Netfilter 实现机制分析

2008-12
唐文
tangwen1123@163.com

1. 前言

Netfilter 作为目前进行包过滤，连接跟踪，地址转换等的主要实现框架，了解其内部机制对于我们更好的利用 Netfilter 进行设计至关重要，因此本文通过阅读内核源码 2.6.21.2，根据自身的分析总结出 Netfilter 的大致实现机制，由于自身水平有限，且相关的参考资料较少，因此其中的结论不能保证完全正确，如果在阅读本文的过程中发现了问题欢迎及时与作者联系。

2. 规则的存储与遍历机制

Ø 规则的存储机制

在 Netfilter 中规则是顺序存储的，一条规则主要包括三个部分：ipt_entry、ipt_entry_matches、ipt_entry_target。ipt_entry_matches 由多个 ipt_entry_match 组成，ipt_entry 结构主要保存标准匹配的内容，ipt_entry_match 结构主要保存扩展匹配的内容，ipt_entry_target 结构主要保存规则的动作。在 ipt_entry 中还保存有与遍历规则相关的变量 target_offset 与 next_offset，通过 target_offset 可以找到规则中动作部分 ipt_entry_target 的位置，通过 next_offset 可以找到下一条规则的位置。规则的存储如下图 2-1 所示。

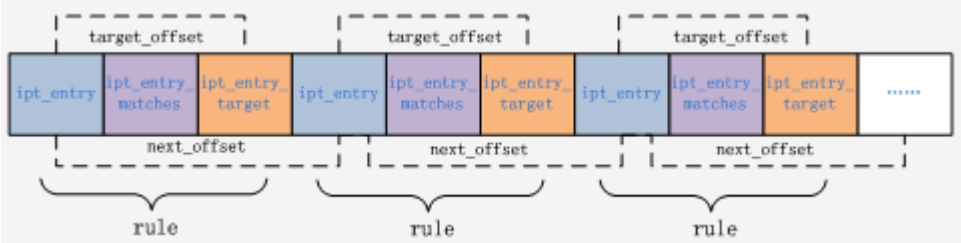

图 2-1 规则的存储

ipt_entry 结构如下图 2-2 所示，其成员 ip 指向结构 ipt_ip，该结构主要保存规则中标准匹配的内容（IP、mask、interface、proto 等），target_offset 的值等于 ipt_entry 的长度与 ipt_entry_matches 的长度之和，next_offset 的值等于规则中三个部分的长度之和。通过 target_offset 与 next_offset 可以实现规则的遍历。

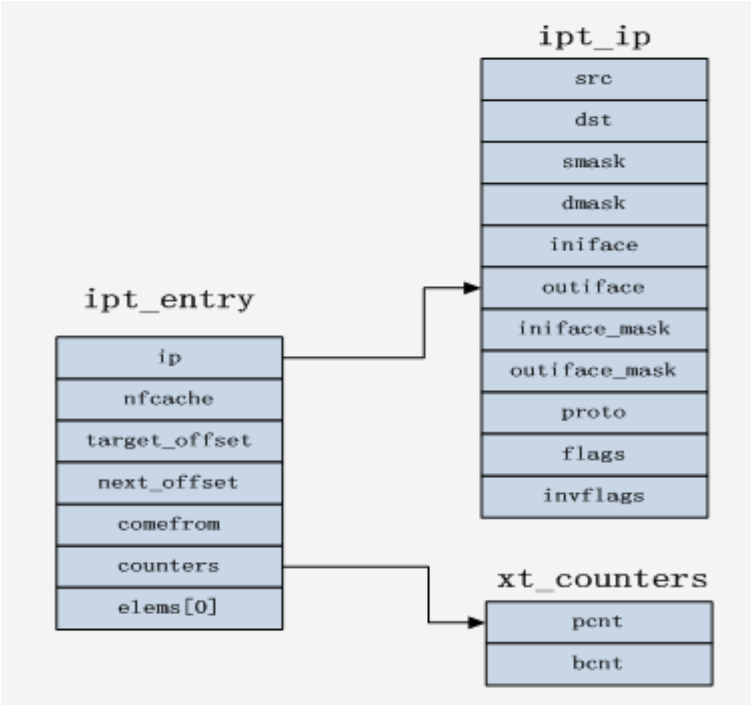

图 2-2 ipt_entry 结构

ipt_entry_match 主要保存规则中扩展匹配内容（tos、ttl、time 等），其是 Netfilter 中内核与用户态交互的关键数据结构，在其内核部分由一个函数指针指向一个 ipt_match 结构，该结构体中包含了对包做匹配的函数，是真正对包做匹配的地方。ipt_entry_target 结构与 ipt_entry_match 结构很类似。

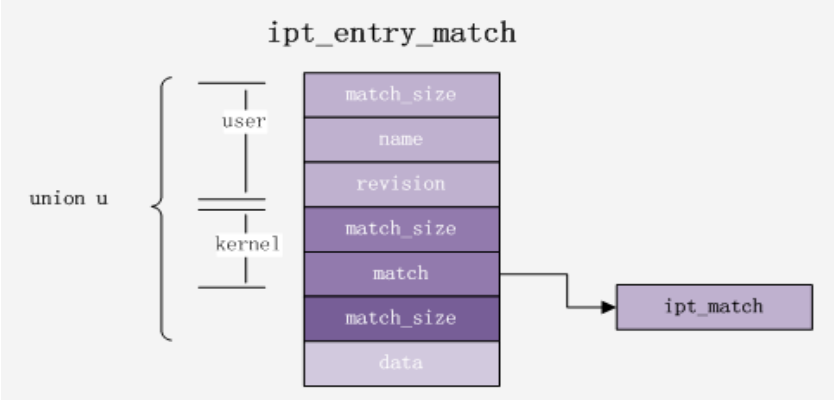

图 2-3 ipt_entry_match 结构

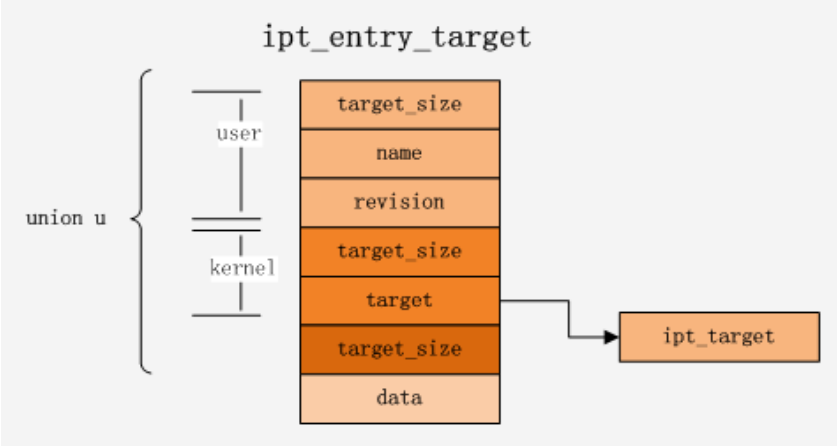

图 2-4 ipt_entry_target 结构

Ø 规则的遍历机制

在 Netfilter 中，函数 ipt_do_table()实现了规则的遍历，该函数根据传入的参数 table 和 hook 找到相应的规则起点，即第一个 ipt_entry 的位置，主要通过函数 get_entry()实现。

```
private = table->private;
table_base = (void *)private->entries[smp_processor_id()];
e = get_entry(table_base, private->hook_entry[hook]);
```

标准匹配是通过函数 ip_packet_match()实现的，该函数主要对包的五元组信息进行匹配，扩展匹配则通过宏 IPT_MATCH_ITERATE 实现，该宏的定义为：

```
#define IPT_MATCH_ITERATE(e, fn, args...)      \
({                                             \
        unsigned int __i;                      \
        int __ret = 0;                         \
        struct ipt_entry_match *__match;       \
                                               \
        for (__i = sizeof(struct ipt_entry);   \
             __i < (e)->target_offset;         \
             __i += __match->u.match_size) {   \
                __match = (void *)(e) + __i;   \
                                               \
                __ret = fn(__match , ## args);\
                if (__ret != 0)                \
                        break;                 \
        }                                      \
        __ret;                                 \
})
```

宏 IPT_MATCH_ITERATE 依次调用各个 ipt_entry_match 所指向的 ipt_match 中 match()处理数据包，在 for 循环中使用了 terget_offset 位置变量查找 match 的位置。

在对数据包进行了匹配后，接着需要进行相应的动作处理，通过函数 ipt_get_target()获取规则动作 ipt_entry_target 的位置：

```
static __inline__ struct ipt_entry_target *
ipt_get_target(struct ipt_entry *e)
{
        return (void *)e + e->target_offset;
}
```

如果还需要继续遍历下一条规则，则继续执行以下语句以找到下一条规则的开始位置：

```
e = (void *)e + e->next_offset;
```

3. 表、匹配、动作存储及管理机制

Ø 表、匹配、动作的存储机制

规则中所使用到的 match、target、table 使用全局变量 xt_af 所指向的相应链表保存，这

些链表是在对 Netfilter 进行初始化或匹配模块扩展时进行更新的，在初始化时，默认的表及动作则添加到相应的链表中。Netfilter 实现了很好的扩展性，如需要对数据包的时间进行匹配，则在 match 的链表中需要首先增加 time 扩展匹配模块，在相应的规则中则通过指向该 time 模块所对应的函数 match()以进行时间的匹配。xt_af 是个一维数组，其按照协议族的不同分别存储，目前我们常用的协议族主要是 AF_INET。

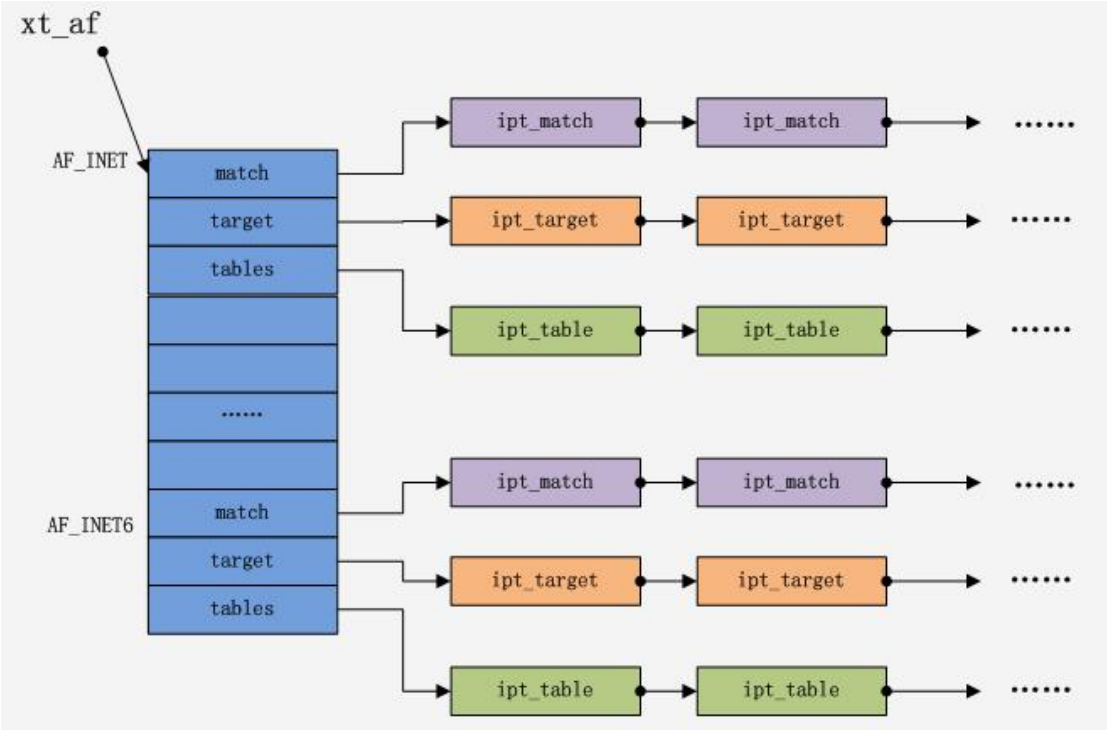

图 3-1 match,target,table 的全局存储

match、target、table 的全局存储如上图 3-1 所示，以下为各部分的详细的结构表示。当扩展一个匹配模块时，其会注册一个 ipt_match 结构到 match 链表中，该结构的主要变量值如下图所示，name 表示扩展模块的名字，match()是该模块最主要的函数，其主要对数据包进行相应的比较，checkentry()主要对包进行相应的完整性检验，destroy()在对模块进行撤销时调用。如果需要自己新加一个扩展模块，则需要构造一个 ipt_match 结构并注册到相应的链表中。ipt_target 的结构与 ipt_match 相似，其最主要的函数是 target()。

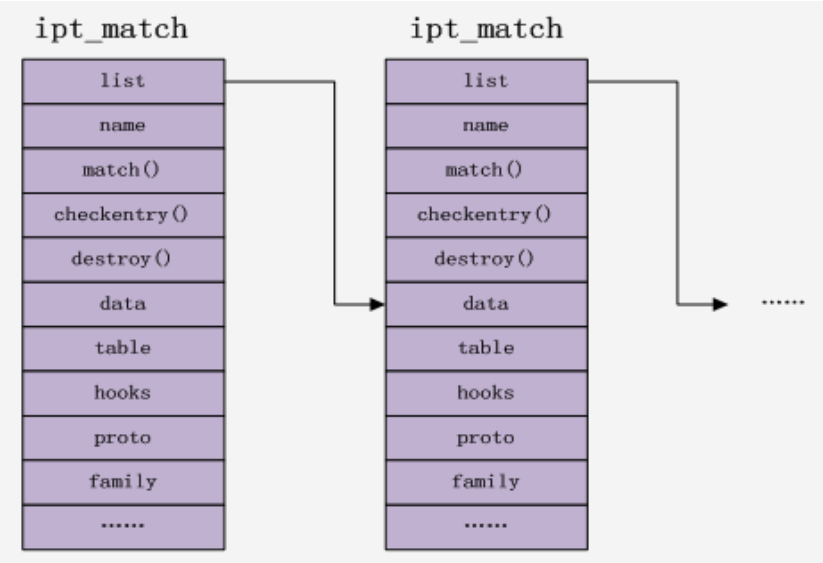

图 3-2 ipt_match 结构的存储

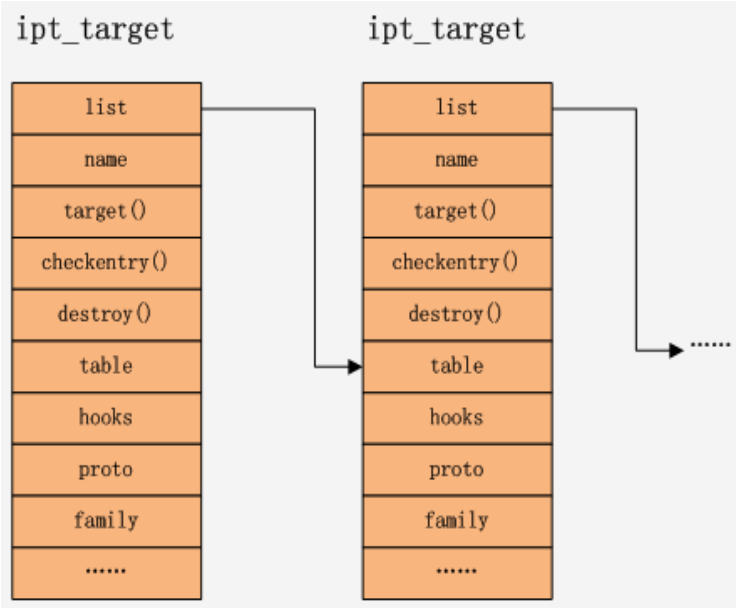

图 3-3 ipt_target 结构的存储

table 主要是用来对规则进行管理，通过 table 中的相应参数可以找到相应的规则所处的入口位置。

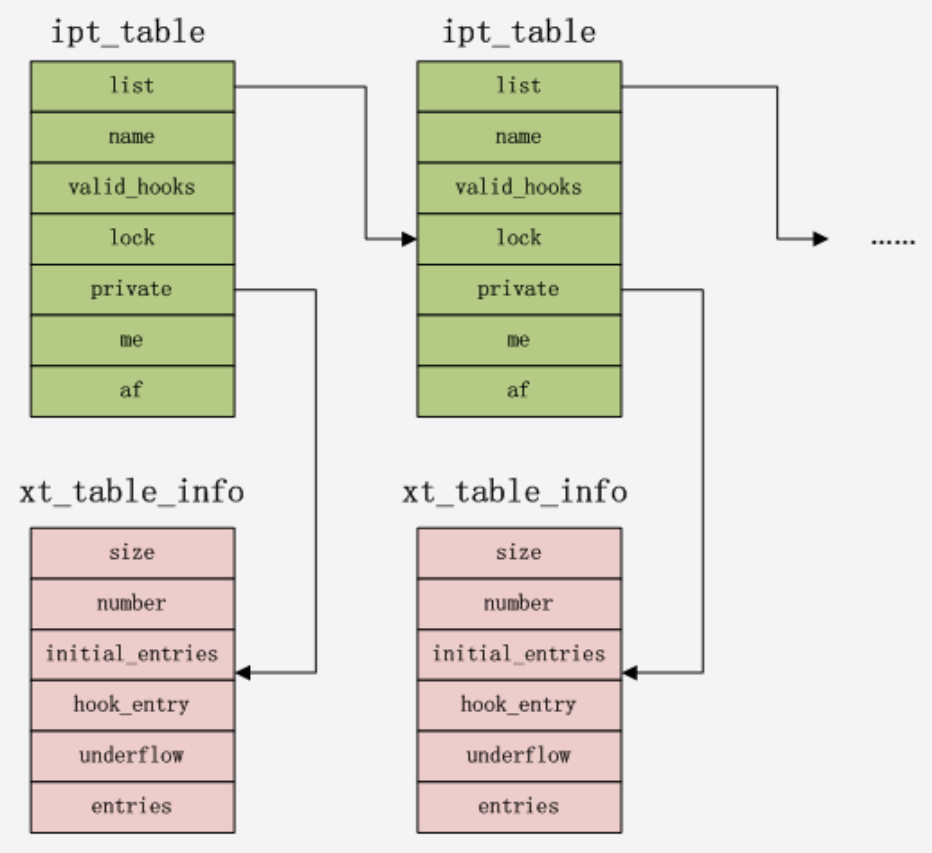

图 3-4 ipt_table 结构的存储

Ø 表、匹配、动作的管理机制

match、target、table 的注册分别调用 xt_register_match()、xt_register_target()、xt_register_table()实现，前两个注册函数很相似，xt_register_table()则稍微复杂些。撤销时则分别调用相应的 register 函数实现。xt_register_match()函数的定义如下(xt_match 与 ipt_match 是一样的)：

```
int
xt_register_match(struct xt_match *match)
{
	int ret, af = match->family;

	ret = mutex_lock_interruptible(&xt[af].mutex);
	if (ret != 0)
		return ret;

	list_add(&match->list, &xt[af].match);
	mutex_unlock(&xt[af].mutex);

	return ret;
}
```

xt_register_table()函数的定义如下(xt_table 与 ipt_table 是一样的)，因为一个 xt_table 结构中还指向另一结构 xt_table_info，该结构主要描述表的相关信息，所以对表注册时需要对这两类结构体进行定义。

```
int xt_register_table(struct xt_table *table,
                      struct xt_table_info *bootstrap,
                      struct xt_table_info *newinfo)
{
        int ret;
        struct xt_table_info *private;
        struct xt_table *t;

        ret = mutex_lock_interruptible(&xt[table->af].mutex);
        if (ret != 0)
                return ret;

        /* Don't autoload: we'd eat our tail... */
        list_for_each_entry(t, &xt[table->af].tables, list) {
                if (strcmp(t->name, table->name) == 0) {
                        ret = -EEXIST;
                        goto unlock;
                }
        }

        /* Simplifies replace_table code. */
        table->private = bootstrap;
        rwlock_init(&table->lock);
        if (!xt_replace_table(table, 0, newinfo, &ret))
                goto unlock;

        private = table->private;
        duprintf("table->private->number = %u\n", private->number);

        /* save number of initial entries */
        private->initial_entries = private->number;

        list_add(&table->list, &xt[table->af].tables);

        ret = 0;
 unlock:
        mutex_unlock(&xt[table->af].mutex);
        return ret;
}
```

4. 钩子函数的存储及管理机制

Ø 钩子函数的存储机制

钩子函数由一个全局二维链表 nf_hooks 保存，其按照协议族归类存储，在每个协议族中，根据钩子点顺序排列，在钩子点内则根据钩子函数的优先级依次排列。钩子函数的存储图如下图 4-1 所示，链表中的每个元素都是指向结构体 nf_hook_ops 中的 hook()函数的指针，nf_hook_ops 实际存储了钩子函数的内容，其结构如图 4-2 所示。在相应的钩子点调用钩子函数时，则根据协议族和钩子点找到相应的链表入口，然后依次调用该链中的每一个钩子函数对数据包进行操作。

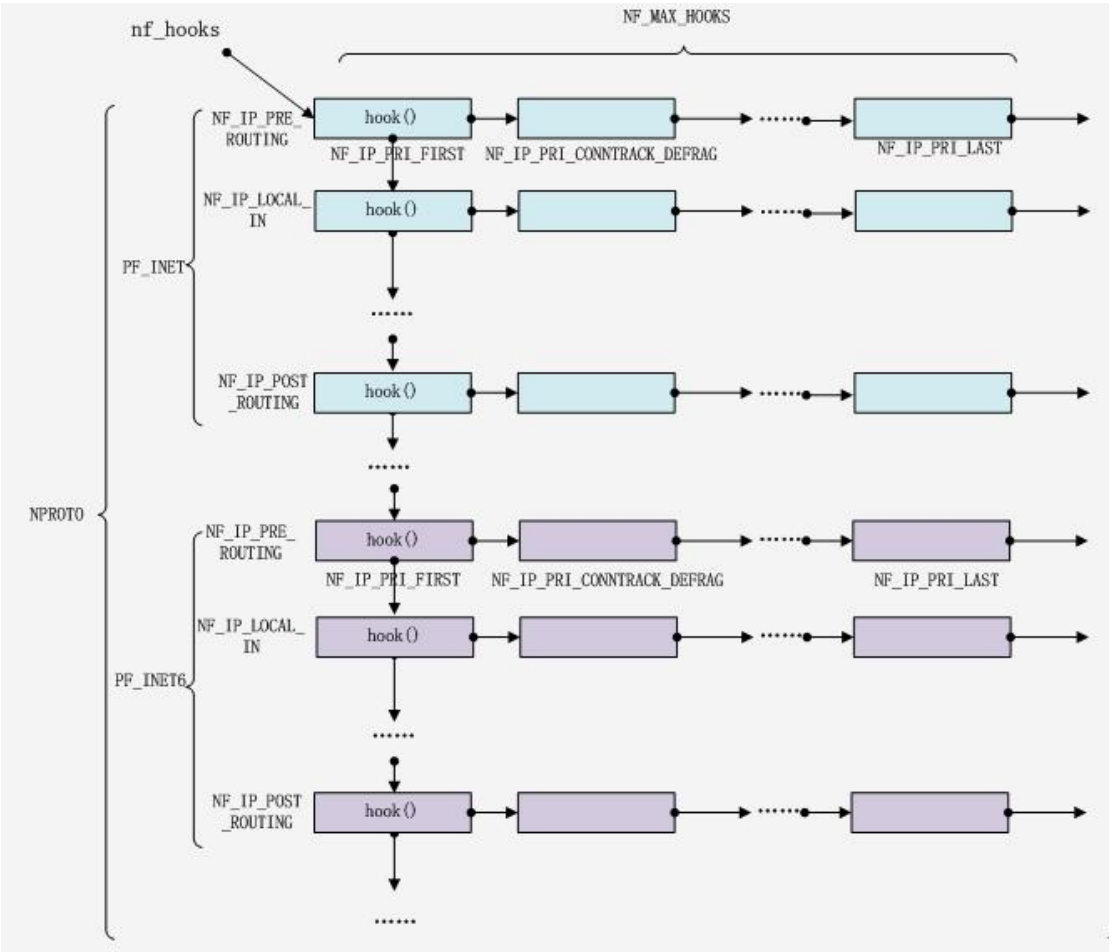

图 4-1 钩子函数的全局存储

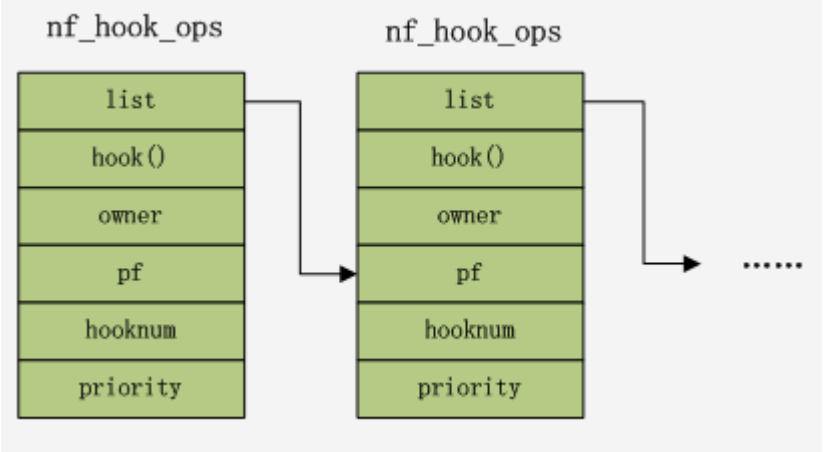

图 4-2 钩子函数的链表

Ø 钩子函数的管理机制

如果需要在相应的钩子点挂载钩子函数，则需要首先定义一个 nf_hook_ops 结构，在其中实现实际的钩子函数，再调用函数 nf_register_hook()将该钩子函数注册到图 4-1 所示的二维链表中，nf_register_hook()函数的定义如下：

```
int nf_register_hook(struct nf_hook_ops *reg)
{
        struct list_head *i;
        int err;

        err = mutex_lock_interruptible(&nf_hook_mutex);
        if (err < 0)
                return err;
        list_for_each(i, &nf_hooks[reg->pf][reg->hooknum]) {
                if (reg->priority < ((struct nf_hook_ops *)i)->priority)
                        break;
        }
        list_add_rcu(&reg->list, i->prev);
        mutex_unlock(&nf_hook_mutex);
        return 0;
}
```

5. Netfilter 的流程框架

在 Netfilter 中的不同钩子点调用了不同的钩子函数，这些钩子函数的调用如图 4-1 所示，其调用的流程框架如下图 5-1 所示。

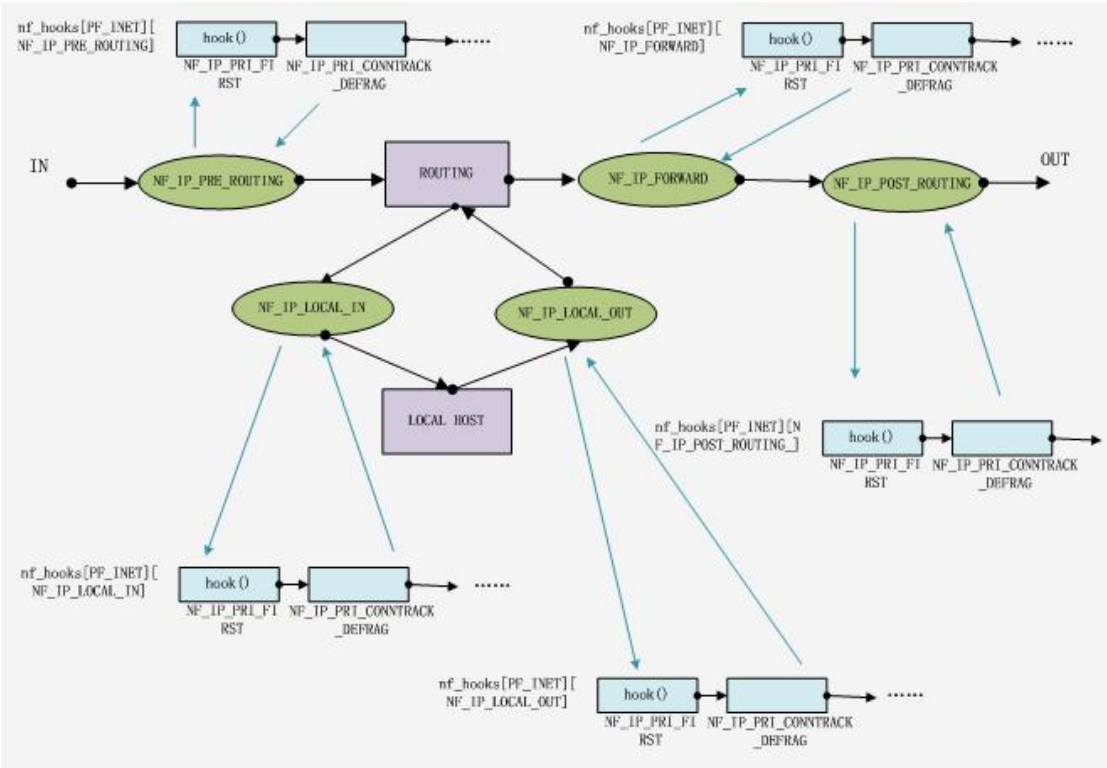

图 5-1 Netfilter 中 hook 函数的调用流程

Netfilter 中默认表 filter 在建立时则在 NF_IP_LOCAL_IN，NF_IP_FORWARD 钩子点注册了钩子函数 ipt_hook()，在 NF_IP_LOCAL_OUT 这个点注册了钩子函数 ipt_local_out_hook()，两个钩子函数都会调用 ipt_do_table()对相对应的表和钩子点的规则进行遍历。调用的流程如下图 5-2 所示。

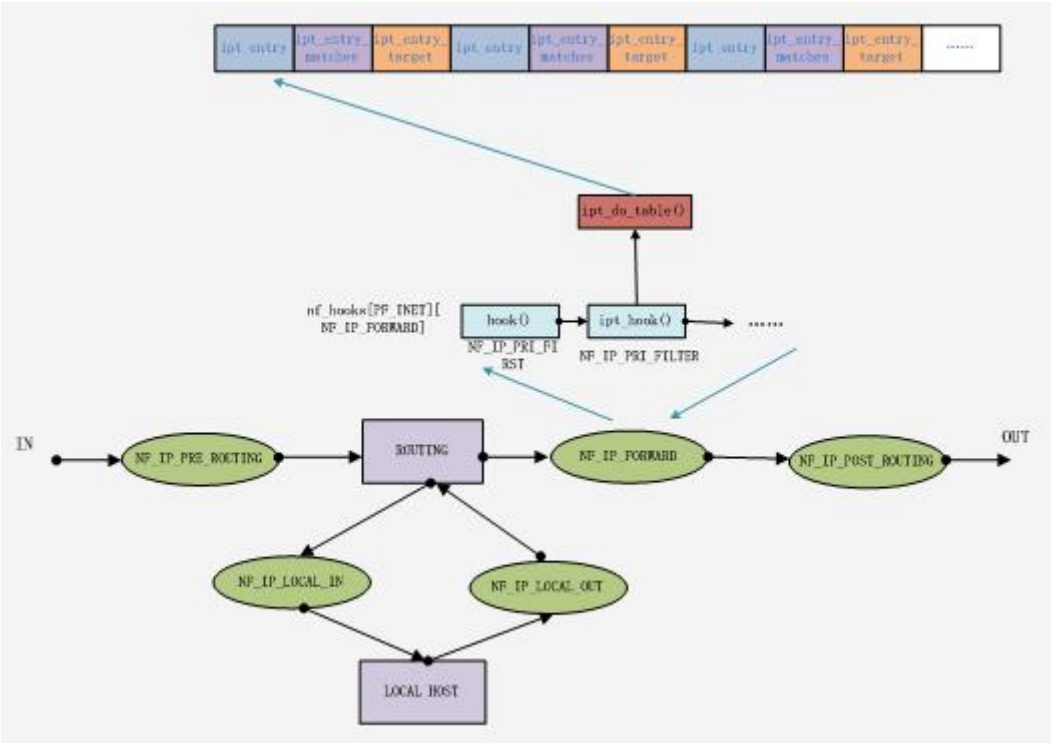

图 5-2 Netfilter 中规则的调用流程

6. 总结

以上只是简单分析了 Netfilter 的整体框架，主要描述了其中的机制，在这个机制上实现了很多功能，除了对基本的功能进行完善和改进外，还出现了很多新的扩展功能。如在此架构上实现的连接跟踪机制和 NAT 机制，以及结合连接跟踪机制与 Netfilter 框架实现的 Layer7 扩展匹配模块等。对此框架的了解，有助于我们更好的利用 Netfilter 框架实现我们的设计，鉴于自身水平有限，因此以上的分析不能保证全部正确。