

netfilter + nf\_conntrack + iptables

原著：pywj777

专注硬核知识分享和技术人一起涅槃重生

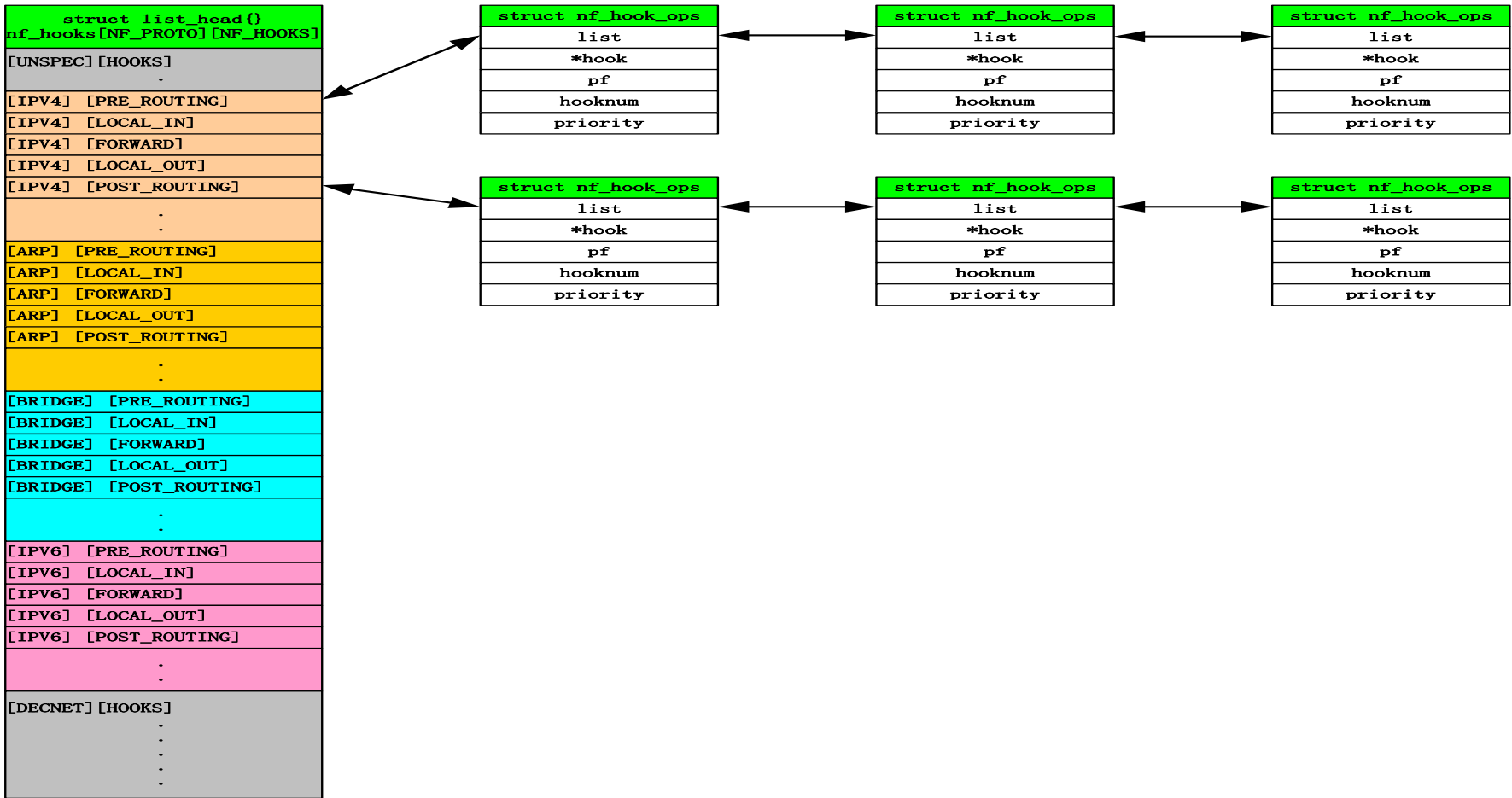
G E E K \_ C O D I N G

极客重生



netfilter

netfilter 的核心框架图



- 1 这个二维数组的每一项代表了一个钩子被调用的点，NF\_PROTO 代表协议栈，NF\_HOOK 代表协议栈中某个路径点。
- 2 所有模块都可以通过 nf\_register\_hook ()函数将一个钩子项挂入想被调用点的链表中（通过 protocol 和 hook 指定一个点）。这样，该钩子项就能够处理指定 protocol 中和指定 hook 点流经的所有数据包。
- 3 netfilter 在不同协议栈的不同点上(例如 arp\_rcv()、ip\_rcv()、ip6\_rcv()、br\_forward()等)放置 NF\_HOOK()函数,当数据包经过了某个协议栈(NF\_PROTO)的某个点(NF\_HOOK)时，该协议栈会通过 NF\_HOOK()函数调用对应钩子链表（nf\_hooks[NF\_PROTO][NF\_HOOK]）中注册的每一个钩子项来处理该数据包。如上一章《IPv4 接收与转发协议栈流程图》中 IPv4 协议处理函数调用的 HOOK 点。

netfilter 提供的全局资源及相应的锁

- 1 nf\_hooks[][]数组链表
  - 1.1 它的定义是 struct list\_head nf\_hooks[NFPROTO\_NUMPROTO][NF\_MAX\_HOOKS] \_\_read\_mostly; 用户通过 nf\_register\_hook()和 nf\_unregister\_hook()在这个全局链表中添加或删除 HOOK 点。并且在协议栈中会通过 NF\_HOOK()->nf\_hook\_slow()来调用这些 hook 点。
  - 1.2 读者与写者之间通过 list\_add\_rcu()、list\_del\_rcu()和 synchronize\_net()、list\_for\_each\_continue\_rcu()来保证数据的一致性。写者与写者之间通过 DEFINE\_MUTEX(nf\_hook\_mutex)信号量来保证多个写者之间的互斥。
- 2 \*nf\_afinfo[]指针数组
  - 2.1 它的定义是 const struct nf\_afinfo \*nf\_afinfo[NFPROTO\_NUMPROTO] \_\_read\_mostly; 用户通过 nf\_register\_afinfo()和 nf\_unregister\_afinfo 在这个指针数组中添加和删除指针项。并且通过 nf\_get\_afinfo()引用这些指针项。
  - 2.2 读者与写者之间通过 rcu\_assign\_pointer()和 synchronize\_rcu()、rcu\_dereference()来保证数据一致性。写者与写者之间通过 DEFINE\_MUTEX(afinfo\_mutex)信号量来保证多个写者之间的互斥。
- 3 nf\_register\_hook(struct nf\_hook\_ops \*reg)、nf\_unregister\_hook(struct nf\_hook\_ops \*reg) 用于在 nf\_hooks[][]数组的指定位置挂载一个钩子项，用于在指定协议栈相应位置处理数据包。
- 4 nf\_register\_hooks(struct nf\_hook\_ops \*reg, unsigned int n)、nf\_unregister\_hooks(struct nf\_hook\_ops \*reg, unsigned int n) 用于在 nf\_hooks[][]数组的指定位置挂载一组钩子项，用于在指定协议栈相应位置处理数据包。

netfilter 为每个钩子函数提供返回值

- NF\_DROP（0） 数据包被丢弃。即不被下一个钩子函数处理，同时也不再被协议栈处理，并释放掉该数据包。协议栈将处理下一个数据包。
- NF\_ACCEPT（1） 数据包允许通过。即交给下一个钩子函数处理、或交给协议栈继续处理（okfn()）。
- NF\_STOLEN（2） 数据包被停止处理。即不被下一个钩子函数处理，同时也不被协议栈处理，但也不释放数据包。协议栈将处理下一个数据包。
- NF\_QUEUE（3） 将数据包交给 nf\_queue 子系统处理。即不被下一个钩子函数处理，同时也不被协议栈处理，但也不释放数据包。协议栈将处理下一个数据包。
- NF\_REPEAT（4） 数据包将被该返回值的钩子函数再次处理一遍。
- NF\_STOP（5） 数据包停止被该 HOOK 点的后续钩子函数处理，并交给协议栈继续处理（okfn()）

nf\_queue 子功能

- 1 nf\_queue 是 netfilter 的一个子功能，当某个钩子函数的返回值 NF\_QUEUE 时，netfilter 就会调用 nf\_queue()函数将数据包交给 nf\_queue 子功能进行处理。
- 2 nf\_queue 的功能，就是调用对该协议数据包感兴趣的函数处理该数据包。然后由该函数决定该数据包的后续处理。
- 3 nf\_queue 提供的 API

3.1 nf\_register\_queue\_handler(u\_int8\_t pf, const struct nf\_queue\_handler \*qh) 注册一个 nf\_queue\_handler 项。参数 pf 与 netfilter 的 NF\_PROTO 相对应，表示协议栈值。参数 qh 是用来处理 pf 指定协议栈中的数据包。该函数就是将 qh 项挂载到 queue\_handler[NFPROTO\_NUMPROTO]指针数组中，位置由 pf 指定。

3.2 nf\_unregister\_queue\_handler(u\_int8\_t pf, const struct nf\_queue\_handler \*qh) 注销一个 nf\_queue\_handler 项。该项由参数 pf 和 qh 指定。

3.3 nf\_unregister\_queue\_handlers(const struct nf\_queue\_handler \*qh) 注销一个 nf\_queue\_handler 项。该项由参数 qh 指定。

3.4 nf\_queue(struct sk\_buff \*skb, u\_int8\_t pf, ...) 由 netfilter 调用，当钩子函数返回 NF\_QUEUE 结果时，netfilter 将数据包交给该函数进行处理。nf\_queue()首先根据参数信息，构建一个 nf\_queue\_entry 结构数据（它包含 skb、输入设备、输出设备、HOOK 点、下一个协议栈处理函数等信息）。其次根据参数 pf 值（该数据包所在的协议栈）在 queue\_handler[NFPROTO\_NUMPROTO]指针数组中找到对应的 nf\_queue\_handler 项（通过 nf\_register\_queue\_handler()函数注册的）。最后将构建好的 nf\_queue\_entry 结构数据交给该注册项指定的函数进行处理。

3.5 nf\_reinject(struct nf\_queue\_entry \*entry, unsigned int verdict) 用于对 entry 中数据包做进一步处理。参数 entry 是上面 nf\_queue()构建的数据（它被传递给注册项做进一步处理），参数 verdict 决定了对该 entry 中数据包如何处理。
- 4 ip\_queue 利用了 nf\_queue 子功能将数据包传递给应用层处理

4.1 它调用 nf\_register\_queue\_handler(NFPROTO\_IPV4, &nfqh)注册一个 nf\_queue\_handler 项 nfqh，用来处理 IPV4 协议栈中的数据包。

4.2 当 IPV4 协议栈的某个钩子函数返回 NF\_QUEUE 结果，netfilter 将调用 nf\_queue()将该数据包交给之前注册的 nfqh 项处理。

4.3 nfqh 指定的函数是 ipq\_enqueue\_packet()，它将数据包通过 netlink 接口传递给应用层处理。（仅仅是将数据包挂到对应 socket 队列中，然后返回继续处理下一个数据包）

4.4 当用户处理完数据包并通过 netlink 接口发送对该数据包的处理结果，ip\_queue 会调用 nf\_reinject()按照用户指定的结果对数据包进行处理。

nf\_log 子功能

- 5 nf\_log 是 netfilter 的一个子功能，它提供了一个调用接口函数 nf\_log\_packet(pf, ...)，它主要调用 pf 协议指定的函数来记录日志。
- 6 nf\_log 提供的 API

6.1 nf\_log\_register(u\_int8\_t pf, struct nf\_logger \*logger) 注册一个 nf\_logger 项。参数 pf 与 netfilter 的 NF\_PROTO 相对应，表示协议栈值。参数 logger 是用来处理 pf 指定协议栈中的记录信息。该函数就是将 logger 项挂载到\*nf\_loggers[NFPROTO\_NUMPROTO]指针数组中，位置由 pf 指定。

6.2 nf\_log\_unregister(struct nf\_logger \*logger) 注销一个 nf\_logger 项。

6.3 nf\_unregister\_queue\_handlers(const struct nf\_queue\_handler \*qh) 注销一个 nf\_queue\_handler 项。该项由参数 qh 指定。

6.4 nf\_log\_packet(pf, ...) 该函数可被协议栈中任何函数调用，用于记录日志信息。它主要调用 pf 协议指定的注册函数来记录日志。
- 7 nfnetlink\_log 在 nf\_log 子系统中注册对应协议的日志记录函数来记录日志，这些函数通过 nfnetlink 接口将日志传递给应用层。

nfnetlink 通信接口

- 1 nfnetlink 是建立在 netlink 基础上的一个与应用层进行通信的接口。它采用了 netlink attributes 接口，该接口使用 TLV<Type, Length, Value>（类型，长度，值）三元组来描述在传输中每一个数据单元，保证了发送方和接收方都以同样的方式来解释传输的数据。这样该接口就可以传输任何数据了。
- 2 nfnetlink 利用 netlink 接口创建一个类型为 NETLINK\_NETFILTER 的内核 socket 接口。它接收应用层传输数据的接口是 nfnetlink\_rcv()，nfnetlink\_rcv()根据 nlmsg\_hdr->nlmsg\_type 值决定交给某个子系统处理。
- 3 nfnetlink 提供的 API

3.1 nfnetlink\_subsys\_register(const struct nfnetlink\_subsystem \*n) 在 nfnetlink 上注册一个子系统，该子系统被注册到 subsys\_table[]数组中，位置由 n->subsys\_id 指定。

3.2 nfnetlink\_subsys\_unregister(const struct nfnetlink\_subsystem \*n) 在 nfnetlink 上注销一个子系统。

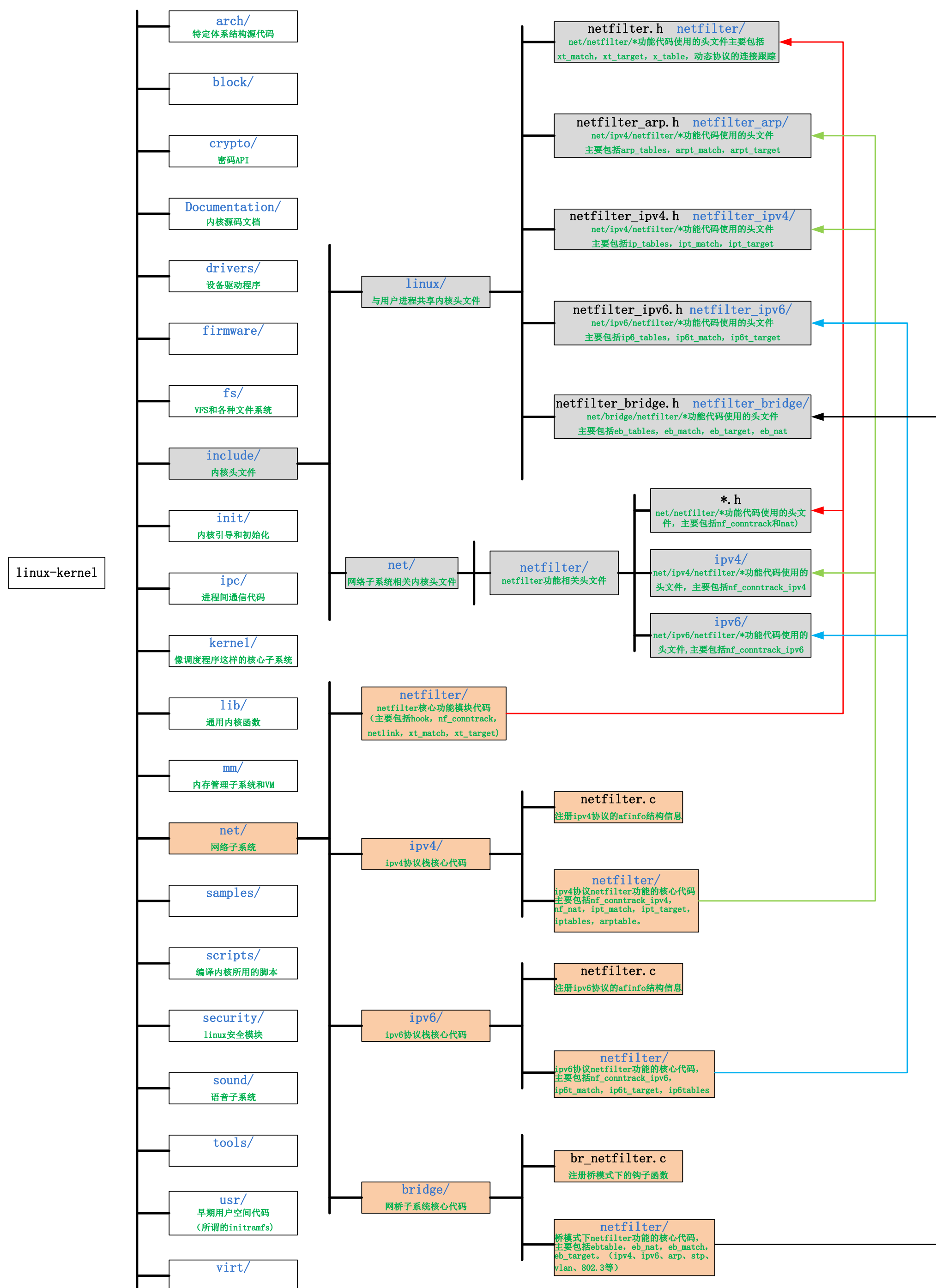
3.3 nfnetlink\_get\_subsys(u\_int16\_t type) 获得一个由 type 指定的子系统，type 值就是 nlmsg\_hdr->nlmsg\_type。

3.4 nfnetlink\_find\_client(u\_int16\_t type, const struct nfnetlink\_subsystem \*ss) 获得 ss 指定的子系统中的某个功能，有 type 指定，type 值就是 nlmsg\_hdr->nlmsg\_type。

3.5 nfnetlink\_unicast(struct sk\_buff \*skb, struct net \*net, u\_int32\_t pid, int flags) 向应用层发送信息，它其实就是直接调用 netlink 的发送函数 netlink\_unicast()。

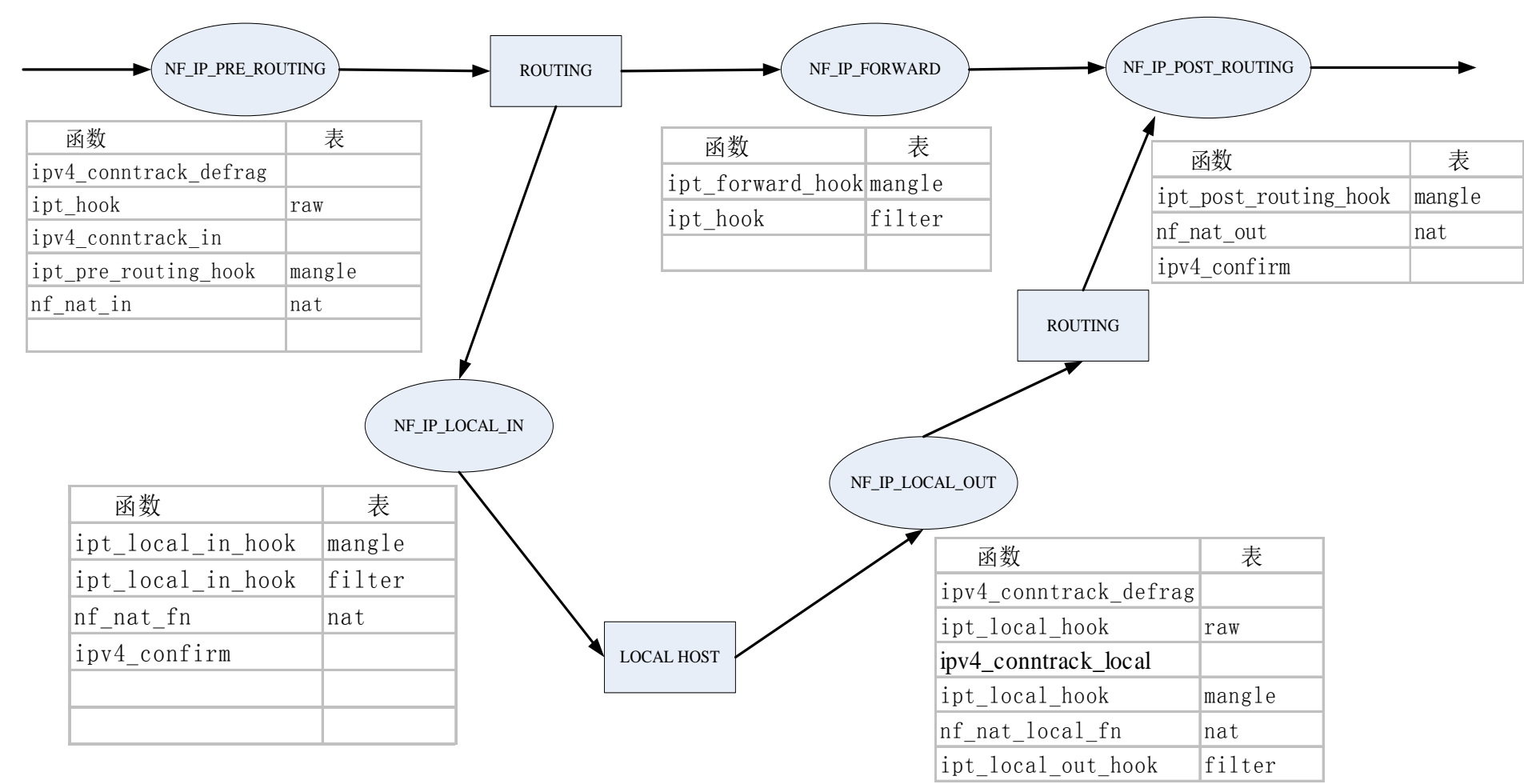
3.6 nfnetlink\_send(struct sk\_buff \*skb, struct net \*net, u32 pid, unsigned group, int echo, gfp\_t flags) 向应用层发送一个通知，它其实就是直接调用 netlink 的通知函数 nlmsg\_notify()。
- 4 nfnetlink\_queue 模块就是在 nfnetlink 接口上注册的一个子系统，功能类似于 ip\_queue，但它使用 nfnetlink 接口与应用层进行通信，并且使用 netlink attributes 接口对数据进行封装和解析。它同时接收应用层的命令，用于在 nf\_queue 子功能中注册对应协议的处理项。
- 5 nfnetlink\_log 模块就是在 nfnetlink 接口上注册的一个子系统，它使用 nfnetlink 接口与应用层进行通信，并且使用 netlink attributes 接口对数据进行封装和解析。它同时接收应用层的命令，用于在 nf\_log 子功能中注册对应协议的处理项。

## netfilter 相关功能在内核中文件分布图



基于 netfilter 的链接跟踪、NAT、包过滤规则、NF-hipac、nf\_queue

在 IPv4 协议栈中 netfilter 挂的 HOOK 函数



## 连接跟踪 nf\_conntrack

### nf\_conntrack 提供的资源

1 与连接跟踪相关的全局资源放在了 net->ct 网络命名空间中，它的类型是 struct netns\_ct。

```
struct netns_ct {
    atomic_t          count; /* 当前连接表中连接的个数 */
    unsigned int      expect_count; /* nf_conntrack_helper 创建的期待子连接 nf_conntrack_expect 项的个数 */
    unsigned int      htable_size; /* 存储连接（nf_conn）的 HASH 桶的大小 */
    struct kmem_cache *nf_conntrack_cachep; /* 指向用于分配 nf_conn 结构而建立的高速缓存（slab）对象 */
    struct hlist_nulls_head *hash; /* 指向存储连接（nf_conn）的 HASH 桶 */
    struct hlist_head *expect_hash; /* 指向存储期待子连接 nf_conntrack_expect 项的 HASH 桶 */
    struct hlist_nulls_head unconfirmed; /* 对于一个链接的第一个包，在 init_conntrack()函数中会将该包 original 方向的 tuple 结构挂入该链，这是因为在此时还不确定该链接会不会被后续的规则过滤掉，如果被过滤掉就没有必要挂入正式的链接跟踪表。在 ipv4_confirm()函数中，会将 unconfirmed 链中的 tuple 拆掉，然后再将 original 方向和 reply 方向的 tuple 挂入到正式的链接跟踪表中，即 init_net.ct.hash 中，这是因为到达 ipv4_confirm()函数时，应该在钩子 NF_IP_POST_ROUTING 处了，已经通过了前面的 filter 表。通过 cat /proc/net/nf_conntrack 显示连接，是不会显示该链中的连接的。但总的连接个数(net->ct.count)包含该链中的连接。当注销 l3proto、l4proto、helper、nat 等资源或在应用层删除所有连接（conntrack -F）时，除了释放 confirmed 连接（在 net->ct.hash 中的连接）的资源，还要释放 unconfirmed 连接（即在该链中的连接）的资源。*/

    struct hlist_nulls_head dying; /* 释放连接时，通告 DESTROY 事件失败的 ct 被放入该链中，并设置定时器，等待下次通告。通过 cat /proc/net/nf_conntrack 显示连接，是不会显示该链中的连接的。但总的连接个数(net->ct.count)包含该链中的连接。当注销连接跟踪模块时，同时要清除正再等待被释放的连接（即该链中的连接）*/

    struct ip_conntrack_stat__percpu *stat; /* 连接跟踪过程中的一些状态统计，每个 CPU 一项，目的是为了减少锁 */
    int sysctl_events; /* 是否开启连接事件通告功能 */
    unsigned int sysctl_events_retry_timeout; /* 通告失败后，重试通告的间隔时间，单位是秒 */
    int sysctl_acct; /* 是否开启每个连接数据包统计功能 */
    int sysctl_checksum;
    unsigned int sysctl_log_invalid; /* Log invalid packets */

#ifdef CONFIG_SYSCTL
    struct ctl_table_header *sysctl_header;
    struct ctl_table_header *acct_sysctl_header;
    struct ctl_table_header *event_sysctl_header;
#endif
    int hash_vmalloc; /* 存储连接（nf_conn）的 HASH 桶是否是使用 vmalloc()进行分配的 */
    int expect_vmalloc; /* 存储期待子连接 nf_conntrack_expect 项的 HASH 桶是否是使用 vmalloc()进行分配的 */
    char *slabname; /* 用于分配 nf_conn 结构而建立的高速缓存（slab）对象的名字 */
};
```

2 连接跟踪通过 nf\_conn 结构进行描述

```
struct nf_conn {
    /* Usage count in here is 1 for hash table/destruct timer, 1 per skb, plus 1 for any connection(s) we are `master' for */
    struct nf_conntrack ct_general; /* 连接跟踪的引用计数 */
    spinlock_t lock;
    /* These are my tuples; original and reply */
    struct nf_conntrack_tuple_hash tuplehash[IP_CT_DIR_MAX]; /* Connection tracking(链接跟踪)用来跟踪、记录每个链接的信息(目前仅支持 IP 协议的连接跟踪)。每个链接由“tuple”来唯一标识，这里的“tuple”对不同的协议会有不同的含义，例如对 tcp,udp 来说就是五元组: (源 IP, 源端口, 目的 IP, 目的端口, 协议号), 对 ICMP 协议来说是: (源 IP, 目的 IP, id, type, code), 其中 id,type 与 code 都是 icmp 协议的信息。链接跟踪是防火墙实现状态检测的基础，很多功能都需要借助链接跟踪才能实现，例如 NAT、快速转发、等等。*/

    /* Have we seen traffic both ways yet? (bitset) */
    unsigned long status; /* 可以设置由 enum ip_conntrack_status 中描述的状态 */
    /* If we were expected by an expectation, this will be it */
    struct nf_conn *master; /* 如果该连接是某个连接的子连接，则 master 指向它的主连接 */
    /* Timer function; drops refcnt when it goes off. */
    struct timer_list timeout;

#ifdef CONFIG_NF_CONNTRACK_MARK
    u_int32_t mark;
#endif
#ifdef CONFIG_NF_CONNTRACK_SECMARK
    u_int32_t secmark;
#endif
    /* Storage reserved for other modules: */
    union nf_conntrack_proto proto; /* 用于保存不同协议的私有数据 */
    /* Extensions */
    struct nf_ct_ext *ext; /* 用于扩展结构 */
};
```



```

#ifdef CONFIG_NET_NS
    struct net *ct_net;
#endif
};

```

- 3 连接跟踪可以设置的标志，即在 `ct->status` 中可以设置的标志，由下面的 `enum ip_conntrack_status` 描述，它们可以共存。这些标志设置后就不会再被清除。

```

enum ip_conntrack_status {
    /* It's an expected connection: bit 0 set.  This bit never changed */
    IPS_EXPECTED_BIT = 0,          /* 表示该连接是个子连接 */
    /* We've seen packets both ways: bit 1 set.  Can be set, not unset. */
    IPS_SEEN_REPLY_BIT = 1,        /* 表示该连接上双方向上都有数据包了 */
    /* Conntrack should never be early-expired. */
    IPS_ASSURED_BIT = 2,           /* TCP: 在三次握手建立完连接后即设定该标志。
                                   UDP: 如果在该连接上的两个方向都有数据包通过，则再有数据包在该连接上通过时，就设定该标志。
                                   ICMP: 不设置该标志 */
    /* Connection is confirmed: originating packet has left box */
    IPS_CONFIRMED_BIT = 3,         /* 表示该连接已被添加到 net->ct.hash 表中 */
    /* Connection needs src nat in orig dir.  This bit never changed. */
    IPS_SRC_NAT_BIT = 4,           /* 在 POSTROUTING 处，当替换 reply tuple 完成时，设置该标记 */
    /* Connection needs dst nat in orig dir.  This bit never changed. */
    IPS_DST_NAT_BIT = 5,           /* 在 PREROUTING 处，当替换 reply tuple 完成时，设置该标记 */
    /* Both together. */
    IPS_NAT_MASK = (IPS_DST_NAT | IPS_SRC_NAT),
    /* Connection needs TCP sequence adjusted. */
    IPS_SEQ_ADJUST_BIT = 6,
    /* NAT initialization bits. */
    IPS_SRC_NAT_DONE_BIT = 7,      /* 在 POSTROUTING 处，已被 SNAT 处理，并被加入到 bysource 链中，设置该标记 */
    IPS_DST_NAT_DONE_BIT = 8,      /* 在 PREROUTING 处，已被 DNAT 处理，并被加入到 bysource 链中，设置该标记 */
    /* Both together */
    IPS_NAT_DONE_MASK = (IPS_DST_NAT_DONE | IPS_SRC_NAT_DONE),
    /* Connection is dying (removed from lists), can not be unset. */
    IPS_DYING_BIT = 9,             /* 表示该连接正在被释放，内核通过该标志保证正在被释放的 ct 不会被其它地方再次引用。有了这个标志，当某个连接要被删除时，即使它还在 net->ct.hash 中，也不会再次被引用。（但好像还是没有太大作用？？？） */
    /* Connection has fixed timeout. */
    IPS_FIXED_TIMEOUT_BIT = 10,    /* 固定连接超时时间，这将不根据状态修改连接超时时间。通过函数 nf_ct_refresh_acct()修改超时时间时检查该标志。但该标志在哪设置的？？？？ */
    /* Conntrack is a template */
    IPS_TEMPLATE_BIT = 11,         /* 由 CT target 进行设置（这个 target 只能用在 raw 表中，用于为数据包构建指定 ct，并打上该标志），用于表明这个 ct 是由 CT target 创建的 */
};

```

- 4 连接跟踪对数据包在用户空间可以表现的状态，由下面 `enum ip_conntrack_info` 表示，被设置在 `skb->nfctinfo` 中。

```

enum ip_conntrack_info {
    /* Part of an established connection (either direction). */
    IP_CT_ESTABLISHED (0),         /* 表示这个数据包对应的连接在两个方向都有数据包通过，并且这是 ORIGINAL 初始方向数据包（无论是 TCP、UDP、ICMP 数据包，只要在该连接的两个方向上已有数据包通过，就会将该连接设置为 IP_CT_ESTABLISHED 状态。不会根据协议中的标志位进行判断，例如 TCP 的 SYN 等）。但它表示不了这是第几个数据包，也说明不了这个 CT 是否是子连接。*/
    /* Like NEW, but related to an existing connection, or ICMP error (in either direction). */
    IP_CT_RELATED (1),             /* 表示这个数据包对应的连接还没有 REPLY 方向数据包，当前数据包是 ORIGINAL 方向数据包。并且这个连接关联一个已有的连接，是该已有连接的子连接，（即 status 标志中已经设置了 IPS_EXPECTED 标志，该标志在 init_conntrack() 函数中设置）。但无法判断是第几个数据包（不一定是第一个）*/
    /* Started a new connection to track (only IP_CT_DIR_ORIGINAL); may be a retransmission. */
    IP_CT_NEW (2),                 /* 表示这个数据包对应的连接还没有 REPLY 方向数据包，当前数据包是 ORIGINAL 方向数据包。该连接不是子连接，但无法判断是第几个数据包（不一定是第一个）*/
    /* >= this indicates reply direction */
    IP_CT_IS_REPLY (3),            /* 这个状态一般不单独使用，通常以下面两种方式使用 */
    IP_CT_ESTABLISHED + IP_CT_IS_REPLY (3), /* 表示这个数据包对应的连接在两个方向都有数据包通过，并且这是 REPLY 应答方向数据包。但它表示不了这是第几个数据包，也说明不了这个 CT 是否是子连接。*/
    IP_CT_RELATED + IP_CT_IS_REPLY (4),    /* 这个状态仅在 nf_conntrack_attach()函数中设置，用于本机返回 REJECT，例如返回一个 ICMP 目的不可达报文，或返回一个 reset 报文。它表示不了这是第几个数据包。*/
    /* Number of distinct IP_CT types (no NEW in reply dirn). */
    IP_CT_NUMBER = IP_CT_IS_REPLY * 2 - 1 (5) /* 可表示状态的总数 */
};

```

- 5 连接跟踪里使用了两个全局 `spin_lock` 锁（`nf_conntrack_lock`、`nf_nat_lock`）和一个局部 `spin_lock` 锁（`ct->lock`）

### 5.1 nf\_conntrack\_lock

5.1.1 ct 从 `ct_hash[]`表中添加/删除时使用该锁，在 `ct_hash` 表中查找 `ct` 时使用 RCU 锁。

5.1.2 ct 从 `unconfirmed` 链上添加/删除时使用该锁，在该 `unconfirmed` 链上的 `ct` 不需要查找。

5.1.3 ct 从 `dying` 链上添加/删除时使用该锁，在该 `dying` 链上的 `ct` 不需要查找。

- 5.1.4 ct 通过 expect 与 mct 关联时使用该锁，目的是防止 mct 被移动或删除。
- 5.1.5 expect 从 expect\_hash[]表中添加/删除/查找时使用该锁。因为 expect 与 ct 紧密关联，所以共用一把锁。expect 仅在初试化连接时被查找。

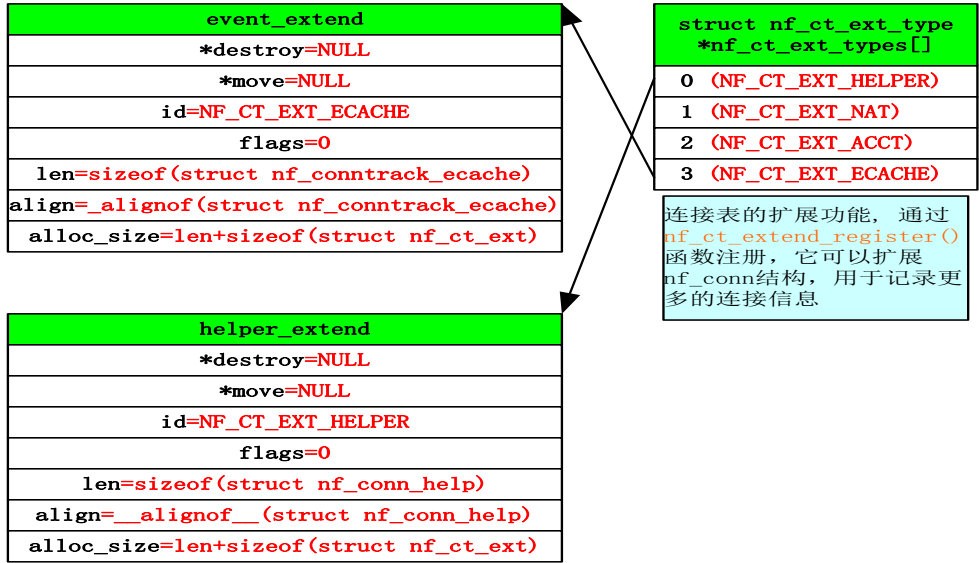
5.2 nf\_nat\_lock

- 5.2.1 ct 从 nat\_bysource[]中添加/删除/查找时使用该锁。nat\_bysource[]在初始化连接时被使用。
- 5.2.2 注册/注销 nf\_nat\_protos 协议时使用该锁。

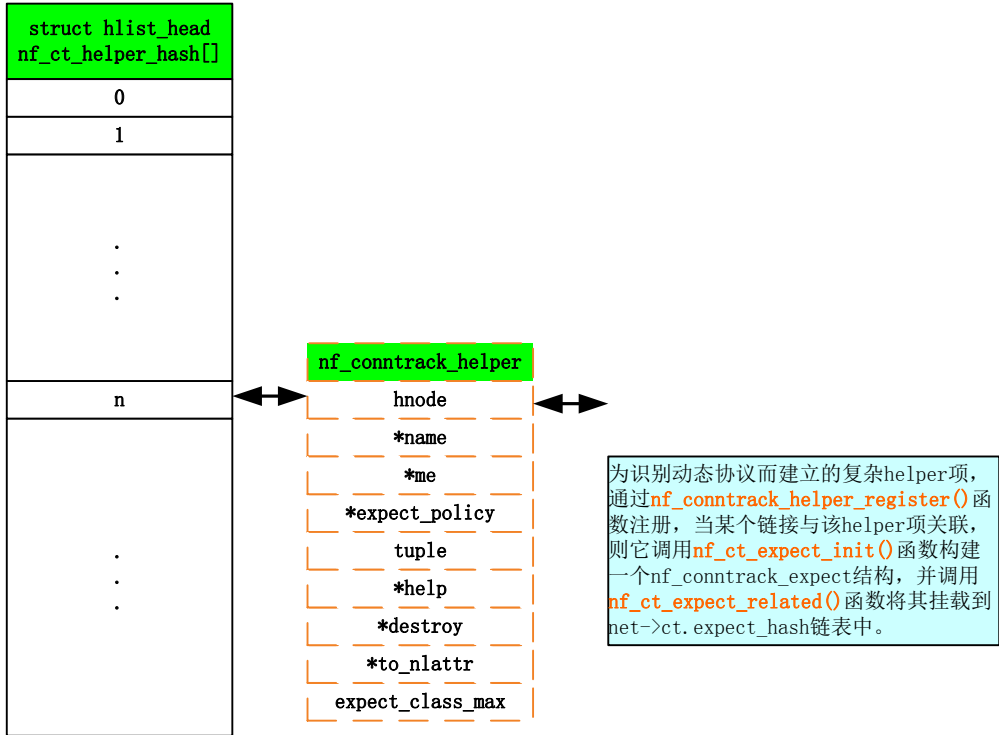
5.3 ct->lock

- 5.3.1 修改某个 ct 的数据时使用该 ct 自己的锁。

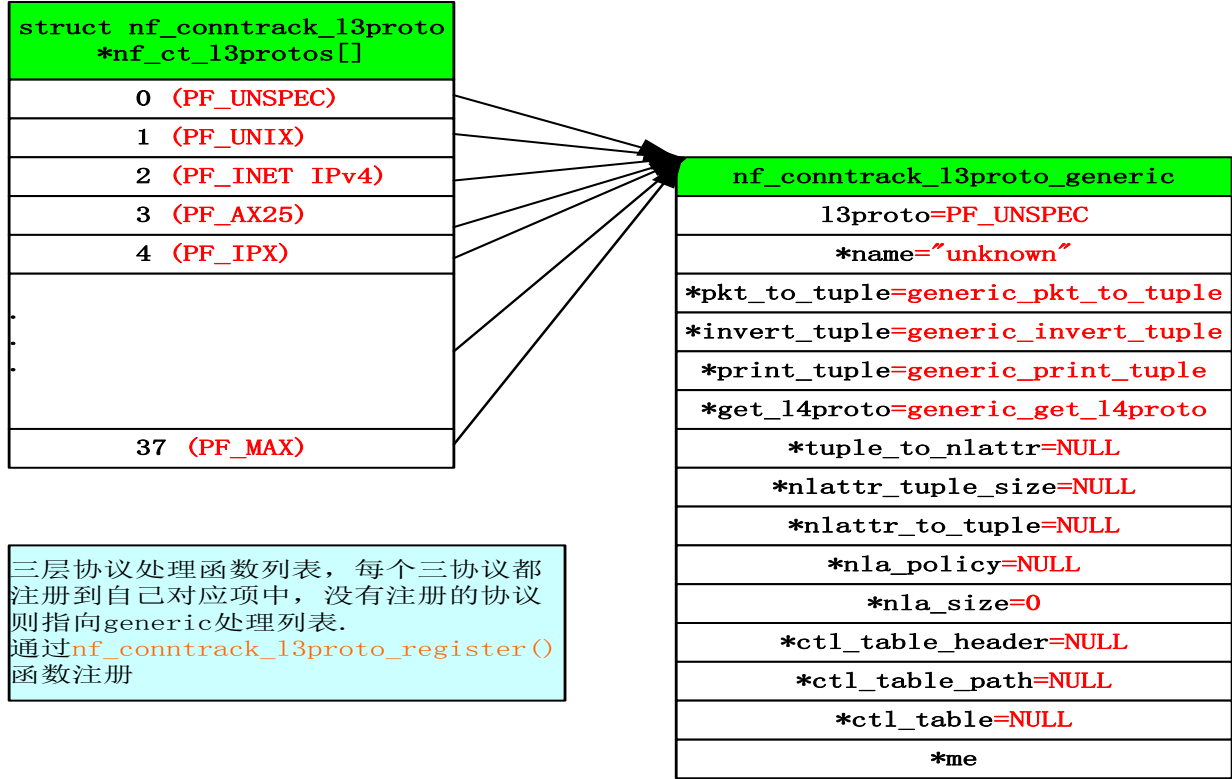
6 扩展连接跟踪结构（nf\_conn）利用 nf\_conntrack\_extend.c 文件中的 nf\_ct\_extend\_register(struct nf\_ct\_ext\_type \*type)和 nf\_ct\_extend\_unregister(struct nf\_ct\_ext\_type \*type)进行扩展，并修改连接跟踪相应代码来利用这部分扩展功能。



7 处理一个连接的子连接协议，利用 nf\_conntrack\_helper.c 文件中的 nf\_conntrack\_helper\_register(struct nf\_conntrack\_helper \*me)来注册 nf\_conntrack\_helper 结构，和 nf\_conntrack\_expect.c 文件中的 nf\_ct\_expect\_related\_report(struct nf\_conntrack\_expect \*expect, u32 pid, int report)来注册 nf\_conntrack\_expect 结构。

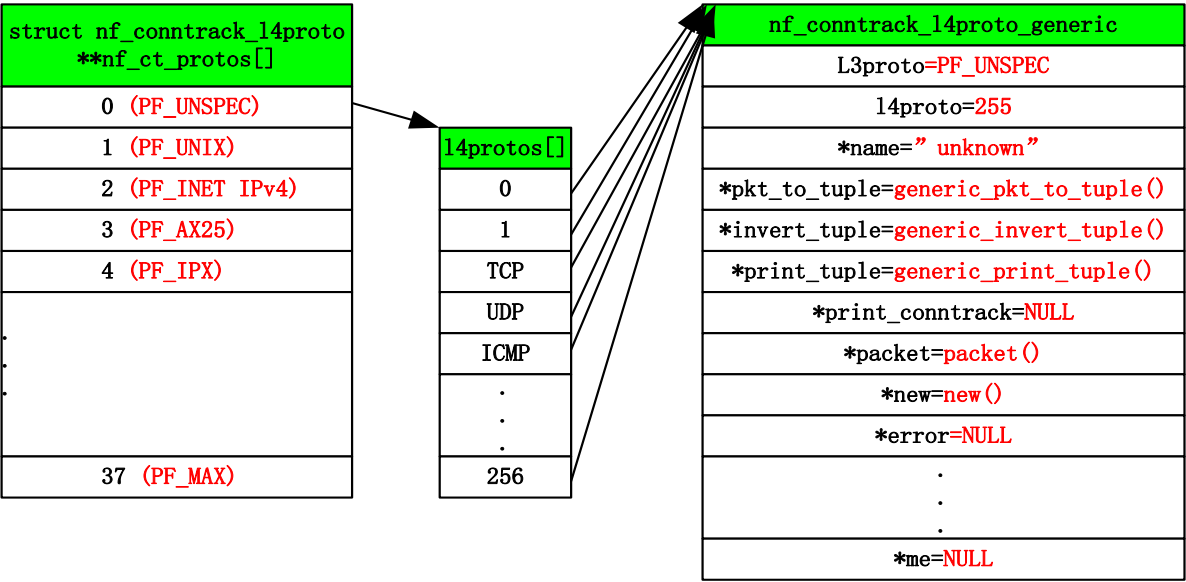


8 三层协议（IPv4/IPv6）利用 nf\_conntrack\_proto.c 文件中的 nf\_conntrack\_l3proto\_register(struct nf\_conntrack\_l3proto \*proto)和 nf\_conntrack\_l3proto\_unregister(struct nf\_conntrack\_l3proto \*proto)在 nf\_ct\_l3protos[]数组中注册自己的三层协议处理函数。





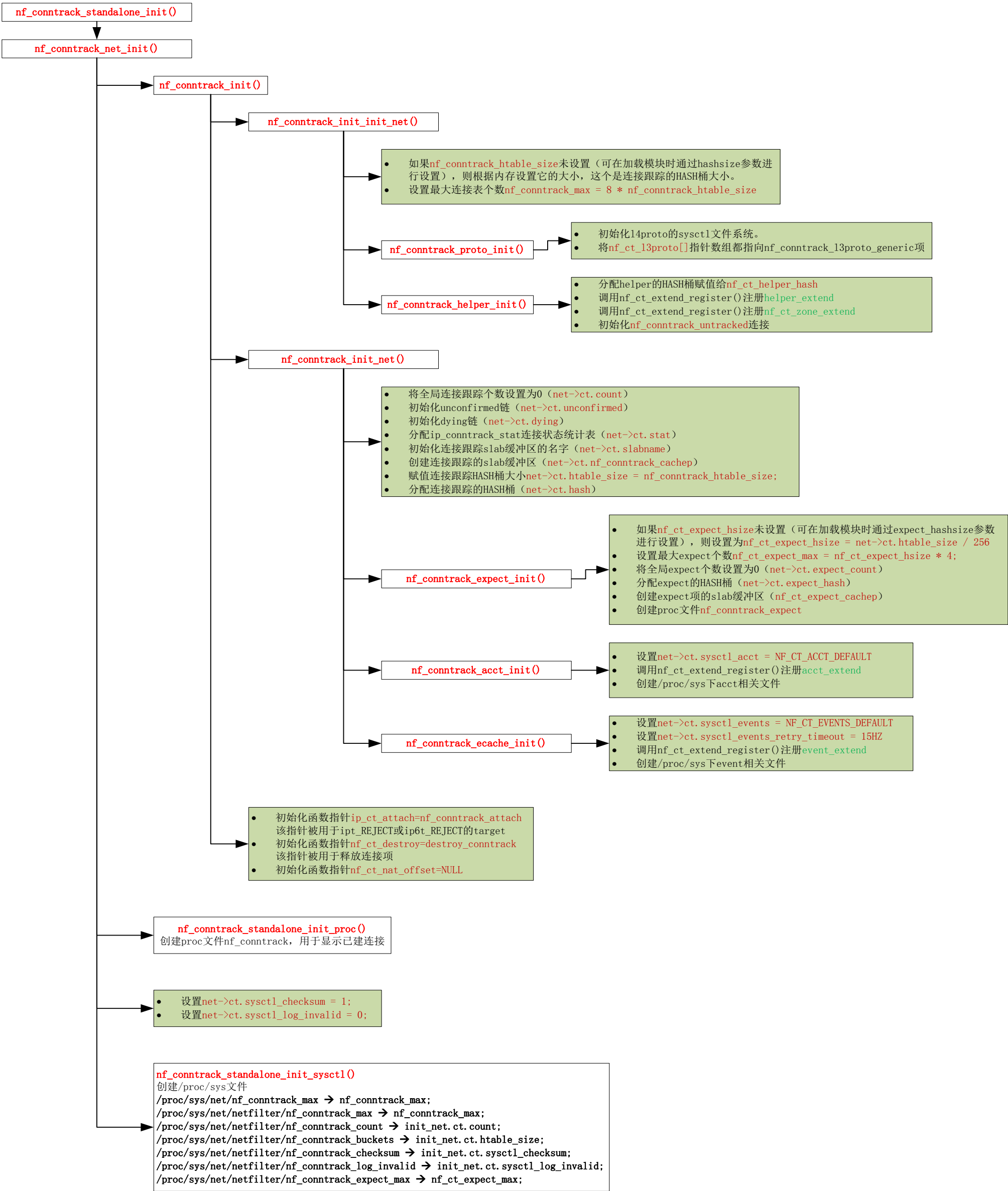
9 四层协议（TCP/UDP）利用 nf\_conntrack\_proto.c 文件中的 nf\_conntrack\_l4proto\_register(struct nf\_conntrack\_l4proto \*l4proto)和 nf\_conntrack\_l4proto\_unregister(struct nf\_conntrack\_l4proto \*l4proto)在 nf\_ct\_protos[]数组中注册自己的四层协议处理函数。



四层协议处理函数列表, 四层协议与三层协议相关联。每个四协议都注册到自己对应项中, 没有注册的协议则指向generic处理列表  
通过nf\_conntrack\_l4proto\_register() 函数注册

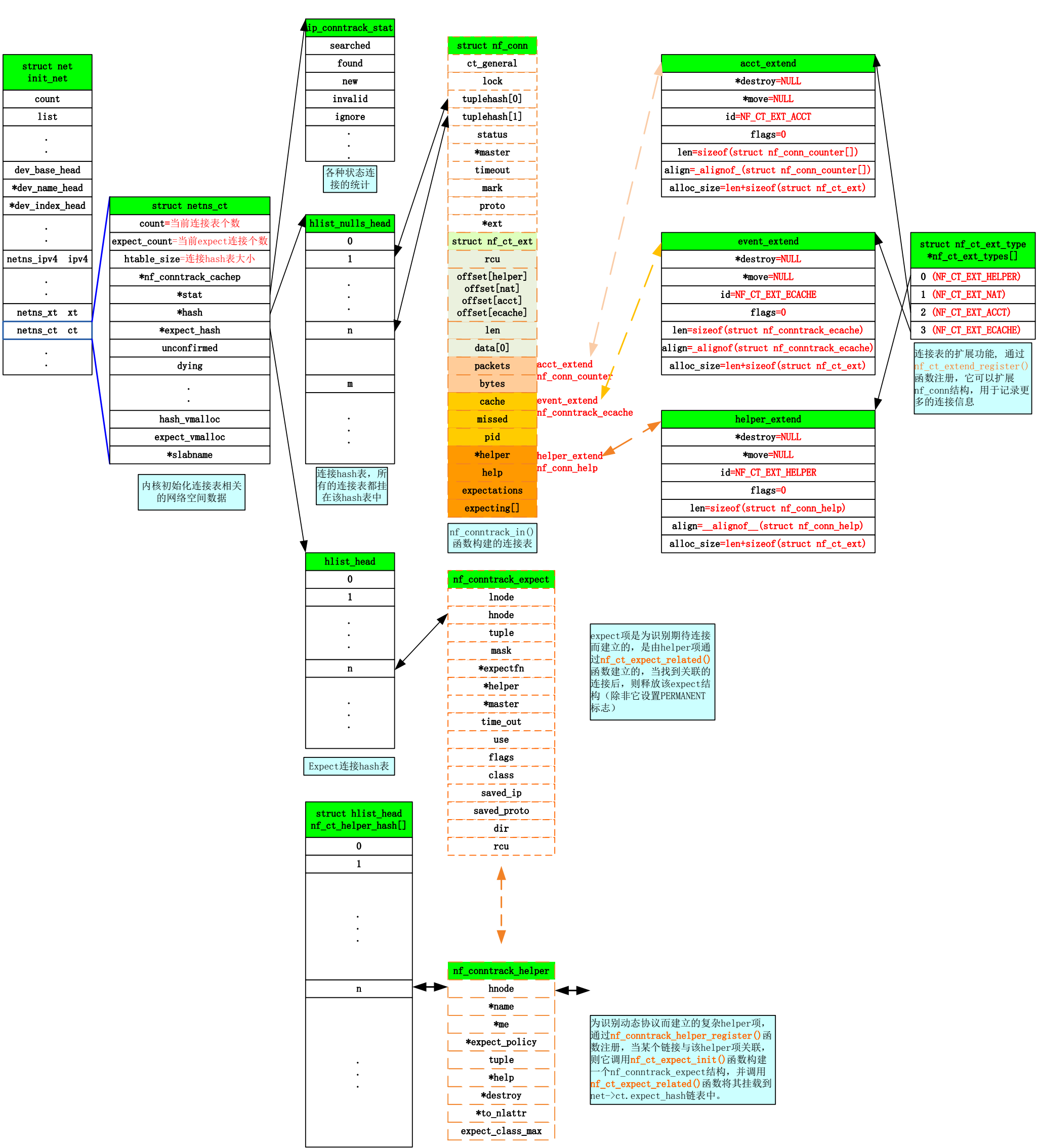
10 建立连接跟踪结构（nf\_conn）利用 nf\_conntrack\_core.c 文件中的 nf\_conntrack\_in()函数进行构建的。nf\_conntrack\_core.c 文件中还包括其它相应的处理函数。

nf\_contrack 的初始化

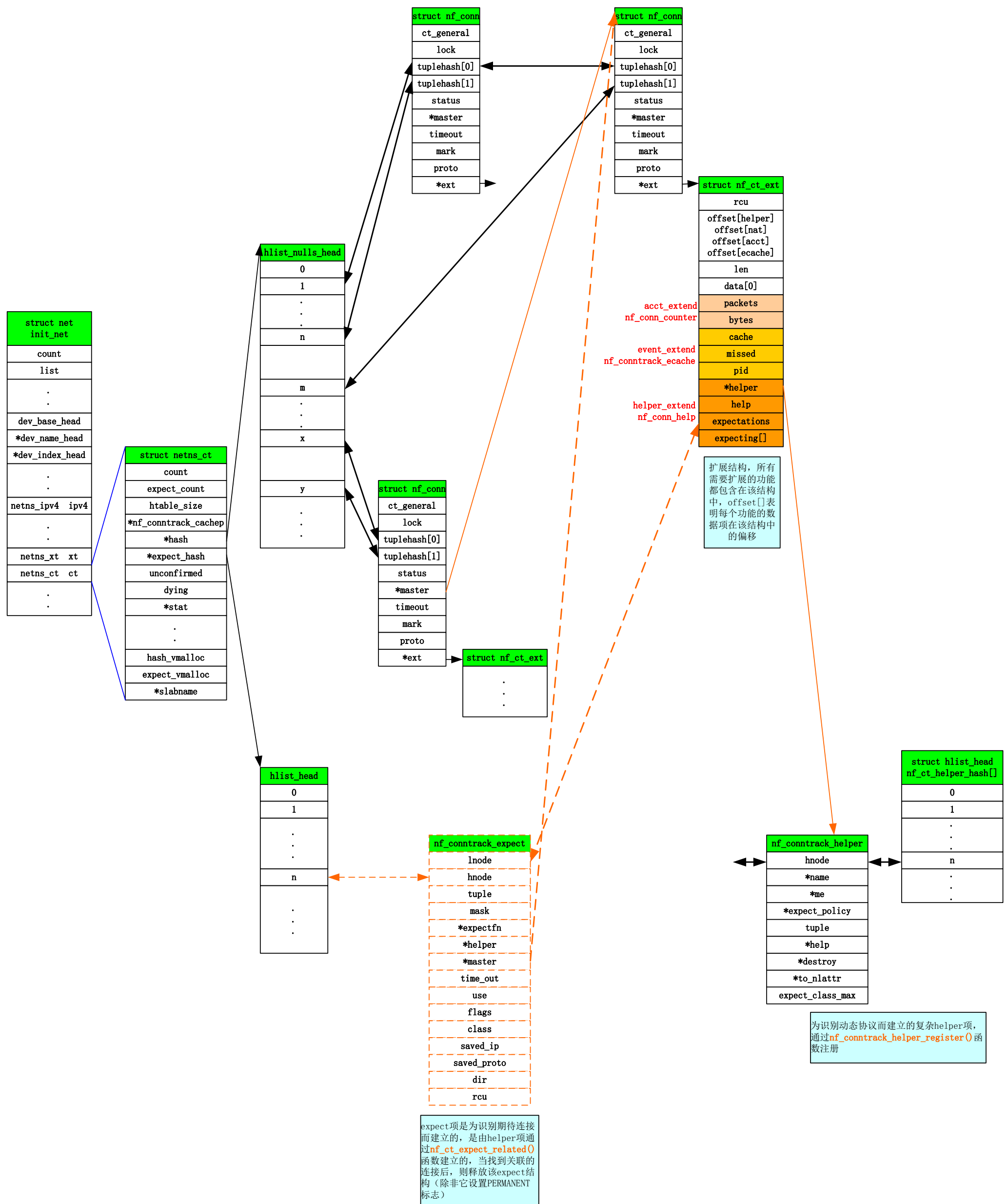


- 1 nf\_contrack 的初始化，就是初始化上面提到的这些资源，它在内核启动时调用 nf\_contrack\_standalone\_init()函数进行初始化的。
- 2 初始化完成后，构建出如下面连接表表现形式中的图所示，只是还没有创建 nf\_conn、nf\_contrack\_expect、nf\_contrack\_helper 项。
- 3 ct\_hash、expect\_hash、helper\_hash 这三个 HASH 桶大小在初始化时就已确定，后面不能再更改。其中 ct\_hash、expect\_hash 可在加载 nf\_contrack.ko 模块时通过参数 hashsize 和 expect\_hashsize 进行设定，而 helper\_hash 不能通过参数修改，它的默认值是 page/sizeof(helper\_hash)。
- 4 nf\_conn 和 nf\_contrack\_expect 都有最大个数限制.nf\_conn 通过全局变量 nf\_contrack\_max 限制,可通过/proc/sys/net/netfilter/nf\_contrack\_max 文件在运行时修改。nf\_contrack\_expect 通过全局变量 nf\_ct\_expect\_max 限制，可通过/proc/sys/net/netfilter/ nf\_contrack\_expect\_max 文件在运行时修改。nf\_contrack\_helper 没有最大数限制，因为这个是通过注册不同协议的模块添加的，大小取决于动态协议跟踪模块的多少，一般不会很大。

连接表的表现形式



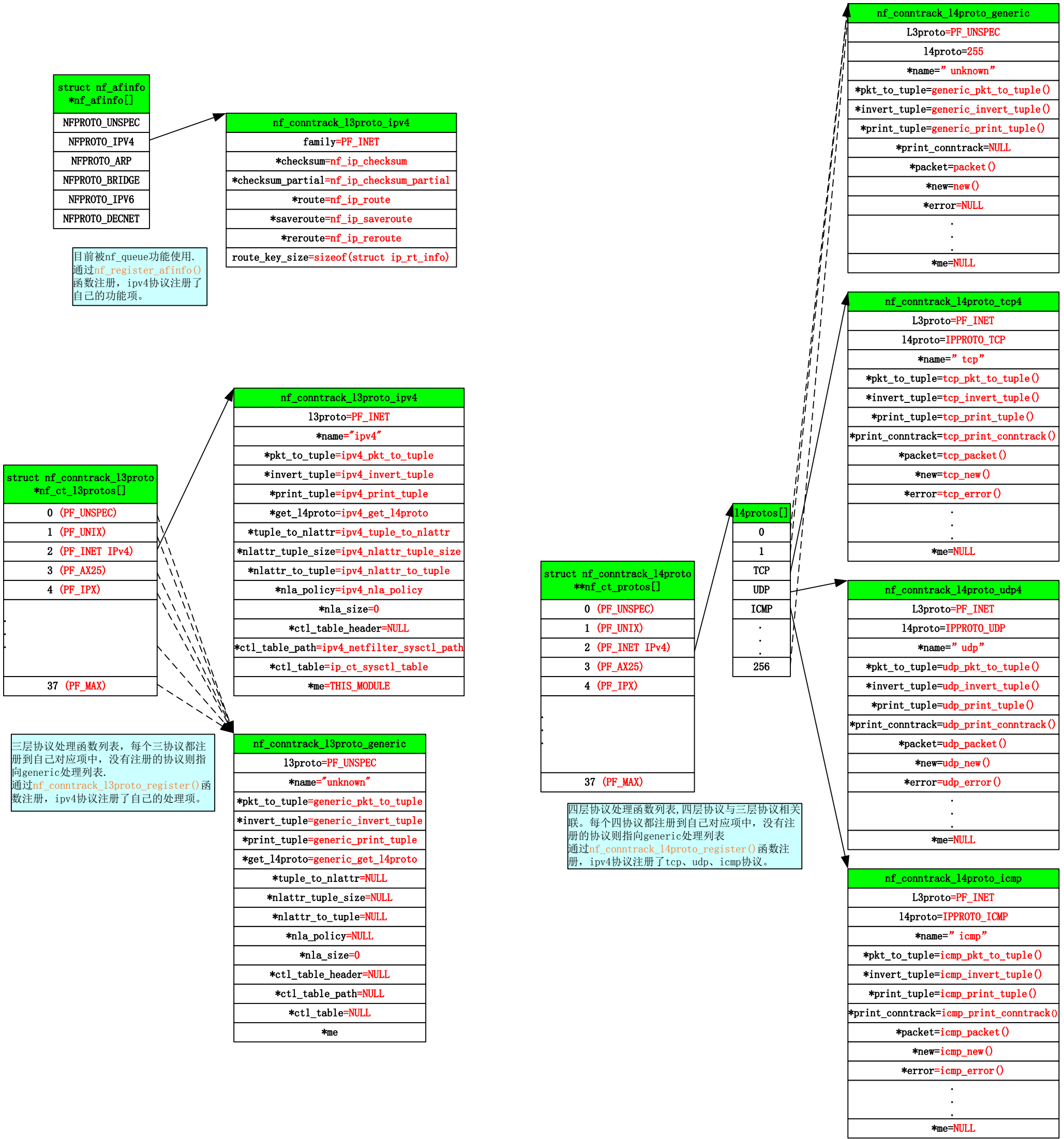
## 连接表和动态协议的表现形式



IPv4 利用 nf\_conntrack 进行链接跟踪

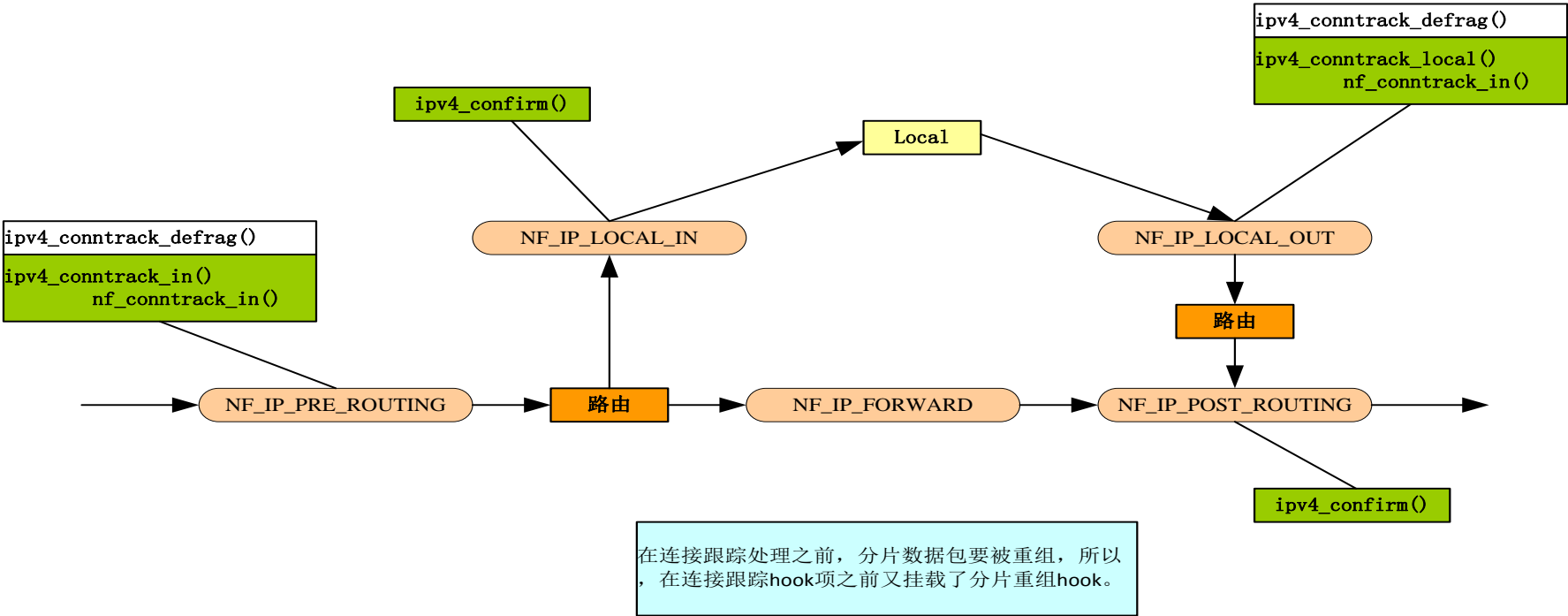
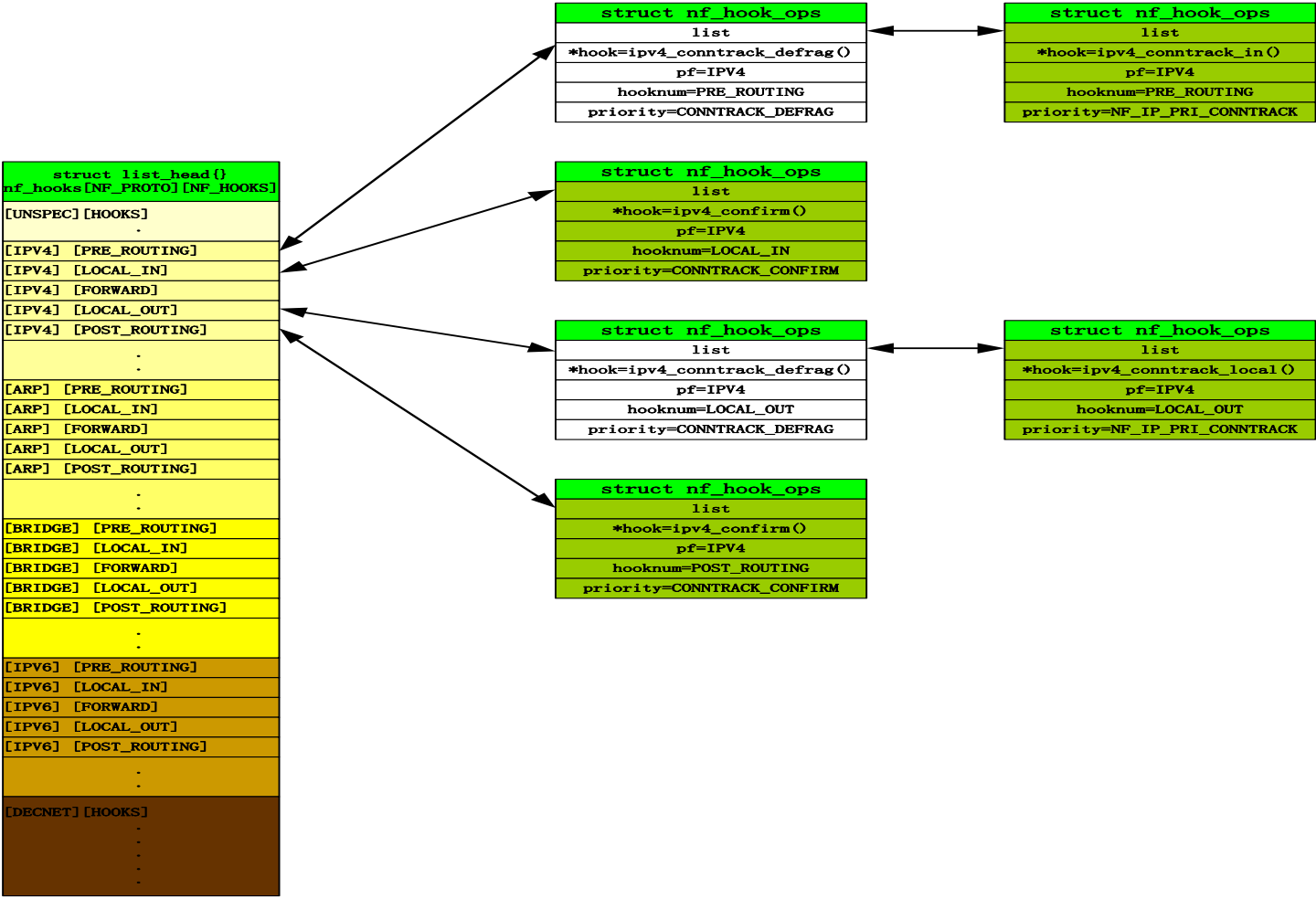
IPv4 连接跟踪的初始化

1    ipv4 协议注册了自己的 3 层协议 IPv4 协议，和 IPv4 相关的三个 4 层协议 TCP、UDP、ICMP。



2    在 netfilter 中利用 `nf_register_hook(struct nf_hook_ops *reg)`、`nf_unregister_hook(struct nf_hook_ops *reg)`函数注册自己的钩子项，调用 `nf_conntrack_in()`函数来建立相应连接。



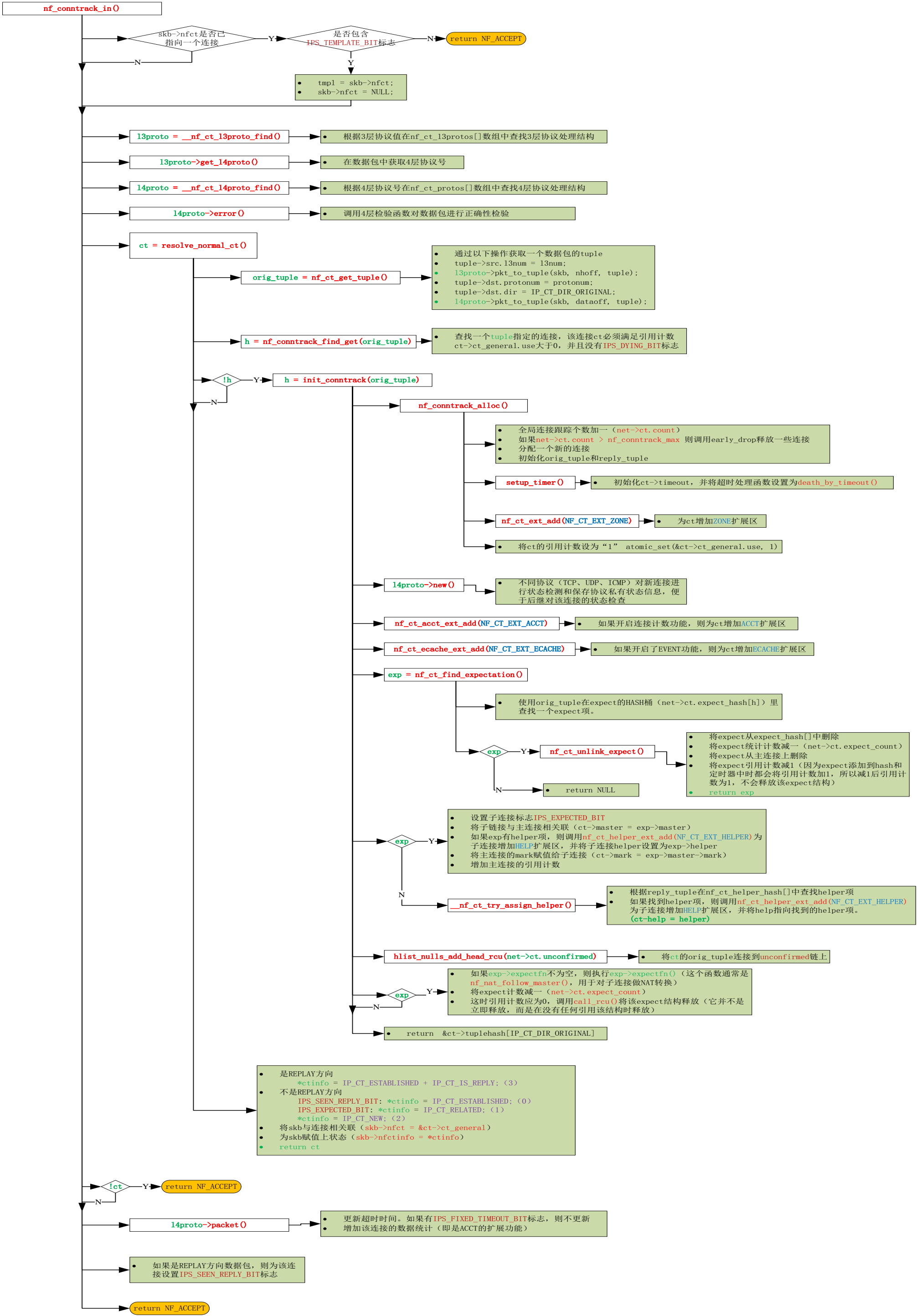


- 3 这样数据包就会经过 `ipv4` 注册的钩子项，并调用 `nf_contrack_in()`函数建立连接表项，连接表项中的 `tuple` 由 `ipv4` 注册的 3/4 层协议处理函数构建。
- 3.1 `ipv4_contrack_in()` 挂载在 `NF_IP_PRE_ROUTING` 点上。该函数主要功能是创建链接，即创建 `struct nf_conn` 结构，同时填充 `struct nf_conn` 中的一些必要的信息，例如链接状态、引用计数、helper 结构等。
- 3.2 `ipv4_confirm()` 挂载在 `NF_IP_POST_ROUTING` 和 `NF_IP_LOCAL_IN` 点上。该函数主要功能是确认一个链接。对于一个新链接，在 `ipv4_contrack_in()`函数中只是创建了 `struct nf_conn` 结构，但并没有将该结构挂载到链接跟踪的 Hash 表中，因为此时还不能确定该链接是否会被 `NF_IP_FORWARD` 点上的钩子函数过滤掉，所以将挂载到 Hash 表的工作放到了 `ipv4_confirm()`函数中。同时，子链接的 helper 功能也是在该函数中实现的。
- 3.3 `ipv4_contrack_local()` 挂载在 `NF_IP_LOCAL_OUT` 点上。该函数功能与 `ipv4_contrack_in()`函数基本相同，但其用来处理本机主动向外发起的链接。
- 4 `nf_contrack_ipv4_compat_init()` --> `register_pernet_subsys()` --> `ip_contrack_net_init()` 创建/proc 文件 `ip_contrack` 和 `ip_contrack_expect`

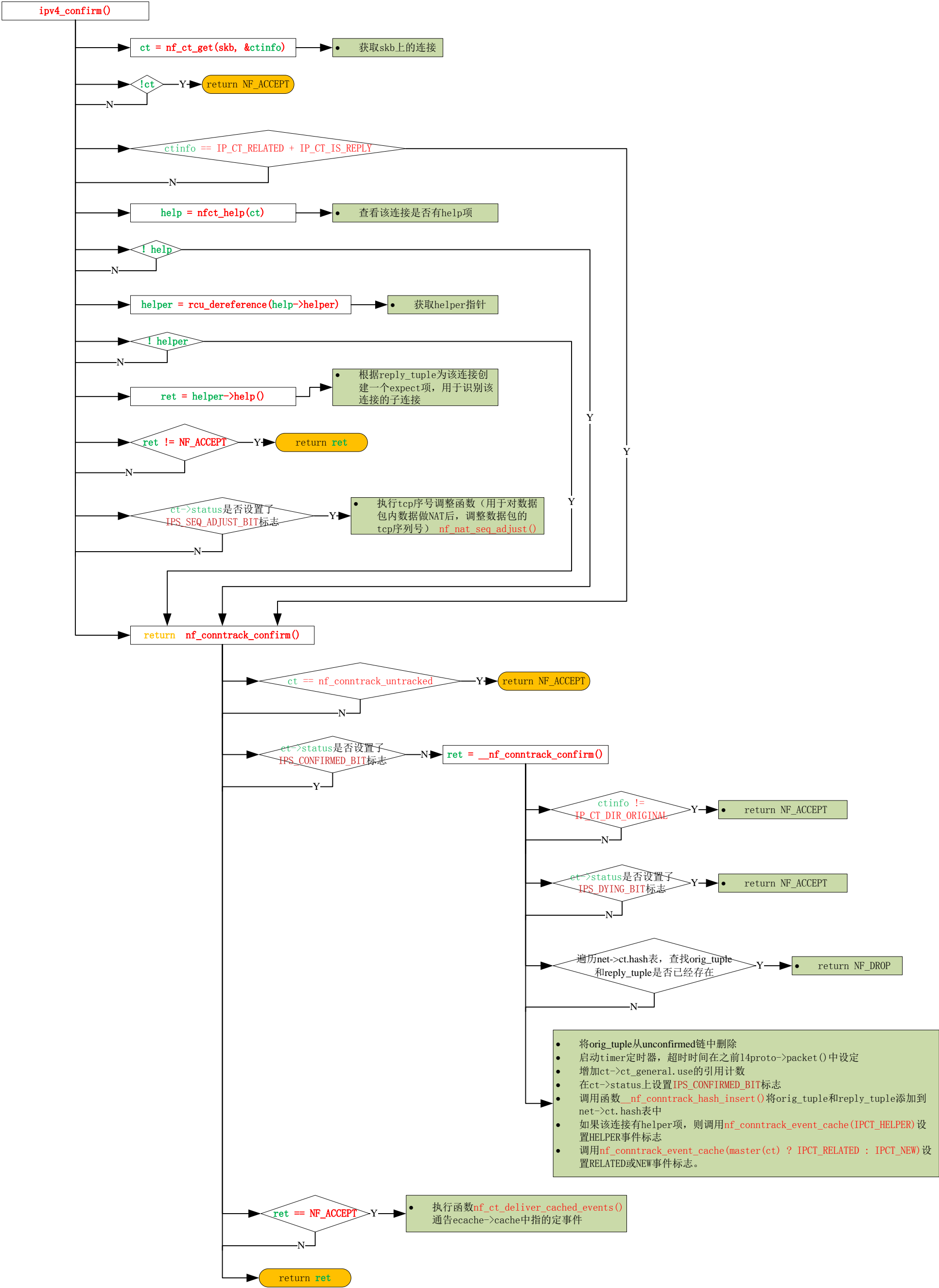
## IPv4 连接跟踪的处理流程

- 1 如上图所示，连链接跟踪主要由三个函数来完成，即 `ipv4_contrack_in()`，`ipv4_confirm()`与 `ipv4_contrack_local()`。其中 `ipv4_contrack_in()`与 `ipv4_contrack_local()` 都是通过调用函数 `nf_contrack_in()`来实现的，所以下面我们主要关注 `nf_contrack_in()`与 `ipv4_confirm()`这两个函数。`nf_contrack_in()`函数主要完成创建链接、添加链接的扩展结构(例如 `helper`, `acct` 结构)、设置链接状态等。`ipv4_confirm()`函数主要负责确认链接(即将链接挂入到正式的链接表中)、执行 `helper` 函数、启动链接超时定时器等。
- 2 另外还有一个定时器函数 `death_by_timeout()`，该函数负责链接到期时删除该链接。

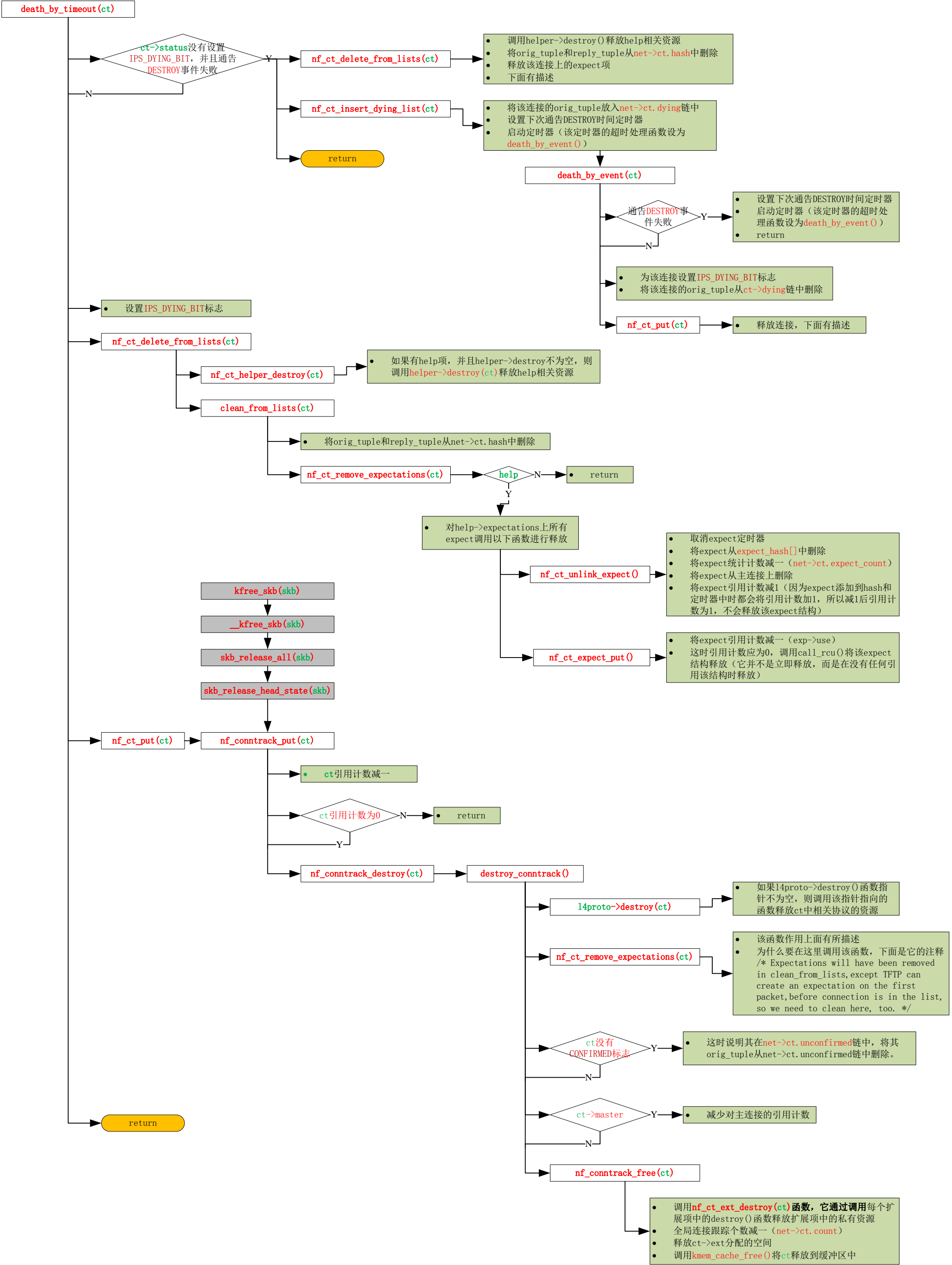
nf\_conntrack\_in



ipv4\_confirm



death\_by\_timeout

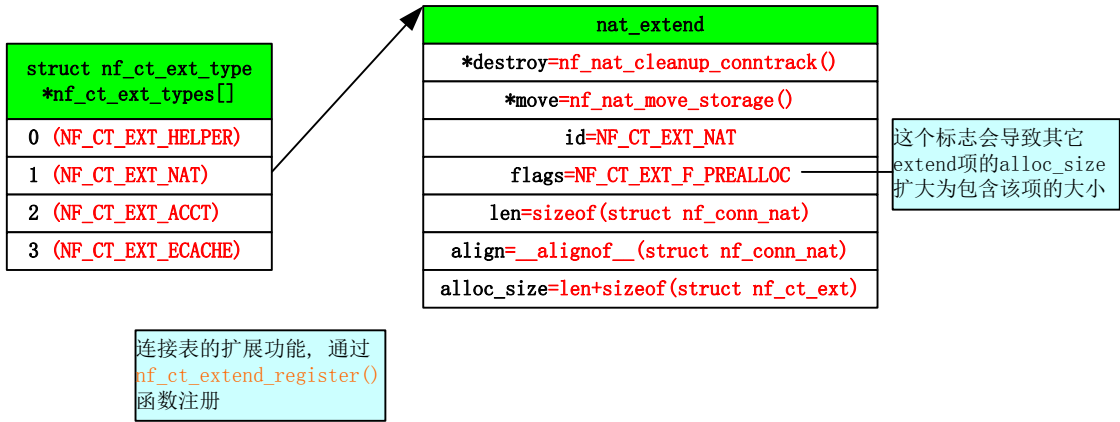


IPv4 利用 nf\_conntrack 进行 NAT 转换

IPv4-NAT 初始化的资源

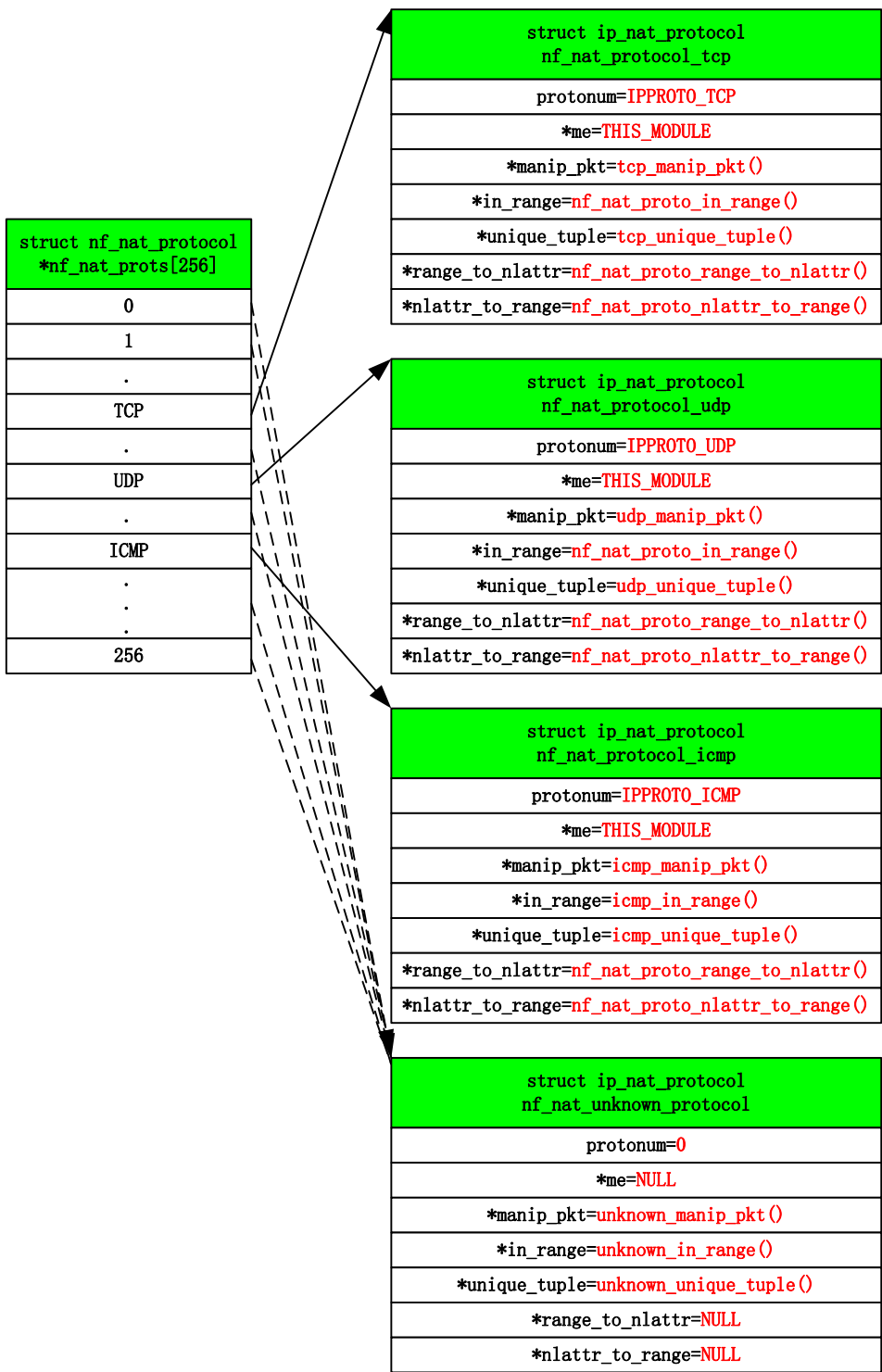
1 NAT 功能的连接跟踪部分初始化，通过函数 nf\_nat\_init()

1.1 调用 nf\_ct\_extend\_register() 注册一个连接跟踪的扩展功能。



1.2 调用 register\_pernet\_subsys() --> nf\_nat\_net\_init() 创建 net->ipv4.nat\_bysource 的 HASH 表，大小等于 net->ct.htable\_size。

1.3 初始化 nf\_nat\_protos[] 数组，为 TCP、UDP、ICMP 协议指定专用处理结构，其它协议都指向默认处理结构。



1.4 为 nf\_conntrack\_untracked 连接设置 IPS\_NAT\_DONE\_MASK 标志。

1.5 将 NAT 模块的全局变量 l3proto 指向 IPV4 协议的 nf\_conntrack\_l3proto 结构。

1.6 设置全局指针 nf\_nat\_seq\_adjust\_hook 指向 nf\_nat\_seq\_adjust() 函数。

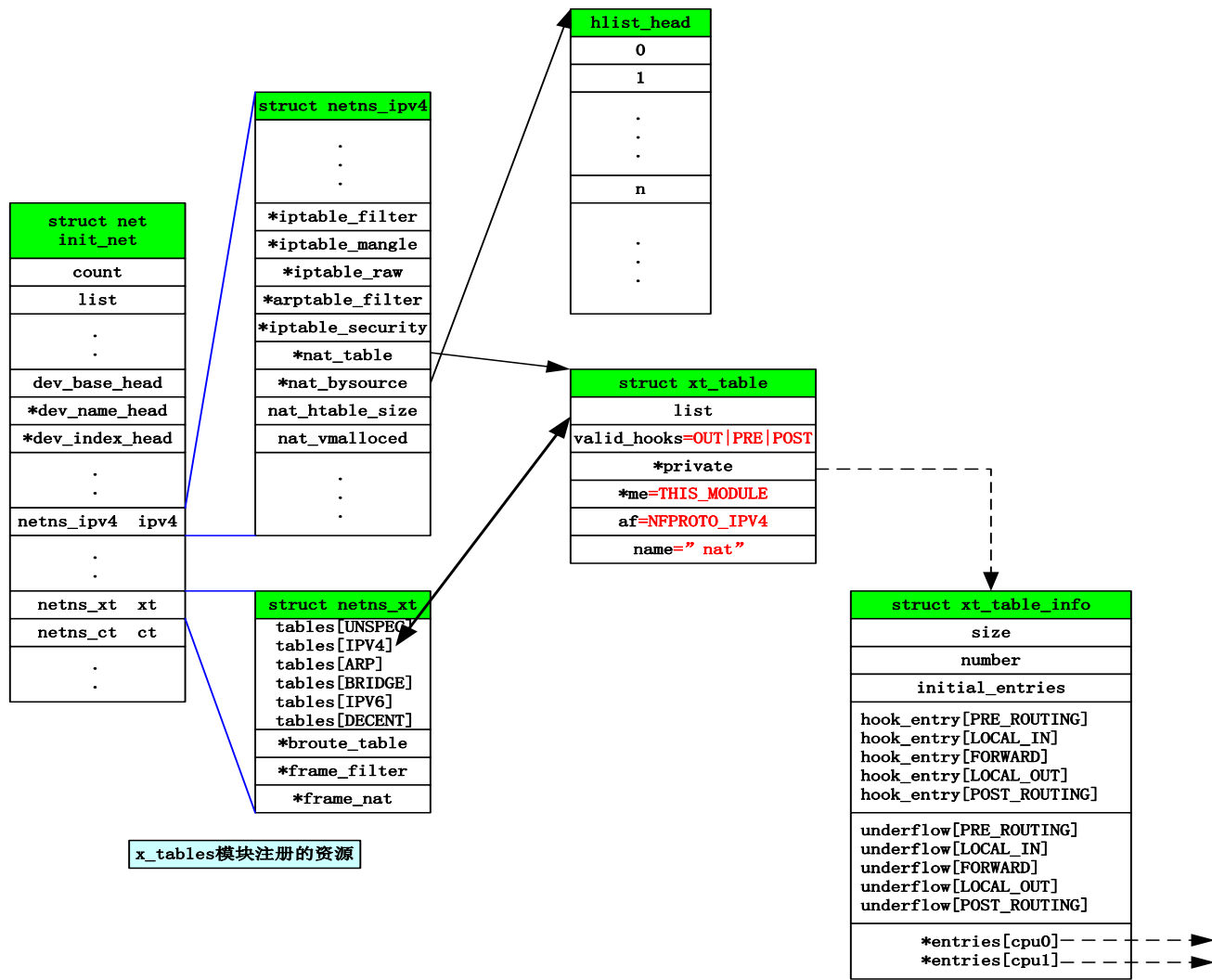
1.7 设置全局指针 nfnetlink\_parse\_nat\_setup\_hook 指向 nfnetlink\_parse\_nat\_setup() 函数。

1.8 设置全局指针 nf\_ct\_nat\_offset 指向 nf\_nat\_get\_offset() 函数。

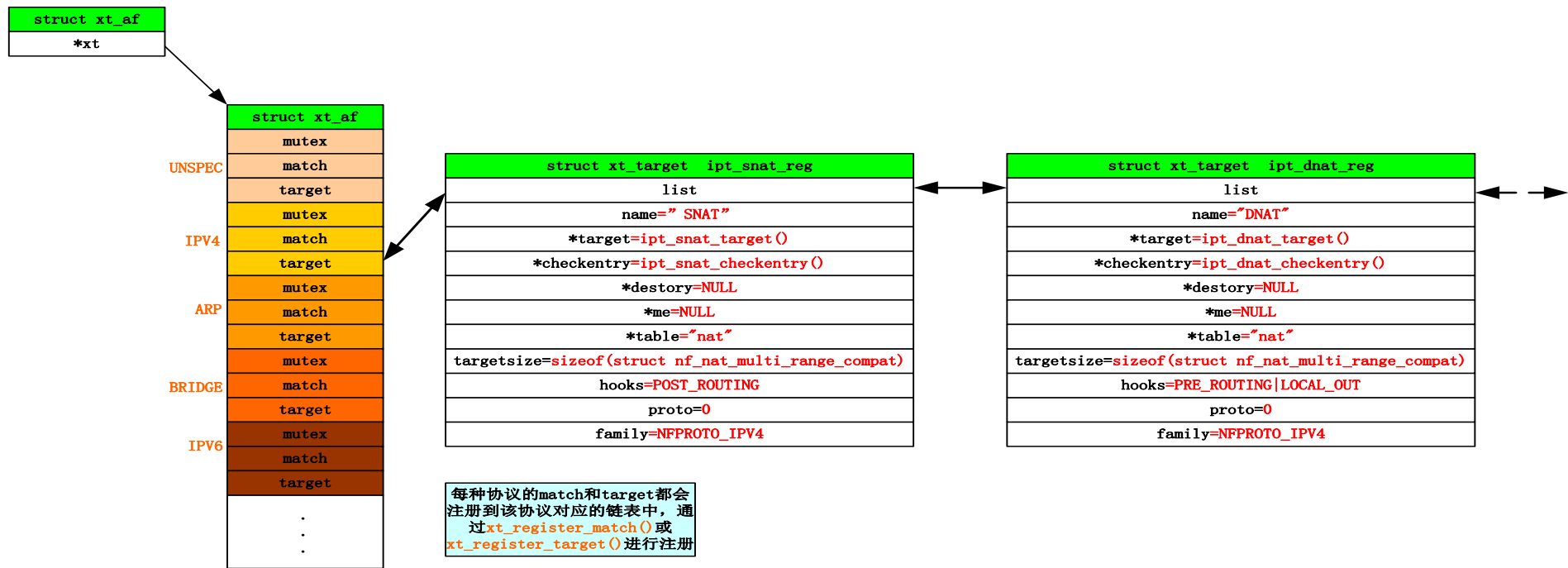
2 NAT 功能的 iptables 部分初始化，通过函数 nf\_nat\_standalone\_init()

2.1 调用 nf\_nat\_rule\_init() --> nf\_nat\_rule\_net\_init() 在 iptables 中注册一个 NAT 表（通过 ipt\_register\_table() 函数）

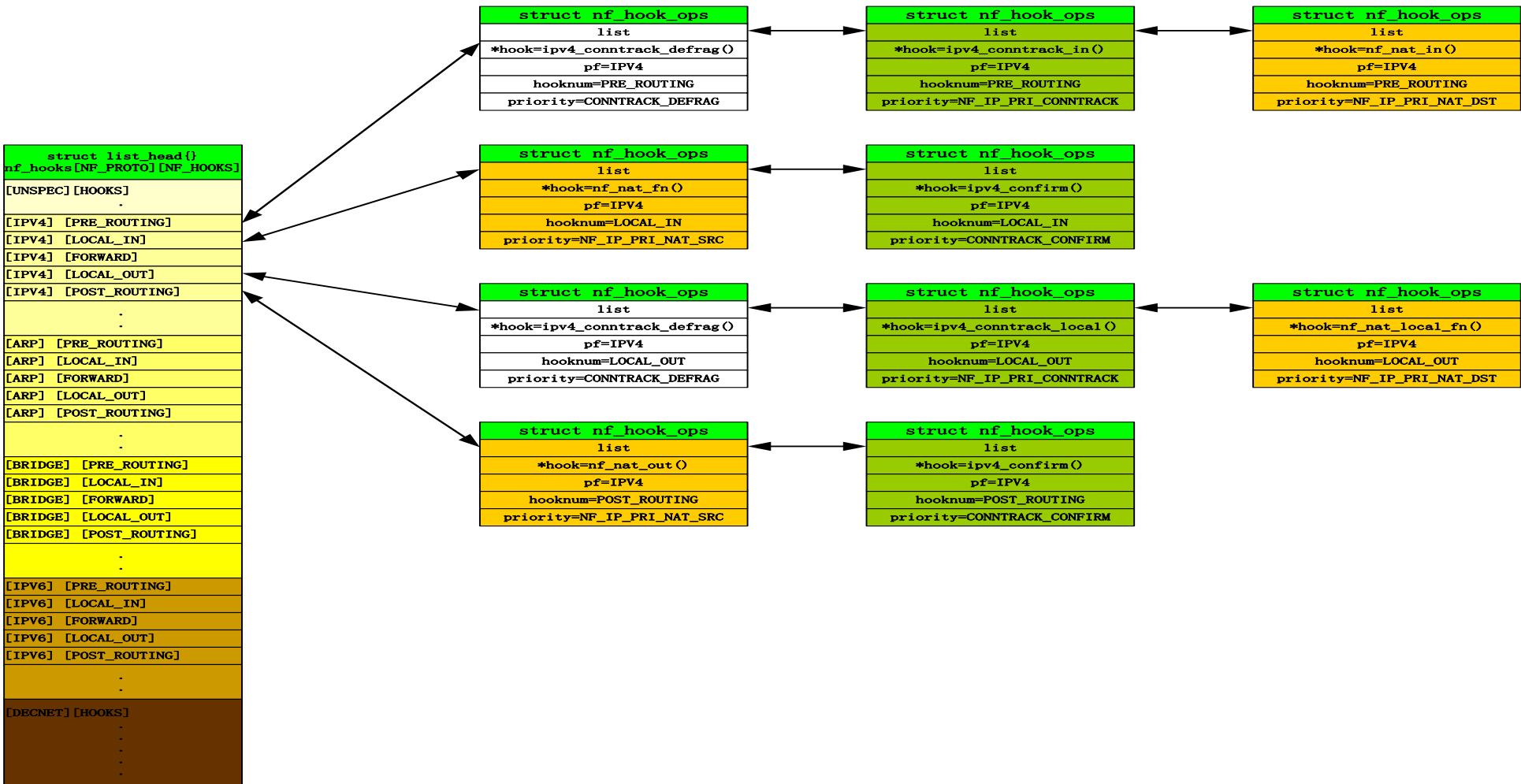


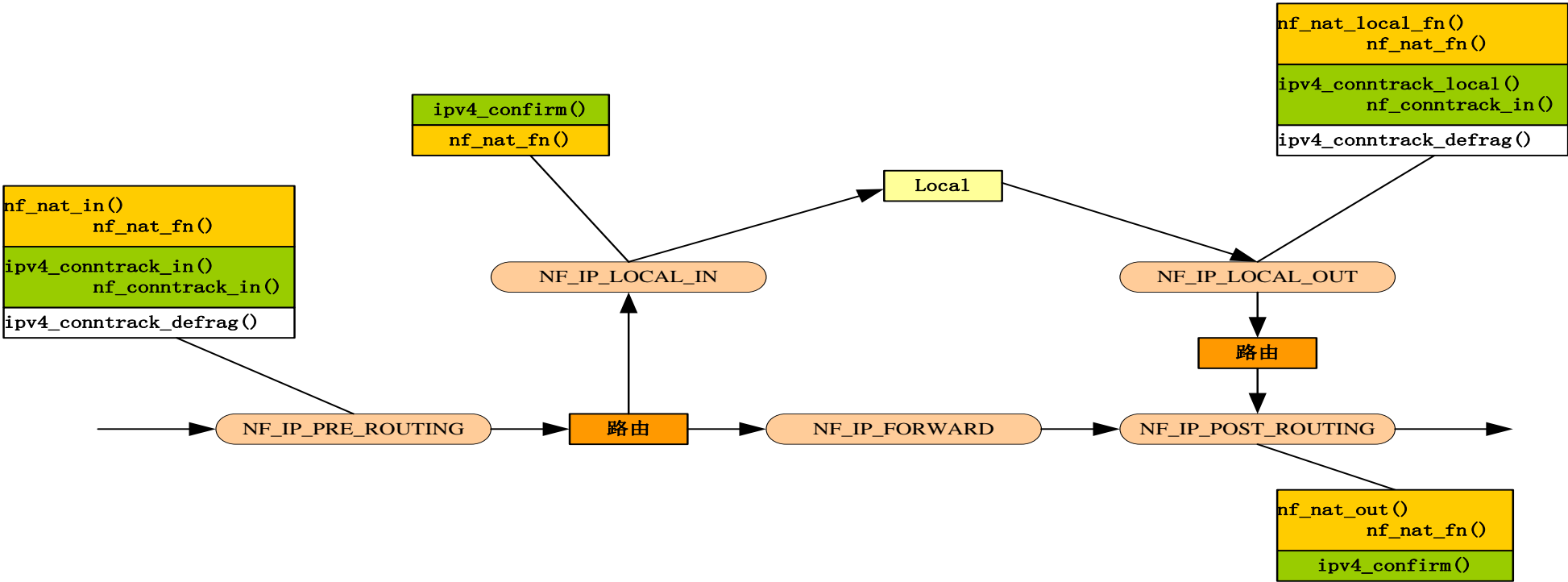


2.2 调用 nf\_nat\_rule\_init() 注册 SNAT target 和 DNAT target（通过 xt\_register\_target()函数）



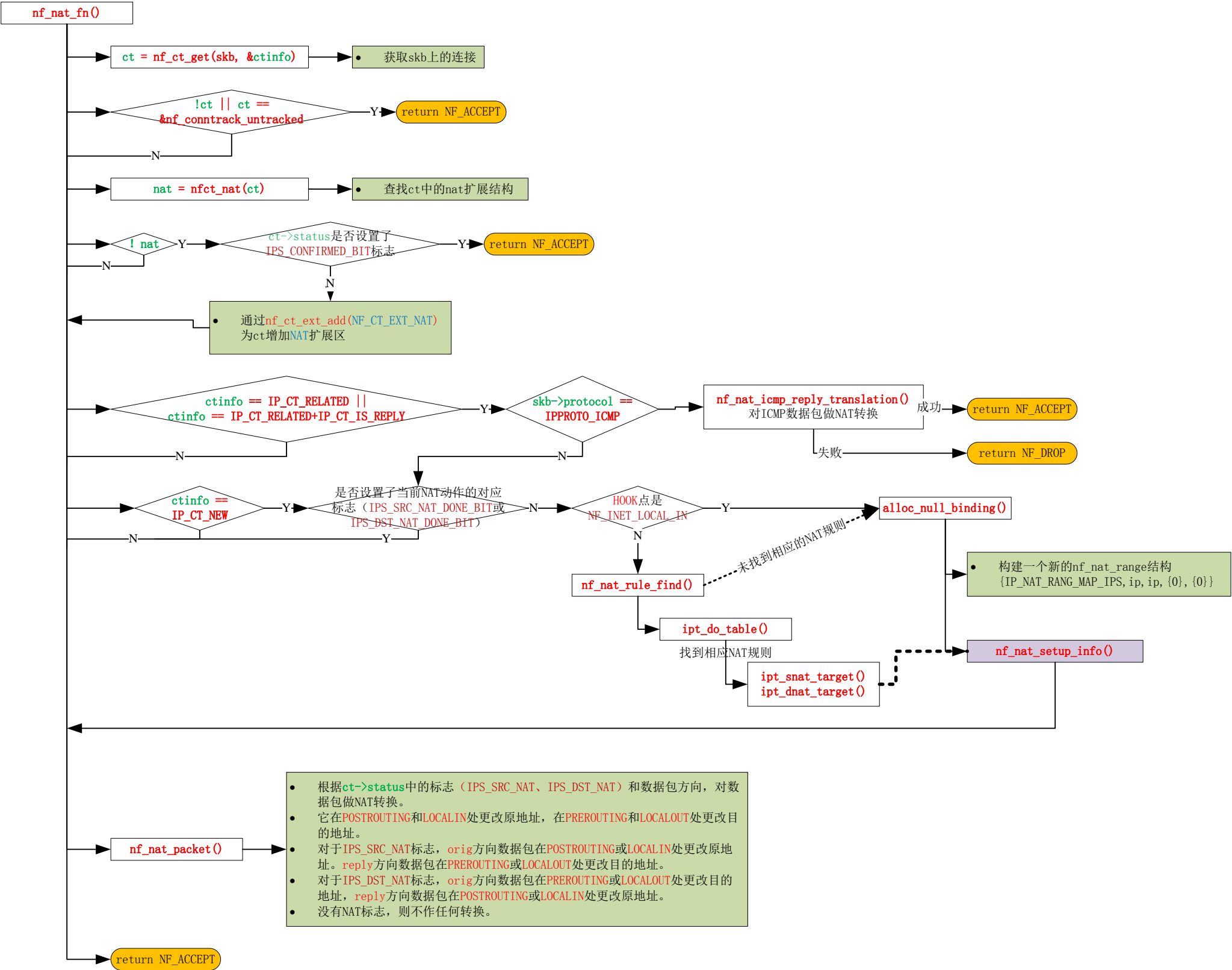
2.3 调用 nf\_register\_hooks() 挂载 NAT 的 HOOK 函数



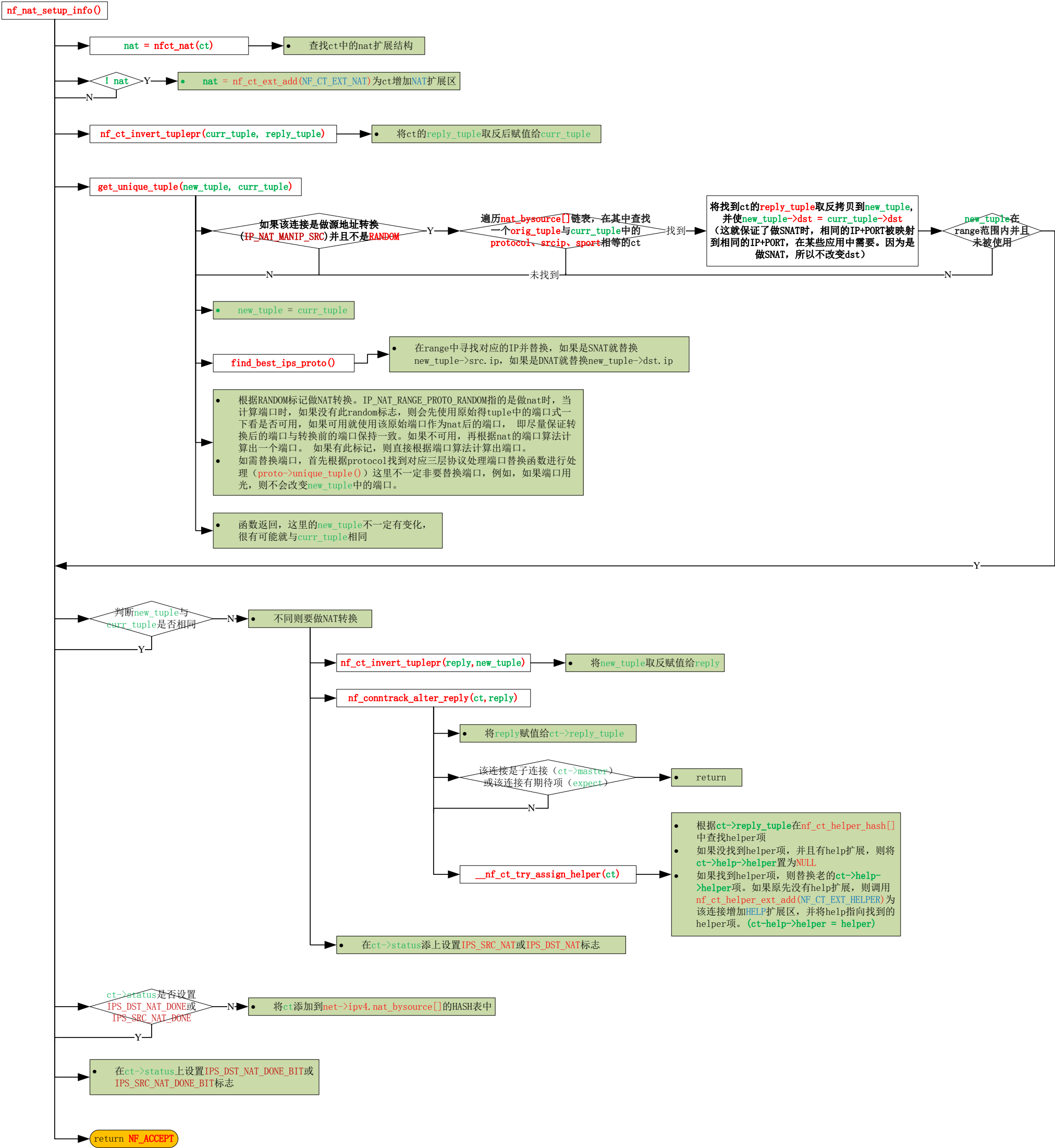


IPv4-NAT 处理流程

nf\_nat\_fn



nf\_nat\_setup\_info

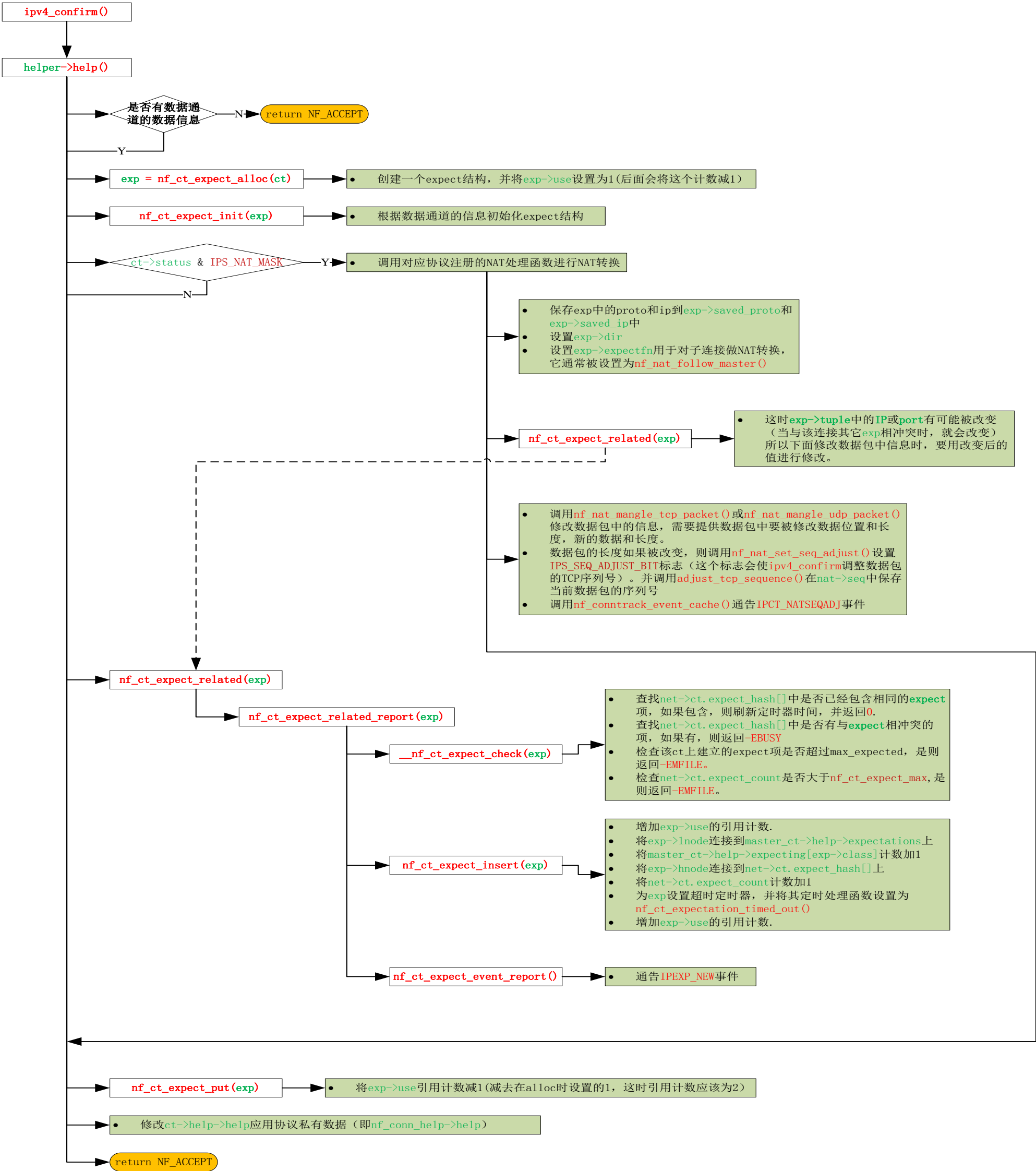


- 1
- 每个 ct 在第一个包就会做好 snat 与 dnat,nat 的信息全放在 reply tuple 中, orig tuple 不会被改变。一旦第一个包建立好 nat 信息后, 后续再也不会修改 tuple 内容了。
- 2
- orig tuple 中的地址信息与 reply tuple 中的地址信息就是原始数据包的信息。例如对 A->B 数据包同时做 snat 与 dnat, PREROUTING 处 B 被 dnat 到 D, POSTROUTING 处 A 被 snat 到 C。则 ct 的内容是: A->B | D->C, A->B 说明了 orig 方向上数据包刚到达墙时的地址内容, D->C 说明 reply 方向上数据包刚到达墙时的地址内容。
- 3
- 在代码中有很多!dir 操作, 原理是: 当为了反向的数据包做事情的时候就取反向 tuple 的数据, 这样才能保证 NAT 后的 tuple 信息被正确使用。
- 4
- bysource 链中链接了所有 CT (做过 NAT 和未做过 NAT), 通过 ct->nat->bysource, HASH 值的计算使用的是 CT 的 orig tuple。其作用是, 当为一个新连接做 SNAT, 需要得到地址映射时, 首先对该链进行查找, 查找此源 IP、协议和端口号是否已经做过了映射。如果做过的话, 就需要在 SNAT 转换时, 映射为相同的源 IP 和端口号。为什么要这么做呢? 因为对于 UDP 来说, 有些协议可能会用相同端口和同一主机不同的端口 (或不同的主机) 进行通信。此时, 由于目的地不同, 原来已有的映射不可使用, 需要一个新的连接。但为了保证通信的的正确性, 此时, 就要映射为相同的源 IP 和端口号。其实就是为 NAT 的打洞服务的。所以 bysource 就是以源 IP、协议和端口号为 hash 值的一个表, 这样在做 snat 时保证相同的 ip+port 影射到相同的 ip+port。
- 5
- IP\_NAT\_RANGE\_PROTO\_RANDOM 指的是做 nat 时, 当计算端口时, 如果没有此 random 标志, 则会先使用原始得 tuple 中的端口试一下看是否可用, 如果可用就使用该原始端口作为 nat 后的端口, 即尽量保证转换后的端口与转换前的端口保持一致。如果不可用, 再根据 nat 的端口算法计算出一个端口。 如果有此标记, 则直接根据端口算法计算出端口。
- 6
- 第一个包之后, ct 的两个方向的 tuple 内容就固定了, 所有的 nat 操作都必须在第一个包就完成。所以 daddr = &ct->tuphash[!dir].tuple.dst.u3;会有这样的操作。

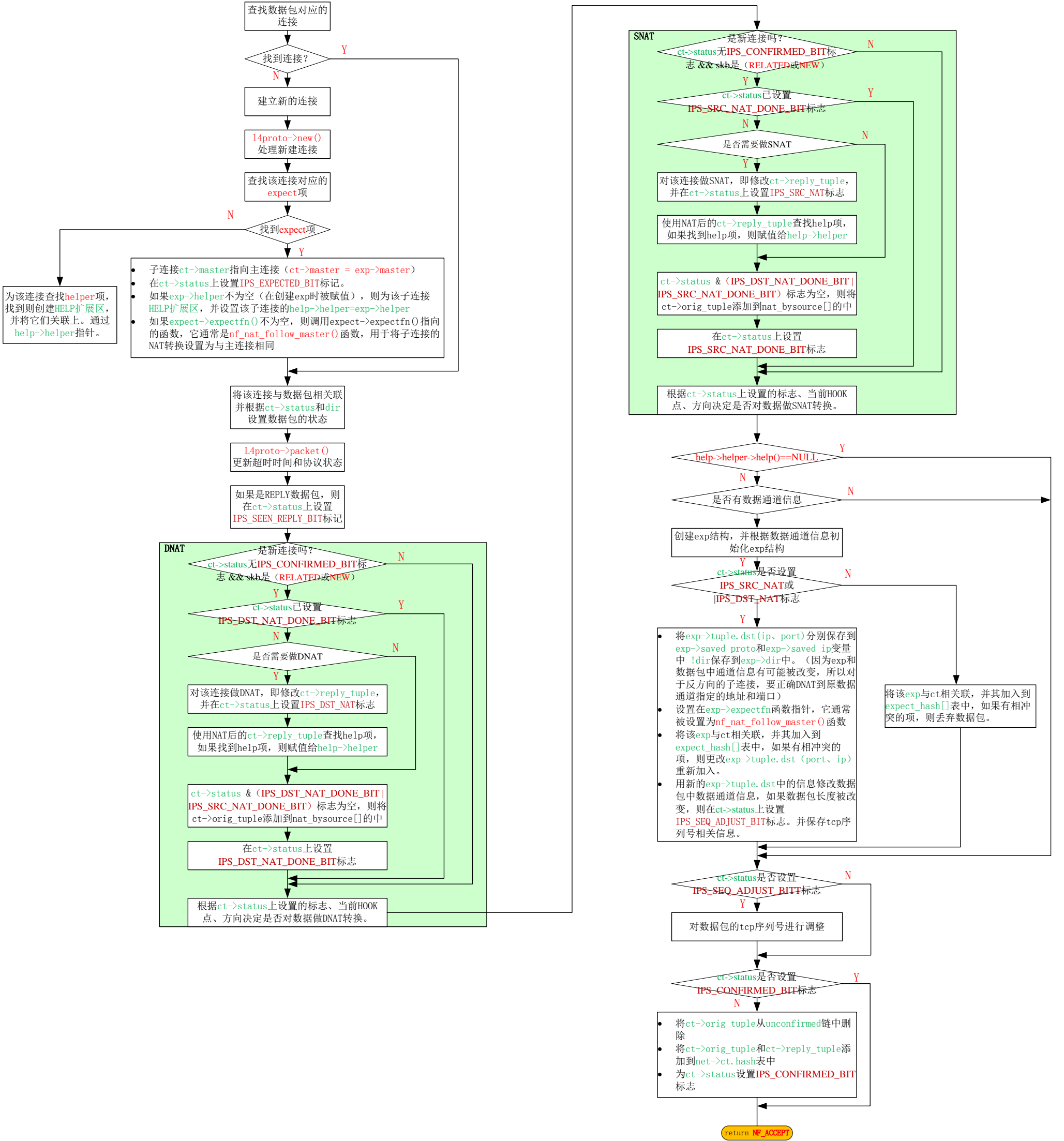
- 7 IPS\_SRC\_NAT 与 IPS\_DST\_NAT，如果被设置，表示经过了 NAT，并且 ct 中的 tuple 被做过 SNAT 或 DNAT。
- 8 数据包永远都是在 PREROUTING 链做目的地址和目的端口转换，在 POSTROUTING 链做原地址和原端口转换。是否要做 NAT 转换则要根数据包方向（dir）和 NAT 标志（IPS\_SRC\_NAT 或 IPS\_DST\_NAT）来判断。
- PREROUTING 链，数据包是 original 方向，并且连接上设置 IPS\_DST\_NAT 标志，或数据包是 reply 方向，并且连接上设置 IPS\_SRC\_NAT 标志，则做 DNAT 转换。
- POSTROUTING 链，数据包是 original 方向，并且连接上设置 IPS\_SRC\_NAT 标志，或数据包是 reply 方向，并且连接上设置 IPS\_DST\_NAT 标志，则做 SNAT 转换。
- 9 IPS\_DST\_NAT\_DONE\_BIT 与 IPS\_SRC\_NAT\_DONE\_BIT，表示该 ct 进入过 NAT 模块，已经进行了源或者目的 NAT 判断，但并不表示 ct 中的 tuple 被修改过。
- 10 源目的 nat 都是在第一个包就判断完成的，假设先添加了原 nat 策略，第一个包通过，这时又添加了目的 nat, 第二个包近来时是不会匹配目的 nat 的 。
- 11 对于一个 ct，nf\_nat\_setup\_info 函数最多只能进入 2 次，第一次 DNAT，第二次 SNAT。在 nf\_nat\_follow\_master 函数中，第一次 SNAT，第二次 DNAT。

IPv4 动态协议的识别和 NAT 转换

动态协议 expect 结构的建立



动态协议 NAT 处理流程图



1 无子连接

- 1.1 一个 ct 用于跟踪一个连接的双方向数据，ct->orig\_tuple 用于跟踪初始方向数据，ct->reply\_tuple 用于跟踪应答方向数据。当根据初始方向数据构建 ct->orig\_tuple 时，同时要构建出 ct->reply\_tuple，用于识别同一连接上应答方向数据。
- 1.2 如果初始方向的数据在通过防火墙后被做了 NAT 转换，为识别出 NAT 数据的应答数据包，则对 ct->reply\_tuple 也要做 NAT 转换。同时 ct 上做好相应 NAT 标记。
- 1.3 所以，上面的信息在初始方向第一个数据包通过后，就要求全部建立好，并且不再改变。
- 1.4 一个连接上不同方向的数据，都有相对应的 tuple (orig\_tuple 和 reply\_tuple)，所以该连接后续数据都将被识别出来。如果 ct 上有 NAT 标记，则根据要去往方向 (即另一个方向) 的 tuple 对数据包做 NAT 转换。所以会有 ct->tuplesh[!dir].tuple 这样的操作。

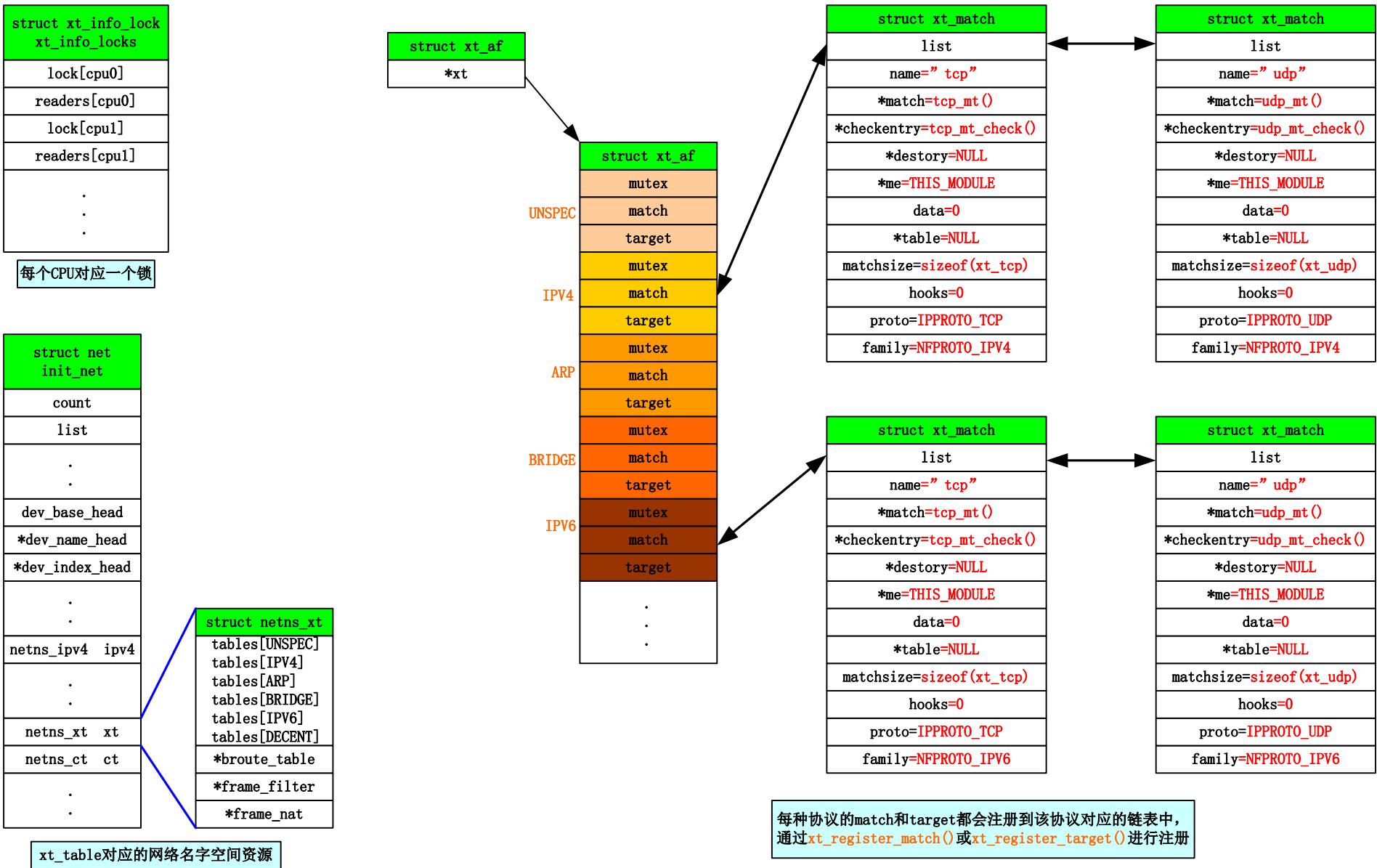
2 有子连接

- 2.1 子连接是由主连接构建的 expect 项识别出来的。



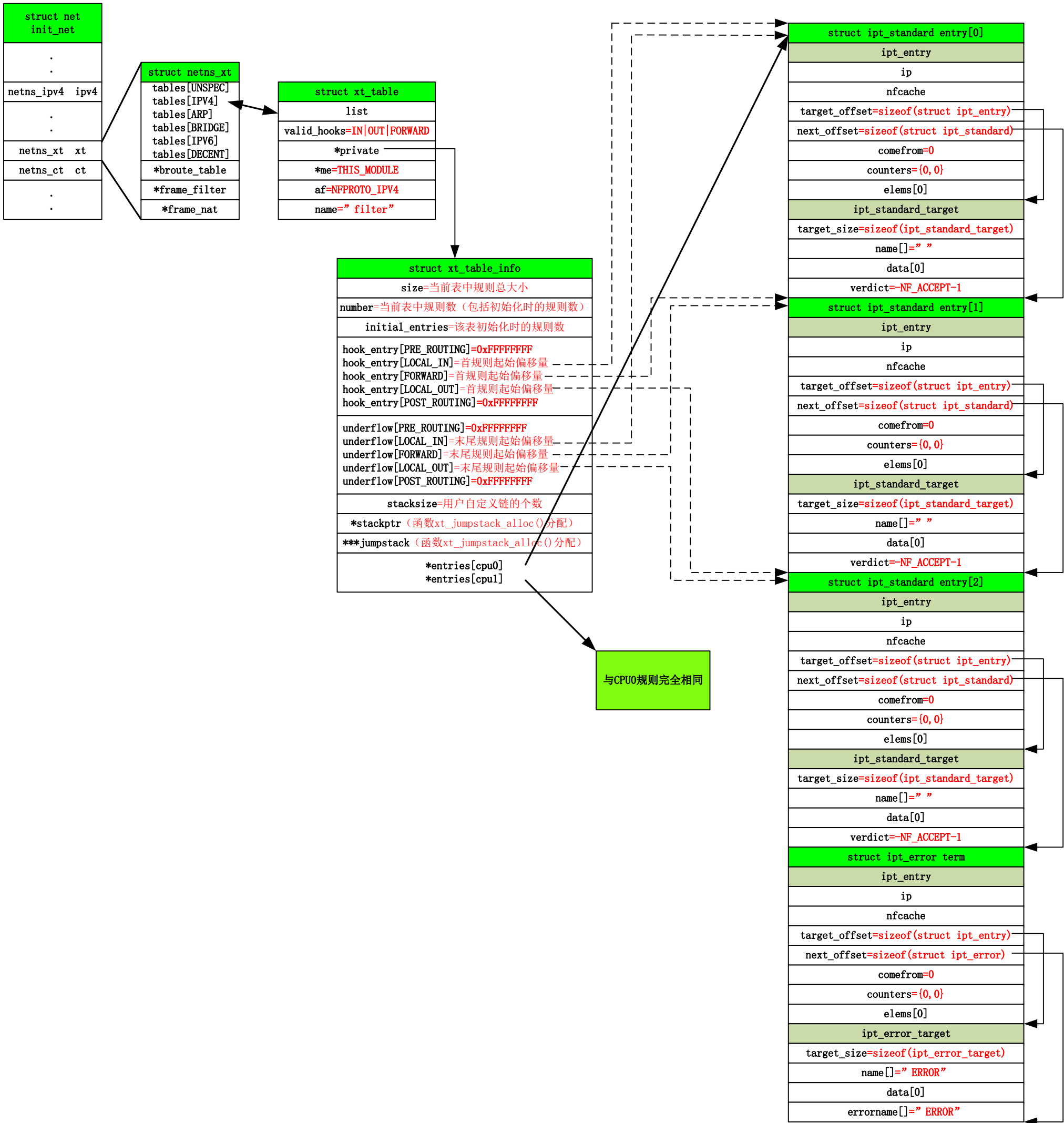
- 2.2 `help` 用于构建 `expect` 项，它期待哪个方向的连接，则用那个方向的 `tuple` 和数据包中数据通道信息构建 `expect` 项。例如期待和当前数据包相反方向的连接，则用相反方向的 `tuple` 中的信息（`ct->tuplehash[!dir].tuple`）。调用 `help` 时，NAT 转换都已完成（`tuple` 中都包含有正确的识别各自方向的信息），所以这时所使用的信息都是正确和所期望的信息。
- 2.3 如果子连接还可能有子连接，则构建 `expect` 项时，初始化一个 `helper` 结构，并赋值给 `expect->helper` 指针。
- 2.4 如果该连接已被做了 NAT 转换，则对数据包中数据通道信息也要做 NAT 转换。这个工作是在 `help` 中完成的。同时，为保证子连接与主连接做相同的 NAT，要为 `expect->expectfn()` 指针初始化一个函数，用于对子连接做 NAT 转换。
- 2.5 在新建一个连接时，如果匹配上某个 `expect` 项，则说明该连接是子连接。这时要根据 `expect` 项中信息对子连接做下面两件事：
  - 2.5.1 如果 `expect->helper` 指针不为空，则为该子连接关联 `expect` 指定的 `help` 项。
  - 2.5.2 如果 `expect->expectfn()` 指针不为空，这调用指针指向的函数，对子连接做 NAT 转换。这样后面 NAT 模块就会根据子连接中信息对子连接数据做 NAT 转换。

Xtables 提供的资源



- 1 struct xt\_af xt[]结构数组
- 该数组用于挂载各个协议的 match 和 target 资源。由于写者（添加、删除）和读者（查找）都是在内核空间进程上下文执行，所以它们只需要用 xt[n].mutex 信号量进行互斥。读者（查找）在将规则关联上一个 match 或 target 时会增加它们所在模块的引用计数，在它释放这个引用计数之前该模块是不会被卸载的，所以另外一个读者（规则匹配）在软中断中可以放心的使用，不需加任何锁。
- 1.1 xt\_register\_match(struct xt\_match \*match)与 xt\_unregister\_match(struct xt\_match \*match)
- 用于在 xt[]数组上挂载对应协议的 match，由于它们都是在内核空间的进程上下文被使用，所以它们使用 mutex\_lock(&xt[af].mutex)信号量进行加锁和解锁。（写者）
- 1.2 xt\_register\_target(struct xt\_target \*target)与 xt\_unregister\_target(struct xt\_target \*target)
- 用于在 xt[]数组上挂载对应协议的 target，由于它们都是在内核空间的进程上下文被使用，所以它们使用 mutex\_lock(&xt[af].mutex)信号量进行加锁和解锁。（写者）
- 1.3 struct xt\_match \*xt\_find\_match(u8 af, const char \*name, u8 revision)与 struct xt\_target \*xt\_find\_target(u8 af, const char \*name, u8 revision)
- 用于在 xt[]数组中查找对应协议的 match 或 target 与对应规则相关联，并增加 match 和 target 所在模块的引用计数。由于它们都是在内核空间的进程上下文被使用，所以它们使用 mutex\_lock(&xt[af].mutex)信号量进行加锁和解锁。同时内核在软中断中进行规则匹配，它引用规则关联的 match 和 target 是安全的，因为 match 和 target 所在模块由于引用计数是不会被释放的。（读者）
- 1.4 由于有一个读者是在软中断的中，并且有多个 CPU 同时使用，是否需要其它保护。答：不需要。因为如果软中断中引用的规则使用了某个 match 或 target，则拥有该 match 和 target 模块的引用计数会被加 1，该模块将不会被卸载（这也就要求在调用 xt\_unregister\_match()或 xt\_unregister\_target()时必须先判断它们所在模块的引用计数，通常它们被放在模块注销函数中）。如果引用计数为 0，则说明没有规则引用该 match 或 target，则在软中断中也不会使用它。
- 2 net.xt.tables[]网络命名空间协议链表
- 该命名空间协议链表用于将不同协议的 table 表挂到对应协议链表中。
- 写者（添加、删除）table 表都在内核空间进程上下文执行，又由于它需要检查该表与注册的 target、match 名字不冲突，所以他们只需要用 xt[n].mutex 信号。
- 读者在软中断中通过 HOOK 引用这些表，所以在写者（添加、删除）之前一定要保证没有读者在操作。添加表操作一定要先通过 xt\_register\_table()添加一个表，然后再通过 xt\_hook\_link()使 HOOK 能够引用这些表；删除表操作一定要先通过 xt\_hook\_unlink()去掉 HOOK 对表的引用，然后再通过 xt\_unregister\_table()删除一个表。
- 2.1 struct xt\_table \*xt\_register\_table(struct net \*net, const struct xt\_table \*input\_table, struct xt\_table\_info \*bootstrap, struct xt\_table\_info \*newinfo)
- 主要是复制 input\_table 到 table 表，并将 newinfo（由调用该函数模块提供的私有数据 xt\_table\_info）与该表的 table->private 指针相关联，然后根据该表指定的协议挂入对应的 net.xt.table[table->af]链表中。它使用 xt[n].mutex 信号进行加锁（如上所述）。
- 2.2 void \*xt\_unregister\_table(struct xt\_table \*table)
- 主要是将 table 从 net.xt.table[table->af]链表中取下来，并返回 table->private 指针指向的 xt\_table\_info 数据。它使用 xt[n].mutex 信号进行加锁（如上所述）。
- 2.3 struct nf\_hook\_ops \*xt\_hook\_link(const struct xt\_table \*table, nf\_hookfn \*fn)与 void xt\_hook\_unlink(const struct xt\_table \*table, struct nf\_hook\_ops \*ops)
- 主要是利用 xt\_table 结构和钩子函数构造出 nf\_hook\_ops 钩子项，然后调用 nf\_register\_hooks()或 nf\_unregister\_hooks()函数来注册或注销 ipv4 协议对应点的钩子函数，这两个函数主要用在内核空间的进程上下文。由于 nf\_register\_hooks()已提供了保护，所以它们不需要任何形式的锁保护。

Iptables 利用 Xtable 初始化 filter 表的结构图



- 1 `struct xt_table *ipt_register_table(struct net *net, const struct xt_table *table, const struct ipt_replace *repl)` (注册并初始化一个表，然后调用 `xt_hook_link()` 引用该表)  
该函数是 iptables 为 filter、nat、mangle 模块提供用于注册相应表结构的接口。它根据当前表要被挂入的 HOOK 点来构建上图所示的 `xt_table_info` 初始规则表，并调用 `xt_register_table()` 函数将 filter 表的 `xt_table` 和 `xt_table_info` 结构挂入 `net.xt.table[IPV4]` 链表中。(上图是 iptables\_filter 模块调用该函数注册的结构图)  
注册完一个表后，就可以通过 `xt_hook_link()` 函数注册一个 HOOK 点来使用这个表中的规则处理数据包。
- 2 `void ipt_unregister_table(struct net *net, struct xt_table *table)` (注销一个表，要在 `xt_hook_unlink()` 之后使用)  
该函数是 iptables 为 filter、nat、mangle 模块提供用于注销相应表结构的接口。它调用 `xt_unregister_table()` 将 `xt_table` 从对应协议链表中取下并释放，然后将返回的 `xt_table_info` 结构中的规则逐一释放 (同时也会释放规则引用的 match 和 target 模块的引用计数)，最后释放 `xt_table_info` 结构。  
为保证释放 table 表时没有其它读者，所以在调用该函数之前要先调用 `xt_hook_unlink()` 函数注销在 HOOK 点挂入的处理函数，保证没有其它 CPU 会再引用到该表。
- 3 `struct xt_info_lock xt_info_locks[CPU]` (用于保证读取修改表中规则的锁，每个 CPU 一个锁)
  - 3.1 `struct xt_table *xt_find_table_lock(struct net *net, u_int8_t af, const char *name)`  
查找 name 指定的表，使用 `xt[af].mutex` 加锁，保证只有一个写者处理该表。并增加表所在模块的引用计数，防止该表被错误释放。

3.2 void xt\_table\_unlock(struct xt\_table \*table)  
与 xt\_find\_table\_lock()配对使用，释放 xt[table->af].mutex 锁。

3.3 static inline void xt\_info\_rdlock\_bh(void) 或 static inline void xt\_info\_rdunlock\_bh(void)  
获取或释放本 CPU 的 xt\_info\_locks[cpu]锁。这个锁主要是用于防止正被使用的规则表（xt\_table\_info 结构）被释放。（它与 get\_counters()进行互斥）

3.4 static inline void xt\_info\_wrlock(unsigned int cpu) 或 static inline void xt\_info\_wrunlock(unsigned int cpu)  
获取指定 CPU 的 xt\_info\_locks[cpu]锁。它主要在 get\_counters()中被调用，用于获取所有 CPU 的写锁，保证所有 CPU 都已完成了对规则表的引用。

3.5 static void get\_counters(const struct xt\_table\_info \*t, struct xt\_counters counters[])  
它可以保证其它 CPU 都已完成了一次对表中所有规则的引用。因为它要对所有其它 CPU 调用 xt\_info\_wrlock(cpu)函数来获取其它 CPU 的 xt\_info\_lock，而其它 CPU 在读取表中规则时，要通过 xt\_info\_rdlock\_bh 获取各自的 xt\_info\_lock 锁，所有当它获取完所有其它 CPU 的 xt\_info\_lock 锁后，就表示其它 CPU 都已完成了对表中规则的引用。这就说明了为什么在 do\_replace 中调用完 get\_counters()后能够安全的释放旧的 xt\_table\_info 结构。

4 static int get\_info(struct net \*net, void \_\_user \*user, const int \*len, int compat) （读取表中信息）  
该函数是用户使用 iptables 命令操作表中规则时，用于获取表中信息的接口。它使用 xt\_find\_table\_lock()和 xt\_table\_unlock()保证没有其它人操作该表。

5 static int get\_entries(struct net \*net, struct ipt\_get\_entries \_\_user \*uptr, const int \*len) （读取表中规则）  
该函数是用户使用 iptables 命令操作表中规则时，用于获取表中规则的接口。它使用 xt\_find\_table\_lock()和 xt\_table\_unlock()保证没有其它人操作该表。

6 unsigned int ipt\_do\_table(struct sk\_buff \*skb, unsigned int hook, const struct net\_device \*in, const struct net\_device \*out, struct xt\_table \*table) （读取表中规则）  
该函数是 iptables 为 filter、nat、mangle 模块提供用于对数据包匹配各表中规则的接口。它根据表对应的 xt\_table\_info 结构中的信息，找到相应的规则，对数据包进行逐一匹配。为保证所引用表中的规则（xt\_table\_info）不被其它写者释放，同时又不影响到其它读者，使用 xt\_info\_rdlock\_bh()和 xt\_info\_rdunlock\_bh()来加锁和解锁。

7 static int do\_replace(struct net \*net, const void \_\_user \*user, unsigned int len) （修改表中规则）  
该函数是 iptables 为 filter、nat、mangle 模块提供用于在对应表中下规则的接口。它根据用户传递过来的规则，构建一个新的 xt\_table\_info 结构和规则，并将它们与对应表的 xt\_table->private 相关联。它通过 xt\_find\_table\_lock()和 xt\_table\_unlock()保证当前只有一个写者在操作该表。通过 local\_bh\_disable()和 local\_bh\_enable()保证更换 table->private 指向新的 xt\_table\_info 结构时不被打断。通过 get\_counters()保证所有其它 CPU 都不再使用旧的 xt\_table\_info 结构，安全释放旧的 xt\_table\_info 结构。

7.1 static int translate\_table(struct net \*net, struct xt\_table\_info \*newinfo, void \*entry0, const struct ipt\_replace \*repl)  
根据 ipt\_replace 结构构建一个 xt\_table\_info 结构，并做一些必要的检查（链是否环路等），同时将表中的规则与相应的 match 和 target 相关联。

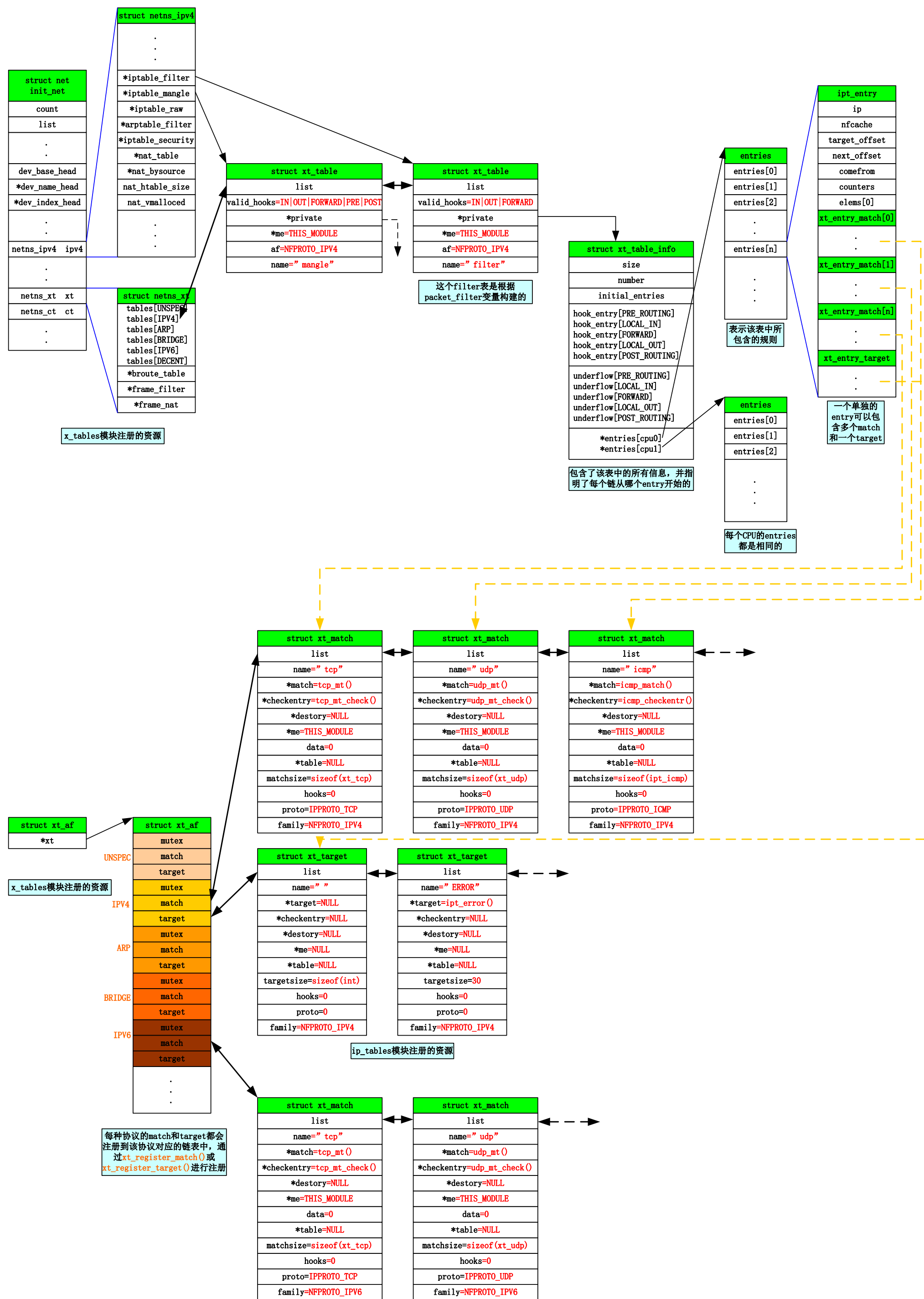
7.2 struct xt\_table\_info \*xt\_replace\_table(struct xt\_table \*table, unsigned int num\_counters, struct xt\_table\_info \*newinfo, int \*error)  
为 newinfo 调用 xt\_jumpstack\_alloc(struct xt\_table\_info \*i)初始化 stack 相关数据，然后使 table->private 指向 newinfo，并返回 oldinfo。

8 总结：

8.1 为每个 CPU 建立一个规则集的原因：是为了在更新规则计数时避免加写锁，每个 CPU 只更新自己规则集中的规则计数。用户上下文通过 write\_lock\_bh()来阻止本地 CPU 和其它 CPU 访问或修改这个规则集，这样它就可以读取或更新规则集了。

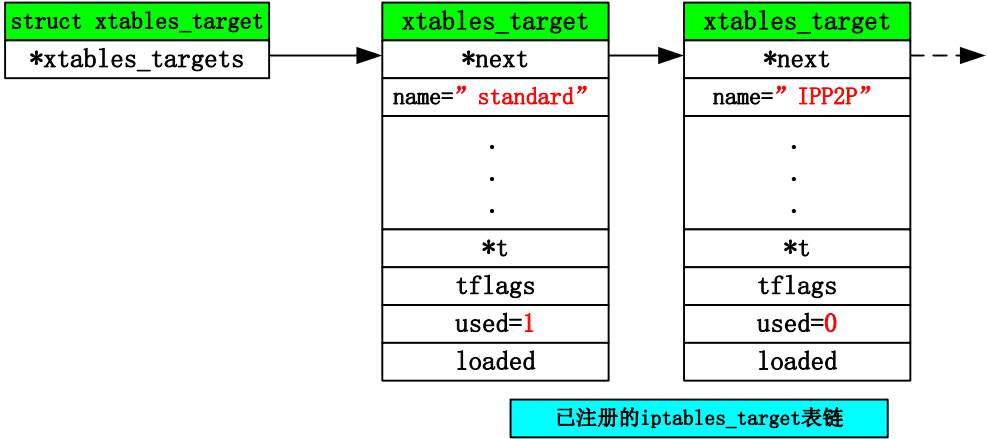
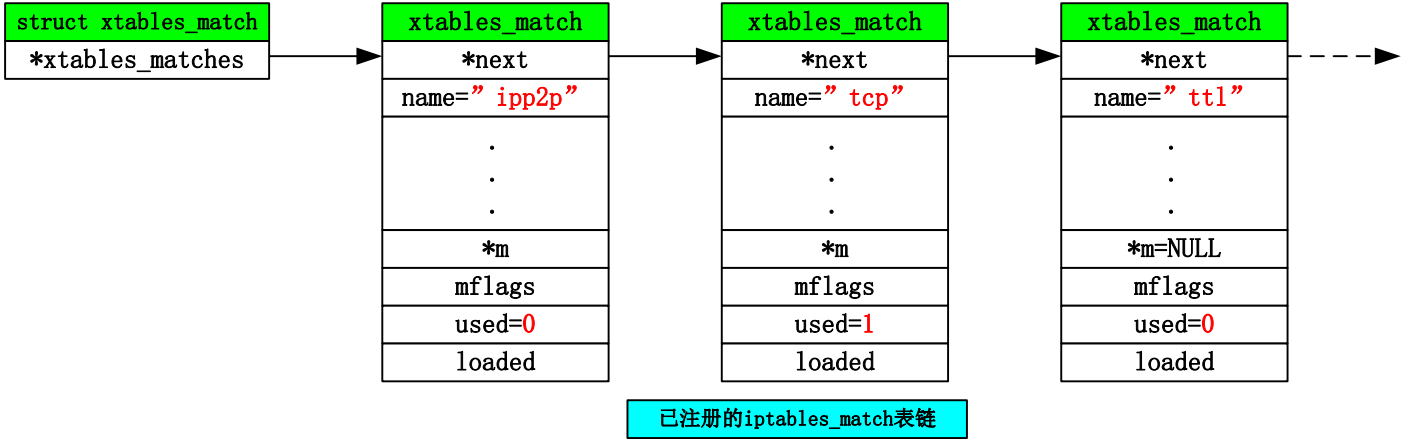
8.2 为每个 CPU 建立一个读写锁（xt\_info\_locks[CPU]）的原因：由于每个 CPU 一个规则集（它们在更新计数器时也只需使用读锁），当用户更新规则或获取规则计数时（它是计算所有 CPU 规则集的计数），就要通过 write\_lock\_bh()阻止本地 CPU 和其它 CPU 访问或修改这个规则集，这就使其它 CPU 都等待这个锁而无法工作。通过使用 xt\_info\_locks[CPU]多 CPU 锁，每计算一个 CPU 规则集计数时，就只对该 CPU 加写锁，从而减少对其它 CPU 的影响。

## Iptables 利用 Xtables 构建的组织形式





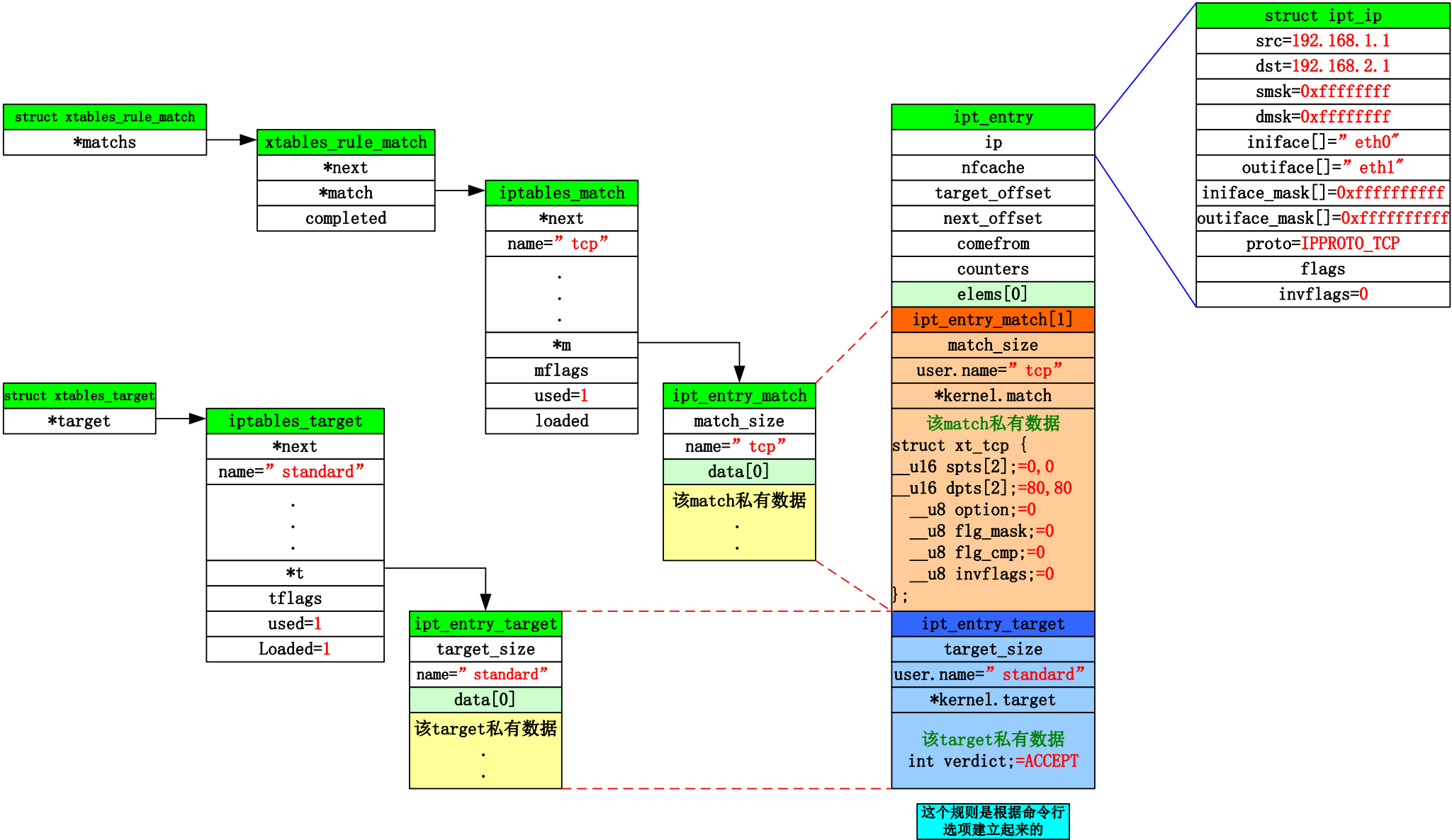
Iptable 用户态的资源



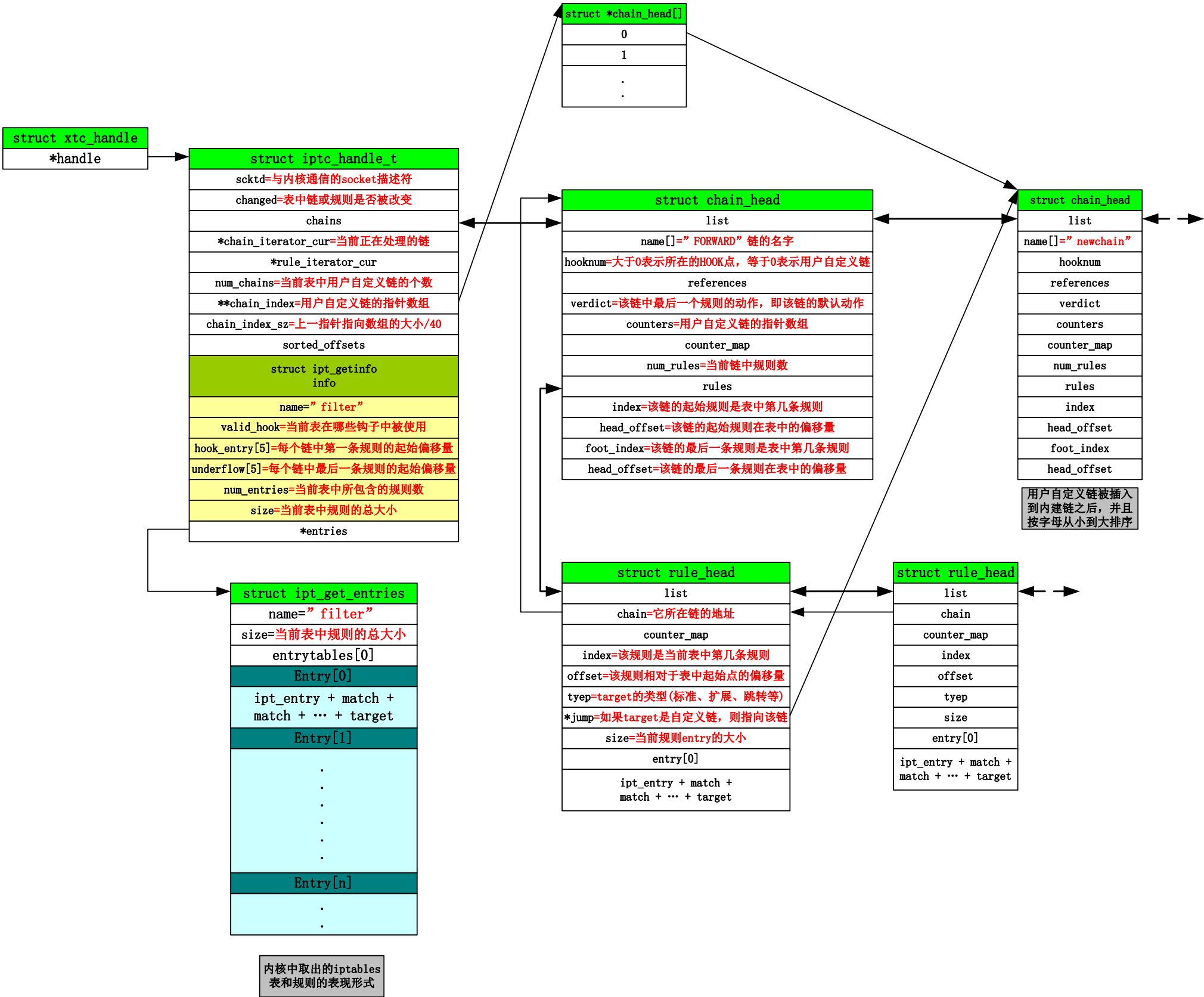
- 1. 用户态的 iptables 包含许多 match 和 target 资源，它们都以链表的形式进行组织。
- 2. 可以使用 xtables\_register\_match()和 xtables\_register\_target()函数扩展 match 与 target 资源。

一条 iptables 规则是如何组织的

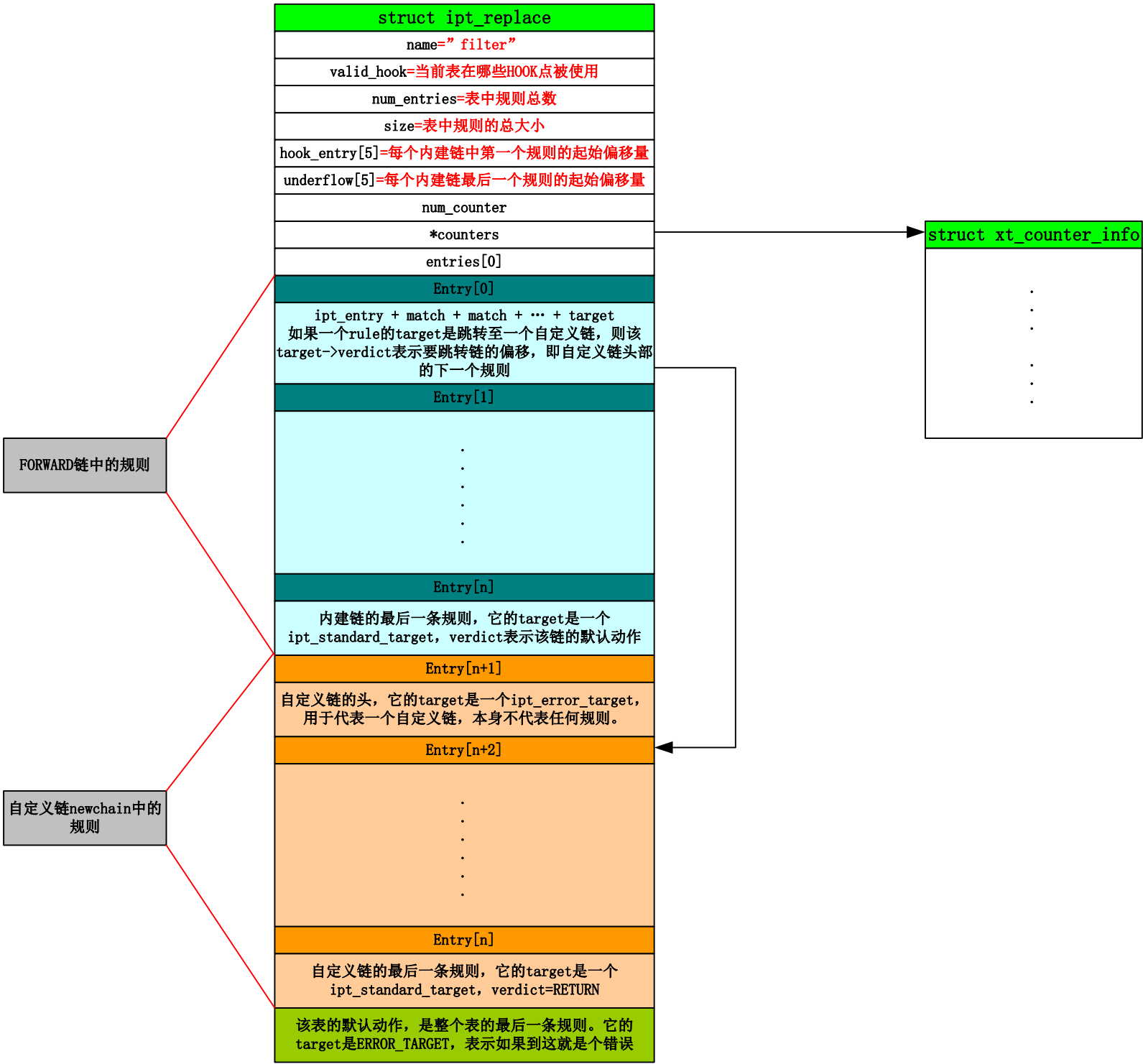
Iptables -t filter -A FORWARD -s 192.168.1.1 -d 192.168.2.1 -i eth0 -o eth1 -p tcp --dport 80 -j ACCEPT



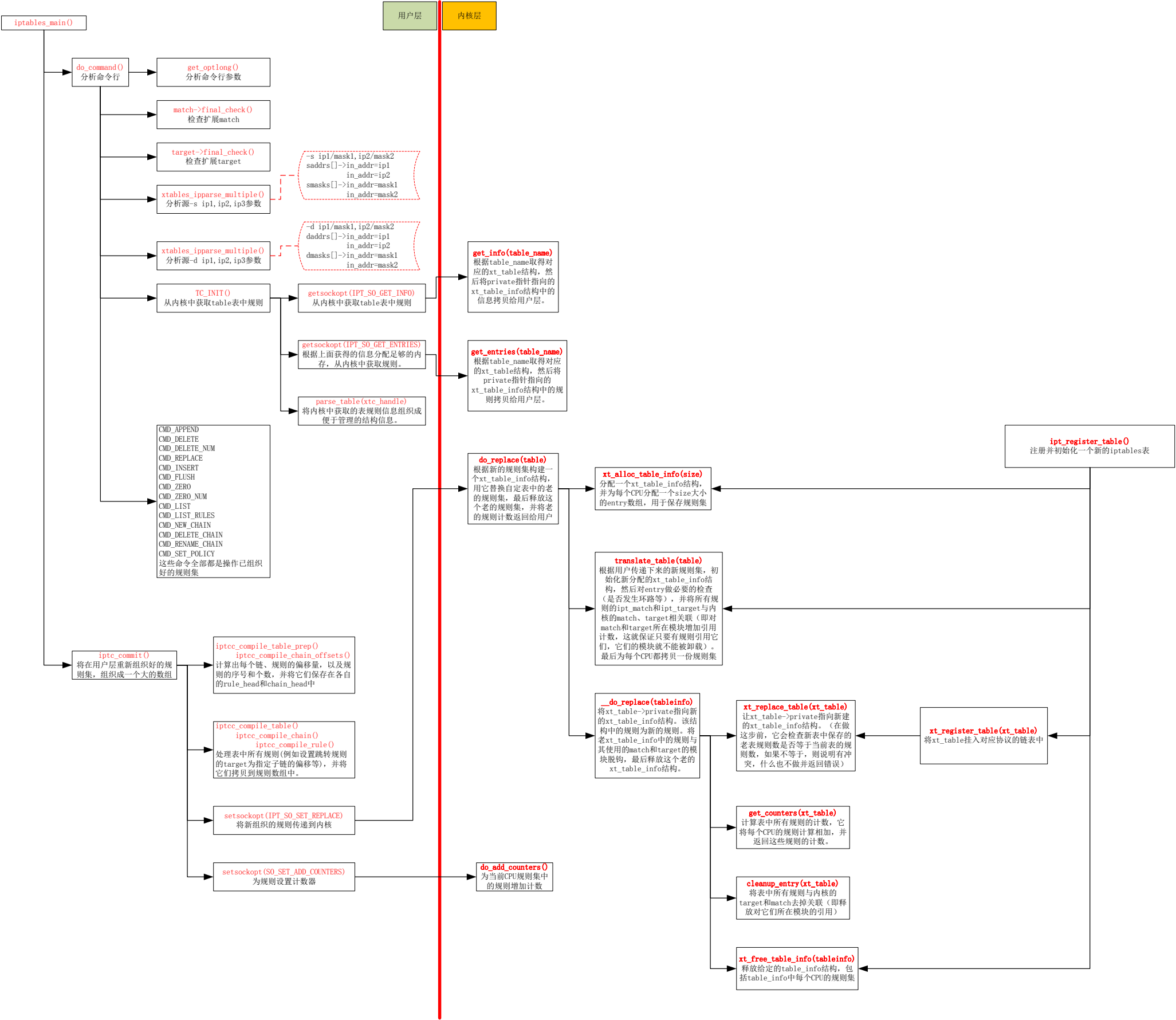
Iptables 对一个表中的规则是如何组织和操作的



Iptables 如何组织规则与内核通信



Iptable 的执行流程



1 Iptables 包含以下 target

- 1.1 IPT\_ACCEPT: (struct ipt\_standard\_target \*)target->verdict = -NF\_ACCEPT-1 < 0 (ipt\_do\_table()碰到这个 target 直接返回 NF\_ACCEPT)
- 1.2 IPT\_DROP: (struct ipt\_standard\_target \*)target->verdict = -NF\_DROP-1 < 0 (ipt\_do\_table()碰到这个 target 直接返回 NF\_DROP)
- 1.3 IPT\_QUEUE: (struct ipt\_standard\_target \*)target->verdict = -NF\_QUEUE-1 < 0 (ipt\_do\_table()碰到这个 target 直接返回 NF\_QUEUE)
- 1.4 IPT\_RETURN: (struct ipt\_standard\_target \*)target->verdict = -NF\_REPATE-1 < 0 (ipt\_do\_table()碰到这个 target 特殊处理)
- 1.5 跳转到子链 target: (struct ipt\_standard\_target \*)target->verdict = 要跳转子链的偏移量 > 0 (ipt\_do\_table()碰到这个 target 会跳转到该子链处理)
- 1.6 IPT\_CONTINUE: IPT\_CONTINUE = XT\_CONTINUE = 0xFFFFFFFF < 0 (ipt\_do\_table()碰到这个 target 会处理下一条规则, 这个 target 被扩展 target 使用)(它不是一个标准 target, 但可被其它扩展 TARGET 用作返回值)
- 1.7 扩展 target: 它是 struct xt\_entry\_target + date[] (ipt\_do\_table()碰到这个 target 会调用 target->target()处理, 并根据返回值做处理。扩展 target 可返回 IPT\_CONTINUE, 或下面 netfilter 定义的值)

2 Netfilter 处理的返回值

- 2.1 NF\_DROP
- 2.2 NF\_ACCEPT
- 2.3 NF\_STOLEN
- 2.4 NF\_QUEUE
- 2.5 NF\_REPEAT
- 2.6 NF\_STOP