_CTRL\_ATTR\_MAX - 1)**

这里为genetlink控制器定义了以CTRL\_ATTR\_UNSPEC为开头到最后的\_\_CTRL\_ATTR\_MAX中的一共7个attr属性类型，后文再进行分析；继续回到genl\_ctrl中，最后netnsok字段为true表示支持net命名空间。

再来看一下genl\_ctrl\_ops的定义：

**static struct genl\_ops genl\_ctrl\_ops[] = {**

**{**

**.cmd = CTRL\_CMD\_GETFAMILY,**

**.doit = ctrl\_getfamily,**

**.dumpit = ctrl\_dumpfamily,**

**.policy = ctrl\_policy,**

**},**

**};**

这里为控制器类型的genetlink family只定义了一种cmd类型的内核操作接口，即CTRL\_CMD\_GETFAMILY，它用于应用空间从内核中获取指定family名称的ID号。因为该ID号在内核注册family时由内核进行分配，应用空间一般只知道需要通信的family name，但是要发起通信就必须知道该ID号，所以内核设计了控制器类型的family并定义了CTRL\_CMD\_GETFAMILY命令的处理接口用于应用程序查找ID号。

然后指明doit和dumpit回调函数为ctrl\_getfamily和ctrl\_dumpfamily，最后指定attr有效性策略为ctrl\_policy：

**static const struct nla\_policy ctrl\_policy[CTRL\_ATTR\_MAX+1] = {**

**[CTRL\_ATTR\_FAMILY\_ID] = { .type = NLA\_U16 },**

**[CTRL\_ATTR\_FAMILY\_NAME] = { .type = NLA\_NUL\_STRING,**

**.len = GENL\_NAMSIZ - 1 },**

**};**

这里为CTRL\_ATTR\_FAMILY\_ID属性限定类型为16位无符号数，为CTRL\_ATTR\_FAMILY\_NAME属性限定为空结尾的字符串类型并限定了长度。

最后来看一下注册的组播组：

**static struct genl\_multicast\_group genl\_ctrl\_groups[] = {**

**{ .name = "notify", },**

**};**

这里添加了name为”notify“的组播组。然后进入genl\_register\_family\_with\_ops\_groups内部来分析一下内核是如何注册这个family簇的：

**#define genl\_register\_family\_with\_ops\_groups(family, ops, grps) \**

**\_genl\_register\_family\_with\_ops\_grps((family), \**

**(ops), ARRAY\_SIZE(ops), \**

**(grps), ARRAY\_SIZE(grps))**

**static inline int**

**\_genl\_register\_family\_with\_ops\_grps(struct genl\_family \*family,**

**const struct genl\_ops \*ops, size\_t n\_ops,**

**const struct genl\_multicast\_group \*mcgrps,**

**size\_t n\_mcgrps)**

**{**

**family->module = THIS\_MODULE;**

**family->ops = ops;**

**family->n\_ops = n\_ops;**

**family->mcgrps = mcgrps;**

**family->n\_mcgrps = n\_mcgrps;**

**return \_\_genl\_register\_family(family);**

**}**

这里根据入参初始化了family的ops等字段，然后调用\_\_genl\_register\_family()继续进行注册

int \_\_genl\_register\_family(struct genl\_family \*family)

{

int err = -EINVAL, i;

if (family->id && family->id < GENL\_MIN\_ID)

goto errout;

if (family->id > GENL\_MAX\_ID)

goto errout;

首先对入参的ID号进行判断，一般来说，为了保证ID号的全局唯一性，程序中一般都设置为GENL\_ID\_GENERATE，由内核统一分配（当然这里注册控制器family除外了）。

err = genl\_validate\_ops(family);

if (err)

return err;

genl\_lock\_all();

if (genl\_family\_find\_byname(family->name)) {

err = -EEXIST;

goto errout\_locked;

}

接下来调用genl\_validate\_ops对ops函数集做校验，对于每一个注册的genl\_ops结构，其中doit和dumpit回调函数必须至少实现一个，然后其针对的cmd命令不可以出现重复，否则返回错误，注册失败。然后上锁开始启动链表操作，首先需要确保的是family name的全局唯一性，因此这里会查找是否有同名的簇已经注册了，若有就不能再注册了。

if (family->id == GENL\_ID\_GENERATE) {

u16 newid = genl\_generate\_id();

if (!newid) {

err = -ENOMEM;

goto errout\_locked;

}

family->id = newid;

} else if (genl\_family\_find\_byid(family->id)) {

err = -EEXIST;

goto errout\_locked;

}

然后判断传入的ID号是否为GENL\_ID\_GENERATE，若是则由内核分配一个空闲的ID号，否则得确保程序指定的ID号没有被使用过。

if (family->maxattr && !family->parallel\_ops) {

family->attrbuf = kmalloc((family->maxattr+1) \*

sizeof(struct nlattr \*), GFP\_KERNEL);

if (family->attrbuf == NULL) {

err = -ENOMEM;

goto errout\_locked;

}

} else

family->attrbuf = NULL;

接着根据注册的最大attr参数maxattr，这里对于genl\_ctrl来说一共分配了CTRL\_ATTR\_MAX+1个指针内存空间，以后用于缓存attr属性（注意仅仅是保存属性的地址而非内容）。

err = genl\_validate\_assign\_mc\_groups(family);

if (err)

goto errout\_locked;

然后调用genl\_validate\_assign\_mc\_groups()函数判断新增组播地址空间，该函数一共做了3件事：（1）判断注册family的group组播名的有效性；（2）为该family分配组播地址比特位并将bit偏移保存到family->mcgrp\_offset变量中（由于generic netlink中不同类型的family簇共用NETLINK\_GENERIC协议类型的group组播地址空间，因此内核特别维护了几个全局变量mc\_groups\_longs、mc\_groups和mc\_group\_start用以维护组播地址的比特位，另外对于几种特殊的family是已经分配了的。无需再行分配，例如这里的crtl控制器）；（3）更新全局nl\_table对应的NETLINK\_GENERIC协议类型netlink的groups标识。

继续回到中\_\_genl\_register\_family()函数中：

list\_add\_tail(&family->family\_list, genl\_family\_chain(family->id));

genl\_unlock\_all();

/\* send all events \*/

genl\_ctrl\_event(CTRL\_CMD\_NEWFAMILY, family, NULL, 0);

for (i = 0; i < family->n\_mcgrps; i++)

genl\_ctrl\_event(CTRL\_CMD\_NEWMCAST\_GRP, family,

&family->mcgrps[i], family->mcgrp\_offset + i);

return 0;

errout\_locked:

genl\_unlock\_all();

errout:

return err;

}

EXPORT\_SYMBOL(\_\_genl\_register\_family);

接下来将family注册到链表中，最后调用genl\_ctrl\_event()函数向内核的控制器family发送CTRL\_CMD\_NEWFAMILY和CTRL\_CMD\_NEWMCAST\_GRP命令消息，当然这里本身就是在创建ctrl控制器family，所以该函数不会做任何的事情，对于注册其他通用family的情况后续在分析，这样ctrl familu就成功创建完成了。

最后再回到genl\_init()中，接下来使用了register\_pernet\_subsys方法为当前系统中的网络命名空间创建Generic Netlink套接字：

static int \_\_net\_init genl\_pernet\_init(struct net \*net)

{

struct netlink\_kernel\_cfg cfg = {

.input = genl\_rcv,

.flags = NL\_CFG\_F\_NONROOT\_RECV,

.bind = genl\_bind,

.unbind = genl\_unbind,

};

/\* we'll bump the group number right afterwards \*/

net->genl\_sock = netlink\_kernel\_create(net, NETLINK\_GENERIC, &cfg);

if (!net->genl\_sock && net\_eq(net, &init\_net))

panic("GENL: Cannot initialize generic netlink\n");

if (!net->genl\_sock)

return -ENOMEM;

return 0;

}

这里定义了genetlink内核套接字的配置，并指定了消息处理函数为genl\_rcv()，套接字绑定和解绑定函数为genl\_bind()和genl\_unbind()（这点需要注意，和NETLINK\_ROUTE不同），随后调用netlink\_kernel\_create()函数完成内核套接字的注册（netlink\_kernel\_create函数在前篇博文中已经详细分析过了），并将生成的套接字赋值到网络命名空间net的genl\_sock中，以后就可以通过net->genl\_sock来找到genetlink内核套接字了。

# 四、demo

示例程序：demo\_genetlink\_kern.c（内核模块）、demo\_genetlink\_user.c（应用层Demo程序）、demo\_genetlink.h

程序主要功能：应用层程序接收用户的输入“字符串”和“数据”向内核发送，内核接收后回发应用层，应用层通过终端打印输出。

## 4.1、创建内核Demo Genetlink

### 1、定义Demo Genetlink

**enum {**

**DEMO\_CMD\_UNSPEC = 0, /\* Reserved \*/**

**DEMO\_CMD\_ECHO, /\* user->kernel request/get-response \*/**

**DEMO\_CMD\_REPLY, /\* kernel->user event \*/**

**\_\_DEMO\_CMD\_MAX,**

**};**

**#define DEMO\_CMD\_MAX (\_\_DEMO\_CMD\_MAX - 1)**

**enum {**

**DEMO\_CMD\_ATTR\_UNSPEC = 0,**

**DEMO\_CMD\_ATTR\_MESG, /\* demo message \*/**

**DEMO\_CMD\_ATTR\_DATA, /\* demo data \*/**

**\_\_DEMO\_CMD\_ATTR\_MAX,**

**};**

**#define DEMO\_CMD\_ATTR\_MAX (\_\_DEMO\_CMD\_ATTR\_MAX - 1)**

定义两种类型的Genetlink cmd指令，其中DEMO\_CMD\_ECHO用于应用层下发数据，DEMO\_CMD\_REPLY用于内核向应用层回发数据；同时定义两种类型的attr属性参数，其中DEMO\_CMD\_ATTR\_MESG表示字符串，DEMO\_CMD\_ATTR\_DATA表示数据。

**static struct genl\_family demo\_family = {**

**.id = GENL\_ID\_GENERATE,**

**.name = DEMO\_GENL\_NAME,**

**.version = DEMO\_GENL\_VERSION,**

**.maxattr = DEMO\_CMD\_ATTR\_MAX,**

**};**

定义demo\_family，其中ID号为GENL\_ID\_GENERATE，表示由内核统一分配，maxattr为DEMO\_CMD\_ATTR\_MAX，其前文中定义的最大attr属性数，内核将为其分配缓存空间。

**static const struct genl\_ops demo\_ops[] = {**

**{**

**.cmd = DEMO\_CMD\_ECHO,**

**.doit = demo\_echo\_cmd,**

**.policy = demo\_cmd\_policy,**

**.flags = GENL\_ADMIN\_PERM,**

**},**

**};**

定义操作函数集operations：demo\_ops，这里只为DEMO\_CMD\_ECHO类型的cmd创建消息处理函数接口（因为DEMO\_CMD\_REPLY类型的cmd用于内核消息，应用层不使用），指定doit消息处理回调函数为demo\_echo\_cmd，同时指定有效组策略为demo\_cmd\_policy。

**static const struct nla\_policy demo\_cmd\_policy[DEMO\_CMD\_ATTR\_MAX+1] = {**

**[DEMO\_CMD\_ATTR\_MESG] = { .type = NLA\_STRING },**

**[DEMO\_CMD\_ATTR\_DATA] = { .type = NLA\_S32 },**

**};**

这里限定DEMO\_CMD\_ATTR\_MESG的属性类型为NLA\_STRING（字符串类型），限定DEMO\_CMD\_ATTR\_DATA的属性类型为NLA\_S32（有符号32位数）。

### 2、内核注册Demo Genetlink

注册：

**static int \_\_init demo\_genetlink\_init(void)**

**{**

**int ret;**

**pr\_info("demo generic netlink module %d init...\n", DEMO\_GENL\_VERSION);**

**ret = genl\_register\_family\_with\_ops(&demo\_family, demo\_ops);**

**if (ret != 0) {**

**pr\_info("failed to init demo generic netlink example module\n");**

**return ret;**

**}**

**pr\_info("demo generic netlink module init success\n");**

**return 0;**

**}**

在模块的初始化函数中，**调用genl\_register\_family\_with\_ops()同时注册demo\_family及demo\_ops**，该函数同前面创建CTRL类型的family簇类似，最终都是调用\_genl\_register\_family\_with\_ops\_grps函数完成创建。这个函数已经大致分析过了，此处的注册流程基本一致，主要区别在于最后的send all events，它向所有的应用层加入CTRL控制器簇组播组的Generic Netlink套接字多播发送CTRL\_CMD\_NEWFAMILY消息，通知应用层有新的family注册了，这样应用层就可以捕获这一消息。详细分析一下：

**static int genl\_ctrl\_event(int event, struct genl\_family \*family,**

**const struct genl\_multicast\_group \*grp,**

**int grp\_id)**

**{**

**struct sk\_buff \*msg;**

**/\* genl is still initialising \*/**

**if (!init\_net.genl\_sock)**

**return 0;**

**switch (event) {**

**case CTRL\_CMD\_NEWFAMILY:**

**case CTRL\_CMD\_DELFAMILY:**

**WARN\_ON(grp);**

**msg = ctrl\_build\_family\_msg(family, 0, 0, event);**

**break;**

**case CTRL\_CMD\_NEWMCAST\_GRP:**

**case CTRL\_CMD\_DELMCAST\_GRP:**

**BUG\_ON(!grp);**

**msg = ctrl\_build\_mcgrp\_msg(family, grp, grp\_id, 0, 0, event);**

**break;**

**default:**

**return -EINVAL;**

**}**

函数首先判断是否已经注册了控制器CTRL簇，这里显然已经注册过了，然后主要的工作在ctrl\_build\_family\_msg()中：

**static struct sk\_buff \*ctrl\_build\_family\_msg(struct genl\_family \*family,**

**u32 portid, int seq, u8 cmd)**

**{**

**struct sk\_buff \*skb;**

**int err;**

**skb = nlmsg\_new(NLMSG\_DEFAULT\_SIZE, GFP\_KERNEL);**

**if (skb == NULL)**

**return ERR\_PTR(-ENOBUFS);**

**err = ctrl\_fill\_info(family, portid, seq, 0, skb, cmd);**

**if (err < 0) {**

**nlmsg\_free(skb);**

**return ERR\_PTR(err);**

**}**

**return skb;**

**}**

首先调用nlmsg\_new()函数创建netlink类型的skb，第一个入参是消息的长度，第二个参数为内存空间分配类型，这里分配的数据空间(包括netlink消息头)一共为一个page。进入nlmsg\_new内部：

static inline struct sk\_buff \*nlmsg\_new(size\_t payload, gfp\_t flags)

{

return alloc\_skb(nlmsg\_total\_size(payload), flags);

}

static inline int nlmsg\_total\_size(int payload)

{

return NLMSG\_ALIGN(nlmsg\_msg\_size(payload));

}

static inline int nlmsg\_msg\_size(int payload)

{

return NLMSG\_HDRLEN + payload;

}

可以看到总共预留的空间为NLMSG\_ALIGN（NLMSG\_HDRLEN+NLMSG\_DEFAULT\_SIZE），这里实际可能用不了这么多的空间，接下来调用ctrl\_fill\_info()填充消息内容：

**static int ctrl\_fill\_info(struct genl\_family \*family, u32 portid, u32 seq,**

**u32 flags, struct sk\_buff \*skb, u8 cmd)**

**{**

**void \*hdr;**

**hdr = genlmsg\_put(skb, portid, seq, &genl\_ctrl, flags, cmd);**

**if (hdr == NULL)**

**return -1;**

**return hdr;**

**}**

这个函数比较长，这里使用插图的形式来观察消息的封装流程（图中未显示空白Pad区）：

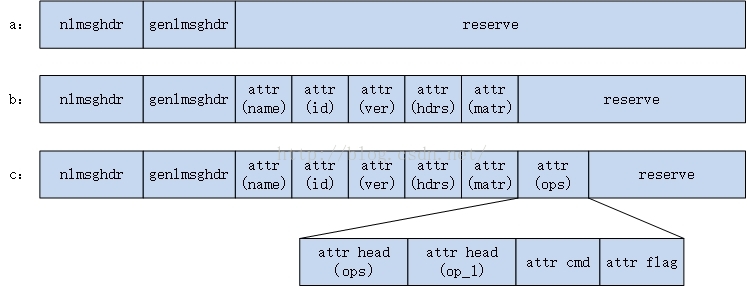


图1 消息的封装流程（a）（b）（c）

首先确定各个入参的内容：family为新注册的demo\_family；portid和seq为0，表示消息的发送端为内核，发送消息序号为0；最后的cmd为CTRL\_CMD\_NEWFAMILY。函数首先调用genlmsg\_put()函数初始化netlink消息头和genetlink消息头；

void \*genlmsg\_put(struct sk\_buff \*skb, u32 portid, u32 seq,

struct genl\_family \*family, int flags, u8 cmd)

{

struct nlmsghdr \*nlh;

struct genlmsghdr \*hdr;

nlh = nlmsg\_put(skb, portid, seq, family->id, GENL\_HDRLEN +

family->hdrsize, flags);

if (nlh == NULL)

return NULL;

return nlh；

}

其中nlmsg\_put()函数向skb缓冲区中获取消息头空间并且初始化netlink消息头，入参中的第5个参数为genetlink消息头和用户私有消息头（这里并未使用）的总空间，实际调用的函数为\_\_nlmsg\_put()：

struct nlmsghdr \*

\_\_nlmsg\_put(struct sk\_buff \*skb, u32 portid, u32 seq, int type, int len, int flags)

{

struct nlmsghdr \*nlh;

int size = nlmsg\_msg\_size(len);

nlh = (struct nlmsghdr \*)skb\_put(skb, NLMSG\_ALIGN(size));

nlh->nlmsg\_type = type;

nlh->nlmsg\_len = size;

nlh->nlmsg\_flags = flags;

nlh->nlmsg\_pid = portid;

nlh->nlmsg\_seq = seq;

if (!\_\_builtin\_constant\_p(size) || NLMSG\_ALIGN(size) - size != 0)

memset(nlmsg\_data(nlh) + len, 0, NLMSG\_ALIGN(size) - size);

return nlh;

}

首先这里的分配的空间大小为size = 传入的len长度 + netlink消息头的长度，然后初始化netlink消息头的各个字段：

nlh->nlmsg\_type ：内核genl\_ctrl family簇的ID 号GENL\_ID\_CTRL；

nlh->nlmsg\_len ：消息长度，即genetlink头+用户私有头+netlink头的长度总和；

nlh->nlmsg\_flags：0；

nlh->nlmsg\_pid：发送端的ID号为0，表示又内核发送；

nlh->nlmsg\_seq：0；

初始化完成后将内存对齐用的空白区刷为0；然后回到genlmsg\_put()函数中继续分析：

hdr = nlmsg\_data(nlh);

hdr->cmd = cmd;

hdr->version = family->version;

hdr->reserved = 0;

return (char \*) hdr + GENL\_HDRLEN;

}

这里通过宏nlmsg\_data获取genetlink消息头的地址，然后开始填充该消息头的各个字段：

hdr->cmd：消息的cmd命令，CTRL\_CMD\_NEWFAMILY；

hdr->version：genl\_ctrl family簇的version；

hdr->reserved：0；

填充完毕后返回消息用户私有头（若有）或实际载荷的首地址，此时的消息skb中的消息填充如图1-a所示。然后再回到ctrl\_fill\_info()函数中，接下来就要开始填充实际的数据了：

if (nla\_put\_string(skb, CTRL\_ATTR\_FAMILY\_NAME, family->name) ||

nla\_put\_u16(skb, CTRL\_ATTR\_FAMILY\_ID, family->id) ||

nla\_put\_u32(skb, CTRL\_ATTR\_VERSION, family->version) ||

nla\_put\_u32(skb, CTRL\_ATTR\_HDRSIZE, family->hdrsize) ||

nla\_put\_u32(skb, CTRL\_ATTR\_MAXATTR, family->maxattr))

goto nla\_put\_failure;

这里将新注册的family结构中的几个字段都填充到了消息中，包括name、id号、版本号、私有头长度以及maxattr（注意属性需要一一对应），调用的函数nla\_put\_string、nla\_put\_u16和nla\_put\_u32都是nla\_put()的封装，而nla\_put实际调用的是\_\_nla\_put()：

int nla\_put(struct sk\_buff \*skb, int attrtype, int attrlen, const void \*data)

{

if (unlikely(skb\_tailroom(skb) < nla\_total\_size(attrlen)))

return -EMSGSIZE;

\_\_nla\_put(skb, attrtype, attrlen, data);

return 0;

}

void \_\_nla\_put(struct sk\_buff \*skb, int attrtype, int attrlen,

const void \*data)

{

struct nlattr \*nla;

nla = \_\_nla\_reserve(skb, attrtype, attrlen);

memcpy(nla\_data(nla), data, attrlen);

}

\_\_nla\_put()的作用是向skb中添加一个netlink attr属性，入参分别为skb地址、要添加的attr属性类型、属性长度和属性实际数据。首先调用了\_\_nla\_reserve在skb中预留出attr属性的内存空间：

struct nlattr \*\_\_nla\_reserve(struct sk\_buff \*skb, int attrtype, int attrlen)

{

struct nlattr \*nla;

nla = (struct nlattr \*) skb\_put(skb, nla\_total\_size(attrlen));

nla->nla\_type = attrtype;

nla->nla\_len = nla\_attr\_size(attrlen);

memset((unsigned char \*) nla + nla->nla\_len, 0, nla\_padlen(attrlen));

return nla;

}

这里首先预留空间长度为nla\_total\_size(attrlen)，即attrlen+NLA\_HDRLEN（属性头长度）+对齐用内存空白；然后初始化属性头的两个字段：

nla->nla\_type：attr属性，即前文中的CTRL\_ATTR\_FAMILY\_NAME等；

nla->nla\_len：attr属性长度（attrlen+NLA\_HDRLEN）；

然后再将attr属性中的实际数据拷贝到预留测空间中，如此一个attr属性就添加完成了，此时的消息skb中的消息填充如图1-b所示。再回到ctrl\_fill\_info()函数中：

if (family->n\_ops) {

struct nlattr \*nla\_ops;

int i;

nla\_ops = nla\_nest\_start(skb, CTRL\_ATTR\_OPS);

if (nla\_ops == NULL)

goto nla\_put\_failure;

for (i = 0; i < family->n\_ops; i++) {

struct nlattr \*nest;

const struct genl\_ops \*ops = &family->ops[i];

u32 op\_flags = ops->flags;

if (ops->dumpit)

op\_flags |= GENL\_CMD\_CAP\_DUMP;

if (ops->doit)

op\_flags |= GENL\_CMD\_CAP\_DO;

if (ops->policy)

op\_flags |= GENL\_CMD\_CAP\_HASPOL;

nest = nla\_nest\_start(skb, i + 1);

if (nest == NULL)

goto nla\_put\_failure;

if (nla\_put\_u32(skb, CTRL\_ATTR\_OP\_ID, ops->cmd) ||

nla\_put\_u32(skb, CTRL\_ATTR\_OP\_FLAGS, op\_flags))

goto nla\_put\_failure;

nla\_nest\_end(skb, nest);

}

nla\_nest\_end(skb, nla\_ops);

}

然后如果新注册的family簇也同时注册了操作接口operations，这里会追加上对应的attr属性参数；但同前面不同的是，这里追加的attr参数是“打包”在一起的，使用的属性为CTRL\_ATTR\_OPS。由于netlink的attr属性是支持多级嵌套的，所以这里的“打包”指的就是新建一级嵌套，首先使用nla\_nest\_start()函数来创建新的一级嵌套：

static inline struct nlattr \*nla\_nest\_start(struct sk\_buff \*skb, int attrtype)

{

struct nlattr \*start = (struct nlattr \*)skb\_tail\_pointer(skb);

if (nla\_put(skb, attrtype, 0, NULL) < 0)

return NULL;

return start;

}

可以看到这里调用的依然是nla\_put()函数，不过这里的入参中指定的attr长度为0，然后数据为NULL，那这里其实就是向skb中添加了一段attr属性头，然后指定它的属性nla\_type为CTRL\_ATTR\_OPS，属性nla\_len为0，注意函数返回的是添加嵌套attr头之前的消息有效数据末尾地址。

然后回到ctrl\_fill\_info()函数中继续往下是一个for循环，在每个循环中向刚才新创建的一级嵌套attr属性中添加属性。它首先会根据operations中实现的回调函数封装flag，然后依旧是调用nla\_nest\_start()函数再次创建新的一级嵌套，不过这次的attrtype为函数的序列，随后在消息中追加上回调函数处理的cmd以及flag，最后调用nla\_nest\_end()函数结束这一层attr嵌套：

static inline int nla\_nest\_end(struct sk\_buff \*skb, struct nlattr \*start)

{

start->nla\_len = skb\_tail\_pointer(skb) - (unsigned char \*)start;

return skb->len;

}

这个函数其实只做了一件事，那就是更新这个嵌套的attr属性头的nla\_len字段为本嵌套属性的实际长度，实现的方式为当前的消息末尾地址减去创建该级嵌套之前的消息末尾地址（这就是nla\_nest\_start()函数要返回start地址的原因了）。回到ctrl\_fill\_info()函数中，在for循环结束以后，依旧调用nla\_nest\_end来结束CTRL\_ATTR\_OPS的那一层attr嵌套，此时的消息skb中的消息填充如图1-c所示。

ctrl\_fill\_info()函数接下来会再判断family->n\_mcgrps字段，若存在组播组，会同operations一样增加一级和operations平级的attr嵌套然后添加CTRL\_ATTR\_MCAST\_GROUPS属性，这里就不详细分析了。在函数的最后调用nla\_nest\_end()完成本次消息封装：

static inline void genlmsg\_end(struct sk\_buff \*skb, void \*hdr)

{

nlmsg\_end(skb, hdr - GENL\_HDRLEN - NLMSG\_HDRLEN);

}

该函数间接调用nlmsg\_end()函数，注意第二个入参为消息attr载荷的首地址减去2个头的长度，即netlink消息头的首地址。

static inline void nlmsg\_end(struct sk\_buff \*skb, struct nlmsghdr \*nlh)

{

nlh->nlmsg\_len = skb\_tail\_pointer(skb) - (unsigned char \*)nlh;

}

这里填充nlh->nlmsg\_len为整个消息的长度(包括attr载荷部分和所有的消息头部分)。到这里，向CTRL控制器簇发送的消息就已经封装完成了，再回到最上层的genl\_ctrl\_event()中：

if (IS\_ERR(msg))

return PTR\_ERR(msg);

if (!family->netnsok) {

genlmsg\_multicast\_netns(&genl\_ctrl, &init\_net, msg, 0,

0, GFP\_KERNEL);

} else {

rcu\_read\_lock();

genlmsg\_multicast\_allns(&genl\_ctrl, msg, 0,

0, GFP\_ATOMIC);

rcu\_read\_unlock();

}

这里根据是否支持net命名空间来选择发送的流程，genlmsg\_multicast\_allns函数从命名中就可以看出会像所有命名空间的控制器簇发送消息，而genlmsg\_multicast\_netns则指定了向init\_net发送，不论哪一种情况，最后都是调用nlmsg\_multicast()函数。不过这里有一点需要注意的就是这里的第三个入参portid为0，这是为了防止向发送端发送报文，这也就表明内核控制器簇套接字是不会接受该广播报文的（内核也不应该接收，否则会panic，可参见netlink\_data\_ready()函数的实现）。

至此Demo Genetlink的内核创建流程就全部结束了，此时应用层可以通过Ctrl簇获取它的ID号并向他发送消息了。下面来分析应用层是如何初始化genetlink套接字的。

## 4.2、二、应用层初始化Genetlink套接字

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

{

......

/\* 初始化socket \*/

nl\_fd = demo\_create\_nl\_socket(NETLINK\_GENERIC);

if (nl\_fd < 0) {

fprintf(stderr, "failed to create netlink socket\n");

return 0;

}

......

}

static int demo\_create\_nl\_socket(int protocol)

{

int fd;

struct sockaddr\_nl local;

/\* 创建socket \*/

fd = socket(AF\_NETLINK, SOCK\_RAW, protocol);

if (fd < 0)

return -1;

memset(&local, 0, sizeof(local));

local.nl\_family = AF\_NETLINK;

local.nl\_pid = getpid();

/\* 使用本进程的pid进行绑定 \*/

if (bind(fd, (struct sockaddr \*) &local, sizeof(local)) < 0)

goto error;

return fd;

error:

close(fd);

return -1;

}

应用层依然通过socket系统调用创建AF\_NETLINK地址簇的SOCK\_RAW套接字，指定协议类型为NETLINK\_GENERIC，创建的流程同前博文《Netlink 内核实现分析（一）：创建》中分析的NETLINK\_ROUTE类似，这里不再赘述。

接下来初始化sockaddr\_nl地址结构并进行绑定操作，为了简单起见，这里使用进程ID进行绑定（该值全局唯一），在实际的程序中可自行安排。绑定的流程前博文也已经分析过了，会调用到netlink回调函数netlink\_bind()，该函数会将绑定的ID号添加到全局nl\_table中。这里有一点需要说明的就是在前一篇博文中已经看到内核genetlink套接字已经指定了bind回调函数为genl\_bind，这里如果指定了多播组地址nl\_groups，会调用到该回调函数进行多播组的绑定操作。简单看一下：

if (nlk->netlink\_bind && groups) {

int group;

for (group = 0; group < nlk->ngroups; group++) {

if (!test\_bit(group, &groups))

continue;

err = nlk->netlink\_bind(net, group + 1);

if (!err)

continue;

netlink\_undo\_bind(group, groups, sk);

return err;

}

}

这里在genetlink支持的最大组播数中进行轮询，检测用户需要绑定的多播组并将其转换为位序号，然后调用netlink\_bind回调函数，这里该函数就是genl\_bind()：

static int genl\_bind(struct net \*net, int group)

{

int i, err = -ENOENT;

down\_read(&cb\_lock);

for (i = 0; i < GENL\_FAM\_TAB\_SIZE; i++) {

struct genl\_family \*f;

list\_for\_each\_entry(f, genl\_family\_chain(i), family\_list) {

if (group >= f->mcgrp\_offset &&

group < f->mcgrp\_offset + f->n\_mcgrps) {

int fam\_grp = group - f->mcgrp\_offset;

if (!f->netnsok && net != &init\_net)

err = -ENOENT;

else if (f->mcast\_bind)

err = f->mcast\_bind(net, fam\_grp);

else

err = 0;

break;

}

}

}

up\_read(&cb\_lock);

return err;

}

由于不同family类型的genetlink都共用同一个组播地址空间，所以这里根据用户输入的组播号来查找对应的family，然后会调用该family对应的mcast\_bind()回调函数，它需要根据family的需求自行实现，可用于做进一步的特殊需求处理，不实现亦可（目前内核中注册的genl\_family均未使用到该接口）。

至此，应用层genetlink套接字初始化完成，下面来分析它是如何发送消息到前文中注册的内核demo genelink套接字的。

## 4.3、用户空间和内核空间通信

用户空间想要发送消息到内核的demo genelink套接字，它首先得知道内核分配的demo family的family id号，因为genelink子系统是根据该id号来区分不同family簇的genelink套接字和分发消息的。此时前文中的ctrl就用于该目的，它可以将family name转换为对应的family id，用户空间也通过family name向ctrl簇查询对应的family id。在应用层序获取了family id后它就可以像内核发送消息，该消息分别包含了字符串和数据，同时内核也在接受后进行回发操作。

另外，在一般的程序中，如果应用层无需向内核发送消息，仅仅需要接收内核发送的消息时，它并不需要通过Ctrl簇获取family id了，仅需要接收内核的genetlink消息并做好cmd和attr类型判断并做出相应的处理即可。

### 1、用户查询Demo Family ID

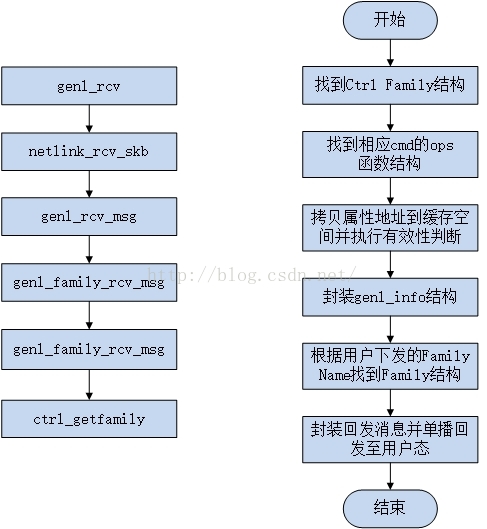


图2 查询Family ID的调用流程

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

{

......

/\* 获取family id \*/

nl\_family\_id = demo\_get\_family\_id(nl\_fd);

if (!nl\_family\_id) {

fprintf(stderr, "Error getting family id, errno %d\n", errno);

goto out;

}

PRINTF("family id %d\n", nl\_family\_id);

......

}

static int demo\_get\_family\_id(int sd)

{

struct msgtemplate ans;

char name[100];

int id = 0, ret;

struct nlattr \*na;

int rep\_len;

/\* 根据gen family name查询family id \*/

strcpy(name, DEMO\_GENL\_NAME);

ret = demo\_send\_cmd(sd, GENL\_ID\_CTRL, getpid(), CTRL\_CMD\_GETFAMILY,

CTRL\_ATTR\_FAMILY\_NAME, (void \*)name, strlen(DEMO\_GENL\_NAME)+1);

if (ret < 0)

return 0;

/\* 接收内核消息 \*/

rep\_len = recv(sd, &ans, sizeof(ans), 0);

if (ans.n.nlmsg\_type == NLMSG\_ERROR || (rep\_len < 0) || !NLMSG\_OK((&ans.n), rep\_len))

return 0;

/\* 解析family id \*/

na = (struct nlattr \*) GENLMSG\_DATA(&ans);

na = (struct nlattr \*) ((char \*) na + NLA\_ALIGN(na->nla\_len));

if (na->nla\_type == CTRL\_ATTR\_FAMILY\_ID) {

id = \*(\_\_u16 \*) NLA\_DATA(na);

}

return id;

}

该demo\_get\_family\_id()函数比较简单，仅仅是封装查询消息并向内核的ctrl簇发送，然后接收内核的回发结果然后解析出其中的family id，具体的消息发送函数由demo\_send\_cmd()封装函数来完成，其中入参分别是socket fd、ctrl family id、消息发送端netlink绑定ID号、消息cmd类型、消息attr属性、消息正文内容、消息正文长度。

static int demo\_send\_cmd(int sd, \_\_u16 nlmsg\_type, \_\_u32 nlmsg\_pid,

\_\_u8 genl\_cmd, \_\_u16 nla\_type,

void \*nla\_data, int nla\_len)

{

struct nlattr \*na;

struct sockaddr\_nl nladdr;

int r, buflen;

char \*buf;

struct msgtemplate msg;

/\* 填充msg (本函数发送的msg只填充一个attr) \*/

msg.n.nlmsg\_len = NLMSG\_LENGTH(GENL\_HDRLEN);

msg.n.nlmsg\_type = nlmsg\_type;

msg.n.nlmsg\_flags = NLM\_F\_REQUEST;

msg.n.nlmsg\_seq = 0;

msg.n.nlmsg\_pid = nlmsg\_pid;

msg.g.cmd = genl\_cmd;

msg.g.version = DEMO\_GENL\_VERSION;

na = (struct nlattr \*) GENLMSG\_DATA(&msg);

na->nla\_type = nla\_type;

na->nla\_len = nla\_len + 1 + NLA\_HDRLEN;

memcpy(NLA\_DATA(na), nla\_data, nla\_len);

msg.n.nlmsg\_len += NLMSG\_ALIGN(na->nla\_len);

buf = (char \*) &msg;

buflen = msg.n.nlmsg\_len;

memset(&nladdr, 0, sizeof(nladdr));

nladdr.nl\_family = AF\_NETLINK;

/\* 循环发送直到发送完成 \*/

while ((r = sendto(sd, buf, buflen, 0, (struct sockaddr \*) &nladdr,

sizeof(nladdr))) < buflen) {

if (r > 0) {

buf += r;

buflen -= r;

} else if (errno != EAGAIN)

return -1;

}

return 0;

}

消息的封装过程同内核态消息封装过程类似，需严格按照genelink消息格式进行封装。首先填充netlink消息头，其中nlmsg\_type字段不使用netlink定义的标准type，填充为目标family的ID号，其他字段同其他类型的netlink类似；然后填充genetlink消息头，这里设定消息cmd字段为CTRL\_CMD\_GETFAMILY，version字段为DEMO\_GENL\_VERSION（同内核保持一致）；最后填充一个attr属性，其中属性头的nla\_type设定为函数传入的属性type，现该值为CTRL\_ATTR\_FAMILY\_NAME，然后将传入的family name拷贝到属性attr的payload载荷中，最后更新各个消息头中的长度字段。

消息分装完成后调用sendto系统调用启动发送流程，指定目的地址的地址簇为AF\_NETLINK，ID号为0（表示内核）。sendto函数同前博文《Netlink 内核实现分析（二）：通信》中分析的sendmsg()系统调用类似（sendto的msg消息封装过程由内核完成），最后都是调用到sock\_sendmsg()函数，具体的中间发送流程前博文中已详细描述，这里不再赘述，直接进入到发送的最后阶段，来看Ctrl簇是如何处理接收到的查询消息的。

在netlink函数调用流程的最后会调用具体协议类型的netlink\_rcv()回调函数，其中genetlink的回调函数在前文中已经看到为genl\_rcv()：

static void genl\_rcv(struct sk\_buff \*skb)

{

down\_read(&cb\_lock);

netlink\_rcv\_skb(skb, &genl\_rcv\_msg);

up\_read(&cb\_lock);

}

这里netlink\_rcv\_skb函数的两个入参其中第一个为消息skb，第二个为genl\_rcv\_msg回调函数；netlink\_rcv\_skb()函数会对消息进行一些通用性的处理，将用户消息封装成genl\_info结构，最后会把消息控制权交给genl\_rcv\_msg()回调函数：

int netlink\_rcv\_skb(struct sk\_buff \*skb, int (\*cb)(struct sk\_buff \*,

struct nlmsghdr \*))

{

struct nlmsghdr \*nlh;

int err;

while (skb->len >= nlmsg\_total\_size(0)) {

int msglen;

nlh = nlmsg\_hdr(skb);

err = 0;

if (nlh->nlmsg\_len < NLMSG\_HDRLEN || skb->len < nlh->nlmsg\_len)

return 0;

/\* Only requests are handled by the kernel \*/

if (!(nlh->nlmsg\_flags & NLM\_F\_REQUEST))

goto ack;

/\* Skip control messages \*/

if (nlh->nlmsg\_type < NLMSG\_MIN\_TYPE)

goto ack;

err = cb(skb, nlh);

if (err == -EINTR)

goto skip;

ack:

if (nlh->nlmsg\_flags & NLM\_F\_ACK || err)

netlink\_ack(skb, nlh, err);

skip:

msglen = NLMSG\_ALIGN(nlh->nlmsg\_len);

if (msglen > skb->len)

msglen = skb->len;

skb\_pull(skb, msglen);

}

return 0;

}

首先判断消息的长度是否不小于netlink消息头的长度（现在的上下文中显然成立），然后进入while循环开始处理存放在skb中的netlink消息（可能有多个）。循环处理中会首先进行一些基本的数据长度判断，然后根据nlmsg\_flags和nlmsg\_type字段判断是否跳过消息处理流程、以及是否回发ACK相应。目前由于设定的nlmsg\_flags为NLM\_F\_REQUEST、nlmsg\_type为GENL\_ID\_CTRL（即NLMSG\_MIN\_TYPE），因此调用genl\_rcv\_msg()回调函数开始消息处理流程：

static int genl\_rcv\_msg(struct sk\_buff \*skb, struct nlmsghdr \*nlh)

{

struct genl\_family \*family;

int err;

family = genl\_family\_find\_byid(nlh->nlmsg\_type);

if (family == NULL)

return -ENOENT;

if (!family->parallel\_ops)

genl\_lock();

err = genl\_family\_rcv\_msg(family, skb, nlh);

if (!family->parallel\_ops)

genl\_unlock();

return err;

}

该函数首先通过nlmsg\_type字段（即family id号）在散列表中查找到对应的注册family，然后如果消息处理不可重入，则这里会上锁，接下来调用genl\_family\_rcv\_msg()函数：

static int genl\_family\_rcv\_msg(struct genl\_family \*family,

struct sk\_buff \*skb,

struct nlmsghdr \*nlh)

{

const struct genl\_ops \*ops;

struct net \*net = sock\_net(skb->sk);

struct genl\_info info;

struct genlmsghdr \*hdr = nlmsg\_data(nlh);

struct nlattr \*\*attrbuf;

int hdrlen, err;

/\* this family doesn't exist in this netns \*/

if (!family->netnsok && !net\_eq(net, &init\_net))

return -ENOENT;

hdrlen = GENL\_HDRLEN + family->hdrsize;

if (nlh->nlmsg\_len < nlmsg\_msg\_size(hdrlen))

return -EINVAL;

函数首先判断网络命名空间，若不支持则当前消息的网络空间必须为init\_net，然后判断消息的长度。

ops = genl\_get\_cmd(hdr->cmd, family);

if (ops == NULL)

return -EOPNOTSUPP;

static const struct genl\_ops \*genl\_get\_cmd(u8 cmd, struct genl\_family \*family)

{

int i;

for (i = 0; i < family->n\_ops; i++)

if (family->ops[i].cmd == cmd)

return &family->ops[i];

return NULL;

}

这里找到消息cmd命令对应的处理函数并保存早ops变量中，查找的方式是通过cmd字段的匹配类型来找的，这里找到的就是前文中注册的demo\_ops结构了。

if ((ops->flags & GENL\_ADMIN\_PERM) &&

!netlink\_capable(skb, CAP\_NET\_ADMIN))

return -EPERM;

接下来判断权限，这里由于已经在demo\_ops中设置了GENL\_ADMIN\_PERM标识，因此本命令操作需要具有CAP\_NET\_ADMIN权限。

if ((nlh->nlmsg\_flags & NLM\_F\_DUMP) == NLM\_F\_DUMP) {

int rc;

if (ops->dumpit == NULL)

return -EOPNOTSUPP;

if (!family->parallel\_ops) {

struct netlink\_dump\_control c = {

.module = family->module,

/\* we have const, but the netlink API doesn't \*/

.data = (void \*)ops,

.dump = genl\_lock\_dumpit,

.done = genl\_lock\_done,

};

genl\_unlock();

rc = \_\_netlink\_dump\_start(net->genl\_sock, skb, nlh, &c);

genl\_lock();

} else {

struct netlink\_dump\_control c = {

.module = family->module,

.dump = ops->dumpit,

.done = ops->done,

};

rc = \_\_netlink\_dump\_start(net->genl\_sock, skb, nlh, &c);

}

return rc;

}

如果用户设定了NLM\_F\_DUMP标识，这里就会调用启动dump流程，回填skb消息（这里的skb将不再是用户下发的消息了）。这里不进行详细的分析，继续往下看：

if (family->maxattr && family->parallel\_ops) {

attrbuf = kmalloc((family->maxattr+1) \*

sizeof(struct nlattr \*), GFP\_KERNEL);

if (attrbuf == NULL)

return -ENOMEM;

} else

attrbuf = family->attrbuf;

这里为attr属性指定接收缓存，在支持重入的情况下这里会另行动态分配内存，否则使用在注册family的\_\_genl\_register\_family函数中分配的内存空间。需要注意的是这里的内存其实只是一个指针数组，用来存放attr属性的地址，并不会存放实际的属性数据。

if (attrbuf) {

err = nlmsg\_parse(nlh, hdrlen, attrbuf, family->maxattr,

ops->policy);

if (err < 0)

goto out;

}

这里将消息的数据拷贝到缓存空间中去，nlmsg\_parse()的几个入参分别为netlink消息头，genelink消息头长度（其实也包括了用户私有头，只不过这里为0罢了），数据属性缓存地址，缓存空间大小和属性有效性策略结构。

static inline int nlmsg\_parse(const struct nlmsghdr \*nlh, int hdrlen,

struct nlattr \*tb[], int maxtype,

const struct nla\_policy \*policy)

{

if (nlh->nlmsg\_len < nlmsg\_msg\_size(hdrlen))

return -EINVAL;

return nla\_parse(tb, maxtype, nlmsg\_attrdata(nlh, hdrlen),

nlmsg\_attrlen(nlh, hdrlen), policy);

}

该函数间接调用netlink通用的属性拷贝函数，其中将第三个参数为attr参数的首地址：nlmsg\_attrdata(nlh, hdrlen)：

static inline struct nlattr \*nlmsg\_attrdata(const struct nlmsghdr \*nlh,

int hdrlen)

{

unsigned char \*data = nlmsg\_data(nlh);

return (struct nlattr \*) (data + NLMSG\_ALIGN(hdrlen));

}

这里将指针跳过netlink的头以及genelink头，指向attr的首地址。

第四个参数为attr属性的长度：nlmsg\_attrlen(nlh, hdrlen)：

static inline int nlmsg\_attrlen(const struct nlmsghdr \*nlh, int hdrlen)

{

return nlmsg\_len(nlh) - NLMSG\_ALIGN(hdrlen);

}

计算方式为消息除去netlink消息头的剩余长度减去genetlink消息头长度后的长度。

int nla\_parse(struct nlattr \*\*tb, int maxtype, const struct nlattr \*head,

int len, const struct nla\_policy \*policy)

{

const struct nlattr \*nla;

int rem, err;

memset(tb, 0, sizeof(struct nlattr \*) \* (maxtype + 1));

nla\_for\_each\_attr(nla, head, len, rem) {

u16 type = nla\_type(nla);

if (type > 0 && type <= maxtype) {

if (policy) {

err = validate\_nla(nla, maxtype, policy);

if (err < 0)

goto errout;

}

tb[type] = (struct nlattr \*)nla;

}

}

if (unlikely(rem > 0))

pr\_warn\_ratelimited("netlink: %d bytes leftover after parsing attributes in process `%s'.\n",

rem, current->comm);

err = 0;

errout:

return err;

}

可以看到该函数会逐一的将属性的地址复制到tb指针数组中去，但是如果传入了有效性策略，那他就会调用validate\_nla函数执行有效性判断。对于这里传入的CTRL\_ATTR\_FAMILY\_NAME属性来说，在ctrl\_policy中已经定义了有效性限制为NLA\_NUL\_STRING，最大长度为GENL\_NAMSIZ-1：

static const struct nla\_policy ctrl\_policy[CTRL\_ATTR\_MAX+1] = {

[CTRL\_ATTR\_FAMILY\_ID] = { .type = NLA\_U16 },

[CTRL\_ATTR\_FAMILY\_NAME] = { .type = NLA\_NUL\_STRING,

.len = GENL\_NAMSIZ - 1 },

};

回到genl\_family\_rcv\_msg()函数中继续往下分析：

info.snd\_seq = nlh->nlmsg\_seq;

info.snd\_portid = NETLINK\_CB(skb).portid;

info.nlhdr = nlh;

info.genlhdr = nlmsg\_data(nlh);

info.userhdr = nlmsg\_data(nlh) + GENL\_HDRLEN;

info.attrs = attrbuf;

info.dst\_sk = skb->sk;

genl\_info\_net\_set(&info, net);

memset(&info.user\_ptr, 0, sizeof(info.user\_ptr));

这里开始封装genl\_info消息结构，填充对应的字段，比较好理解，其中snd\_portid填充为发送端的套接字ID号，attrs为前文中分配的attr缓存空间首地址，接下来启动最终的调用处理流程：

if (family->pre\_doit) {

err = family->pre\_doit(ops, skb, &info);

if (err)

goto out;

}

err = ops->doit(skb, &info);

if (family->post\_doit)

family->post\_doit(ops, skb, &info);

如果在注册family时指定了pre\_doit和post\_doit回调函数，将在分别调用ops->doit()函数的前后调用他们，对于Ctrl簇而言并没有定义，这里会直接调用ops->doit()回调函数，对于CTRL\_CMD\_GETFAMILY来说就是ctrl\_getfamily()了：

static int ctrl\_getfamily(struct sk\_buff \*skb, struct genl\_info \*info)

{

struct sk\_buff \*msg;

struct genl\_family \*res = NULL;

int err = -EINVAL;

if (info->attrs[CTRL\_ATTR\_FAMILY\_ID]) {

u16 id = nla\_get\_u16(info->attrs[CTRL\_ATTR\_FAMILY\_ID]);

res = genl\_family\_find\_byid(id);

err = -ENOENT;

}

首先函数匹配CTRL\_ATTR\_FAMILY\_ID，由于并未传入该属性数据，因此这里该属性的地址为NULL，然后接着判断另一个属性类型：

if (info->attrs[CTRL\_ATTR\_FAMILY\_NAME]) {

char \*name;

name = nla\_data(info->attrs[CTRL\_ATTR\_FAMILY\_NAME]);

res = genl\_family\_find\_byname(name);

err = -ENOENT;

}

这里就开始处理CTRL\_CMD\_GETFAMILY属性了，只做了一件事，就是通过用户传入的family name获取到对应的family结构。

if (res == NULL)

return err;

if (!res->netnsok && !net\_eq(genl\_info\_net(info), &init\_net)) {

/\* family doesn't exist here \*/

return -ENOENT;

}

msg = ctrl\_build\_family\_msg(res, info->snd\_portid, info->snd\_seq,

CTRL\_CMD\_NEWFAMILY);

if (IS\_ERR(msg))

return PTR\_ERR(msg);

return genlmsg\_reply(msg, info);

这里依然使用ctrl\_build\_family\_msg()函数封装回发消息（该函数的分析见前文），注意回发消息的cmd为CTRL\_CMD\_NEWFAMILY（它会将查询结果family的全部内容回传），指定的port\_id号为消息查询端的id（并不是内核的id号0），消息的sequence也同查询消息一致。

函数最后调用genlmsg\_reply()向应用层回发消息：

static inline int genlmsg\_reply(struct sk\_buff \*skb, struct genl\_info \*info)

{

return genlmsg\_unicast(genl\_info\_net(info), skb, info->snd\_portid);

}

可以看到它就是nlmsg\_unicast的一个封装而已（nlmsg\_unicast的实现分析见《Netlink 内核实现分析（二）：通信》）。至此查询消息的发送和内核的处理流程分析完毕，下面回到示例程序demo\_get\_family\_id()中：

/\* 接收内核消息 \*/

rep\_len = recv(sd, &ans, sizeof(ans), 0);

if (ans.n.nlmsg\_type == NLMSG\_ERROR || (rep\_len < 0) || !NLMSG\_OK((&ans.n), rep\_len))

return 0;

/\* 解析family id \*/

na = (struct nlattr \*) GENLMSG\_DATA(&ans);

na = (struct nlattr \*) ((char \*) na + NLA\_ALIGN(na->nla\_len));

if (na->nla\_type == CTRL\_ATTR\_FAMILY\_ID) {

id = \*(\_\_u16 \*) NLA\_DATA(na);

}

这里找到回发消息中的第二个attr（消息结构参见图1-c），然后获取出其中的family id号。至此用户程序成功获取的了demo family的id号，接下来就可以向他发送消息了。

### 2、向内核Demo Family发送消息

/\* 发送字符串消息 \*/

my\_pid = getpid();

string = argv[1];

data = atoi(argv[2]);

ret = demo\_send\_cmd(nl\_fd, nl\_family\_id, my\_pid, DEMO\_CMD\_ECHO,

DEMO\_CMD\_ATTR\_MESG, string, strlen(string) + 1);

if (ret < 0) {

fprintf(stderr, "failed to send echo cmd\n");

goto out;

}

/\* 发送数据消息 \*/

ret = demo\_send\_cmd(nl\_fd, nl\_family\_id, my\_pid, DEMO\_CMD\_ECHO,

DEMO\_CMD\_ATTR\_DATA, &data, sizeof(data));

if (ret < 0) {

fprintf(stderr, "failed to send echo cmd\n");

goto out;

}

本示例程序比较简单，直接使用程序的入参作为发送的数据。发送依然是调用demo\_send\_cmd函数实现，但是入参同获取family id时的有所不同，首先发送字符串消息时第二个入参设置为刚刚获取的demo family id，然后发送端套接字ID为当前进程的pid号，然后发送cmd为DEMO\_CMD\_ECHO，发送的属性依次为DEMO\_CMD\_ATTR\_DATA（其实cmd和attr属性并没有明确的一一对应关系，用户可根据需求自行组合，同时一个cmd消息也可以带很多的attr属性，这点从内核ctrl回发的消息就可以看出），最后发送的消息内容分别为用户输入的字符串和数据。

### 3、内核Demo Family 回发消息

应用层序向demo family发送DEMO\_CMD\_ECHO消息后，内核会调用到前文中注册时指定的doit回调函数demo\_echo\_cmd（具体的数据发送流程同前文中分析的Ctrl查询消息，不再详细分析），来看一下demo\_echo\_cmd()函数所做的处理。

static int demo\_echo\_cmd(struct sk\_buff \*skb, struct genl\_info \*info)

{

if (info->attrs[DEMO\_CMD\_ATTR\_MESG])

return cmd\_attr\_echo\_message(info);

else if (info->attrs[DEMO\_CMD\_ATTR\_DATA])

return cmd\_attr\_echo\_data(info);

else

return -EINVAL;

}

该函数会判断接收的属性类型，并做出相应的处理（注意：为了简单起见，该doit回调函数最多一次只能处理一种类型的attr属性），先来看cmd\_attr\_echo\_message()函数

static int cmd\_attr\_echo\_message(struct genl\_info \*info)

{

struct nlattr \*na;

char \*msg;

struct sk\_buff \*rep\_skb;

size\_t size;

int ret;

/\* 读取用户下发的消息 \*/

na = info->attrs[DEMO\_CMD\_ATTR\_MESG];

if (!na)

return -EINVAL;

msg = (char \*)nla\_data(na);

pr\_info("demo generic netlink receive echo mesg %s\n", msg);

/\* 回发消息 \*/

size = nla\_total\_size(strlen(msg)+1);

/\* 准备构建消息 \*/

ret = demo\_prepare\_reply(info, DEMO\_CMD\_REPLY, &rep\_skb, size);

if (ret < 0)

return ret;

/\* 填充消息 \*/

ret = demo\_mk\_reply(rep\_skb, DEMO\_CMD\_ATTR\_MESG, msg, size);

if (ret < 0)

goto err;

/\* 完成构建并发送 \*/

return demo\_send\_reply(rep\_skb, info);

err:

nlmsg\_free(rep\_skb);

return ret;

}

这里从DEMO\_CMD\_ATTR\_MESG属性地址处取出用户下发的消息内容，然后调用demo\_prepare\_reply构建回发消息头。其中入参依次为接收genl\_info消息，回发cmd类型，skb指针地址，回发数据长度。

static int demo\_prepare\_reply(struct genl\_info \*info, u8 cmd, struct sk\_buff \*\*skbp, size\_t size)

{

struct sk\_buff \*skb;

void \*reply;

/\*

\* If new attributes are added, please revisit this allocation

\*/

skb = genlmsg\_new(size, GFP\_KERNEL);

if (!skb)

return -ENOMEM;

if (!info)

return -EINVAL;

/\* 构建回发消息头 \*/

reply = genlmsg\_put\_reply(skb, info, &demo\_family, 0, cmd);

if (reply == NULL) {

nlmsg\_free(skb);

return -EINVAL;

}

\*skbp = skb;

return 0;

}

这里依然调用genlmsg\_new()函数申请skb套接字缓存空间，然后直接调用genlmsg\_put\_reply()函数构建回发消息的netlink消息头和genetlink消息头：

static inline void \*genlmsg\_put\_reply(struct sk\_buff \*skb,

struct genl\_info \*info,

struct genl\_family \*family,

int flags, u8 cmd)

{

return genlmsg\_put(skb, info->snd\_portid, info->snd\_seq, family,

flags, cmd);

}

该函数仅仅是genlmsg\_put的一个封装而已，注意入参info->snd\_portid为用户层的netlink套接字的id号。回到cmd\_attr\_echo\_message()函数中，接下来填充消息属性：

/\* 填充消息 \*/

ret = demo\_mk\_reply(rep\_skb, DEMO\_CMD\_ATTR\_MESG, msg, size);

if (ret < 0)

goto err;

static int demo\_mk\_reply(struct sk\_buff \*skb, int aggr, void \*data, int len)

{

/\* add a netlink attribute to a socket buffer \*/

return nla\_put(skb, aggr, len, data);

}

这里调用nla\_put()函数将字符串消息填充到第一个attr属性中，同时指定attr的属性类型为DEMO\_CMD\_ATTR\_MESG，最后调用demo\_send\_reply()将消息往应用层发送：

/\* 完成构建并发送 \*/

return demo\_send\_reply(rep\_skb, info);

static int demo\_send\_reply(struct sk\_buff \*skb, struct genl\_info \*info)

{

struct genlmsghdr \*genlhdr = nlmsg\_data(nlmsg\_hdr(skb));

void \*reply = genlmsg\_data(genlhdr);

genlmsg\_end(skb, reply);

return genlmsg\_reply(skb, info);

}

首先调用genlmsg\_end()更新消息头重的长度字段，然后调用genlmsg\_reply启动回发流程：

static inline int genlmsg\_reply(struct sk\_buff \*skb, struct genl\_info \*info)

{

return genlmsg\_unicast(genl\_info\_net(info), skb, info->snd\_portid);

}

该函数为genlmsg\_unicast()的一个封装。这样demo family的回发字符串息就发送出去了。下面再来简单的看一下cmd\_attr\_echo\_data()函数，它同cmd\_attr\_echo\_message()函数基本类似，唯一的区别就是调用了内核提供的nla\_get\_s32()和nla\_put\_s32()这两个封装函数来获取和设置s32类型的attr属性，不做过多的论描。

static int cmd\_attr\_echo\_data(struct genl\_info \*info)

{

struct nlattr \*na;

s32 data;

struct sk\_buff \*rep\_skb;

size\_t size;

int ret;

/\* 读取用户下发的数据 \*/

na = info->attrs[DEMO\_CMD\_ATTR\_DATA];

if (!na)

return -EINVAL;

data = nla\_get\_s32(info->attrs[DEMO\_CMD\_ATTR\_DATA]);

pr\_info("demo generic netlink receive echo data %d\n", data);

/\* 回发数据 \*/

size = nla\_total\_size(sizeof(s32));

ret = demo\_prepare\_reply(info, DEMO\_CMD\_REPLY, &rep\_skb, size);

if (ret < 0)

return ret;

/\* 为了简单这里直接调用netlink库函数(对于需求的丰富可以自行封装) \*/

ret = nla\_put\_s32(rep\_skb, DEMO\_CMD\_ATTR\_DATA, data);

if (ret < 0)

goto err;

return demo\_send\_reply(rep\_skb, info);

err:

nlmsg\_free(rep\_skb);

return ret;

}

内核消息全部单播发送出去以后，下面来看应用层的接收流程。

### 4、应用层接收内核Demo Family回发消息

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

{

......

/\* 接收用户消息并解析(本示例程序中仅解析2个) \*/

demo\_msg\_recv\_analysis(nl\_fd, argc-1);

......

}

void demo\_msg\_recv\_analysis(int sd, int num)

{

int rep\_len;

int len;

struct nlattr \*na;

struct msgtemplate msg;

unsigned int data;

char \*string;

while (num--) {

/\* 接收内核消息回显 \*/

rep\_len = recv(sd, &msg, sizeof(msg), 0);

if (rep\_len < 0 || demo\_msg\_check(msg, rep\_len) < 0) {

fprintf(stderr, "nonfatal reply error: errno %d\n", errno);

continue;

}

PRINTF("received %d bytes\n", rep\_len);

PRINTF("nlmsghdr size=%zu, nlmsg\_len=%d, rep\_len=%d\n",

sizeof(struct nlmsghdr), msg.n.nlmsg\_len, rep\_len);

rep\_len = GENLMSG\_PAYLOAD(&msg.n);

na = (struct nlattr \*) GENLMSG\_DATA(&msg);

len = 0;

/\* 一个msg里可能有多个attr，所以这里循环读取 \*/

while (len < rep\_len) {

len += NLA\_ALIGN(na->nla\_len);

switch (na->nla\_type) {

case DEMO\_CMD\_ATTR\_MESG:

/\* 接收到内核字符串回显 \*/

string = (char \*) NLA\_DATA(na);

printf("echo reply:%s\n", string);

break;

case DEMO\_CMD\_ATTR\_DATA:

/\* 接收到内核数据回显 \*/

data = \*(int \*) NLA\_DATA(na);

printf("echo reply:%u\n", data);

break;

default:

fprintf(stderr, "Unknown nla\_type %d\n", na->nla\_type);

}

na = (struct nlattr \*) (GENLMSG\_DATA(&msg) + len);

}

}

}

应用程序调用demo\_msg\_recv\_analysis()函数接收内核消息，其中num表示接收消息的个数。该函数中循环调用recv函数阻塞式的接收内核netlink消息。当有消息接收到以后调用demo\_msg\_check()函数判断消息的有效性：

int demo\_msg\_check(struct msgtemplate msg, int rep\_len)

{

if (msg.n.nlmsg\_type == NLMSG\_ERROR || !NLMSG\_OK((&msg.n), rep\_len)) {

struct nlmsgerr \*err = NLMSG\_DATA(&msg);

fprintf(stderr, "fatal reply error, errno %d\n", err->error);

return -1;

}

return 0;

}

这里首先判断消息头中的nlmsg\_type字段，如果该字段为NLMSG\_ERROR表示接收到了错误的消息，应该立即丢弃。如果接收到的消息类型无误，则接下来判断消息的长度是否足够，使用的是NLMSG\_OK宏（见netlink.h）。然后接收函数循环读取attr属性并根据属性的attr类型单独进行处理，本示例中仅仅在终端中打印。需要补充的是，本程序中并没有对接收到的消息cmd类型进行判断，其实为了程序的可靠性考虑，最好增加这一方面的判断（虽然netlink的id号保证了不会收到其他id的genetlink消息，但是当某family的cmd类型较多时容易引起混乱）。至此demo family的genetlink单播通信过程就大致分析完毕，另外关于多播的通信流程也比较简单，不难掌握，以后有时间再行补充。

# 五、总结

本文和前一篇《Generic Netlink内核实现分析（一）：初始化》分析了内核与用户态Generic netlink的创建、创建及通信流程。Generic netlink作为内核netlink机制的一种特殊应用，更好的实现了内核态与用户态之间的全双工通信的拓展，同时本文还给出了一个简单的示例程序用以分析原理，亦可用作日后扩展的原型程序。