传统socket

int socket(int domain, int type, int protocol);

domain：指定一个通信域

struct net\_proto\_family {//表示一个域

int family;

int (\*create)(struct net \*net, struct socket \*sock,

int protocol, int kern);

struct module \*owner;

};

例如：

static const struct net\_proto\_family inet\_family\_ops = {

.family = PF\_INET,

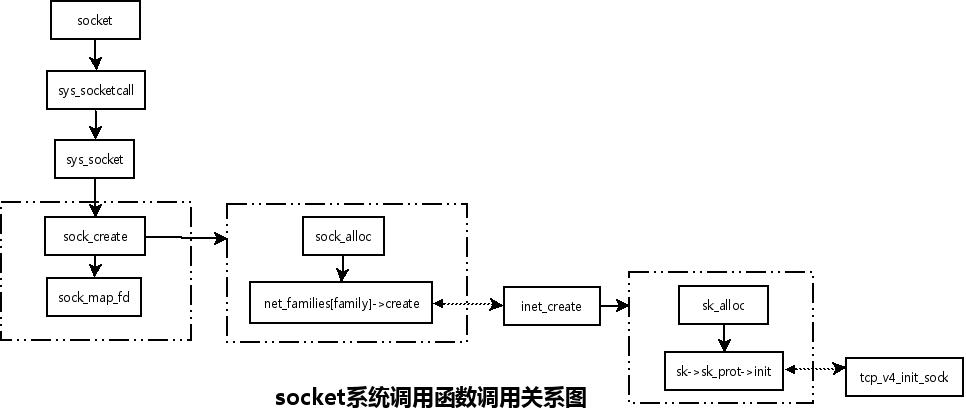
.create = inet\_create,

.owner = THIS\_MODULE,

};

注册：sock\_register(&inet\_family\_ops );

socket函数调用到内核的上面的inet\_create

****

PF\_PACKET：

static const struct net\_proto\_family packet\_family\_ops = {

.family = PF\_PACKET,

.create = packet\_create,

.owner = THIS\_MODULE,

};

struct packet\_type {//包接收类型:包含钩子函数 还有type：protocol类型(对应于socket第三个参数)

\_\_be16 type; /\* This is really htons(ether\_type). \*/

struct net\_device \*dev; /\* NULL is wildcarded here \*/

int (\*func) (struct sk\_buff \*,

struct net\_device \*,

struct packet\_type \*,

struct net\_device \*);

struct sk\_buff \*(\*gso\_segment)(struct sk\_buff \*skb,

u32 features);

int (\*gso\_send\_check)(struct sk\_buff \*skb);

struct sk\_buff \*\*(\*gro\_receive)(struct sk\_buff \*\*head,

struct sk\_buff \*skb);

int (\*gro\_complete)(struct sk\_buff \*skb);

void \*af\_packet\_priv;

struct list\_head list;

};

/\*操作函数 包括bind connect sendmesg等等 如果是PF\_INET 那么这个就根据protocol选择udp或者tcp的处理函数 如上图(在PF\_INET协议上支持的套接字接口有流套接字(SOCK\_STREAM)，数据报套接字(SOCK\_DGRAM)和原始套接字(SOCK\_RAW),结构为struct inet\_protosw 的proto\_ops)\*/

struct proto\_ops {

int family;

struct module \*owner;

int (\*release) (struct socket \*sock);

int (\*bind) (struct socket \*sock,

struct sockaddr \*myaddr,

int sockaddr\_len);

int (\*accept) (struct socket \*sock,

struct socket \*newsock, int flags);

int (\*getname) (struct socket \*sock,

struct sockaddr \*addr,

int \*sockaddr\_len, int peer);

unsigned int (\*poll) (struct file \*file, struct socket \*sock,

struct poll\_table\_struct \*wait);

int (\*ioctl) (struct socket \*sock, unsigned int cmd,

unsigned long arg);

.....

}

static struct proto packet\_proto = { //packet\_proto->obj\_size 决定了分配的大小 如果是PF\_INET 那么这个就根据protocol选择udp或者tcp的 //proto(结构同上struct inet\_protosw 的proto)

.name = "PACKET",

.owner = THIS\_MODULE,

.obj\_size = sizeof(struct packet\_sock),

};

socket->sys\_\_socket ->socket\_\_creat->packet\_family\_ops->create;

static int packet\_create(struct net \*net, struct socket \*sock, int protocol, int kern)

{

sk = sk\_alloc(net, PF\_PACKET, GFP\_KERNEL, &packet\_proto);//分配sk结构体 sk->sk\_prot = packet\_proto; packet\_proto->obj\_size 决定了分配的大小，struct packet\_sock第一个成员就是sk，所以有(struct packet\_sock\*)sk就成了一个struct packet\_sock

sock->ops = &packet\_ops;//struct proto\_ops packet\_ops;//

packet\_sock->prot\_hook.func = packet\_rcv;/\*struct packet\_type packet\_sock->prot\_hook;作为packet\_sock的成员\*/

packet\_sock->prot\_hook.type = protocol;

packet\_sock->prot\_hook.af\_packet\_priv = sk；//sk传给包类型，方便接收到包后找到相应的sock

dev\_add\_pack(&po->prot\_hook);//把packet\_type加入相应的type或者全局接收列表中

}

**应用层**的bind ioctl connenct sendmesg等都是调到内核的sock->ops(packet\_ops)实现的。

bind——Sys\_bind——packet\_bind(sock->proto\_ops->bind)

sendmsg——sys\_sendmsg——packet\_sendmsgsock->proto\_ops->sendmsgt)

rcvmsg()——sys\_sendmsg——packet\_revmsg()(sock->proto\_ops->revmsg)

........

**包的接收：**

网卡驱动--->netif\_rx()--->netif\_receive\_skb()->deliver\_skb()->packet\_type.func->packet\_rcv;

static int packet\_rcv(struct sk\_buff \*skb, struct net\_device \*dev, struct packet\_type \*pt, struct net\_device \*orig\_dev)

{

sk = pt->af\_packet\_priv;//获取sk

struct packet\_sock \*po = (struct packet\_sock\*)sk；//获取packet\_sock

res = run\_filter(skb, sk, snaplen);//根据过滤规则过滤

\_\_skb\_queue\_tail(&sk->sk\_receive\_queue, skb);//包传到sk的queue上

}