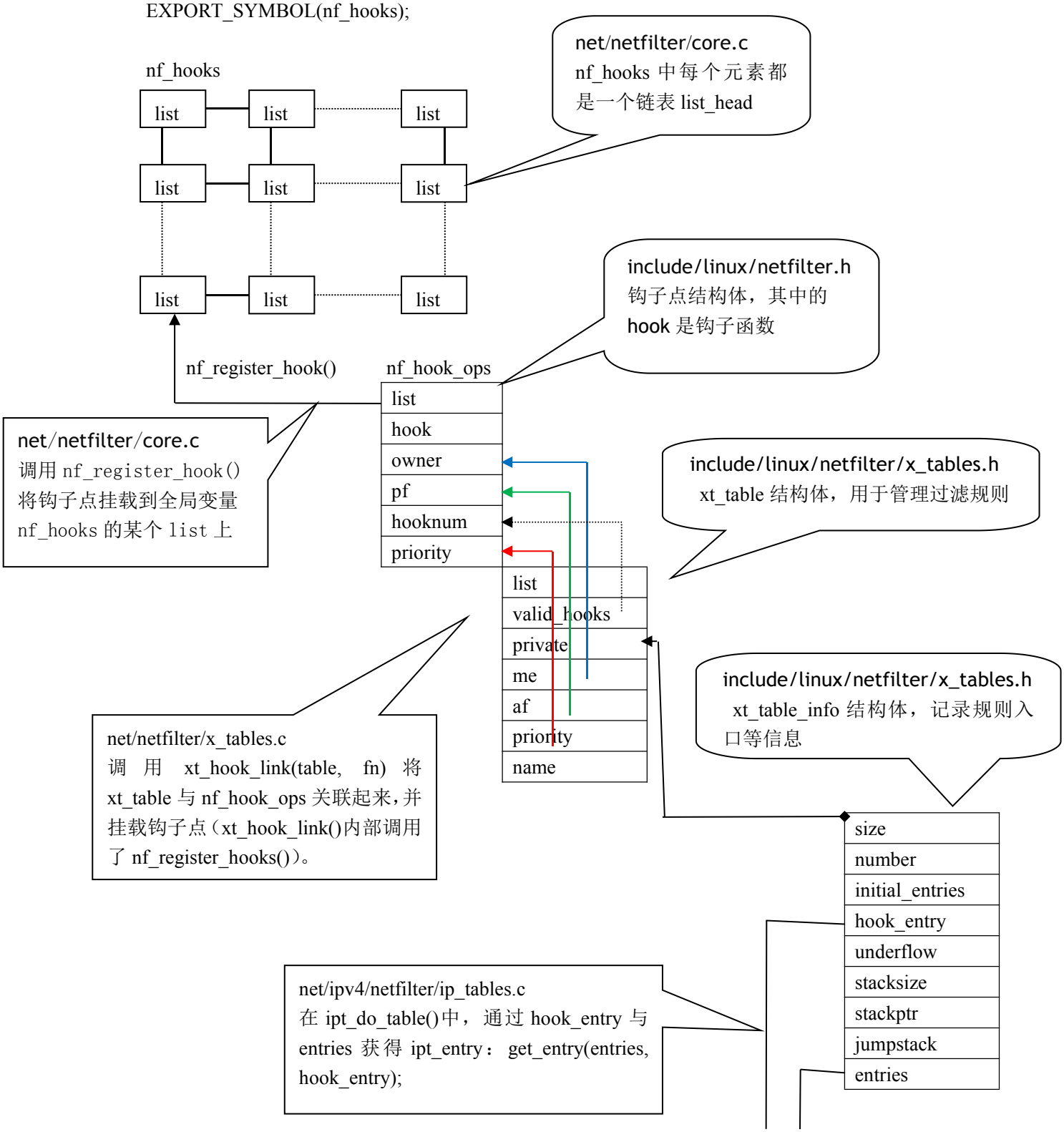
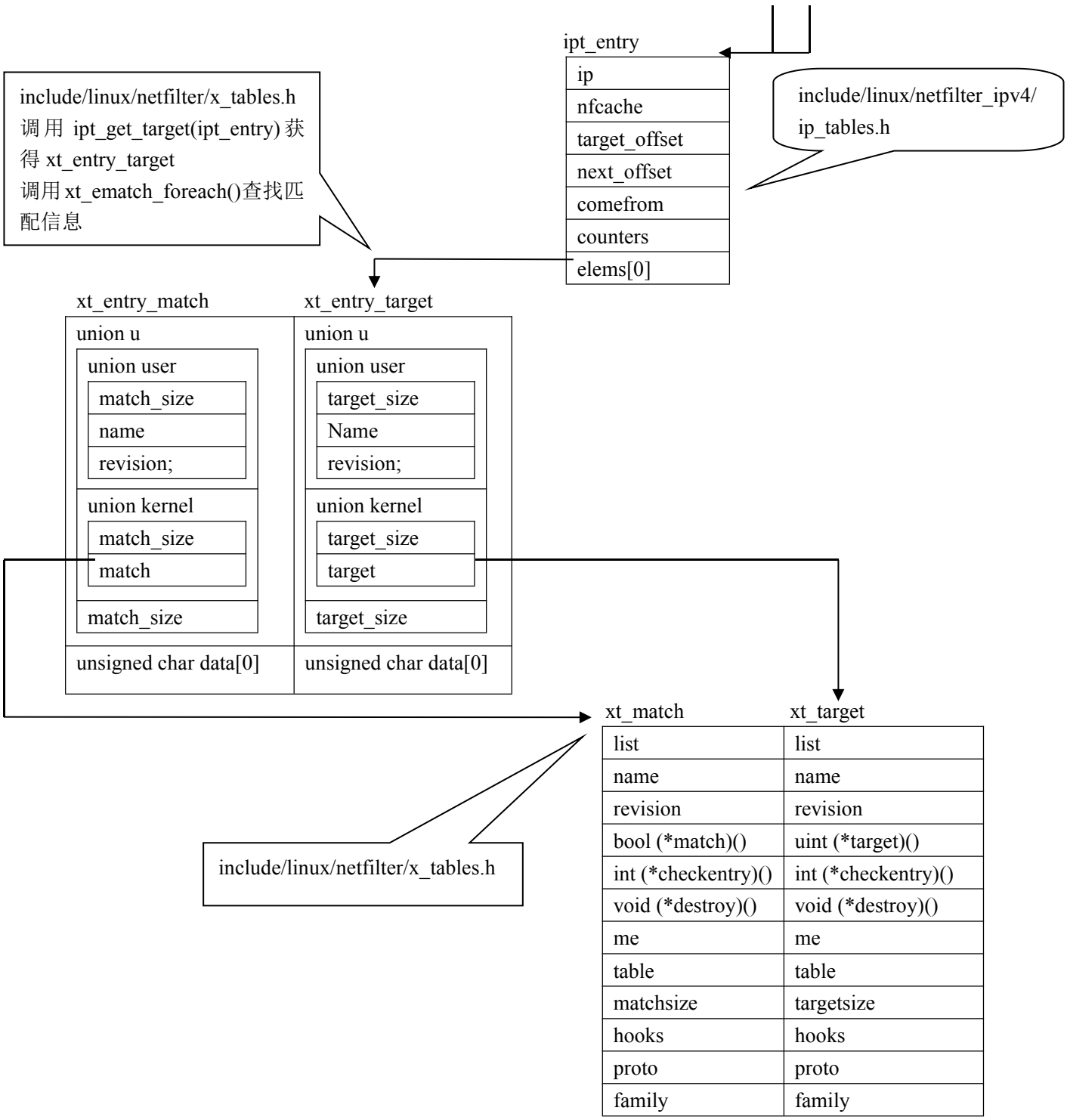
Netfilter 架构分析

一、全局图

在文件 net/netfilter/core.c 中定义了一全局变量 nf_hooks，用于记录钩子点。nf_hooks 第一维代表协议数，第二维代表钩子数。

struct list_head nf_hooks[NFPROTO_NUMPROTO][NF_MAX_HOOKS] __read_mostly;
EXPORT_SYMBOL(nf_hooks);



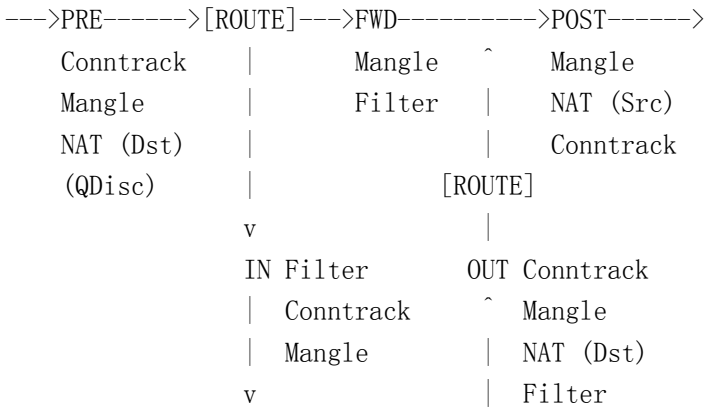


xt_match	xt_target
list	list
name	name
revision	revision
bool (*match)()	uint (*target)()
int (*checkentry)()	int (*checkentry)()
void (*destroy)()	void (*destroy)()
me	me
table	table
matchsize	targetsize
hooks	hooks
proto	proto
family	family

二、钩子函数(hook)与过滤规则表(xt_table)

前面已经提到，钩子函数与过滤规则的管理是通过全局变量 nf_hooks 来实现的，那么，什么时候会调用钩子函数呢？钩子函数又是如何利用已经注册好的过滤规则的呢？

在 Linux 内核中定义了网络数据包的流动方向，数据包被网卡捕获后，它在内核网络子系统里的传输路径是：pre-routing→route(in or forward)→(out)→post-routing。在 netfilter 上注册的钩子函数如下所示(这些钩子函数按它们被调用的顺序排列)：



共有五个位置设置了钩子点，PRE、IN、FWD、OUT、POST。钩子函数被注册到相应位置之后，它们就会那里等待数据包的到来，在接收数据包的地方，钩子函数被调用，数据包先由钩子函数捕获，进行处理，然后再转发或者丢弃。

钩子函数的声明：include/linux/netfilter.h

```
typedef unsigned int nf_hookfn(unsigned int hooknum,
                               struct sk_buff *skb,
                               const struct net_device *in,
                               const struct net_device *out,
                               int (*okfn)(struct sk_buff *));
```

下面围绕 iptable_filter 为中心进行分析：

net/ipv4/netfilter/iptable_filter.c
调用 xt_hook_link()使 xt_table 与 nf_hook_ops 关联起来，并挂载钩子点。

```
static int __init iptable_filter_init(void)
{
        。。。。。。
        /* Register hooks */
        filter_ops = xt_hook_link(&packet_filter, iptable_filter_hook);
        。。。。。。
}
```

net/ipv4/netfilter/iptable_filter.c
定义一个钩子函数 iptable_filter_hook()，调用 ipt_do_table 来使用规则表里的规则。

```
static unsigned int iptable_filter_hook()
{
        。。。。。。
        net = dev_net((in != NULL) ? in : out);
        return ipt_do_table(skb, hook, in, out, net->ipv4.iptable_filter);
}
```

获得规则入口 ipt_entry

遍历匹配表，匹配则执行下面动作

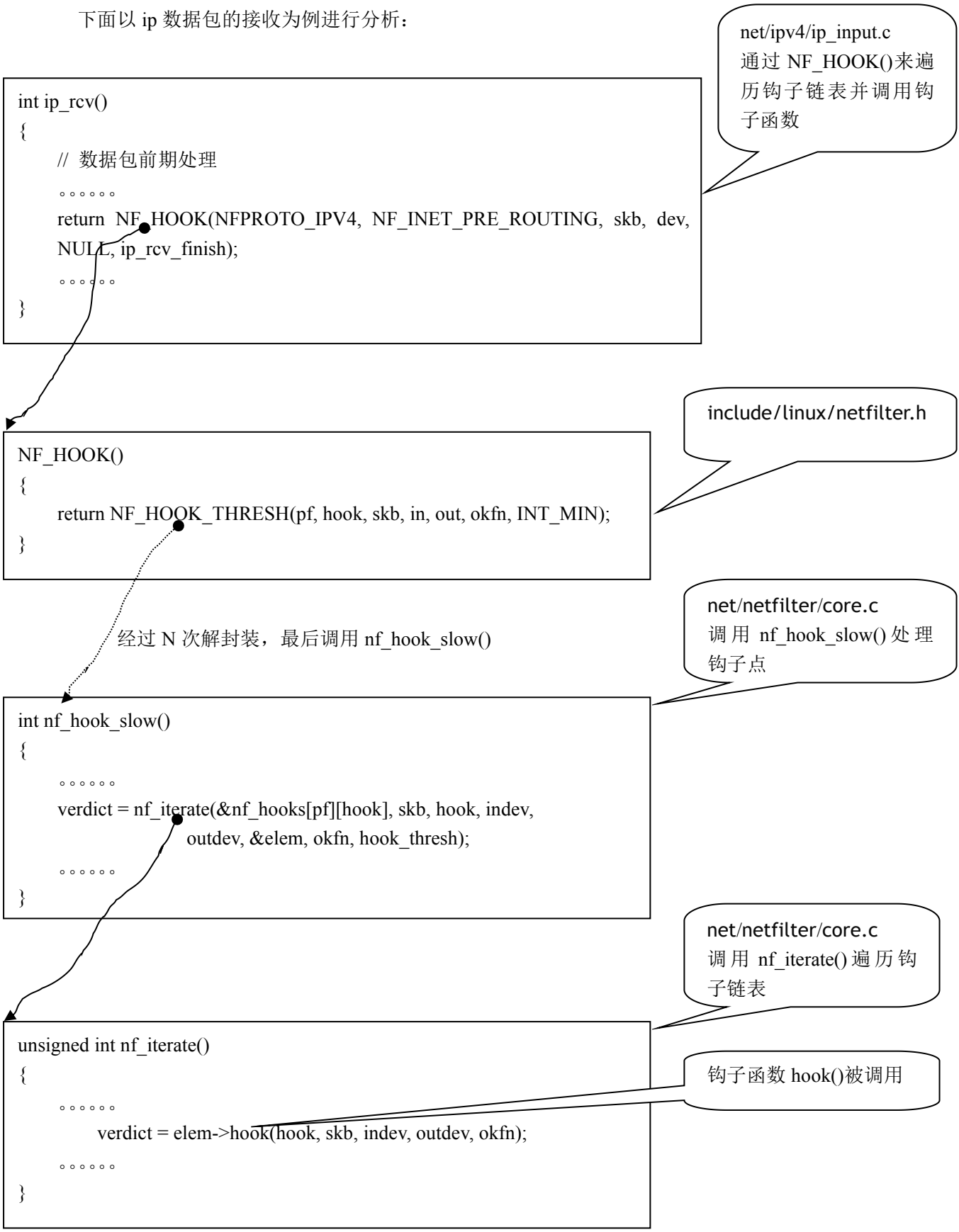
获得规则目标 xt_entry_target

调用目标方法 xt_target.target()

```
unsigned int ipt_do_table()
{
        。。。。。。
        table_base = private->entries[cpu];
        e = get_entry(table_base, private->hook_entry[hook]);
        do {
                。。。。。。
                xt_ematch_foreach(ematch, e) {
                        acpar.match         = ematch->u.kernel.match;
                        acpar.matchinfo = ematch->data;
                        if (!acpar.match->match(skb, &acpar))
                                goto no_match;}
                。。。。。。
                t = ipt_get_target(e);
                。。。。。。
                verdict = t->u.kernel.target->target(skb, &acpar);
                。。。。。。
        } while (!acpar.hotdrop);
        。。。。。。
}
```

从上面已经知道了过滤规则的使用是由钩子函数来实现的，那么，什么时候、在什么地方调用钩子函数呢？

下面以 ip 数据包的接收为例进行分析：

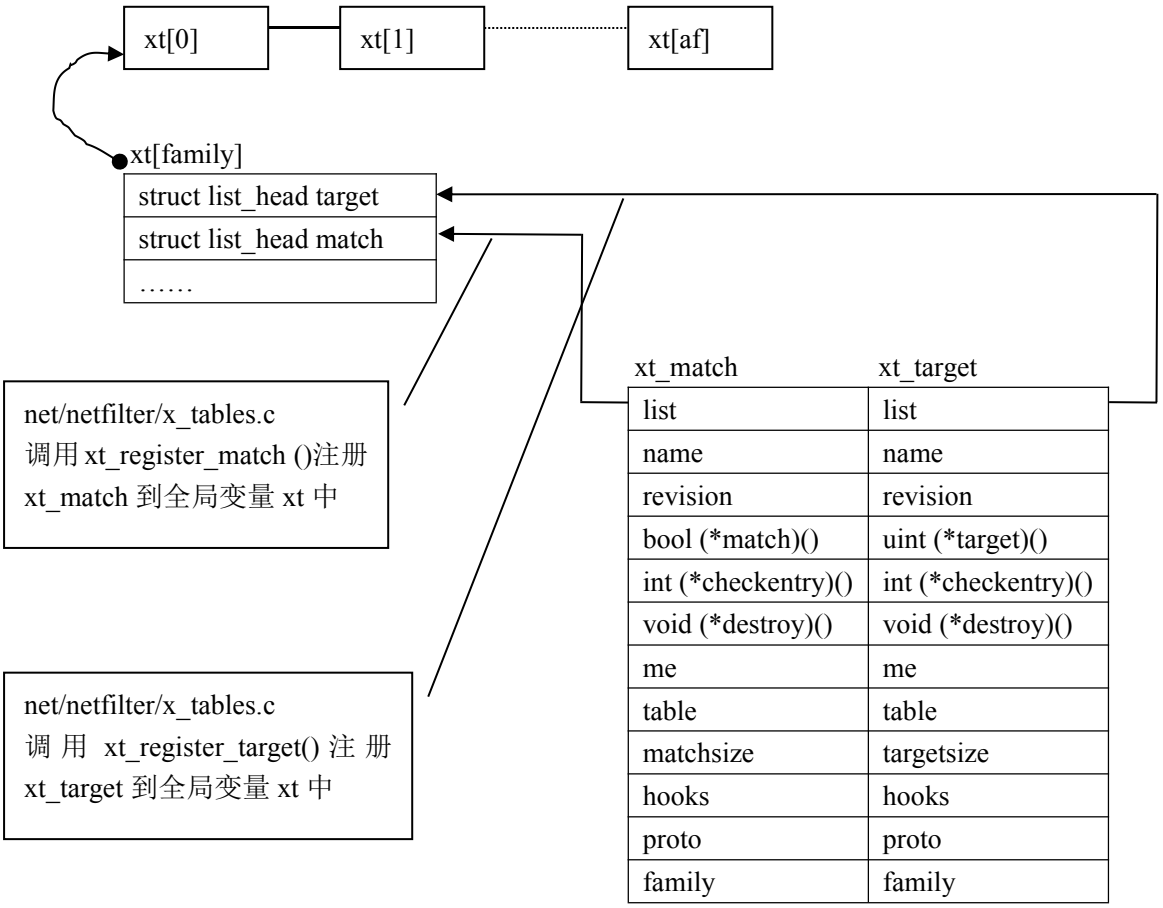


三、规则的管理

一条规则由 ipt_entry 表示， ipt_entry_match 中的 xt_match 表示匹配规则，ipt_entry_target 中的 xt_target 表示匹配后的动作。

在文件 net/netfilter/x_tables.c 中定义了一个全局变量 xt：

static struct xt_af *xt;



在文件 include/net/net_namespace.h 中定义了 net 结构体：

```
struct net {
。。。。。。
#ifdef CONFIG_NETFILTER
        struct netns_xt         xt;
。。。。。。
#endif
。。。。。。
};
```

在文件 include/net/netns/x_tables.h 中定义了 netns_xt 结构体：

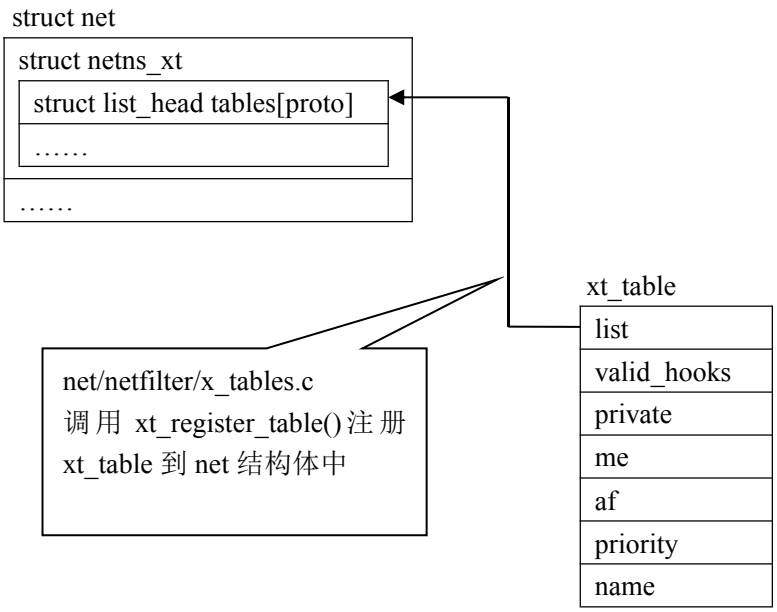
```
struct netns_xt {

    struct list_head tables[NFPROTO_NUMPROTO];

    。。。。。。

};
```

规则表 xt_table 的管理：



四、ip table 内建的 filter 表

定义在文件 net/ipv4/netfilter/iptable_filter.c 中

xt_table 表：

```
static const struct xt_table packet_filter = {
.name           = "filter",//规则表默认的名字
.valid_hooks    = FILTER_VALID_HOOKS,//定义了三个钩子点 IN/FORWARD/OUT
.me         = THIS_MODULE,
.af         = NFPROTO_IPV4,//协议
.priority   = NF_IP_PRI_FILTER,//优先级
};
```

per network 操作：

```
static struct pernet_operations iptable_filter_net_ops = {
    .init = iptable_filter_net_init,
    .exit = iptable_filter_net_exit,
};
```

初始化 filter 表：

```
static int __init iptable_filter_init(void)
{
	。。。。。。
	/* 注册 pernet_operations */
	ret = register_pernet_subsys(&iptable_filter_net_ops);
	if (ret < 0)
		return ret;

	/* 注册 hooks */
	filter_ops = xt_hook_link(&packet_filter, iptable_filter_hook);
	。。。。。。
}
```

五、内核空间与用户空间的通信

运行于用户态的 iptables 规则的下发通过套接字操作接口 setsockopt()与 getsockopt() 实现，内核态结构体 struct nf_sockopt_ops 里的相关系统调用使该规则能被内核识别，并更新规则表。

1、内核态的定义在文件 net/ipv4/netfilter/ip_tables.c 中

net/ipv4/netfilter/ip_tables.c
socket 系统调用结构体

```
static struct nf_sockopt_ops ipt_sockopts = {
        .pf             = PF_INET,
        .set_optmin     = IPT_BASE_CTL,
        .set_optmax     = IPT_SO_SET_MAX+1,
        .set            = do_ipt_set_ctl,
#ifdef CONFIG_COMPAT
        .compat_set     = compat_do_ipt_set_ctl,
#endif
        .get_optmin     = IPT_BASE_CTL,
        .get_optmax     = IPT_SO_GET_MAX+1,
        .get            = do_ipt_get_ctl,
#ifdef CONFIG_COMPAT
        .compat_get     = compat_do_ipt_get_ctl,
#endif
        .owner          = THIS_MODULE,
};
```

net/ipv4/netfilter/ip_tables.c
do_ipt_set_ctl()的实现

```
static int
do_ipt_set_ctl(struct sock *sk, int cmd, void __user *user, unsigned int len)
{
        。。。
        switch (cmd) {
        case IPT_SO_SET_REPLACE:
                。。。
        case IPT_SO_SET_ADD_COUNTERS:
                。。。
        default:
                。。。
        }
        。。。
}
```

```
do_ipt_get_ctl(struct sock *sk, int cmd, void __user *user, int *len)
{
    。。。
    switch (cmd) {
    case IPT_SO_GET_INFO:
        。。。
    case IPT_SO_GET_ENTRIES:
        。。。
    case IPT_SO_GET_REVISION_MATCH:
    case IPT_SO_GET_REVISION_TARGET: {
        。。。
    }
    default:
        。。。
    }
    。。。
}
```

2、用户态的调用在 iptables 源码目录的 libiptc/libiptc.c 文件中。

iptables 源码目录下 ./libiptc/libiptc.c setsockopt()与 getsockopt()的调用

```
ret = setsockopt(handle->sockfd, TC_IPPROTO, SO_SET_REPLACE,
      repl, sizeof(*repl) + repl->size);
```

```
if (getsockopt(sockfd, TC_IPPROTO, SO_GET_INFO, &info, &s) < 0) {
        close(sockfd);
        return NULL;
```

从内核态 do_ipt_set_ctl()的实现可知，用户态的 setsockopt()仅支持 SET_REPLACE 与 SET_ADD_COUNTERS 两种操作。

从内核态 do_ipt_get_ctl()的实现可知，用户态的 getsockopt()支持 GET_INFO、GET_ENTRIES、GET_REVISION_MATCH 与 GET_REVISION_TARGET 操作。

内核态与用户态的宏定义如下，它们对应的值是一样的：

内核态	用户态
IPT_SO_SET_REPLACE	SO_SET_REPLACE
IPT_SO_SET_ADD_COUNTERS	SO_SET_ADD_COUNTERS
IPT_SO_GET_INFO	SO_GET_INFO
IPT_SO_GET_ENTRIES	SO_GET_ENTRIES
IPT_SO_GET_REVISION_MATCH	SO_GET_REVISION_MATCH
IPT_SO_GET_REVISION_TARGET	SO_GET_REVISION_TARGET

六、参考文献

1、 Linux netfilter Hacking HOWTO
Rusty Russell and Harald Welte, mailing list netfilter@lists.samba.org

2、 Netfilter 框架图形化概要分析
by shenguanghui, shenghxscc@126.com 2006-02-24

3、 iptables 内核框架和应用层 iptables 命令图形化概要分析
by shenguanghui shenghxscc@126.com 2006-02-24

4、 sk_buff_head 图形化简介
by shenguanghui (shenghxscc@126.com)

5、网络代码分析第二部分－－网络子系统在IP层的收发过程剖析
R.wen (rwen2012@126.com)

6、洞悉 linux 下的 Netfilter&iptables
http://blog.chinaunix.net/uid-23069658-id-3160506.html

7、Netfilter 实现机制分析
2008-12 唐文 tangwen1123@163.com