

Linux2.6 内核总结（庞宇）

Linux2.6 内核总结	1
中断	3
X86 系统的中断控制器和中断机制	3
8259A 可编程中断控制器	3
APIC 高级可编程中断控制器	3
MSI 与 MSI-X 消息信号中断机制（PCI-E）	3
中断亲和（irq-affinity）	3
RPS (Receive Packet Steering)	4
SMP	5
全局变量	5
使用接口	5
多核启动顺序	5
同步	6
流水线处理	6
缓存一致性	6
优化屏障和内存屏障.....	6
每 CPU 变量	7
原子操作	7
自旋锁.....	8
RCU 互斥.....	8
信号量.....	10
代码执行环境.....	10
中断处理程序（内核中断上下文）	10
软中断处理程序（内核中断上下文）	11
内核抢占（内核进程上下文）	11
内存管理（x86 系统）	12
硬件地址的基本概念.....	12
软件地址的基本概念.....	12
进程与地址	13
linux 内核的内存管理.....	15
slub 机制(slab 与 slob)	16
内存的分配	17
内存的释放	17
linux-2.6 kernel 针对不同配置，内核内存的分配结果	18
linux-2.4 kernel 内存管理中的一些问题	19
网络知识.....	21
网络命名空间（net_namespace）	21
网络中重要数据结构.....	22
sk_buff 数据结构	22
sk_buff 数据结构的管理.....	26
sk_buff 的分片与重组（IP 数据报文）	30
帧的接收	32
内核为驱动提供的接收接口.....	32
驱动如何接收数据包	32
接收队列的组织形式	32
帧的发送	33
内核为驱动提供的发送接口.....	33
驱动程序对发送控制的接口.....	33
发送软中断作用和它的队列组织形式.....	33
与设备、发送、接收相关的状态.....	34
Intel 网卡驱动（igb）举例.....	35
初始化后的全图结构图	35
初始化的处理流程图	36
当中断产生时的处理流程.....	38
IPv4 接收与转发协议栈流程图	41
Linux2.6 网络协议栈处理流程图	42
netfilter.....	43
netfilter 的核心框架图	43
netfilter 提供的全局资源及相应的锁	43
netfilter 为每个钩子函数提供返回值	43
nf_queue 子功能	44
nf_log 子功能.....	44
nfnetlink 通信接口	44
netfilter 相关功能在内核中文件分布图	45
基于 netfilter 的链接跟踪、NAT、包过滤规则、NF-hipac、nf_queue	46

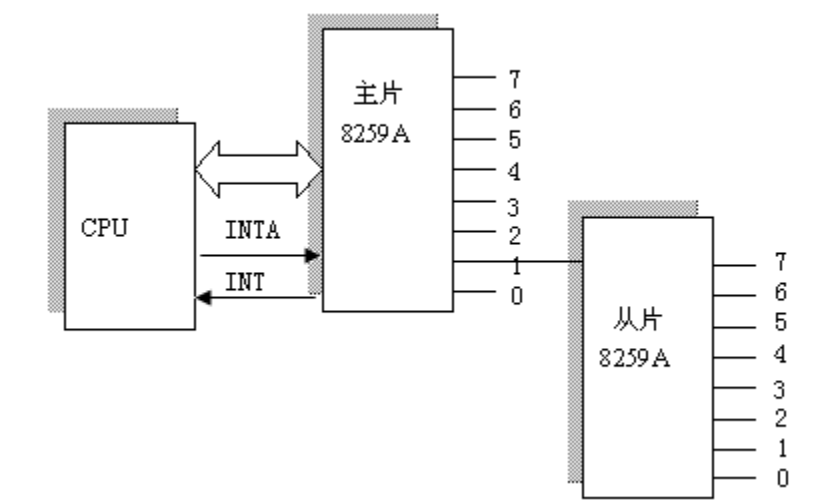
在 IPv4 协议栈中 netfilter 挂的 HOOK 函数	46
连接跟踪 nf_conntrack	47
nf_conntrack 提供的资源	47
nf_conntrack 的初始化	51
连接表的表现形式	52
连接表和动态协议的表现形式	53
IPv4 利用 nf_conntrack 进行链接跟踪	54
IPv4 连接跟踪的初始化	54
IPv4 连接跟踪的处理流程	55
IPv4 利用 nf_conntrack 进行 NAT 转换	60
IPv4-NAT 初始化的资源	60
IPv4-NAT 处理流程	62
IPv4 动态协议的识别和 NAT 转换	65
动态协议 expect 结构的建立	65
动态协议 NAT 处理流程图	66
包过滤规则	68
Xtables 提供的资源	68
Iptables 利用 Xtable 初始化 filter 表的结构图	69
Iptables 利用 Xtables 构建的组织形式	71
Iptable 用户态的资源	72
一条 iptables 规则是如何组织的	72
Iptables 对一个表中的规则是如何组织和操作的	73
Iptables 如何组织规则与内核通信	74
Iptable 的执行流程	75
Iptables 的 hook 与规则	76
Iptables 的互斥	76
连接表的建立、动态协议的识别、NAT 转换之间的处理流程	77
NF-hipac	78
NF-hipac 用户态规则的组织形式	78
NF-hipac 内核态规则的组织形式	79
NF-hipac 内核初始化后组织形式	80
dt_rule 与 chain_rule 之间的关系（dt_rule 是由 chain_rule 构建出来的）	81
NF-hipac 是如何将规则组织成树形结构的	82
NF-hipac 添加规则的执行流程图	84
用户空间与内核的接口	85
procfs (/proc 文件系统)	85
sysctl (/proc/sys 文件系统)	85
sysfs (/sys 文件系统)	85
ioctl (文件或套接字系统调用)	85
socket (套接字系统调用)	85
netlink (套接字系统调用)	85
debugfs (调试内核文件系统)	85
relayfs (快速转发数据文件系统)	85
用户与内核接口的对比	85
conntrack 连接管理	86
netlink 介绍	86
nfnetlink 介绍	86
nf_conntrack_netlink 介绍	86
IPv6	90
IPv6 地址间的关系	90
备注	91
以太网帧的限制	91
连接跟踪	91
性能优化	91
tcpdump	91

中断

X86 系统的中断控制器和中断机制

8259A 可编程中断控制器

传统的 PIC（Programmable Interrupt Controller）是由两片 8259A 风格的外部芯片以“级联”的方式连接在一起。每个芯片可处理多达 8 个不同的 IRQ。因为从 PIC 的 INT 输出线连接到主 PIC 的 IRQ2 引脚，所以可用 IRQ 线的个数达到 15 个。

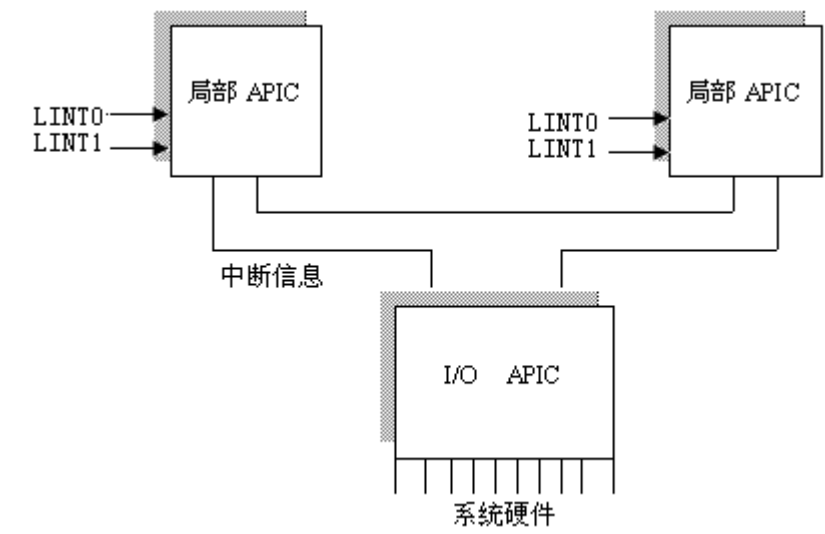


APIC 高级可编程中断控制器

8259A 只适合单 CPU 的情况，为了充分挖掘 SMP 体系结构的并行性，能够把中断传递给系统中的每个 CPU 至关重要。基于此理由，Intel 引入了一种名为 I/O 高级可编程控制器的新组件，来替代老式的 8259A 可编程中断控制器。该组件包含两大组成部分：一是“本地 APIC”，主要负责传递中断信号到指定的处理器；举例来说，一台具有三个处理器的机器，则它必须相对的要三个本地 APIC。另外一个重要的部分是 I/O APIC，主要是收集来自 I/O 装置的 Interrupt 信号且在当那些装置需要中断时发送信号到本地 APIC，系统中最多可拥有 8 个 I/O APIC。每个本地 APIC 都有 32 位的寄存器，一个内部时钟，一个本地定时设备以及为本地中断保留的两条额外的 IRQ 线 LINT0 和 LINT1。所有本地 APIC 都连接到 I/O APIC，形成一个多级 APIC 系统。

目前大部分单处理器系统都包含一个 I/O APIC 芯片，可以通过以下两种方式来对这种芯片进行配置：

- 作为一种标准的 8259A 工作方式。本地 APIC 被禁止，外部 I/O APIC 连接到 CPU，两条 LINT0 和 LINT1 分别连接到 INTR 和 NMI 引脚。
- 作为一种标准外部 I/O APIC。本地 APIC 被激活，且所有的外部中断都通过 I/O APIC 接收。



MSI 与 MSI-X 消息信号中断机制（PCI-E）

这种中断机制是 PCI-E 的新标准，它是利用写寄存器值来触发中断。它相比上面两种中断模式所带来的好处是各个设备可以自定义自己的中断号，保证各个设备使用唯一的 interrupt 号，从而不需要共享中断号。（上面两种模式，设备的中断号是由设备连线决定的，多个设备可能会共享同一中断号）。

MSI-X 中断机制类似于 MSI 中断机制，但它增加了虚拟通道的概念，每个虚拟通道可分配独立的中断号。

中断亲和（irq-affinity）

在 SMP 体系结构中，我们可以通过设置中断亲和力（irq affinity），将一个或多个中断源（中断号）绑定到特定的 CPU 上运行。（这里指的是中断号，而不是某个设备，因为多个设备可能会共享同一中断号）。

在 /proc/irq 目录中，对于已经注册中断处理程序的硬件设备，都会在该目录下存在一个以该中断号命名的目录 IRQ#，IRQ# 目录下有一个 smp_affinity 文件（SMP 体系结构才有该文件），它是一个 CPU 的位掩码，可以用来设置该中断的亲和力，默认值为 0xffff，表明把中断发送到所有的 CPU 上去处理（其实是将中断送到第一个设置为 1 的 bit 位指定的 CPU 上）。

Linux 提供统一的设置 irq-affinity 接口，它内部能够处理 APIC 和 MSI 两种不同的中断机制（因为 SMP 系统中不使用 8259A 中断机制）。但 irq-affinity 在 MSI 中断机制上表现的更好，因为使用 MSI 中断机制的设备都独立使用各自的中断号，而使用 APIC 中断机制的设备可能会共享相同中断号（irq-affinity 是针对中断号，而不是针对设备的）。

RPS (Receive Packet Steering)

RPS 算法可根据 `skb` 报文的五元组信息计算出 `rxhash` 值保存在 `skb` 结构中，然后根据 NIC 的接收队列的 `rps_map` 信息计算出一个 `cpu` 编号，然后将 `skb` 报文加入到指定的 `cpu` 的接收队列，并启动其软中断。

测试用例：

`echo 1 > /sys/class/net/eth0/queues/rx-0/rps_cpus` 表示只让 `core 0` 处理数据包

`echo 2 > /sys/class/net/eth0/queues/rx-0/rps_cpus` 表示只让 `core 1` 处理数据包

`echo 3 > /sys/class/net/eth0/queues/rx-0/rps_cpus` 表示让 `core 0` 和 `core 1` 处理数据包

`cat /proc/net/softnet_stat` 可以查看每个核的软中断处理报文的数目

SMP

全局变量

NR_CPUS	通过 make menuconfig 配置的系统支持 CPU 个数	
nr_cpumask_bits = NR_CPUS	系统中所有 CPU 占用的 bit 位数，每个 CPU 占用 1 位。	
nr_cpu_ids	当前系统中可能会使用的 CPU 个数，它是在启动时根据用户配置和硬件平台实际 CPU 个数设置的。	
nr_node_ids	系统中 Numa 个数。	
struct cpumask { unsigned long bit[NR_CPUS/sizeof(long)]; }		
struct cpumask *cpu_possible_mask = unsigned long cpu_possible_bit[NR_CPUS/sizeof(long)]		每个被置 1 的位表示当前系统中可能包含的 CPU，例如 FF 表示有 8 个 CPU
struct cpumask *cpu_online_mask = unsigned long cpu_online_bit[NR_CPUS/sizeof(long)]		每个被置 1 的位代表一个被启动的 CPU
struct cpumask *cpu_present_mask = unsigned long cpu_present_bit[NR_CPUS/sizeof(long)]		如果为开启 CPU-HOTPLUG，则与 cpu_possible 相同
struct cpumask *cpu_active_mask = unsigned long cpu_active_bit[NR_CPUS/sizeof(long)]		每一个被置 1 的位表示通过 cpu_up()启动的 CPU

使用接口

DEFINE_PER_CPU(type, name)	定义一个 type 类型的 name 变量，在编译的时候把它放在.data.percpu 段的位置中，当系统启动时，会将.data.percpu 段中数据拷贝到每一个 CPU 的私有数据段空间中，而这个段地址由各自 CPU 的 fs 段选择子指定。
smp_processor_id()	这个宏被翻译成 asm("movq %%fs:%p1,%0" : "=r" (pfo_ret__) : "m" (cpu_number)); 它从本 CPU 的 fs 段中获取 cpu_number 的值,即获取当前 CPU 的编号。 在系统初始化时通过 DEFINE_PER_CPU(type, name)定义的 per-cpu 变量会被分别拷贝到每个 CPU 独有数据区中，并将各个 CPU 数据区中的 cpu_number 值设置为该 CPU 的编号值，同时个数据区的基地址会保存到各自 CPU 的 fs 段中，这样通过 fs 段获取的变量就是该 CPU 独有的变量。

多核启动顺序

- 1 主 CPU 加电启动
- 1.1 执行 arch/x86/boot/header.S 程序。这是整个 bzImage 的最前面部分代码，前 512 字节代码被早期 linux 用于 MBR 的 boot secoter(由 BIOS 加载)。现在的 linux 版本已不再支持(必须由 bootloader 加载)，所以下面这部分代码被用于检查 bzImage 是否还是被 BIOS 直接加载的，如果是，则打印信息并等待重启。下面这部分代码对应 ULK3 的远古时代--引导装入程序(但是它有点特殊，因为它现在并不用作启动扇区了，而是用做正确性检查)

1.2 跳转到 arch/x86/boot/compressed/head_32.S 程序。保护模式下的内核代码，用于解压缩 vmlinux，相当于 ULK3 的文艺复兴时期--第一个 startup_32()函数。

1.3 跳转到 arch/x86/kernel/head_32.S 程序的 startup_32 入口点。保护模式下的内核代码，相当于 ULK3 的文艺复兴时期--第二个 startup_32()函数，为第一个 linux 进程(进程 0)建立执行环境(开启页表映射机制，建立进程 0 内核堆栈等)。对于 SMP 结构的系统,这个时候在运行的只是其中的一个处理器,就是所谓的"主 CPU"(也称为"引导处理器"BP)，而其它的"次 CPU"(也称为"应用处理器"AP)则处于停机状态，等待主 CPU 的将其启动。次 CPU 在被主 CPU 启动进入内核时，同样也要到该代码段执行，但它的起始入口点是 ENTRY(startup_32_smp);

1.4 跳转到 init/main.c 文件中的 start_kernel()函数执行初始化，只有主 CPU 才会调用该函数进行初始化。相当于 ULK3 的现代时期。

1.5 创建 kernel_init 进程做进一步工作，其中包括调用 smp_init()—>cpu_up()-->do_boot_cpu()启动所有从 CPU，调用 do_basic_setup()函数做进一步初始化，执行/sbin/init 程序等。

1.6 调用 cpu_idle()等待调度进程执行。
- 2 从 CPU 被主 CPU 启动
- 2.1 执行 trampoline.S 程序，它主要是一个中转程序。主 CPU 在 smp_init()函数中初始化所有次 CPU，它通过 APIC 启动次 CPU 运行时，要把启动地址发给次 CPU。可是，由于 APIC 的内部结构，实际上只能发送一个 8 位的物理页面号，这样就给启动地址加上了限制:首先，必须与页边界对其;其次，必须在 1M 以下。所以主 CPU 要先将 trampoline.S 代码复制到 1M 以下内存中，然后启动次 CPU 从 trampoline.S 开始执行，次 CPU 再通过它跳转到 startup_32_smp 处继续执行。

2.2 跳转到 arch/x86/kernel/head_32.S 程序的 startup_32_smp 入口点，startup_32_smp 就是所有次 CPU 的入口点。

2.3 调用 cpu_idle()等待调度进程执行。

同步

流水线处理

- 1
- 流水线对于单核和多核处理的方式都一样。
- 2
- 现代处理器，为了加快处理速度，使用了流水线处理方法，但流水线处理方法会带来数据冒险（当一条指令用到的数据被前面流水线中指令改变的话，就会出现冒险）。CPU 利用数据前递（将结果值直接从一个流水线阶段传到较早阶段的技术）和加载互锁（用暂停来处理加载/使用冒险的方法）来解决流水线数据冒险问题。
- 3
- 对于应用程序来说，可以不用关注流水线问题（无论是单核还是多核），不需要使用任何机制保证它的正确性（对于软件是透明的），因为 CPU 会处理的很好。

缓存一致性

- 1
- 为了提高数据访问效率，每个 CPU 上都有一个容量很小（现在一般是 1M 这个数量级），速度很快的缓存，用于缓存最常访问的那些数据。由于操作内存的速度实在太慢，数据被修改时也只更新缓存，并不直接写出到内存中去，这一来就造成了缓存中的数据与内存不一致。
- 2
- 单核：如果系统中只有一个 CPU，所有线程看到的都是缓存中的最新数据，这就不存在一致性问题。
- 3
- 多核：但如果系统中有多个 CPU，同一份内存可能会被缓存到多个 CPU 中，如果在不同 CPU 中运行的不同线程看到同一份内存的缓存值不一样就麻烦了，因此有必要维护这多种缓存的一致性。当然要做到这一点只要一有修改操作，就通知所有 CPU 更新缓存，或者放弃缓存下次访问的时候再重新从内存中读取。但这个 Stupid 的实现显然不会有好的性能，为解决这一问题，产生了很多维护缓存一致性的协议，MESI 就是其中一种。

3.1

MESI 协议的名称由来是指这一协议为缓存的每个数据单位（称为 cache line，在 Intel CPU 上一般是 64 字节）维护两个状态位，使得每个数据单位可能处于 M、E、S 或 I 这四种状态之一。各种状态含义如下：
M: 被修改的。处于这一状态的数据只在本 CPU 中有缓存，且其数据已被修改，没有更新到内存中
E: 独占的。处于这一状态的数据只在本 CPU 中有缓存，且其数据没有被修改，与内存一致
S: 共享的。处于这一状态的数据在多个 CPU 中有缓存
I: 无效的。本 CPU 中的这份缓存已经无效了。

3.2

当 CPU 要读取数据时，只要缓存的状态不是 I 都可以从缓存中读，否则就要从主存中读。这一读操作可能会被某个处于 M 或 E 状态的 CPU 截获，该 CPU 将修改的数据写出到内存，并将自己设为 S 状态后这一读操作才继续进行。只有缓存状态是 E 或 M 时，CPU 才可以修改其中的数据，修改后缓存即处于 M 状态。如果 CPU 要修改数据时发现其缓存不处于 E 或 M 状态，则需要发出特殊的 RFO 指令（Read For Ownership），将其它 CPU 的缓存设为 I 状态。因此，如果一个变量在某段时间内只被一个线程频繁修改，则对应的缓存早就处于 M 状态，这时 CAS 操作就不会涉及到总线操作。所以频繁的加锁并不一定会影响系统并发度，关键看锁冲突的情况严重不严重，如果经常出现冲突，即缓存一会被这个 CPU 独占，一会被那个 CPU 独占，这时才会不断产生 RFO，影响到并发性能。

3.3

结论：在 SMP 结构中，多个 CPU 之间缓存一致性问题对于软件而言是透明的，应用程序可以不用关注缓存一致性问题。

优化屏障和内存屏障

- 1
- 当处理多处理器之间或处理器与硬件设备之间的同步问题时，有时需要在你的程序代码中以指定的顺序发出读内存（读入）和写内存（存储）指令，这时就要考虑到屏障，其它情况不需考虑。
- 2
- 由于以下情况会导致指令执行顺序乱序，所以需要优化屏障或内存屏障：

2.1

编译器在优化时会打乱指令执行顺序。

2.2

单 CPU 流水线优化（乱序读/乱序写）会打乱指令执行顺序。

2.3

多 CPU（store buffer/invalidate queue）会打乱指令执行顺序。
- 3
- 优化屏障：当使用优化编译器时，你千万不要认为指令会严格按它们在源代码中出现的顺序执行。例如，编译器可能重新安排汇编语言指令以使寄存器以最优的方式使用。优化屏障原语保证编译程序不会混淆放在原语操作之前的汇编语言指令和放在原语操作之后的汇编语言指令，它可以防止编译器跨屏障对载入或存储操作进行优化。同时使编译器不会重新组织存储或载入操作而防止改变 C 代码的效果和现有数据的依赖关系。

3.1

在 linux 中，优化屏障就是 barrier()宏，它展开为 asm volatile(“”:memory”)。指令 asm 告诉编译程序要插入汇编语言片段（这种情况下位空）。volatile 关键字禁止编译器把 asm 指令与程序中其它指令重新组合。memory 关键字强制编译器假定 RAM 中的所有内存单元已被汇编语言指令修改；因此，编译器不能使用存放在 CPU 寄存器中的内存单元的值来优化 asm 指令后的代码。

3.2

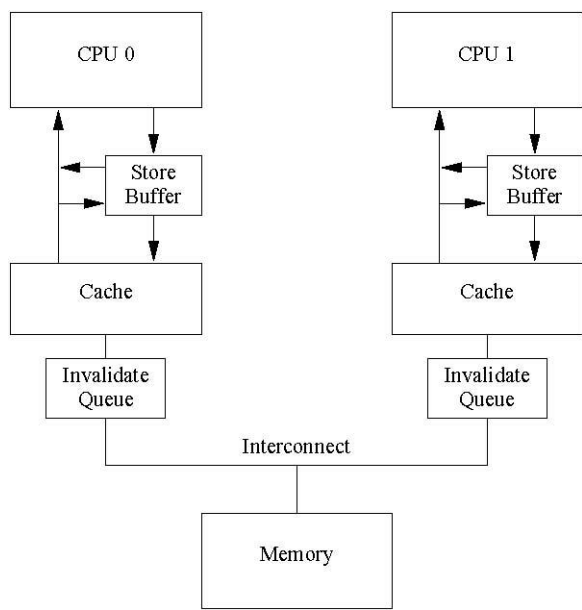
当不与硬件设备或其它 CPU 同步时，是不需要优化屏障的，因为编译器绝不会对相互依赖代码进行优化的。

3.3

注意，优化屏障并不保证当前 CPU 把汇编语言指令不混在一起指行--->这是内存屏障的工作。
- 4
- 内存屏障：内存屏障确保在原语之后的操作开始执行之前，原语之前的操作已经完成。Linux 使用了六个内存屏障原语，这些原语也被当做优化屏障，因为我们必须保证编译程序不在屏障前后移动汇编语言指令。“读内存屏障”仅仅作用于从内存读操作的指令，而“写内存屏障”仅仅作用于写内存的指令。内存屏障即用于多处理器系统，也用于单处理器系统。当内存屏障用于防止仅出现在多处理器系统上的竞争条件时，就使用 smp_***()原语；在单处理器系统上，它们什么也不做，仅仅是充当编译器屏障。*mb()原语用于单处理器和外设或多处理器和外设系统上的竞争条件。

4.1

既然 SMP 之间的缓存一致性由 MESI 保证，为什么在 SMP 系统中还要有 smp_***()内存屏障呢。如下图所示：



由于 MESI 的存在，当 CPU 要写非 E 或 M 状态的缓存时，总是要发送“read invalidate”信号给其它 CPU，当接收到所有 CPU 对该信号的应答后，才会执行写操作，而在等待应答信号时 CPU 被延迟处理后续指令。为解决这个瓶颈，现代 CPU 中提供了“store buffer”用于存放等待信号的指令，而继续执行后续指令，当应答信号到来时，再执行“store buffer”中的指令。这就导致了写乱序。所以提供了 `smp_mb()` 内存屏障，保证后续指令都等待“store buffer”中的指令执行完毕或将后续对 cache 的操作都排在“store buffer”中。但是，等待其它 CPU 对“invalidate”应答可能会需要很长时间。其它 CPU 在接收到“invalidate”信号后并不一定马上进行处理，因为这时这个 cache 可能正在使用。为了对“invalidate”信号做出快速应答，现代 CPU 使用“invalidate queue”存储“invalidate”信号，并立即进行应答，然后在适当的时候再对该“invalidate”信号进行处理。这就导致了读乱序。所以提供了 `smp_mb()` 内存屏障，保证后续指令都等待“invalidate queue”中的信号被处理完毕后才被执行。

- 4.2 `mb()` 适用于 MP 或 UP 系统与外设之间的内存屏障。通过使用 lock 指令保证内存和 cache 之间的同步。
- 4.3 `rmb()` 适用于 MP 或 UP 系统与外设之间的读内存屏障。通过使用 lock 指令保证内存和 cache 之间的同步。
- 4.4 `wmb()` 适用于 MP 或 UP 系统与外设之间的写内存屏障。通过使用 lock 指令保证内存和 cache 之间的同步。
- 4.5 `smp_mb()` 仅适用于 MP 之间的内存屏障。由于 x86 不会对先写后读重新排序，但会对先读后写重新排序，所以 `smp_mb()` 被定义为 `mb()`。
- 4.6 `smp_rmb()` 仅适用于 MP 之间的读内存屏障。x86 中没有“invalidate queue”，因此它不会对两个读操作重新排序（又有 MESI）。所有 `smp_rmb()` 仅仅是编译器屏障。（在一些老的 CPU 上，两个读操作可能会被从新排序，这时 `smp_rmb` 被定义为 `rmb()`）。
- 4.7 `smp_wmb()` 仅适用于 MP 之间的写内存屏障。x86 中没有“store buffer”，因此它不会对两次写操作重新排序（又有 MESI）。所有 `smp_wmb()` 仅仅是编译器屏障。（在某些特殊的 CPU 上是会有 store buffer 的，这时 `smp_wmb()` 被定义为 `wmb()`）。

总结：
写屏障（***_wmb()）就是保证写屏障之前的写指令全部完成，并且通知到了所有其它 CPU，即“store buffer”中等待信号的指令都接收到信号，并完成修改。
读屏障（***_rmb()）就是保证读屏障之后的指令，引用的都是最新的数据，即“invalidate queue”中的信号都已被处理（其它 CPU 写屏障之前修改的数据都已在本 CPU 上体现出来）

每 CPU 变量

- 1 每 CPU 变量在单核中是没有意义的，只有在多核中才有意义。
- 2 最好的同步技术是把设计不需要同步的内核放在首位，每一种显示的同步原语都有不容忽视的性能开销。
- 3 最简单也是最重要的同步技术包括把内核变量声明为每 CPU 变量。每 CPU 变量主要是数据结构的数组，系统的每个 CPU 对应数组的一个元素。同时它可减少 CPU 缓存失效。
- 4 一个 CPU 不应该访问与其它 CPU 对应的数组元素，另外，它可以随意读或修改它自己的元素而不用担心出现竞争条件，因为它是唯一有资格这么做的 CPU。但是，这也意味着每 CPU 变量基本上只能在特殊情况下使用，也就是当它确定在系统的 CPU 上的数据在逻辑上是独立的。只有这个处理器能访问这个数据的规则纯粹是一个编程约定。你需要确保本地处理器只会访问它自己的唯一数据。系统本身并不存在任何措施禁止你从事的欺骗活动。
- 5 虽然每 CPU 变量为来自不同 CPU 的并发访问提供保护，但对来自异步函数（中断处理程序和可延迟函数）的访问不提供保护，在这种情况下需要另外的同步原语。
- 6 使用每个 CPU 数据会省去许多（或最小化）数据上锁，它唯一的安全要求就是要禁止内核抢占。而这点代价相比上锁要小得多，而且接口会自动帮你做这个步骤（因为在获取当前处理器号时需要调用 `get_cpu()` 函数，而该函数会自动调用 `preempt_disable()` 禁止内核抢占。相应的 `put_cpu()` 也会自动调用 `preempt_enable()` 允许内核抢占）。每个 CPU 数据在中断上下文或进程上下文中使用都很安全。但要注意，你不能再访问每个 CPU 数据过程中睡眠，否则，你就又可能醒来后已经到了其它处理器了。

原子操作

- 1 原子操作保证在单核和多核系统都能正确执行。并且执行效果相同。
- 2 原子操作可以确保指令以原子的方式执行→执行过程不被打断。
- 3 内核提供两组原子操作接口：一组对整数进行操作，另一组对单独的位进行操作。在 linux 支持的所有体系结构上都实现了这两组接口。大多数体系结构要么本来就支持简单的原子操作，要么就为单步执行提供了锁内存总线的指令。

3.1 整数的原子操作

- 3.1.1 针对整数的原子操作只能对 `atomic_t` 类型的数据进行处理。
- 3.1.2 如果需要将 `atomic_t` 类型转换为 `int` 型，可以使用 `atomic_read()`来完成。
- 3.1.3 原子整数操作函数是 `atomic_*`()。
- 3.1.4 原子整数操作最常见的用途就是实现计数器。
- 3.1.5 原子性确保指令执行期间不被打断，要么全部执行完，要么根本不执行。原子操作只保证原子性，顺序性通过屏障指令来实施。

3.2 原子位操作

- 3.2.1 位操作函数是对普通的内存地址进行操作的。它的参数是一个指针和一个位号，第 0 位是给定地址的最低有效位。在 32 位机上，第 31 位是给定地址的最高有效位，而第 32 位是下一个字的最低有效位。虽然使用原子操作在多数情况下是对一个字长的内存进行访问，因而位号应该位于 0~31 之间（在 64 位机器中是 0~63 之间），但是，对位号的范围并没有限制。
- 3.2.2 由于原子位操作是对普通的指针进行的操作，所有不像原子整形对应的 `atomic_t`，这里没有特殊的数据类型。相反，只要指针指向了任何你希望的数据，你就可以对他进行操作。
- 3.2.3 原子位函数形式是 `*_bit(int nr, void *addr)`。
- 3.2.4 为方便起见，内核还提供了一组与上述原子操作对应的非原子位函数。非原子位函数与原子位函数的操作完全相同，但它不保证原子性，且名字前缀多两个下划线。如果不需要原子性操作，那么这些非原子的位函数相比原子的位函数可能会执行的更快些。

自旋锁

1 下面介绍的自旋锁，在单核中编译为空，如果内核开启了抢占机制，则被编译为设置内核抢占机制是否被启用的开关。在多核中，自旋锁是用户互斥多个 CPU 之间共同访问同一数据的锁机制，同时它也屏蔽内核抢占。（自旋锁具有内存屏障功能，所以在多核中使用自旋锁时，可以不必使用内存屏障）

2 普通自旋锁

- 2.1 在单处理器机器上，编译的时候并不会加入自旋锁。它仅仅被当做一个设置内核抢占机制是否被启用的开关。如果禁止内核抢占，那么在编译时自旋锁会被完全剔除出内核。
- 2.2 多核中普通自旋锁是用户互斥多个 CPU 之间共同访问同一数据的锁机制，同时它也禁止内核抢占。它应该注意以下几点：
 - 2.2.1 自旋锁是不可递归的。
 - 2.2.2 持有自旋锁后不能睡眠，并且自旋锁会自动禁止内核抢占。
 - 2.2.3 加锁的一个大原则是：针对代码加锁会使得程序难以理解，并且容易引发竞争条件，正确的做法应该是对数据而不是代码加锁。

3 读/写自旋锁

- 3.1 读/写自旋锁的引入是为了增加内核的并发能力，当读者多而写者少时，它会提高效率。因为多个读者可同时获得读锁。
- 3.2 无论读锁还是写锁都会禁止内核抢占。
- 3.3 不能把一个读锁“升级”为写锁，如：`read_lock(&mr_rwlock); write_lock(&mr_rwlock);` 将会带来死锁，因为写锁会不断自旋，等待所有的读者释放锁，其中包括它自己。所以当确实需要写操作时，要在一开始就请求写锁。如果写和读不能清晰的分开的话，那么使用一般的自旋锁就行了，不要使用读/写自旋锁。
- 3.4 多个读者可以安全的获得同一读锁，事实上，即使一个线程递归的获得同一读锁也是安全的。
- 3.5 大量读者必定会使挂起的写者处于饥饿状态，在设计锁时一定要记住这一点。

4 顺序自旋锁

- 4.1 Linux2.6 中引入了顺序锁（`seqlock`），它与读/写自旋锁非常相似，只是它为写赋予了较高的优先级；事实上，即使在读者正在读的时候也允许写者继续运行。这种策略的好处是写者永远不会等待（除非另外一个写者正在写），缺点是有些时候读者不得不反复多次读相同的数据直到它获得有效的副本。
- 4.2 注意，读顺序锁不禁用内核抢占，也不必禁用内核抢占；另一方面，由于写者获取自旋锁，所以它自动禁用内核抢占。
- 4.3 并不是每一种资源都可以使用顺序锁来保护。一般来说，必须在满足下述条件时才能使用顺序锁：
 - 4.3.1 被包含的数据结构不包括被写者修改和被读者间接引用的指针（否则，写者可能在读者的眼鼻下就修改指针）。
 - 4.3.2 读者的临界区代码没有副作用（否则，多个读者的操作会与单独的读操作有不同的结果）。
- 4.4 读者的临界区代码应该简短，而且写者应该不常获取顺序锁，否则，反复的读访问会引起严重的开销。

RCU 互斥

- 1 RCU(Read-Copy Update)，顾名思义就是读-拷贝修改，它是基于其原理命名的。对于被 RCU 保护的共享数据结构，读者不需要获得任何锁就可以访问它，但写者在访问它时首先拷贝一个副本，然后对副本进行修改，最后使用一个回调（`callback`）机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据。这个时机就是所有引用该数据的 CPU 都退出对共享数据的操作。
- 2 RCU 实际上是一种改进的 `rwlock`，读者几乎没有什么同步开销，它不需要锁，不使用原子指令，而且在除 alpha 的所有架构上也不需要内存栅（`Memory Barrier`），因此不会导致锁竞争，内存延迟以及流水线停滞。不需要锁也使得使用更容易，因为死锁问题就不需要考虑了。写者的同步开销比较大，它需要延迟数据结构的释放，复制被修改的数据结构，它也必须使用某种锁机制同步并行的其它写者的修改操作。
- 3 RCU 与 `rwlock` 的不同之处是：它既允许多个读者同时访问被保护的数据，又允许多个读者和多个写者同时访问被保护的数据（注意：是否可以有多个写者并行访问取决于写者之间使用的同步机制），读者没有任何同步开销，而写者的同步开销则取决于使用的写者间同步机制。但 RCU 不能替代 `rwlock`，因为如果写比较多时，对

读者的性能提高不能弥补写者导致的损失。

4 读者在访问被 RCU 保护的共享数据期间不能被阻塞，这是 RCU 机制得以实现的一个基本前提，也就说当读者在引用被 RCU 保护的共享数据期间，读者所在的 CPU 不能发生上下文切换，spinlock 和 rwlock 都需要这样的前提。写者在访问被 RCU 保护的共享数据时不需要和读者竞争任何锁，只有在有多于一个写者的情况下需要获得某种锁以与其他写者同步。写者修改数据前首先拷贝一个被修改元素的副本，然后在副本上进行修改，修改完毕后它向垃圾回收器注册一个回调函数以便在适当的时机执行真正的修改操作。等待适当时机的这一时期称为 grace period，而 CPU 发生了上下文切换称为经历一个 quiescent state，grace period 就是所有 CPU 都经历一次 quiescent state 所需要的等待的时间。垃圾收集器就是在 grace period 之后调用写者注册的回调函数来完成真正的数据修改或数据释放操作的。

5 读者 RCU API（对于读者，RCU 仅需要抢占失效？？？）

5.1 void rcu_read_lock (void)
void rcu_read_unlock (void)

5.2 void rcu_read_lock_bh (void)
void rcu_read_unlock_bh (void)

这两个只在修改是通过 call_rcu_bh 进行的情况下使用，因为 call_rcu_bh 将把 softirq 的执行完毕也认为是一个 quiescent state，因此如果修改是通过 call_rcu_bh 进行的，在进程上下文的读端临界区必须使用这一变种

6 写者 RCU API

6.1 void synchronize_rcu (void)

该函数由 RCU 写端调用，它将阻塞写者，直到经过 grace period 后，即所有的读者已经完成读端临界区，写者才可以继续下一步操作。如果有多个 RCU 写端调用该函数，他们将在一个 grace period 之后全部被唤醒。

6.2 void synchronize_sched (void)

该函数用于等待所有 CPU 都处在可抢占状态，它能保证正在运行的中断处理函数处理完毕，但不能保证正在运行的 softirq 处理完毕。注意，synchronize_rcu 只保证所有 CPU 都处理完正在运行的读端临界区。

6.3 void call_rcu (struct rcu_head *head, void (*func)(struct rcu_head *rcu))

该函数不会使写者阻塞，因而可以在中断上下文或 softirq 使用，而 synchronize_rcu 和 synchronize_sched 只能在进程上下文使用。该函数将把函数 func 挂接到 RCU 回调函数链上，然后立即返回。一旦所有的 CPU 都已经完成端临界区操作，该函数将被调用来释放删除的将绝不在被应用的数据。参数 head 用于记录回调函数 func，一般该结构会作为被 RCU 保护的数据结构的一个字段，以便省去单独为该结构分配内存的操作。需要指出的是，函数 synchronize_rcu 的实现实际上使用函数 call_rcu。

6.4 void call_rcu_bh (struct rcu_head *head, void (*func)(struct rcu_head *rcu))

该函数功能几乎与 call_rcu()完全相同，唯一差别就是它把 softirq 的完成也当作经历一个 quiescent state，因此如果写端使用了该函数，在进程上下文的读端必须使用 rcu_read_lock_bh。

6.5 多个写者之间互斥使用 spin_lock 之类的锁，修改数据使用 synchronize_kernel()和 call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg)等函数注册一个回调函数在适当的时候进行写操作。

7 RCU 增加指针操作的 RCU 版本

在 SMP 系统中，当对同一个全局指针变量赋值不同地址时（即操作一个全局变量），就要通过某种形式的锁加以保护（spin_lock，或 rw_lock）。RCU 利用内存栅对指针提供了这样的保护操作，使得读和写者之间不需要锁（但读者要用 rcu_read_lock()将读临界区保护起来，使 CPU 不能发生上下文切换），但多个写者之间需要某些形式的锁加以保护。

7.1 #define rcu_assign_pointer(p, v)

该函数把地址 v 赋值给指针 p，使用内存栅保证了新指针的修改对所有读者是可见。它的内部使用了 smp_wmb()写内存屏障（store buffer 中指令都执行完）保证其之前操作在通知到其它 CPU 之后才更新 p=v 指针（即 v 赋值给 p 之前，v 的更新要完成），而相应的读者如 rcu_dereference(p)使用读内存屏障（invalidate queue 队列中信号被处理完），这就保证了指针变量的一致性。即如果看到 v，则一定是 v 更新后的值，否则是 p 之前的值。如果是将 p 指针设为 NULL，也就是删除指针指向的变量，这就要求调用者不能立刻释放被移走的 v，必须调用 synchronize_rcu()或 call_rcu()来延迟释放直到 RCU 经历了一个 grace period 时间。

7.2 #define rcu_dereference (p)

该宏用于在 RCU 读端临界区（使用 rcu_read_lock()保护的区域）获得一个 RCU 保护的指针，该指针可以在以后安全地引用。它内部使用读内存屏障（invalidate queue 队列中信号被处理完），与 rcu_assign_pointer ()的写内存屏障相对应，这就保证了指针变量的一致性。即如果看到 v，则一定是 v 更新后的值，否则是 p 之前的值。

8 RCU 增加了链表操作的 RCU 版本。

在 SMP 系统中，对同一链表同时有读操作和写操作，就需要通过某种形式的锁加以保护（spin_lock，或 rw_lock）。RCU 利用内存栅对链表提供了这样的保护操作，使得读和写者之间不需要锁（但读者要用 rcu_read_lock()将读临界区保护起来，使 CPU 不能发生上下文切换），但多个写者之间需要某些形式的锁加以保护。

8.1 void list_add_rcu (struct list_head *new, struct list_head *head)

该函数把链表项 new 插入到 RCU 保护的链表 head 的开头。使用内存栅保证了在引用这个新插入的链表项之前，新链表项的链接指针的修改对所有读者是可见的。它的内部主要使用了 smp_wmb()写内存屏障（store buffer 中指令都执行完）保证 new->next=next 和 new->prev=prev 指针更新都通知到其它 CPU 之后才更新 prev->next=new 和 next->prev=new 指针，而相应的读者如 list_for_each_entry_rcu()使用读内存屏障（invalidate queue 队列中信号被处理完），这就保证了链表的统一性，要么其它 CPU 看到 new 被加入到链表中，要么看不到 new。

8.2 void list_add_tail_rcu (struct list_head *new, struct list_head *head)

该函数类似于 list_add_rcu，它将把新的链表项 new 添加到被 RCU 保护的链表的末尾。

8.3 void list_del_rcu (struct list_head *entry)

该函数从 RCU 保护的链表中移走指定的链表项 entry，并且把 entry 的 prev 指针设置为 LIST_POISON2，但是并没有把 entry 的 next 指针设置为 LIST_POISON1，因为该指针可能仍然在被读者用于便利该链表。list_del_rcu()没有任何形式的内存屏障，这样它在修改完 next->prev=prev 和 prev->next=next 指针后，不需要等待 store buffer 指令执行完毕就执行其它指令。这对于其它读者和写者没有影响，它们要么看到这个被移走的 entry，要么看不到。这也就要求调用者不能立刻释放被移走的 entry，必须调用 synchronize_rcu()或 call_rcu()来延迟释放直到 RCU 经历了一个 grace period 时间。

8.4 void list_replace_rcu (struct list_head *old, struct list_head *new)

它使用新的链表项 new 取代旧的链表项 old，内存栅保证在引用新的链表项之前，它的链接指针的修正对所有读者可见。原理与 list_add_rcu()类似。

- 8.5 `#define list_for_each_entry_rcu (pos, head, member)`
该宏用于遍历由 RCU 保护的链表 head，只要在读端临界区（使用 `rcu_read_lock()` 保护的区域）使用该函数，它就可以安全地和其它_rcu 链表操作函数（如 `list_add_rcu`）并发运行。它内部使用读内存屏障（`invalidate queue` 队列中信号被处理完），与 `list_add_rcu()` 的写内存屏障相对应，这就保证了链表的统一性，使得看到的 new 要么是完整的添加到链表中，要么根本看不到 new。
- 8.6 `#define list_for_each_entry_continue_rcu (pos, head, member)`
该宏用于在退出点之后继续遍历由 RCU 保护的链表 head。
- 8.7 `#define __list_for_each_rcu (pos, head)`
 `#define list_for_each_continue_rcu (pos, head)`
- 8.8 `void hlist_del_rcu (struct hlist_node *n)`
它从由 RCU 保护的哈希链表中移走链表项 n，并设置 n 的 ppre 指针为 LIST_POISON2，但并没有设置 next 为 LIST_POISON1, 因为该指针可能被读者使用用于遍历链表。原理与 `list_del_rcu()` 类似。
- 8.9 `void hlist_add_head_rcu(struct hlist_node *n, struct hlist_head *h)`
该函数用于把链表项 n 插入到被 RCU 保护的哈希链表的开头，但同时允许读者对该哈希链表的遍历。内存栅确保在引用新链表项之前，它的指针修正对所有读者可见。原理与 `list_add_rcu()` 类似。
- 8.10 `#define __hlist_for_each_rcu (pos, head)`
该宏用于遍历由 RCU 保护的哈希链表 head，只要在读端临界区（使用 `rcu_read_lock()` 保护的区域）使用该函数，它就可以安全地和其它_rcu 哈希链表操作函数（如 `hlist_add_rcu`）并发运行。原理与 `list_for_each_entry_rcu()` 类似。
- 8.11 `#define hlist_for_each_entry_rcu (tpos, pos, head, member)`
类似于 `hlist_for_each_rcu`，不同之处在于它用于遍历指定类型的数据结构哈希链表，当前链表项 pos 为一包含 `struct list_head` 结构的特定的数据结构。

信号量

- 1 信号量在单核和多核工作都是一样的。
- 2 Linux 中的信号量是一种睡眠锁。如果有一个任务试图获得一个已经被占用的信号量时，信号量会将其推进一个等待队列，然后让其睡眠。这时处理器能重获自用，从而去执行其他代码。当持有信号量的进程将信号量释放后，处于等待队列中的那个任务将被唤醒，并获得该信号量。
- 3 信号量的获取内部使用 `spin_lock_irqsave()` 保护，所以它能够原子的获取，不会被任何事情打断。
- 4 由于争用信号量的进程在等待锁重新变为可用时会睡眠，所以信号量适用于锁会被长时间持有的情况。
- 5 相反，锁被短时间持有时，使用信号量就不太适宜了。因为睡眠、维护等待队列以及唤醒所花费的开销可能比锁被占用的全部时间还要长。
- 6 由于执行线程在锁被争用时会睡眠，所以只能在进程上下文中才能获取信号量锁，因为在中断上下文是不能进行调度的。
- 7 你可以在持有信号量时去睡眠（当然你也可能并不需要睡眠），因为当其他进程试图获得同一信号量时不会因此而死锁（因为该进程也只是去睡眠而已，而你最终会继续执行的）。
- 8 在你占用信号量的同时不能占用自旋锁。因为在你等待信号量时可能会睡眠，而在持有自旋锁时是不允许睡眠的。
- 9 如果需要在自旋锁和信号量中做选择，应该根据锁被持有的时间长短做判断。同时在中断上下文中只能使用自旋锁，而在任务睡眠时只能使用信号量。
- 10 另外，信号量不同于自旋锁，它不会禁止内核抢占，所以持有信号量的代码可以被抢占。这意味着信号量不会对调度的等待时间带来负面影响。
- 11 信号量有一个有用的特性，它可以同时允许任意数量的锁持有者，而自旋锁在一个时刻最多允许一个任务持有它。信号量同时允许的持有者数量可以在声明信号量时指定。

代码执行环境

- 1 当内核启动初始化完成后，所有 CPU 都执行在进程的地址空间中，无论是在进程上下文，还是在中断上下文。
- 2 启动后，所有 CPU（无论单核或多核），在任何时间点上的活动范围都是下面三者之一：

2.1 运行于用户空间，处于进程上下文。

2.2 运行于内核空间，处于进程上下文。代表某个特定的进程执行。

2.3 运行于内核空间，处于中断上下文。与任何进程无关，处理某个特定中断。

中断处理程序（内核中断上下文）

- 1 linux 的中断处理程序是无需重入的。当一个给定的中断处理程序正在执行时，相应的中断线在所有处理器上都会被屏蔽掉，以防止在同一中断线上接收另一个新的中断。通常情况下，所有其他的中断都是打开的，所以这些不同中断线上的其他中断都能被处理，但当前中断线总是被禁止的。所以，同一个中断处理程序绝对不会被同时调用以处理嵌套的中断。但如果多个中断共享相同的数据，则要考虑互斥。
- 1.1 单核：同一个中断线的处理程序不需考虑重入。但不同中断线处理程序使用共享数据，则要使用 `local_irq_disable()` 等函数屏蔽当前 CPU 中断。
- 1.2 多核：同一个中断线的处理程序不需考虑重入。但不同中断线处理程序使用共享数据，则要使用 `spin_lock_irqsave()` 和 `spin_lock_irqstore()` 函数（因为不知当前是否允许中断）屏蔽当前 CPU 中断并加互斥锁。这样在当前 CPU 上不会再被其它中断打断，并且也能保证与其它 CPU 互斥（无论其它 CPU 是执行在中断上下

文还是进程上下文)。

软中断处理程序（内核中断上下文）

1 软中断处理程序是在中断上下文中进行处理的，软中断在以下情况下会被调度：

- 1.1 从一个硬件中断代码处返回时。（软中断在被中断进程地址空间的中断上下文中执行）
- 1.2 在 ksoftirqd 内核线程中。（软中断在该进程地址空间的中断上下文中执行）

2 同种类型的两个软中断可以同时运行在一个系统的多个处理器上。但是，同一处理器上的一个软中断绝不会抢占另一个软中断，因此，根本没必要禁止下半部。

- 2.1 **单核**：什么也不需要做。
- 2.2 **多核**：使用 `spin_lock()`或 `rwlock()`等函数屏蔽对全局变量的竞态访问。

3 软中断处理程序会被中断处理程序给打断，当软中断处理程序和中断处理程序共享数据时。

- 3.1 **单核**：使用 `local_irq_disable()`等函数可屏蔽中断的竞争。（它只能关闭当前处理器的中断）
- 3.2 **多核**：使用 `spin_lock_irq()`或 `spin_lock_irqsave()`等函数屏蔽中断的同时加互斥锁。（它只能关闭当前处理器的中断）
注意：一定要在获取锁之前，首先禁止本地中断，否则，中断处理程序就会打断正持有锁的内核代码，有可能会试图去争用这个已被持有的自旋锁。
注意：需要关闭的只是当前处理器上的中断。如果中断发生在不同的处理器上，即使中断处理程序在同一锁上自旋，也不会妨碍锁的持有者（在不同处理器上）最终释放锁。

内核抢占（内核进程上下文）

1 同一内核进程上下文可能同时在多个 CPU 上执行，可使用 `spin_lock()`等函数屏蔽多核竞争。

- 1.1 **单核**：`spin_lock()`仅仅禁止内核抢占。
- 1.2 **多核**：`spin_lock()`会加锁防止多个 CPU 同时访问同一数据。

2 在内核进程上下文可能被软中断给打断，当进程上下文和软中断共享数据时：

- 2.1 **单核**：使用 `local_bh_disable()`等函数屏蔽软中断。（它只能关闭当前处理器的软中断）
- 2.2 **多核**：使用 `spin_lock_bh()`等函数屏蔽软中断的同时加互斥锁。（它只能关闭当前处理器的软中断）
注意：一定要在获取锁之前，首先禁止本地软中断，否则，软中断处理程序就会打断正持有锁的内核代码，有可能会试图去争用这个已被持有的自旋锁。
注意：需要关闭的只是当前处理器上的软中断。如果软中断发生在不同的处理器上，即使软中断处理程序在同一锁上自旋，也不会妨碍锁的持有者（在不同处理器上）最终释放锁。

3 在内核进程上下文可能被中断给打断，当进程上下文和中断共享数据时：

- 3.1 **单核**：使用 `local_irq_disable()`等函数屏蔽中断。（它只能关闭当前处理器的中断）
- 3.2 **多核**：使用 `spin_lock_irq()`或 `spin_lock_irqsave()`等函数屏蔽中断的同时加互斥锁。（它只能关闭当前处理器的中断）
注意：一定要在获取锁之前，首先禁止本地中断，否则，中断处理程序就会打断正持有锁的内核代码，有可能会试图去争用这个已被持有的自旋锁。
注意：需要关闭的只是当前处理器上的中断。如果中断发生在不同的处理器上，即使中断处理程序在同一锁上自旋，也不会妨碍锁的持有者（在不同处理器上）最终释放锁。

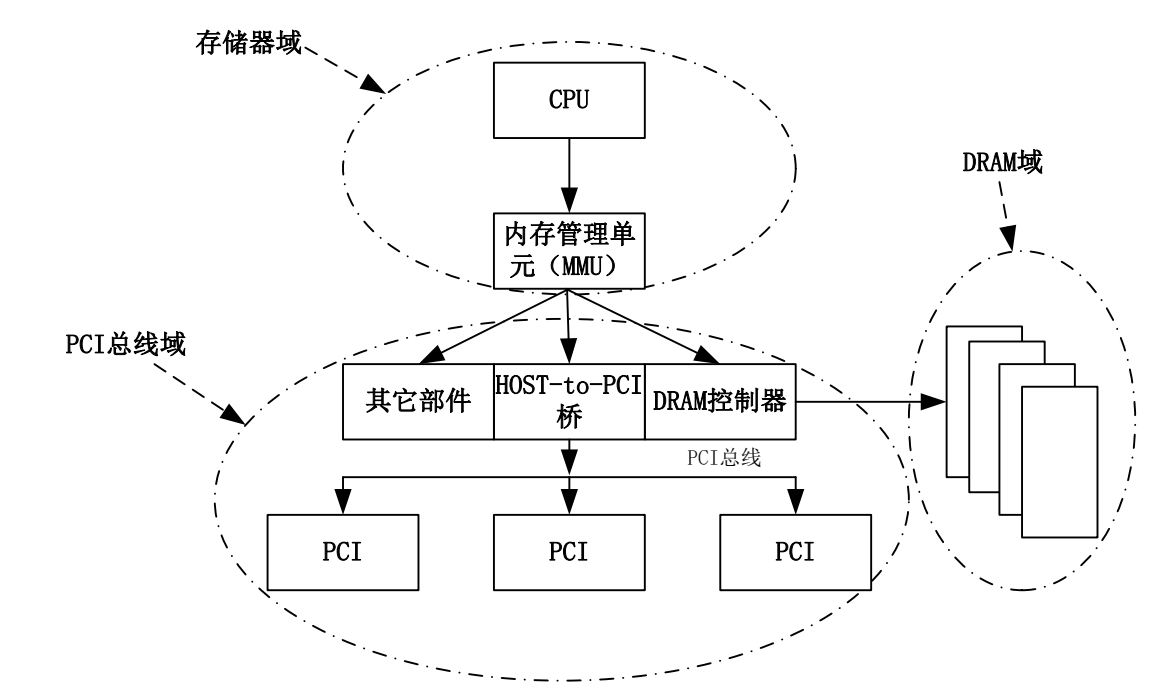
4 在内核进程上下文可能被抢占。

- 4.1 内核抢占在以下情况时发生
 - 4.1.1 从中断处理程序（软中断或硬中断）返回到内核空间之前（在中断处理程序中，内核是不可抢占的）。
 - 4.1.2 内核在进程上下文释放 `spinlock` 锁时（如果自旋锁被持有，内核是不可抢占的，同时也不能发生上下文切换）。
 - 4.1.3 内核在进程上下文调用 `perrmpt_enable()`函数允许内核抢占时。
 - 4.1.4 当内核任务中显示的调用 `schedule()`时。
 - 4.1.5 内核任务阻塞时。
- 4.2 `preempt_enable()`或 `preempt_disable()`函数用于允许或禁止内核抢占。
- 4.3 上面用于防止竞争的 `spin_lock()`、`spin_lock_*`()、`local_irq_disable()`、`local_bh_disable()`等函数自动禁止内核抢占。

内存管理（x86 系统）

硬件地址的基本概念

- DRAM 域地址**：是 DRAM 控制器所能访问的地址空间集合。
- PCI 总线域地址**：是 PCI 设备所能直接访问的地址空间集合。
- 存储器域地址**：是 CPU 所能访问的地址空间集合。
- CPU 访问 DRAM 域或 PCI 总线域地址空间时，都需要进行地址转换（将存储器域地址转换为相应域的地址）。例如：CPU 访问 DRAM 域时，需要进行存储器域地址空间到 DRAM 域地址空间的转换（由 DRAM 控制器完成）；CPU 访问 PCI 总线域时，需要进行存储器域地址空间到 PCI 总线域地址空间的转换（由 HOST 主桥完成）。在 x86 处理器系统中，会将 DRAM 域和 PCI 总线域映射到存储器域空间中，并且其大多数 DRAM 域中的地址与存储器域中的地址一一对应而且相等，而存储器域的 PCI 地址与 PCI 总线域的地址也一一对应而且相等。它们在存储器域空间的映射彼此独立，互不冲突，映射关系由 BIOS 提供（e820 地址映射表）。
- PCI 设备访问 DRAM 域地址空间时，首先要经过 HOST 主桥将 PCI 总线域地址转换为存储器域地址，然后再由 DRAM 控制器将存储器域地址转换为 DRAM 域地址。



软件地址的基本概念

- 逻辑地址**：是一个 32 位长的地址。所有进程地址都使用逻辑地址。
- 线性地址（也称为虚拟地址）**：是一个 32 位长地址，可以用来表示高达 4G 的地址，值的范围从 0x00000000 到 0xffffffff。它是通过分段单元的硬件电路将逻辑地址转换为线性地址，即将逻辑地址加上一个段起始地址就得到了线性地址。在 linux 中，所有的段都从 0x00000000 开始，所以在 linux 下逻辑地址等于线性地址，即进程地址也是线性地址。后面我们就不再讨论逻辑地址，而都使用线性地址代表进程所使用的地址。
- 物理地址**：是通过分页单元的硬件电路将线性地址转换为物理地址。该地址就是上面所说的存储器域地址，是 CPU 所能访问的地址空间的集合。
- linux 启动后通过 BIOS 获得 e820 存储器域地址图表，如下面所示（这是一个包含 4G DRAM 设备的 BIOS-e820 图表）：

```
BIOS-provided physical RAM map:
BIOS-e820: 0000000000000000 - 000000000009ec00 (usable)
BIOS-e820: 000000000009ec00 - 00000000000a0000 (reserved)
BIOS-e820: 00000000000e0000 - 0000000000010000 (reserved)
BIOS-e820: 0000000000010000 - 000000000002000000 (usable)    (512M)
BIOS-e820: 000000000002000000 - 000000000002020000 (reserved)
BIOS-e820: 000000000002020000 - 000000000004000000 (usable)    (512M)
BIOS-e820: 000000000004000000 - 000000000004020000 (reserved)
BIOS-e820: 000000000004020000 - 00000000000bad59000 (usable)    (2048M)
BIOS-e820: 00000000bad59000 - 00000000bada6000 (ACPI NVS)
BIOS-e820: 00000000bada6000 - 00000000badae000 (ACPI data)
BIOS-e820: 00000000badae000 - 00000000badc1000 (reserved)
BIOS-e820: 00000000badc1000 - 00000000badc2000 (ACPI NVS)
BIOS-e820: 00000000badc2000 - 00000000badd3000 (reserved)
BIOS-e820: 00000000badd3000 - 00000000badd6000 (ACPI NVS)
BIOS-e820: 00000000badd6000 - 00000000badf6000 (reserved)
BIOS-e820: 00000000badf6000 - 00000000badf8000 (usable)
BIOS-e820: 00000000badf8000 - 00000000bae09000 (reserved)
BIOS-e820: 00000000bae09000 - 00000000bae16000 (ACPI NVS)
BIOS-e820: 00000000bae16000 - 00000000bae3b000 (reserved)
BIOS-e820: 00000000bae3b000 - 00000000bae7e000 (ACPI NVS)
BIOS-e820: 00000000bae7e000 - 00000000bb000000 (usable)
```



```
BIOS-e820: 00000000bb800000 - 00000000bfa00000 (reserved)
BIOS-e820: 00000000fed1c000 - 00000000fed40000 (reserved)
BIOS-e820: 00000000ff000000 - 0000000100000000 (reserved)
BIOS-e820: 0000000100000000 - 000000013fe00000 (usable)    (1024M)
```

- 4.1 内核根据这个图表获得可用物理内存大小，并建立一个 page 数组（每个 page 项管理 4K 内存）对所有可用内存进行管理。同时根据 Memory split（用户/内存线性地址空间的分配比例，我们的是 1G/3G user/kernel split）将 page 数组划分为不同的管理区（DMA、NORMAL、HIGHMEM），并为 DMA、NORMAL 区的内存建立好页目录页表的映射关系。
- 4.2 例如上图 0~3G 存储器域地址大部分用于映射 DRAM 域（也有部分被保留做其它用途，在 page 数组中对保留区管理的 page 结构会标识为不可用），3G~4G 被用作其它用途，4G~5G 存储器域地址被用于映射 DRAM 域（我们的设备使用了 4G DRAM）。我们可以得到以下几点：

4.2.1 前 3G 存储器域地址都被用于映射 DRAM 域，是为了内核线性地址空间能够多映射一些 DRAM 空间。因为内核线性地址空间通过页目录/页表映射为存储器域地址，而被映射的这部分存储器域地址映射的 DRAM 域越多，则意味着内核线性地址空间映射的 DRAM 空间越多。

4.2.2 由于最高 1G 的 DRAM 地址被 4G~5G 的存储器域地址所映射，所以为了能够访问最高 1G 的 DRAM 空间，我们必须打开 PAE 选项，该选项会使 CPU 的寻址空间增加到 36 位，即 CPU 可访问 64G 存储器域地址。

4.2.3 由于我们为内核分配的 3G 线性地址空间，而这其中还有 512M（通过内核启动命令 vmalloc=512M 设置的）被用于 VMALLOC，所以内核只有 2.5G 线性地址用于映射存储器域地址，也就是说只有 2.5G 内存被 DMA、NORMAL 区管理，其余 1.5G 内存被 HIGHMEM 区管理（这个区的内存还未进行页目录/页表映射）。

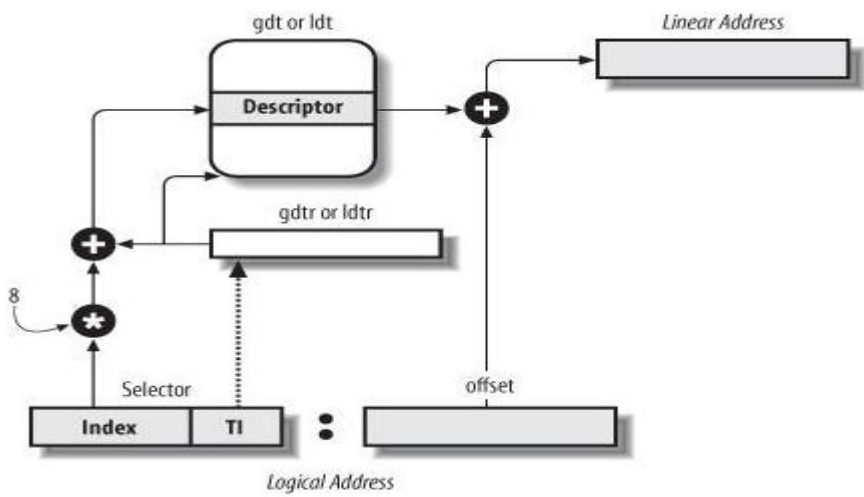
4.2.4 内存的管理与地址映射是两个不同概念。内存是由 page 结构进行管理的，每个 page 结构管理 4K 内存。地址映射（线性地址向存储器域地址映射）是由页目录/页表（它是 CPU 的硬件机制）完成的。对于映射，分为 4K 页和 4M 页的映射，4K 页映射需要通过页目录和一级页表进行映射，而 4M 页可通过页目录项直接映射。之所以要支持 4M 页映射，目的是为了减小 TLB miss。只要 CPU 支持 PSE（扩展分页）功能，则内核默认会使用 4M 页映射（但某些地址还必须使用 4K 页映射）。

进程与地址

- 1 在 X86 系统中，所有进程都使用的是逻辑地址，它要通过分段单元硬件电路转换为线性地址，再通过分页单元硬件电路转换为物理地址（即存储器域地址）。如下图所示：

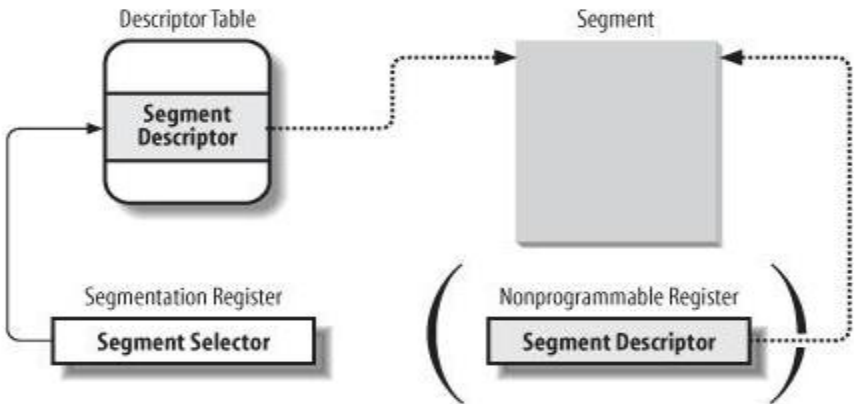


- 1.1 分段单元的逻辑如下图所示

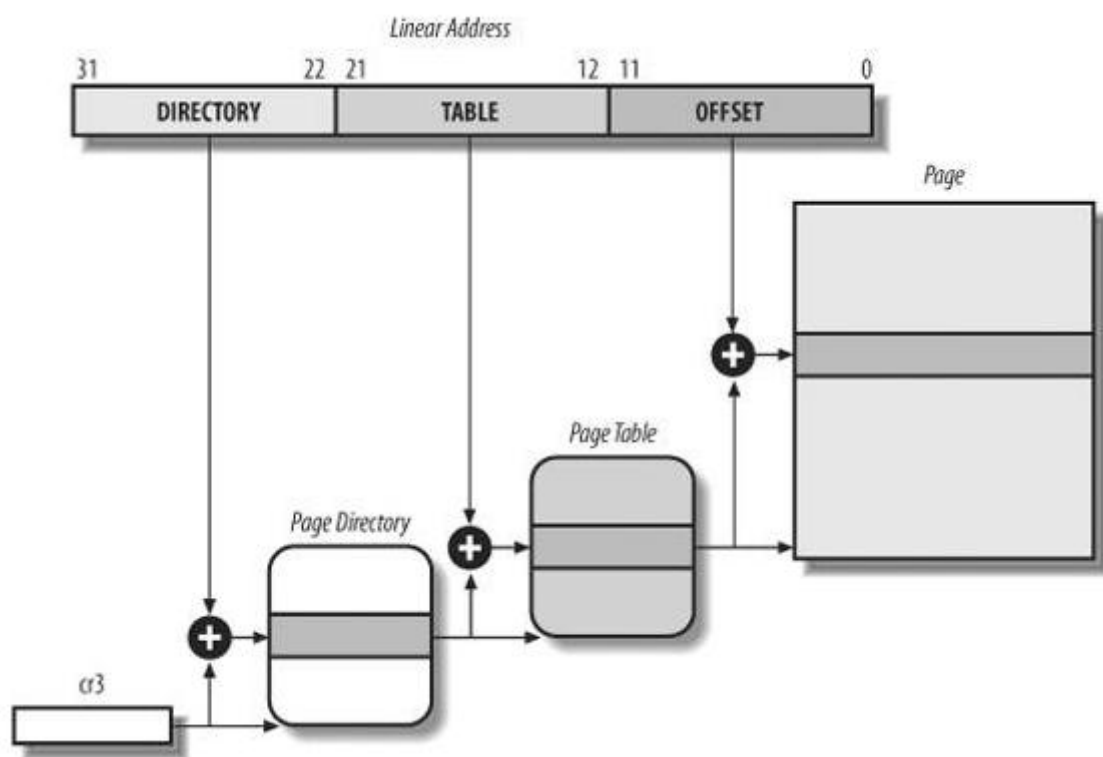


逻辑地址到线性地址的转换过程：CPU 根据段寄存器中的 TI 字段，以决定段描述符是保存在 GDT 全局描述符表中（gdtr 寄存器保存它的线性地址）还是在 LDT 局部描述符表中（ldtr 寄存器保存它的线性地址），然后用段寄存器的 index 字段在描述符表中（GDT 或 LDT）找到对应的段描述符，最后将逻辑地址与段描述符中的 base 字段值相加就得到了线性地址。

CPU 为加速逻辑地址到线性地址的转换，80x86 处理器提供了一种附加的非编程的寄存器（段描述符高速缓冲寄存器），它可存储 8 字节的段描述符。当装载 CPU 的段寄存器（CS 或 DS 等）时，相应的段描述符就由内存装入到对应的段描述符高速缓冲寄存器中，从那时起，针对那个段的逻辑地址向线性地址转换就可以不访问主存中的 GDT 了，处理器只需直接引用段描述符高速缓冲寄存器即可。仅当段寄存器内容改变时，才有必要访问 GDT 或 LDT。如下图所示：



- 1.2 分页单元的逻辑如下图所示



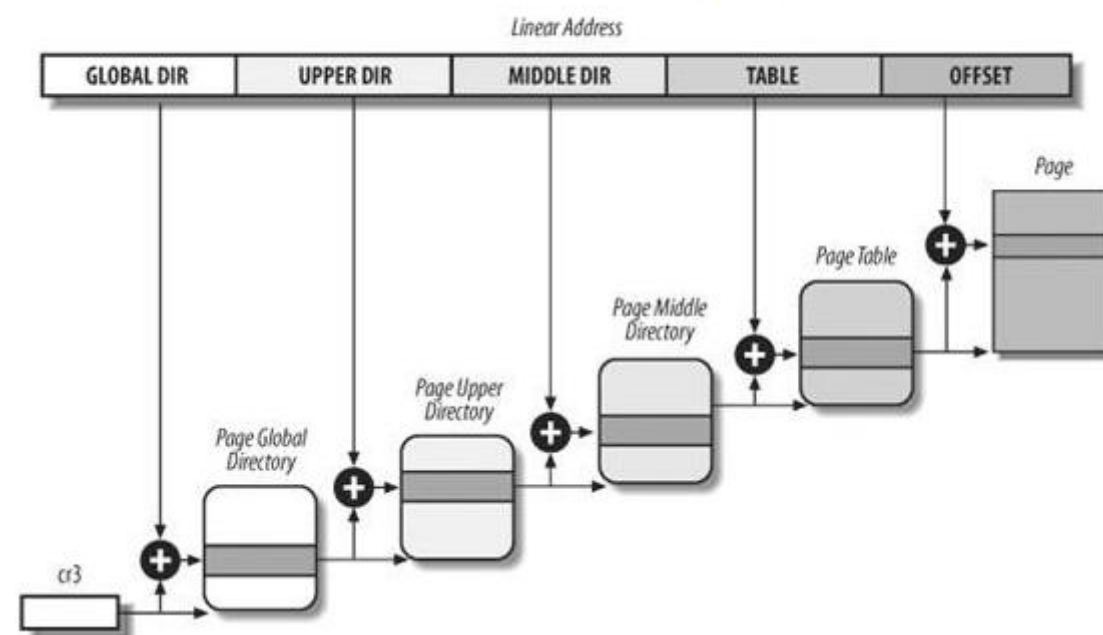
线性地址到物理地址的转换过程： CPU 将线性地址通过页目录和页表的映射方式进行地址转换。页目录的物理地址存放在控制寄存器 `cr3` 中。线性地址中的 `DIRECTORY` 字段决定页目录中的目录项，而且目录项指向适当的页表。线性地址的 `TABLE` 字段又决定页表中的表项，而表项含有页所在页框的物理地址。`OFFSET` 字段决定页框内的相对位置。也就是 `Page Table` 中 `table` 指定的表项中 `BASE` 地址加上线性地址中 `OFFSET` 字段就得到了物理地址。

CPU 为加速线性地址到物理地址的转换，80x86 处理器包含了一个称为 TLB（Translation Lookaside Buffer）的高速缓存用于加快线性地址的转换。当一个线性地址被第一次使用时，通过慢速访问 RAM 中的页目录/页表计算出相应的物理地址。同时将这个线性地址和映射的物理地址存放在一个 TLB 表项中，以便以后对同一个线性地址引用可以快速的得到转换。

在多处理器系统中，每个 CPU 都有自己的 TLB。与硬件高速缓存相反，TLB 中的对应项不必同步，这是因为运行在现有 CPU 上的进程可以使用同一线性地址与不同的物理地址发生联系。处理器不能自动同步它们自己的 TLB 高速缓存，因为决定线性地址和物理地址之间映射何时不再有效的是内核，而不是硬件。Intel 处理器提供了以下使 TLB 无效的技术：

1. 写 `cr3` 寄存器使处理器自动刷新相对于非全局页的 TLB 表项。
2. `invlpg` 汇编语言指令使映射指定线性地址的单个 TLB 表项无效。
3. 通过清除 `cr4` 的 `PGE` 标志禁用全局页，并再次将旧的 `cr4` 值写入 `cr4` 寄存器，达到刷新所有 TLB 表项（包括那些全局页对应的 TLB 表项，即那些 `Global` 标志被置位的页）。

Figure 2-12. The Linux paging model



linux 采用了一种同时适用于 32 位和 64 位系统的普通分页模型。对于 64 位系统需要采用四级分页模型，但是对于没有启用物理地址扩展的 32 位系统，两级页表已经足够了。linux 通过使“page upper directory”位和“page middle directory”位全为 0，从根本上取消了“page upper directory”和“page middle directory”字段。不过“page upper directory”和“page middle directory”在指针序列中的位置被保留，以便同样的代码在 32 位和 64 位系统下都能使用。内核为“page upper directory”和“page middle directory”保留了一个位置，这是通过把他们的页目录项设置为 1，并把这两个目录项映射到页全局目录的一个适当的目录项而实现的。

- 2 **GDT 全局描述符表（Global Descriptor Table）：** 用于存放 8 字节的段描述符，GDT 在主存中的地址和大小存放在 CPU 的 `gdtr` 控制寄存器中。在单处理器系统中只有一个 GDT，而在多处理器系统中每个 CPU 对应一个 GDT。GDT 中可以存放以下段描述符：
 - 2.1 **代码段描述符：** 描述符规定了一个代码段的基地址、大小和存取权限等信息。（该描述符置 `S` 标志位 1，非系统段）。CPU 有一个代码段寄存器 `CS`，指向当前 CPU 正使用的代码段，同时对应一个代码段描述符高速缓冲寄存器，用于缓存 `CS` 寄存器在 GDT 中指定的代码段描述符。
 - 2.2 **数据段描述符：** 描述符规定了一个数据段的基地址、大小和存取权限等信息。（该描述符置 `S` 标志位 1，非系统段）。CPU 有一个数据段寄存器 `DS`，指向当前 CPU 正使用的数据段，同时对应一个数据段描述符高速缓冲寄存器，用于缓存 `DS` 寄存器在 GDT 中指定的数据段描述符。
 - 2.3 **任务状态段描述符 TSSD（Task State Segment）：** 描述符规定了一个任务状态段的基地址和任务状态段的大小等信息。任务状态段 TSS 是一个 104 字节的数据结构或控制块，用于记录一个任务的关键性状态信息（例如通用寄存器、段寄存器、`CR3`、堆栈指针等）。CPU 有一个任务寄存器 `TR`，指向当前 CPU 正执行任务的 TSS，同时对应一个任务状态段描述符高速缓冲寄存器，用于缓存 `TR` 寄存器在 GDT 中指定的任务状态段描述符。
- 3 **LDT 局部描述符表（Local Descriptor Table）：** 用于存放 8 字节的段描述符（只能存放代码段描述符和数据段描述符），LDT 在主存中的地址和大小存放在 CPU 的 `gdtr` 控制寄存器中。每个进程都可以有一个 LDT，但 linux 几乎不使用该描述符表。

4 IDT 中断描述符表（Interrupt Descriptor Table）：用于存放 8 字节的门描述符，IDT 在主存中的地址和大小存放在 CPU 的 idtr 控制寄存器中。在允许中断之前，必须用 lidt 汇编指令初始化 idtr。IDT 中可以包含以下三种类型的描述符：

4.1 任务门描述符：当中断信号发生时，必须取代当前进程的那个进程的 TSS 选择符存放在任务门中。

4.1.1 linux 并不使用硬件提供的任务门执行进程切换，而是使用软件执行进程切换。每个进程描述符包含一个类型为 thread_struct 的 thread 字段，只要进程被切换出去，内核就把其硬件上下文保存在这个结构中。这个数据结构包含的字段涉及大部分 CPU 寄存器，但不包括诸如 eax、ebx 等等这些通用寄存器，它们的值保留在内核堆栈中。

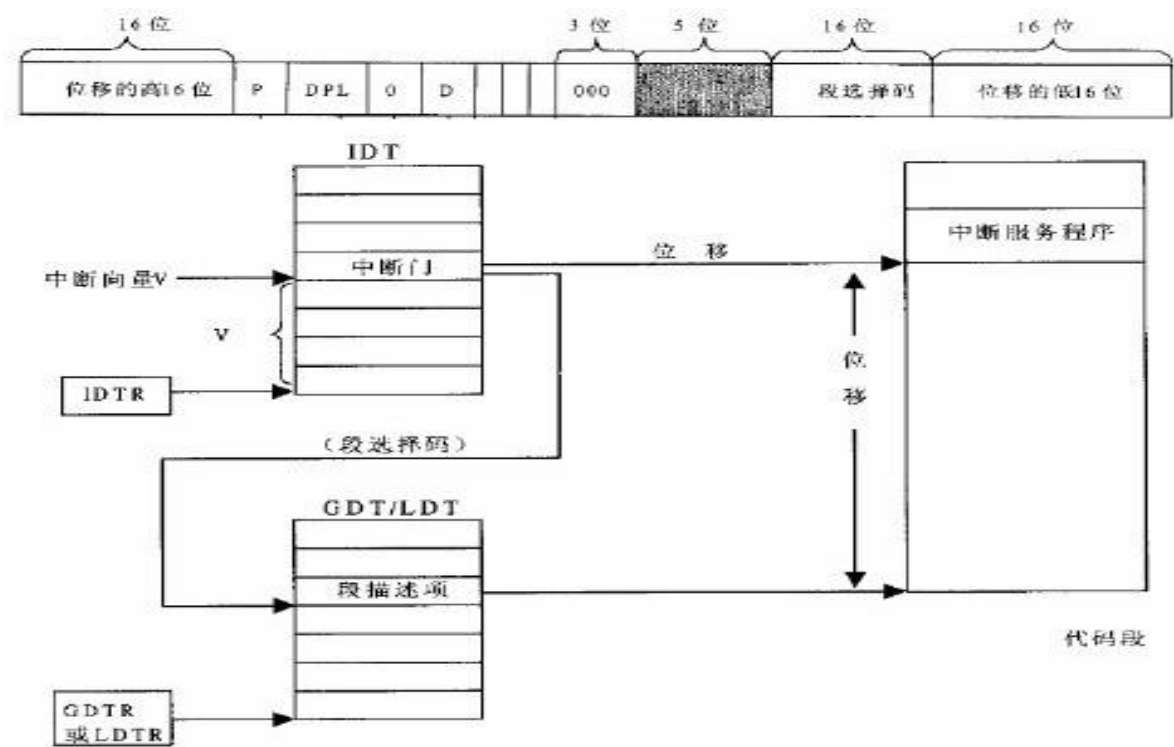
4.1.2 不过 i386 CPU 要求软件设置 TR 及 TSS，内核中便只好“走过场”的设置好 TR 及 TSS 以满足 CPU 的要求。但是，内核并不使用任务门、也不允许使用 JMP 或 CALL 指令实施任务切换。内核只是在初始化阶段设置 TR，使之指向一个 TSS，从此以后就再不改变 TR 的内容了。也就是说，每个 CPU 只有一个 TSS，在初始化以后的全部运行过程中，各 CPU 的进程永远各自使用本 CPU 的同一个 TSS。同时，内核也不依靠 TSS 保存每个进程切换时的寄存器副本，而且将这些寄存器的副本保存在各个进程自己的系统空间堆栈中。

这样一来，TSS 中的绝大部分内容已经失去原来的意义。但是，当 CPU 因中断或系统调用而从用户空间进入系统空间时，会由于运行级别的变化而自动更换堆栈。而新的堆栈指针，包括堆栈寄存器 SS 的内容和堆栈指针寄存器 ESP 的内容，则取自当前任务的 TSS（当前任务 TSS 的地址可由任务寄存器 TR 通过 GDT 找到）。于是，对于 linux 内核来说，TSS 中有意义的就只剩下了 0 级的堆栈指针，也就是 SS0 和 EPO 两项了。Intel 原来的意图是让 TR 的内容，随着不同的 TSS，随着任务的切换而走马灯似的转。可是在 linux 内核中却变成了“铁打的营盘流水的兵”：就一个 TSS，像一座营盘，一经建立就再也不动了。而里面的内容，也就是当前任务的系统堆栈指针，则随着进程的调度切换而流水似的变动。这里的原因在于：改变 TSS 中 SS0 和 ESP0 所花的开销比通过装入 TR 以更换一个 TSS 要小得多。因此，linux 内核中，TSS 并不是属于某个进程的资源，而是全局性的公共资源。在多处理器的情况下，尽管内核中确实有多个 TSS，但是每个 CPU 仍旧只有一个 TSS，一经装入就不再改变了。

4.2 中断门描述符：包含段选择符和中断或异常处理程序的段内偏移量。当控制权转移到一个适当的段时，处理器清 IF 标志，从而关闭将来会发生的可屏蔽中断。

4.3 陷阱门描述符：与中断门相似，只是控制权传递到一个适当的段时处理器不修改 IF 标志。

4.3.1 linux 利用中断门处理中断，利用陷阱门处理异常，同时系统调用也是使用陷阱门（向量号是 0x80），中断门与陷阱门都要指向一个子程序，所以必须结合使用段选择码和段内位移。中断门与陷阱门处理的区别是通过中断门进入中断服务程序时 CPU 会自动将中断关闭。其结构图和处理机制示意图如下：



当中断或异常（系统调用）产生时，首先确定中断或异常相关的中断向量 i，然后取出 IDTR 寄存器指向的 IDT 表中第 i 项（我们假设它是一个中断门或陷阱门），最后将第 i 项的段选择符添加到 CS 寄存器中（这会导致 CPU 将 CS 寄存器在 GDT 表中的对应段描述符添加到段描述符高速缓冲寄存器中），将偏移量添加到 EIP 寄存器中，这样 CPU 中的 CS 和 EIP 寄存器将包含下一条将要执行的指令的逻辑地址。由于 CS 段寄存器被改变，所以特权级（CPL）也会发生变化。

当中断或异常（系统调用）处理完成后，相应的处理程序必须产生一条 iret 指令，该指令将保存在栈中的值（中断或异常发生之前寄存器的值）装载到 CS、EIP 等寄存器（这会导致 CPU 将 CS 寄存器在 GDT 表中的对应段描述符添加到段描述符高速缓冲寄存器中）。这样 CPU 中的 CS 和 EIP 寄存器将包含下一条将要执行的指令的逻辑地址（中断或异常发生之前的下一条指令）。

无论是中断或系统调用，都要修改段寄存器（CS、DS、SS、ESP 等），目的是为了切换代码执行的特权级别 CPL（CPL 存放在 CS 寄存器的 RPL 字段内，每当一个代码段选择子装入 CS 寄存器中时，处理器自动地把 CPL 存放到 CS 的 RPL 字段）和堆栈空间（用户态堆栈与内核态堆栈不同）。但在修改段寄存器 CS 之前，如果是由 INT n 指令或 INTO 指令引起的转移，还要检查中断门、陷阱门或任务门描述符中的 DPL 是否满足 $CPL \leq DPL$ （对于其它的异常或中断，门中的 DPL 被忽略）。

5 每当执行中断、系统调用（从用户态到内核态）、进程切换时，都要更新段寄存器（如 CS、SS、DS 等）。

6 每当执行进程切换时，除了会更新段寄存器外，还会更新页目录指针 CR3。

linux 内核的内存管理

1 线性地址空间（被分为 user 和 kernel 两个空间）

1.1 USER 线性地址空间，指的是可被应用层访问的线性地址空间。其大小（32 位处理器） = 4G - KERNEL 线性地址空间大小。

1.2 KERNEL 线性地址空间，指的是可被内核使用的线性地址空间。其大小 = LOWMEM 线性地址空间大小 + VMALLOC 线性地址空间大小。

1.2.1 LOWMEM 线性地址空间，它占用了 KERNEL 线性地址空间中的一部分。其大小 = KERNEL 线性地址空间大小 - VMALLOC 线性地址空间大小。

1.2.2 LOWMEM 功能，这段地址空间在启动时已做好地址映射，其映射的物理内存是连续的，并与物理内存地址一一对应。

1.2.3 VMALLOC 线性地址空间，它占用了 KERNEL 线性地址空间中的另一部分。其大小 = KERNEL 线性地址空间大小 - 物理内存大小 > __VMALLOC_RESERVE ？

KERNEL 线性地址空间 - 物理内存大小 : `__VMALLOC_RESERVE`

1.2.4 VMALLOC 功能, 用于在内核中分配线性地址连续, 但物理地址不需连续的大内存区。这段地址空间初始化时未做任何地址映射。

1.3 linux 内核线性地址空间变量的含义

1.3.1 `PAGE_OFFSET = __PAGE_OFFSET = CONFIG_PAGE_OFFSET`: 这个值是用于划分 USER 线性地址和 KERNEL 线性地址的分界点。

1.3.2 `__VMALLOC_RESERVE`: 表示为 VMALLOC 线性地址空间最小应保留的空间大小, 默认是 128M。这个值可在系统启动时通过内核参数 `vmalloc` 来改变。

1.3.3 `VMALLOC_END`: VMALLOC 线性地址空间的结束地址。其值 = 最大线性地址 - 其它 ROM 内存空间大小

1.3.4 `VMALLOC_START`: VMALLOC 线性地址空间的起始地址。其值 = `VMALLOC_END` - VMALLOC 线性大小 (最小为 `VMALLOC_RESERVE`)

1.3.5 `VMALLOC_OFFSET`: 固定大小为 8M, 用于再 LOWMEM 线性地址空间和 VMALLOC 线性地址空间之间建立一个隔离带, 防止它们互相影响。

1.3.6 `MAXMEM`: 其值为(`VMALLOC_END` - `PAGE_OFFSET` - `__VMALLOC_RESERVE`), 表示内核能够直接映射的最大 RAM 容量, 这个值也是相对固定的(与物理内存实际的大小没有关系), 因为 `VMALLOC_END`、`PAGE_OFFSET`、`__VMALLOC_RESERVE` 是相对固定的。

1.3.7 `MAXMEM_PFN`: 其值为 `PFN_DOWN(MAXMEM)`, 表示内核能够直接管理的内存页个数。

1.3.8 `MAX_ARCH_PFN`: 表示 CPU 最大可访问的内存页数。对应 32 位 CPU, 如果其开启了 PAE, 则 `MAX_ARCH_PFN=(1ULL<<(36-PAGE_SHIFT))`。如果未开启 PAE, 则 `MAX_ARCH_PFN=(1ULL<<(32-PAGE_SHIFT))`

1.3.9 `max_pfn`: 是从 bios 发现的最大的能管理的内存页个数, 与实际内存大小有关, 但它的大小不能超过 `MAX_ARCH_PFN`。

当 `max_pfn <= MAXMEM_PFN` 时说明实际的物理内存比内核能够线性映射的内存数小, 此时就不需要 Highmem 来管理了。此时就算开启了 HIGHMEM, `ZONE_HIGHMEM` 区能够管理的内存大小也是 0。

当 `max_pfn > MAXMEM_PFN` 时说明实际的物理内存比内核能够线性映射的内存数大, 这样多余的那部分物理内存就不能进行线性映射了, 必须通过页表映射(即 `vmalloc` 来管理), 该物理内存应该由 `ZONE_HIGHMEM` 区来管理。而如果没有开启 HIGHMEM 选项, 则这部分内存就丢失了, 不再被管理。

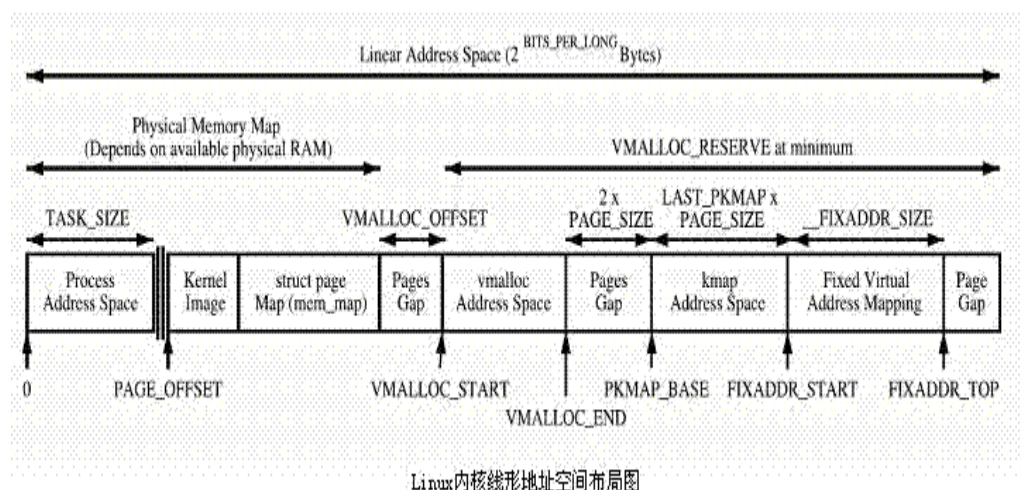
1.3.10 `max_low_pfn`: 低端内存(相对于 highmem)的最大页数, 这个就是 `ZONE_NORMAL` 区能管理的最大页数。

如果实际物理内存比内核能够线性映射的内存数小, `max_low_pfn = max_pfn`, 即所有的内存都归低端内存区管理。

如果实际物理内存比内核能够线性映射的内存数大, `max_low_pfn = MAXMEM_PFN`, 即低端内存区能够管理的最大值就是内核能够线性映射的最大值。

1.3.11 `highmem_pages`: 高端内存的最大页数, 这个就是 `ZONE_HIGHMEM` 区管理的最大页数。其值为(`max_pfn - max_low_pfn`)。

1.4 linux 内核线性地址空间的布局



Linux内核线性地址空间布局图

2 物理地址空间 (被 page 结构按功能区进行管理)

2.1 `ZONE_DMA` 物理内存空间, 在 x86 体系结构上它是 0~16M 的内存范围。由 `page` 结构进行管理, 这个区包含的页能被老设备执行 DMA 操作。

2.2 `ZONE_NORMAL` 物理内存空间, 在 x86 体系上它是 16M~KERNEL->LOWMEM 线性地址对应的内存范围。由 `page` 结构进行管理。

(内核在启动时, 就已经将上面这两个区的内存空间映射到 KERNEL 线性地址空间中了。所以内核不需做任何操作, 就可以直接访问这些物理内存。但为了保证所访问内存页没有被其它功能使用, 因此在访问内存之前, 要使用 `get_free_page()` 等函数获取空闲页, 并将该页标记为使用。返回的物理页不需要做任何地址映射, 仅仅是将物理地址加上内核空间偏移量就获得了其线性地址)

2.3 `ZONE_HIGHMEM` 物理内存空间, 其大小 = 实际物理内存大小 - KERNEL->LOWMEM 线性地址空间大小。由 `page` 结构进行管理, 用于管理物理内存大小超过 KERNEL->LOWMEM 线性地址空间大小以外的物理内存。这段内存空间没有被映射到内核线性地址空间, 在使用时, 要通过 `alloc_page()` 分配一个 `page` 页, 然后再将该页映射到线性地址空间, 这样才能通过线性地址访问。内核 (通常用 `vmalloc()` 函数) 将 HIGHMEM 物理内存页映射到 VMALLOC 线性地址空间, 因此内核可使用的 HIGHMEM 物理内存大小与 VMALLOC 线性地址空间大小有关。另外, HIGHMEM 物理内存页通常被映射到 USER 线性地址空间。

2.4 `ZONE_NORMAL` 与 `ZONE_HIGHMEM` 主要的区别是 `ZONE_NORMAL` 中的物理内存不需要做页表映射直接用就行了, 其虚拟地址的最大值是 `MAXMEM`。而 `ZONE_HIGHMEM` 中的物理内存需要做页表映射才能使用, 其虚拟地址的范围是 `VMALLOC_START`~`VMALLOC_END`。所以不能做线性映射的内存都归 `ZONE_HIGHMEM` 来管理了。如果开启的 HIGHMEM 选项, 则 `ZONE_HIGHMEM` 的大小就是剩下的物理内存, 如果没开启, 则这部分物理内存就无法管理了。

slub 机制(slab 与 slob)

很久很久以前: 一个叫做 Mark Hemment 的哥儿们写了 Slab。在接下来的一些年里, 其他人对 Slab 进行了完善。一年半以前, SLOB 问世了。SLOB 的目标是针对嵌入式系统的, 主要是适用于那些内存非常有限的系统, 比如 32MB 以下的内存, 它不太注重 large smp 系统, 虽然最近在这方面有一些小的改进。几个月之前, SLUB 闪亮登场。它基本上属于对 Slab 的重设计(redesign), 但是代码更少, 并且能更好的适应 large NUMA 系统。SLUB 被很认为是 Slab 和 Slob 的取代者, 大概在 2.6.24/2.6.25 将会被同志们抛弃。而 SLUB 将是未来 Linux Kernel 中的首选。

简单的说: Slab 是基础, 是最早从 Sun OS 那引进的; Slub 是在 Slab 上进行的改进, 在大型机上表现出色 (不知道在普通 PC 上如何), 据说还被 IA-64 作为默认; 而 Slob 是针对小型系统设计的, 当然了, 主要是嵌入式。

内存的分配

- 1 如果需要连续的物理页，可使用如下内存分配方式：
 - 1.1 `alloc_pages(gfp_mask,order)`等函数，这类函数以页为单位进行分配，由于内核采用伙伴系统算法管理内存页，所以在 x86 体系结构上页分配方式最多可分配 4M 内存。这类函数是否会睡眠以及在哪个区段分配内存由 `gfp_mask` 决定。它返回第一个所分配页框描述符的线性地址。
 - 1.2 `__get_free_pages(gfp_mask, order)`等函数，这类函数实际是调用函数 `alloc_pages()`，但它返回第一个分配页框的线性地址。它只能从 DMA 和 NORMAL 区分配内存，并通过该内存物理地址减去 `PAGE_OFFSET` 获得其线性地址。其它性质与 `alloc_pages()` 相同。
 - 1.3 `kmalloc(size,flags)`函数，该函数以字节为单位，它最大可分配 128K 内存。该函数是否会睡眠以及在哪个区段分配内存由 `flags` 决定。但它不能指定 `__GFP_HIGHMEM` 标志在高端内存中分配内存。
 - 1.4 每个分配函数都对应一个释放函数。
- 2 如果不需要物理上连续的页，而仅仅需要虚拟地址上连续的页，那么就使用 `vmalloc(size)`函数分配内存。
 - 2.1 该函数会睡眠，不能用在中断上下文。
 - 2.2 由于 `vmalloc()`分配内存指定了 `__GFP_HIGHMEM` 标志，所以它先从高端内存（默认是高于 896M）分配。
 - 2.3 它可获得的地址在 `VMALLOC_START`～`VMALLOC_END` 空间，默认大小是 128M。
 - 2.4 由于在物理上不连续，所以该函数分配的内存不能在处理器之外使用。
 - 2.5 通过 `vmalloc()`分配的内存使用起来效率不高，因为它会导致 TLB 抖动。
- 3 如果创建和销毁很多较大的数据结构，则应考虑建立 slab 高速缓存。
 - 3.1 `kmem_cache_create()`建立高速缓存（它只使用 `ZONE_NORMAL` 和 `ZONE_DMA` 区域），该函数会睡眠，不能用在中断上下文。
 - 3.2 `kmem_cache_destroy()`释放高速缓存，该函数会睡眠，不能用在中断上下文。
 - 3.3 `kmem_cache_alloc(cachep,flags)`。该函数在 `cachep` 指定的告诉缓存中获取对象，是否睡眠由 `flags` 决定。
 - 3.4 `kmem_cache_free(cachep,objp)`。该函数将 `objp` 指向的对象释放到 `cachep` 指定的高速缓存中。该函数不会睡眠。
- 4 如果想从高端内存进行分配，使用下面的分配方法。
 - 4.1 使用 `alloc_pages(gfp_mask,order)`，在 `gfp_mask` 中指定 `__GFP_HIGHMEM` 标志进行分配，但该函数返回的是一个 `page` 结构的指针，而不是指向某个逻辑地址的指针，所以需要使用 `kmap()`函数把高端内存映射到内核的地址空间。
 - 4.2 `vmalloc()`函数也会从高端内存进行分配，它返回的是映射好的逻辑地址。但它能分配的内存大小是 `VMALLOC_START`～`VMALLOC_END` 空间（默认是 128M 内存空间）。并且该函数会睡眠，不能用在中断上下文中。
- 5 分配内存时，如果指定 `__GFP_DMA` 标志，则只有 `ZONE_DMA` 区段会被搜索。如果没有指定标志，则 `ZONE_NORMAL` 和 `ZONE_DMA` 区段都会被搜索，`ZONE_NORMAL` 区段优先搜索。如果指定 `__GFP_HIGHMEM` 标志，则三个区段都会被搜索，搜索区段的优先顺序是 `ZONE_HIGHMEM`、`ZONE_NORMAL`、`ZONE_DMA`。
- 6 编程中常用到的类型标志
 - 6.1 `GFP_KERNEL` 标志，可用于内核空间的进程上下文，可能会睡眠。
 - 6.2 `GFP_ATOMIC` 标志，可用于内核空间的进程上下文、中断上下文、`spinlock` 锁之间，不会睡眠。

内存的释放

- 1 每种内存分配函数都对应一个释放函数，释放时应使用对应的释放函数，并注意该函数是否会睡眠。
- 2 有些内存释放函数并不会将内存真正释放到空闲页队列中，而是释放到它的缓存中（slab 缓冲区）。
- 3 内核周期性（差不多每两秒一次）调用 `cache_reap()`函数回收 slab 高速缓冲区的内存页。
- 4 当内核调用 `__alloc_pages()`发现所有适合内存分配的内存管理区包含的空闲页低于“警告”值时，它激活 `kswapd` 内核线程来回收内存。
- 5 当内存分配失败时，激活内存紧缺回收函数 `try_to_free_pages()`进行回收。
- 6 当内核无法再释放出更多内存时，它会用删除进程的方法来释放出更多内存。
- 7 有些页是不可被回收的：空闲页、保留页（置有 `PG_RESERVED` 标志）、内核动态分配页、内核态堆栈页、临时锁定页（`PG_LOCKET` 标志置位）、内存锁定页（`VM_LOCKED` 标志置位）。
- 8 可被回收的页也是有一些可直接被丢弃、有些需要被同步、有些需要被放入交换分区。

linux-2.6 kernel 针对不同配置，内核内存的分配结果

```
1G/3G user/kernel splitit  (__VMALLOC_RESERVE = 128 << 20)
0MB HIGHMEM available.      //HIGHMEM 物理内存大小
2047MB LOWMEM available.    //LOWMEM 内核可直接使用物理内存大小
mapped low ram: 0 - 7ffd0000
low ram: 0 - 7ffd0000
node 0 low ram: 00000000 - 7ffd0000
node 0 bootmap 00008000 - 00017ffc
Memory: 2071924k/2096960k available (2801k kernel code, 24644k reserved, 1065k data, 328k init, 0k highmem)
virtual kernel memory layout:
    fixmap   : 0xfff17000 - 0xfffff000   ( 928 kB)
    pkmap    : 0xff800000 - 0xffc00000   (4096 kB)
    vmalloc  : 0xc07d0000 - 0xff7fe000   (1008 MB)    //内核为 VMALLOC 预留的线性地址空间大小
    lowmem   : 0x40000000 - 0xbffd0000   (2047 MB)    //内核直接映射的线性地址空间大小
        .init : 0x413c7000 - 0x41419000   ( 328 kB)
        .data : 0x412bc701 - 0x413c6e64   (1065 kB)
        .text : 0x41000000 - 0x412bc701   (2801 kB)
```

```
1G/3G user/kernel splitit  (__VMALLOC_RESERVE = 128 << 20, vmalloc=1512M)
527MB HIGHMEM available.
1519MB LOWMEM available.
mapped low ram: 0 - 5effe000
low ram: 0 - 5effe000
node 0 low ram: 00000000 - 5effe000
node 0 bootmap 00008000 - 00013e00
Memory: 2071912k/2096960k available (2801k kernel code, 24656k reserved, 1065k data, 328k init, 540488k highmem)
virtual kernel memory layout:
    fixmap   : 0xfff17000 - 0xfffff000   ( 928 kB)
    pkmap    : 0xff800000 - 0xffc00000   (4096 kB)
    vmalloc  : 0x9f7fe000 - 0xff7fe000   (1536 MB)
    lowmem   : 0x40000000 - 0x9effe000   (1519 MB)
        .init : 0x413c7000 - 0x41419000   ( 328 kB)
        .data : 0x412bc701 - 0x413c6e64   (1065 kB)
        .text : 0x41000000 - 0x412bc701   (2801 kB)
```

```
2G/2G user/kernel split  (__VMALLOC_RESERVE = 128 << 20)
135MB HIGHMEM available.
1911MB LOWMEM available.
mapped low ram: 0 - 777fe000
low ram: 0 - 777fe000
node 0 low ram: 00000000 - 777fe000
node 0 bootmap 00008000 - 00016f00
Memory: 2071924k/2096960k available (2801k kernel code, 24644k reserved, 1065k data, 328k init, 139080k highmem)
virtual kernel memory layout:
    fixmap   : 0xfff17000 - 0xfffff000   ( 928 kB)
    pkmap    : 0xff800000 - 0xffc00000   (4096 kB)
    vmalloc  : 0xf7ffe000 - 0xff7fe000   ( 120 MB)
    lowmem   : 0x80000000 - 0xf77fe000   (1911 MB)
        .init : 0x813c7000 - 0x81419000   ( 328 kB)
        .data : 0x812bc6b1 - 0x813c6e64   (1065 kB)
        .text : 0x81000000 - 0x812bc6b1   (2801 kB)
```

```
2G/2G user/kernel split (for full 2G low memory)  (__VMALLOC_RESERVE = 128 << 20)
7MB HIGHMEM available.
2039MB LOWMEM available.
mapped low ram: 0 - 7f7fe000
low ram: 0 - 7f7fe000
node 0 low ram: 00000000 - 7f7fe000
node 0 bootmap 00008000 - 00017f00
Memory: 2071928k/2096960k available (2801k kernel code, 24640k reserved, 1065k data, 328k init, 8008k highmem)
virtual kernel memory layout:
    fixmap   : 0xfff17000 - 0xfffff000   ( 928 kB)
    pkmap    : 0xff800000 - 0xffc00000   (4096 kB)
    vmalloc  : 0xf7ffe000 - 0xff7fe000   ( 120 MB)
    lowmem   : 0x78000000 - 0xf77fe000   (2039 MB)
        .init : 0x793c7000 - 0x79419000   ( 328 kB)
        .data : 0x792bc701 - 0x793c6e64   (1065 kB)
        .text : 0x79000000 - 0x792bc701   (2801 kB)
```

2G/2G user/kernel split (for full 2G low memory) (__VMALLOC_RESERVE = 128 << 20, vmalloc=512M)

399MB HIGHMEM available.

1647MB LOWMEM available.

mapped low ram: 0 - 66ffe000

low ram: 0 - 66ffe000

node 0 low ram: 00000000 - 66ffe000

node 0 bootmap 00008000 - 00014e00

Memory: 2071920k/2096960k available (2801k kernel code, 24648k reserved, 1065k data, 328k init, 409416k highmem)

virtual kernel memory layout:

fixmap : 0xffff17000 - 0xfffff000 (928 kB)
pkmap : 0xff800000 - 0xffc00000 (4096 kB)
vmalloc : 0xdf7fe000 - 0xff7fe000 (512 MB)
lowmem : 0x78000000 - 0xdeffe000 (1647 MB)
.init : 0x793c7000 - 0x79419000 (328 kB)
.data : 0x792bc701 - 0x793c6e64 (1065 kB)
.text : 0x79000000 - 0x792bc701 (2801 kB)

3G/1G user/kernel split (__VMALLOC_RESERVE = 128 << 20)

1159MB HIGHMEM available.

887MB LOWMEM available.

mapped low ram: 0 - 377fe000

low ram: 0 - 377fe000

node 0 low ram: 00000000 - 377fe000

node 0 bootmap 00008000 - 0000ef00

Memory: 2072684k/2096960k available (2801k kernel code, 23884k reserved, 1065k data, 328k init, 1187656k highmem)

virtual kernel memory layout:

fixmap : 0xffff17000 - 0xfffff000 (928 kB)
pkmap : 0xff800000 - 0xffc00000 (4096 kB)
vmalloc : 0xf7ffe000 - 0xff7fe000 (120 MB)
lowmem : 0xc0000000 - 0xf77fe000 (887 MB)
.init : 0xc13c7000 - 0xc1419000 (328 kB)
.data : 0xc12bc701 - 0xc13c6e64 (1065 kB)
.text : 0xc1000000 - 0xc12bc701 (2801 kB)

3G/1G user/kernel split (for full 1G low memory) (__VMALLOC_RESERVE = 128 << 20)

903MB HIGHMEM available.

1143MB LOWMEM available.

mapped low ram: 0 - 477fe000

low ram: 0 - 477fe000

node 0 low ram: 00000000 - 477fe000

node 0 bootmap 00008000 - 00010f00

Memory: 2071916k/2096960k available (2801k kernel code, 24652k reserved, 1065k data, 328k init, 925512k highmem)

virtual kernel memory layout:

fixmap : 0xffff17000 - 0xfffff000 (928 kB)
pkmap : 0xff800000 - 0xffc00000 (4096 kB)
vmalloc : 0xf7ffe000 - 0xff7fe000 (120 MB)
lowmem : 0xb0000000 - 0xf77fe000 (1143 MB)
.init : 0xb13c7000 - 0xb1419000 (328 kB)
.data : 0xb12bc701 - 0xb13c6e64 (1065 kB)
.text : 0xb1000000 - 0xb12bc701 (2801 kB)

linux-2.4 kernel 内存管理中的一些问题

- 1 内核能直接使用的内存只能是 0~896M 空间（由于内核只有 1G 线性地址空间）。即使系统中有 2G 物理内存，并且开启 CONFIG_HIGHMEM 和 CONFIG_X86_PAE 选项，内核空间也只能使用 896M 内存。这在 linux2.6 内核中可通过编译选项来改变。（如果开启了 CONFIG_HIGHMEM 选项，则用户空间可以使用高于 1G 的内存空间。如果开启了 CONFIG_X86_PAE 选项，则 X86-cpu 总线变为 36 位，即 CPU 可访问 0~64G 内存）
- 2 通过调用__alloc_pages()分配高端内存页，然后通过 kmap()映射该内存页，内核空间也只能再多使用 128M 内存（即 0~1G 空间）。因为内核空间的线性地址空间只有 1G。
- 3 由于内核只有 1G 线性地址空间的限制，所以在我们的设备上即使使用 2G 的物理内存，也等于浪费了 1G 的物理内存。（但如果开启 CONFIG_HIGHMEM 选项，1G~2G 内存空间可被用户空间使用）
- 4 通下面的方法可增大内核的线性地址空间，但它会减少用户进程的线性地址空间(2.4 内核的改变方法)

//将内核与用户地址空间改成 2g/2g，需要改下面 2 个地方

include/asm-i386/page.h

#define __PAGE_OFFSET (0x80000000) //0xC0000000 改为 0x80000000

arch/i386/vmlinux.lds

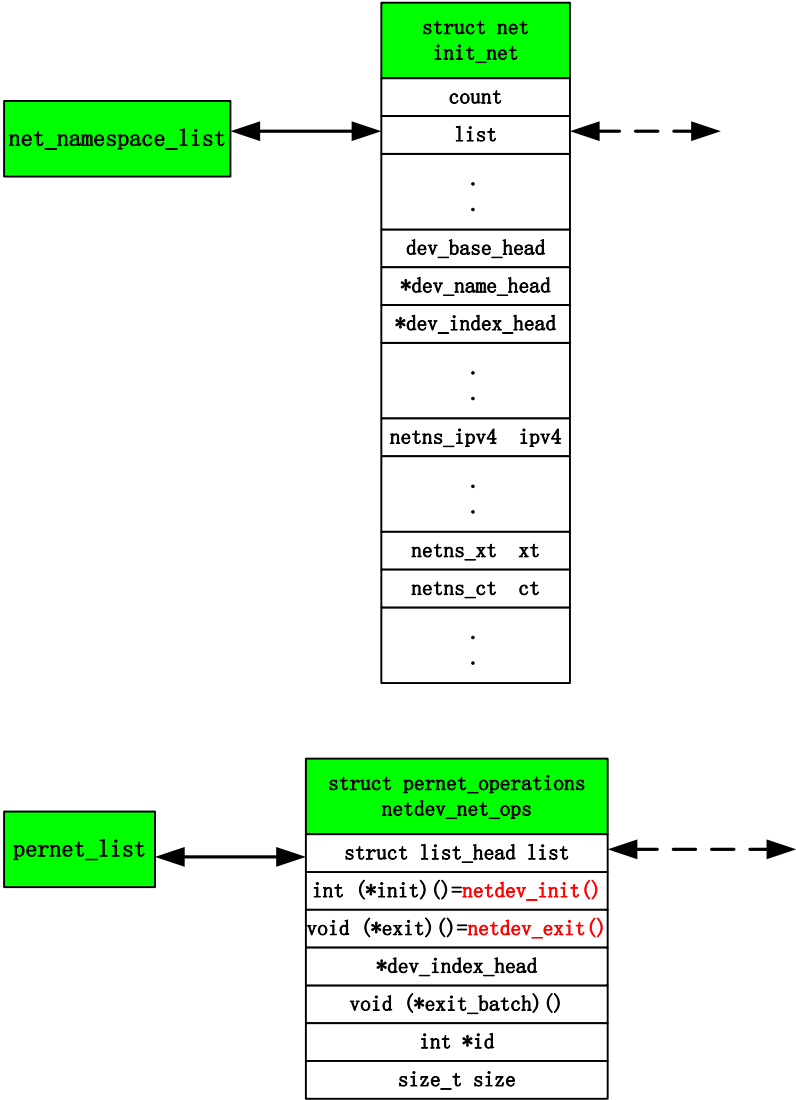
OUTPUT_ARCH(i386)

```
ENTRY(_start)
SECTIONS
{
/* #ifdef __CHEETAH__ */
. = 0x80000000 + 0x100000; //0xC0000000+0x100000 改为 0x80000000+0x100000
/* #endif */
_text = .; /* Text and read-only data */
.text : {
*(.text)
*(.fixup)
*(.gnu.warning)
} = 0x9090
```


网络知识

网络命名空间（net_namespace）

1 网络命名空间结构图



2 网络命名空间的使用说明

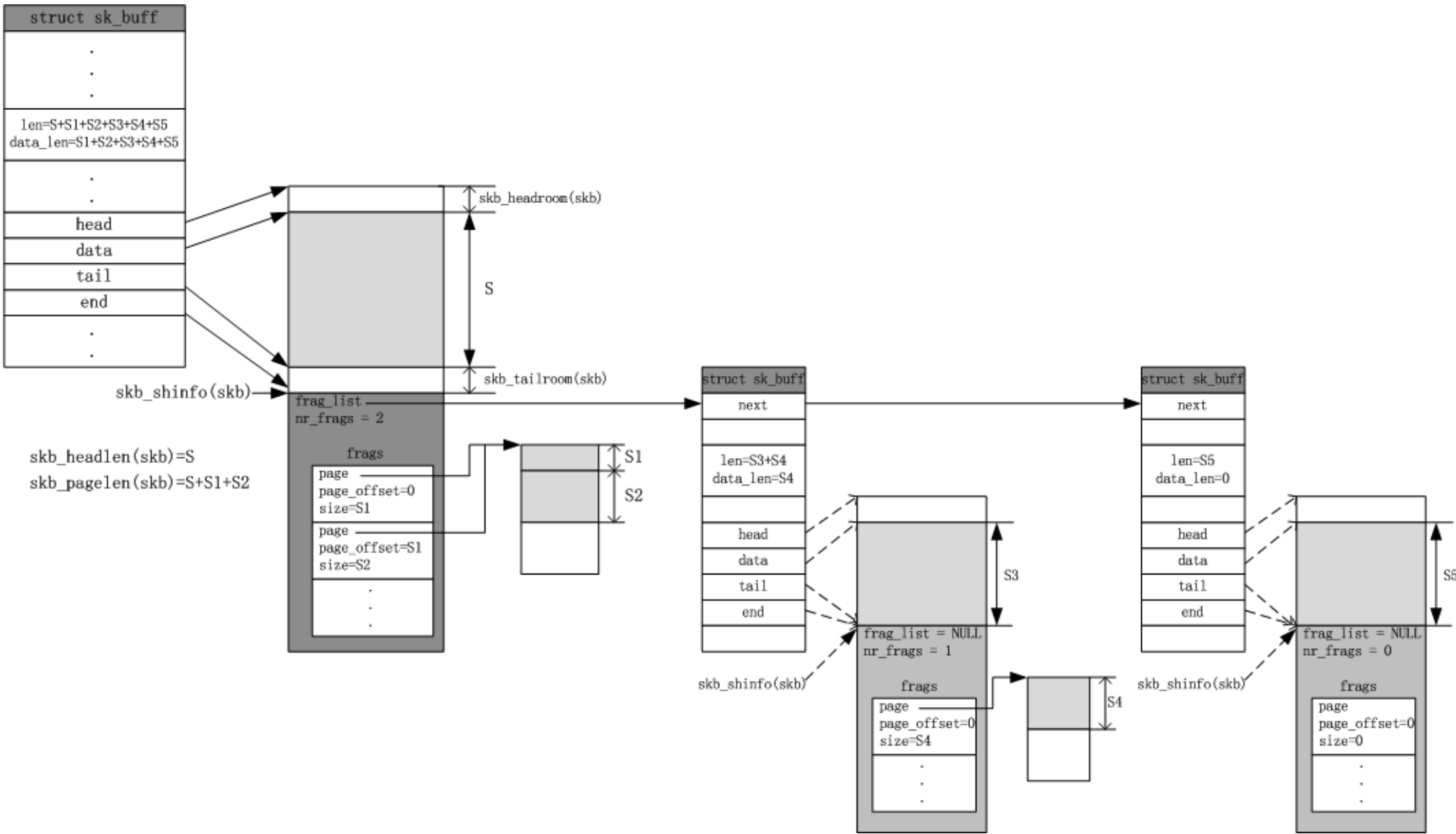
- 2.1 网络命名空间的本意的是为每一种网络实体创建一个 struct net 的变量（当前只有一个网络实体 init_net），并将它们加入到 net_namespace_list 链表中。每当注册一个 struct pernet_operations 结构时，就用该结构的(*init)()指针指向的函数初始化 net_namespace_list 链表中的每一个 net 实体。每当添加一个 net 实体时，都要用 pernet_list 链上已注册的所有 pernet_operations 结构进行初始化。
- 2.2 register_pernet_subsys(struct pernet_operations *ops)函数用于注册一个 pernet_operations 结构。它先将 pernet_operations 结构挂入 pernet_list 链表中（不使用名字空间时则不会挂入该链表），然后用注册的 pernet_operations->init 指针指向的函数初始化所有 net_namespace_list 链表中的 net 实体（不使用名字空间时则只初始化 init_net 实体）。

网络中重要数据结构

sk_buff 数据结构

下图是一个 IP 数据报文通过 sk_buff 进行管理的表现形式。这样一个管理形式代表一个完整的 IP 报文。sk_buff 通过 3 个缓存管理区来管理一个 IP 数据报文。

- 1
- 主缓存区: 这个缓存区是一个连续的内存区块, 这个内存区块通过 `skb->head`、`skb->data`、`skb->tail` 和 `skb->end` 指针进行管理。主缓存区中的数据可以通过移动 `skb->data` 和 `skb->tail` 轻易的添加或删除数据。
- 2
- 分散/聚集 I/O 缓存区: 这个缓存区是一组单独的内存页 (page), 这些内存页通过 `frags[]` 结构数组进行管理。分散/聚集 I/O 缓存区只有在网络设备支持分散/聚集 I/O 功能时才会被用到, 用于防止内存的浪费 (否则内核就要把主缓存区直接分配 PMTU 大小, 不管上层是否要传输这么多数据)。
- 3
- skb_shinfo(SKB)->frag_list 链表的缓存区: 这个缓存区是一组单独的 sk_buff 管理结构, 而每个 sk_buff 又有可能包含这 3 种缓存区。skb_shinfo(SKB)->frag_list 链表的缓存区用于处理分片数据包 (无论是本地发出的 IP 数据报文被分片, 还是接收到 IP 分片报文被重组)。



对上图进行一些说明:

- 1
- 上图中 frag_list 中的 sk_buff, 有可能还有自己的 frag_list 缓存区, 但未在上图中表现出来。
- 2
- 我们假设网络设备支持分散/聚集 I/O 功能, 所以上图中 sk_buff 包含分散/聚集 I/O 缓存区。
- 3
- 无论是被分片的 IP 数据报文 (ip_fragment), 还是被重组的 IP 数据报文 (ip_defrag), 都是按上图的管理形式进行管理。每个的分片报文都由一个 sk_buff 管理, 并且按照偏移量的顺序被挂载在第一个分片报文的 frag_list 链表中。
- 4
- 内核中管理 sk_buff 的函数, 大部分 (例如 pskb_**) 都能处理上面的管理形式 (即同时处理上面 3 个缓存区)。但也有一些函数 (skb_**) 仅处理 sk_buff 的主缓存区, 所以, 除非明确知道自己所操作的 sk_buff 仅包含主缓存区数据, 否则就不要使用这些仅操作主缓存区的管理函数。

```
struct sk_buff {
    /* These two members must be first. */
    struct sk_buff *next; /* Next buffer in list */
    struct sk_buff *prev; /* Previous buffer in list */

    ktime_t timestamp; /* Time we arrived */

    struct sock *sk; /* Socket we are owned by
    * 指向一个拥有这个 sk_buff 的 sock 结构的指针, 这个指针在网络包由本机发出或者由本机进程接收时有效, 如果这个 sk_buff 只在
    * 转发中使用, 则这个指针是 NULL
    */
    struct net_device *dev; /* Device we arrived on/are leaving by */

    /*
    * This is the control buffer. It is free to use for every layer. Please put your private variables there. If you want to keep them across layers you have to do a skb_clone() first.
    * This is owned by whoever has the skb queued ATM.
    */
}
```

```
char                cb[48] __aligned(8);

unsigned long       _skb_dst;      /* destination entry (with norefcnt bit) */
#ifdef CONFIG_XFRM
struct sec_path *sp;              /* the security path, used for xfrm */
#endif

unsigned int        truesize;      /* Buffer size 表示实际分配的缓存区大小，但不包含 skb_shared_info 附件信息结构大小。
    * truesize = sizeof(struct sk_buff) + data 数据区 +分散/聚集 I/O 缓存区数据（skb_shinfo(SKB)->frags 指定的缓存区数据）量 +
    * skb_shinfo(SKB)->frag_list 链表中每个 sk_buff->truesize 长度。
    * 如果申请了 len 字节的缓冲区，alloc_skb 函数会把它初始化成 len + sizeof(sk_buff)。
    * truesize 经常有会被表示为不同含义（例如并不包含 skb_shinfo(SKB)->frag_list 列表中每个 sk_buff->truesize 的长度）
    */
unsigned int        len;          /* Length of actual data. 表示缓冲区中实际的数据量。
    * len = 主缓存区数据（skb->data 和 skb->tail 之间的数据）量 + 分散/聚集 I/O 缓存区数据（skb_shinfo(SKB)->frags 指定的缓存区数
    * 据）量 + skb_shinfo(SKB)->frag_list 链表的缓存区数据量。
    * 当缓存区从一个网络分成移往下一个网络分层时，其值就会变化，因为在协议栈中往上移动时报头会被丢弃，但是往下移动时报
    * 头就会添加进来。len 也会把协议报头算在内，如“数据预留和对其（skb_reserve、skb_put、skb_push、skb_pull）等函数会调整
    * len 大小。
    */
unsigned int        data_len;     /* Data length. 表示除主缓存区以外的其它缓存区的数据量。
    * data_len = 分散/聚集 I/O 缓存区数据（skb_shinfo(SKB)->frags 指定的缓存区数据）量 + skb_shinfo(SKB)->frag_list 链表的缓存区数据
    * 量。
    */

__u16              mac_len,      /* Length of link layer header */
                hdr_len;        /* writable header length of cloned skb */

/*
 * #define CHECKSUM_NONE          0
 * #define CHECKSUM_UNNECESSARY  1
 * #define CHECKSUM_COMPLETE     2
 * #define CHECKSUM_PARTIAL      3
 *
 * 上面的 4 个标志被用于设置 ip_summed 变量，它们在接收和发送时分别代表不同含义。
 * 变量 csum、csum_start、csum_offset 值的含义，由当前数据包是接收还是发送和 ip_summed 设置的标志来决定。
 *
 * A. Checksumming of received packets by device.
 *
 * CHECKSUM_NONE:          device failed to checksum this packet skb->csum is undefined.
 *
 * CHECKSUM_UNNECESSARY:  device parsed packet and wouldbe verified checksum.
 *                          skb->csum is undefined.
 *                          It is bad option, but, unfortunately, many of vendors do this. Apparently with secret goal to sell you new device, when you will add new
 *                          protocol to your host. F.e. IPv6. 8)
 *
 * CHECKSUM_COMPLETE:     the most generic way. Device supplied checksum of _all_ the packet as seen by netif_rx in skb->csum.
 *                          NOTE: Even if device supports only some protocols, but is able to produce some skb->csum, it MUST use COMPLETE, not UNNECESSARY.
 *
 * CHECKSUM_PARTIAL:      identical to the case for output below. This may occur on a packet received directly from another Linux OS, e.g., a virtualised Linux kernel on
 *                          the same host. The packet can be treated in the same way as UNNECESSARY except that on output (i.e., forwarding) the checksum must be
 *                          filled in by the OS or the hardware.
 *
 * B. Checksumming on output.
 *
 * CHECKSUM_NONE:          skb is checksummed by protocol or csum is not required.
 *
 * CHECKSUM_PARTIAL:      device is required to csum packet as seen by hard_start_xmit from skb->csum_start to the end and to record the checksum at
 *                          skb->csum_start + skb->csum_offset.
 *
 * Device must show its capabilities in dev->features, set at device setup time.
 * NETIF_F_HW_CSUM        it is clever device, it is able to checksum everything.
 * NETIF_F_NO_CSUM        loopback or reliable single hop media.
 * NETIF_F_IP_CSUM        device is dumb. It is able to csum only TCP/UDP over IPv4. Sigh. Vendors like this way by an unknown reason. Though, see comment above
 *                          about CHECKSUM_UNNECESSARY. 8)
 * NETIF_F_IPV6_CSUM      about as dumb as the last one but does IPv6 instead.
 *
 * Any questions? No questions, good.          --ANK
 */
union {
    __wsum        csum;
```

```
struct {
    __u16    csum_start;
    __u16    csum_offset;
};

__u32        priority;                /* Packet queueing priority
 * 这个变量描述发送或转发包的 QoS 类别。如果包是本地生成的，socket 层会设置 priority 变量。如果包是将要被转发的，
 * rt_tos2priority 函数会根据 ip 头中的 Tos 域来计算付给这个变量的值。
 */

kmemcheck_bitfield_begin(flags1);

__u8         local_df:1,              /* allow local fragmentation */
             cloned:1,               /* Head may be cloned (check refcnt to be sure) */
             ip_summed:2,            /* 看 csum 上面的注释 */
             nohdr:1,               /* Payload reference only, must not modify header */
             nfctinfo:3;            /* Relationship of this skb to the connection */

__u8         pkt_type:3,             /* 表示帧的类型。分类是由 L2 的目的地址来决定的。可能的取值都在 include/linux/if_packet.h 中定义。对以太网设备来说，
                                     这个变量由 eth_type_trans()函数初始化。包含下面的值：
                                     PACKET_HOST      已接收帧的目的 MAC 地址与收到它的网络设备的 MAC 地址相等。换句话说，这个包是发给本机的。
                                                         由于该值是 0，所以数据包初始化就是这个类型。
                                     PACKET_MULTICAST 已接收帧的目的 MAC 地址是该接口已注册的一个多播地址。
                                     PACKET_BROADCAST 已接收帧的目的 MAC 地址是接收接口的广播地址。
                                     PACKET_OTHERHOST 已接收帧的目的 MAC 地址与收到它的网络设备的 MAC 地址不同（不管是单播，多播还是广播）。
                                                         因此，在桥模式下该类型数据包会被转发，在路由模式下该类型数据包会被丢弃。
                                     PACKET_OUTGOING 数据包正被发送。此表示的用户是 Decnet 协议，并且给每个网络设备分流器一个输出封包副本的
                                                         函数（参考 dev_queue_xmit_nit 函数）。
                                     PACKET_LOOPBAK   数据包正发送至 lookback 设备。由于有这个标记，在处理 loopback 设备时，内核可以跳过一些真实
                                                         设备所需要的操作。
                                     PACKET_FASTROUTE 用 fastroute 功能路由封包。2.6 内核版本不再支持 fastroute。*/
             fclone:2,              /* skbuff clone status */
             ipvs_property:1,       /* skbuff is owned by ipvs */
             peeked:1,              /* this packet has been seen already, so stats have been done for it, don't do them again */
             nf_trace:1;            /* netfilter packet trace flag */

__be16       protocol:16;           /* protocol 是从 L2 层的设备驱动程序的角度来看，就是用在下一个较高层（L3）的协议。它是大端字节序（即网络字节序）。
 * 典型协议有 IP、IPv6、以及 ARP；完整的列表在 include/linux/if_ether.h 中(ETH_P_XXX)。由于每种协议都有自己的函数处
 * 理例程用来处理输入的封包，因此驱动程序使用这个字段通知其上层该使用哪个处理例程。每个驱动程序会调用 netif_rx
 * 用来启动上面的网络分层的处理例程，所以，在该函数被调用前 protocol 字段必须初始化。（以太网驱动程序中用
 * eth_type_trans() 返回值设置该字段）
 */

kmemcheck_bitfield_end(flags1);

void         (*destructor)(struct sk_buff *skb); /* Destruct function
 * 这个函数指针可以初始化成一个在缓冲区释放时完成某些动作的函数。如果缓冲区不属于一个 socket，这个函
 * 数指针通常是不会被赋值的。如果缓冲区属于一个 socket，这个函数指针会被赋值为 sock_rfree 或 sock_wfree
 * （分别由 skb_set_owner_r 或 skb_set_owner_w 函数初始化）。这两个 sock_xxx 函数用于更新 socket 的队列中的
 * 内存容量。
 */

#ifdef CONFIG_NF_CONNTRACK || defined(CONFIG_NF_CONNTRACK_MODULE)
/*
 * 当内核支持连接跟踪，则 skb->nfct 指向连接跟踪的&ct->general，它是一个 nf_conntrack 结构。
 * struct nf_conntrack {
 *     atomic_t use;
 * };
 */
struct nf_conntrack *nfct;          /* Associated connection, if any */
struct sk_buff      *nfct_reasm;    /* netfilter conntrack re-assembly pointer */
#endif

#ifdef CONFIG_BRIDGE_NETFILTER
/*
 * 当 skb_buff 同步桥模式，并且桥下支持 netfilter，则为*nf_bridge 指针分配该结构，用于保存一下私有信息。
 * struct nf_bridge_info {
 *     atomic_t use;
 *     struct net_device *physindev;
 *     struct net_device *physoutdev;
 *     unsigned int mask;
 *     unsigned long data[32 / sizeof(unsigned long)];
 */
```



```

    * };
    */
    struct nf_bridge_info    *nf_bridge;
#endif

    int            skb_iif;                /* ifindex of device we arrived on */
#ifdef CONFIG_NET_SCHED
    __u16          tc_index;                /* traffic control index */
#ifdef CONFIG_NET_CLS_ACT
    __u16          tc_verd;                /* traffic control verdict */
#endif
#endif

    __u32          rxhash;                /* the packet hash computed on receive */

    kmemcheck_bitfield_begin(flags2);
    __u16          queue_mapping:16;        /* Queue mapping for multiqueue devices */
#ifdef CONFIG_IPV6_NDISC_NODETYPE
    __u8           ndisc_nodetype:2;        /* router type (from link layer) */
#endif
    kmemcheck_bitfield_end(flags2);

    /* 0/14 bit hole */

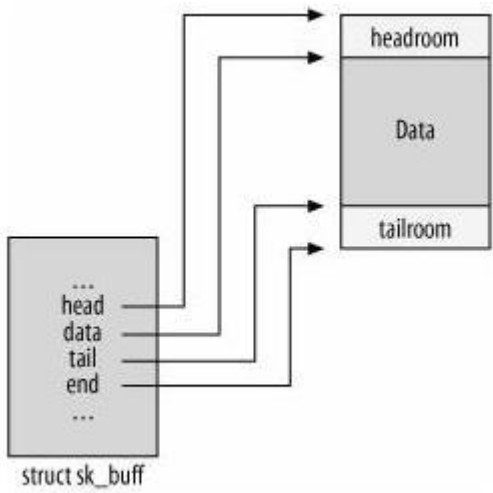
#ifdef CONFIG_NET_DMA
    dma_cookie_t    dma_cookie;            /* a cookie to one of several possible DMA operations done by skb DMA functions */
#endif
#ifdef CONFIG_NETWORK_SECMARK
    __u32          secmark;                /* security marking */
#endif
    union {
        __u32      mark;                    /* Generic packet mark */
        __u32      dropcount;                /* ***** */
    };

    __u16          vlan_tci;                /* vlan tag control information */

    sk_buff_data_t  transport_header;        /* Transport layer header */
    sk_buff_data_t  network_header;          /* Network layer header */
    sk_buff_data_t  mac_header;              /* Link layer header */

    /* These elements must be at the end, see alloc_skb() for details.  */

```



```

/*
 * 它们表示缓冲区和数据部分的边界。在每一层申请缓冲区时，它会分配比协议头或协议数据大的空间。Head 和 end 指向缓冲区的头部和尾部，而 data 和 tail 指
 * 向实际数据的头部和尾部。每一层会在 head 和 data 之间填充协议头，或者在 tail 和 end 之间添加新的协议数据。注意，这里的 end 和 tail 均不包含末尾地址，
 * 即它们都是指向相关缓冲区的末尾地址+1。注意：接收数据包时，数据包在哪一层协议处理，data 指针就指向那一层协议的头部，不要擅自改变 data 指针，除
 * 非你要将数据包交给另一层协议处理。
 */
sk_buff_data_t  tail;                    /* Tail pointer */
sk_buff_data_t  end;                    /* End pointer */
unsigned char    *head,                  /* Head of buffer */
                *data;                    /* Data head pointer */

//unsigned int   truesize;                /* Buffer size */
atomic_t         users;                  /* User count - see {datagram,tcp}.c */
};

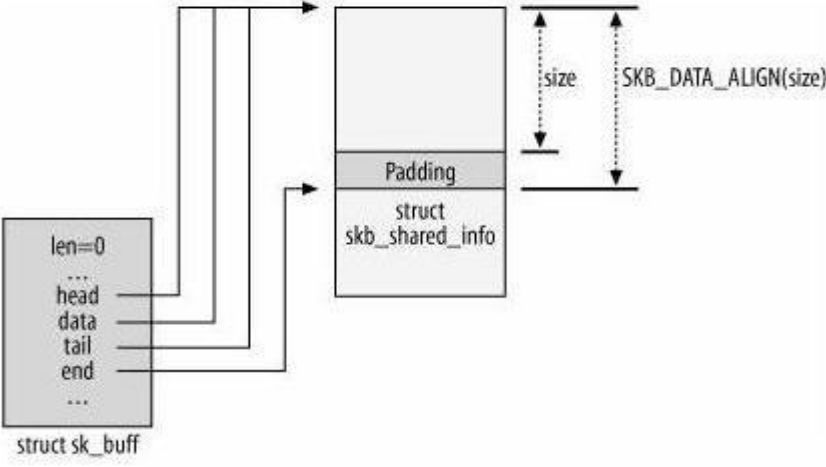
```

/*

- * 在数据缓冲区的末尾，有一个数据结构 `skb_share_info`，它保存了数据块的附加信息。这个数据结构紧跟在 `end` 指针所指的地址之后（`end` 指针指示数据的末尾）。

```

/* sk_buff 中没有指向 sk_share_info 结构指针，如果要访问这个结构，就需要使用 skb_info 宏，这个宏简单的返回 end 指针。
*/
#define skb_shinfo(SKB)      ((struct skb_shared_info *)((SKB)->end))



/*
 * This data is invariant across clones and lives at the end of the header data, ie. at skb->end.
 * skb_shared_info{}结构并没有定义在 struct sk_buff{}里面，而是在创建 skbuff 时与主缓存区一起动态分配的，位于主缓存区的后面，我们可以用 struct sk_buff{}的成员
 * end 来访问。事实上，内核就是用宏 skb_shinfo(SKB)来访问 skbuff 的 skb_shared_info{}结构的。
 */
struct skb_shared_info {
    unsigned short    nr_frags;      /* 表示 frags[]中挂载了多少个分散/聚集 I/O 缓冲区。并不表示 frag_list 链表中 IP 片段的数目。 */
    unsigned short    gso_size;
    /* Warning: this field is not always filled in (UFO)! */
    unsigned short    gso_segs;
    unsigned short    gso_type;
    __be32            ip6_frag_id;
    union skb_shared_tx tx_flags;
    struct sk_buff *frag_list;      /* frag_list 里的数据代表的是独立缓冲区，也就是每个缓冲区都必须作为单独的 IP 片段而独立传输 */
    struct skb_shared_hwtstamps hwtstamps;

    atomic_t dataref;      /* Warning : all fields before dataref are cleared in __alloc_skb()
        * 表示主缓存区的共享计数，skb_clone 会增加该引用计数。需要指出的是，分散聚集 I/O 缓存区和 frag_list 链表中的 skbuff 都有它们各自独
        * 立的共享计数。因此增加 dataref 引用计数，并不需要相应增加分散聚集 I/O 缓存区和 frag_list 链表中的 skbuff 的引用计数。这是因为虽然
        * dataref 增加了，但主缓存区（包括 skb_shared_info）仍然只有一份，它仍然只拥有一份对分散聚集 I/O 缓存区和 frag_list 链表中的 skbuff
        * 的引用。
        */
};

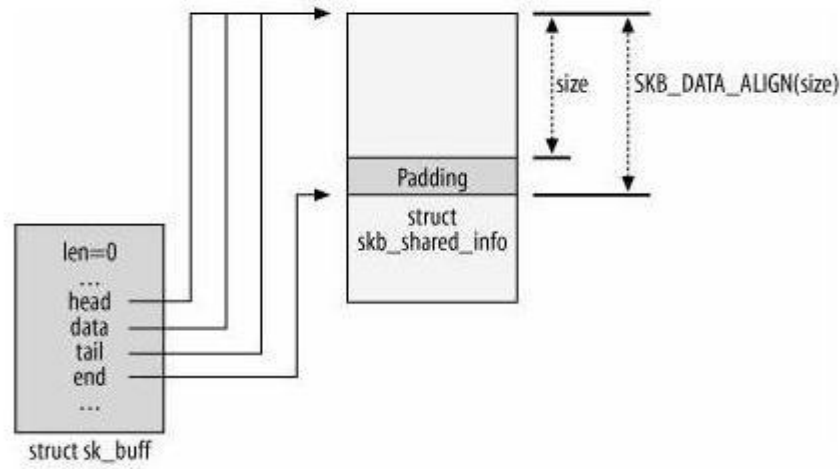
/*
 * 下面的结构变量用于支持分散/聚集 I/O 处理的，它与 IP 数据分段是各自独立的。分散/聚集 I/O 只是让程序和硬件可以使用非相邻内存区域，就好像它们是相邻
 * 的那样。然而每个片段依然必须尊重其最大尺寸（PMTU）的限制。也就是说，即使 PAGE_SIZE 大于 PMTU，当 sk_buff 里的数据（由 skb->data 所指）加上由 frags
 * 所引用的数据片段达到 PMTU 时，就需要建立一个新的 sk_buff。
 * 记住：frags 向量里的数据是主缓冲区中数据的扩展，这些 frags 引用的数据片段不需要任何报头，因为一个 sk_buff 实例的所有数据片段都是和同一个 IP 封包相
 * 关。
 * 分散/聚合 I/O 功能是否使用，要看输出的网络设备是否具备该功能，如果该网络设备具备该功能，则 netdev->features 会设置 NETIF_F_SG 标志。
 * #define MAX_SKB_FRAGS (65536/PAGE_SIZE + 2) //To allow 64K frame to be packed as single skb without frag_list
 * struct skb_frag_struct {
 *     struct page *page;      //指向内存页面指针。
 *     __u32 page_offset;      //页面开端相对偏移量。
 *     __u32 size;             //该片段的大小。
 * };
 */
skb_frag_t    frags[MAX_SKB_FRAGS];    /* 在支持分散/聚集 I/O 时，除第一个之外的每个缓冲区都存在这个指针数组指向的内存页中。 */

void *        destructor_arg;          /* Intermediate layers must ensure that destructor_arg remains valid until skb destructor */
};
```

sk_buff 数据结构的管理

sk_buff 的分配

- 1 `alloc_skb(size, gfp_mask)` 用于分配 sk_buff 管理结构和其管理的数据缓冲区。这就意味着，需要分配两块内存（一个是缓冲区，一个是缓冲区的管理结构 sk_buff）。参数 gfp_mask 是分配函数标志（被用于 kmalloc 等函数）。参数 size 表示调用者所要求的数据缓冲区大小，它要通过 SKB_DATA_ALIGN 宏强制对齐。并且在数据缓冲区尾部要包含一个 skb_shared_info 结构（该结构主要是用来处理 IP 分片）。分配完后它还要初始化它们，将 sk_buff 与数据缓冲区相关联。如下图所示：



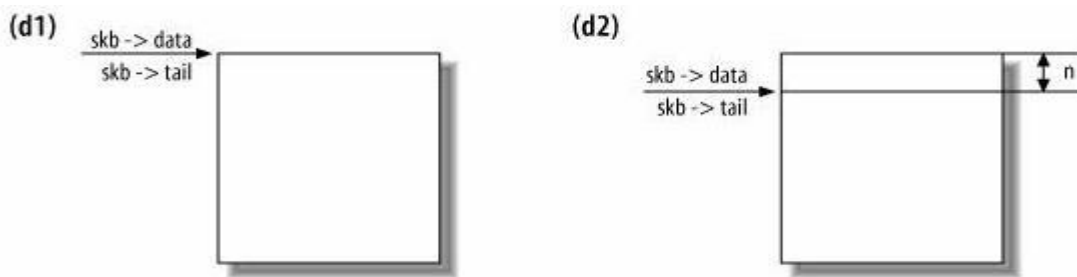
- 2 **dev_alloc_skb(length)** 也用于分配 sk_buff 管理结构和其管理的数据缓冲区。它主要被设备驱动使用，通常用在中断上下文中。这是一个 alloc_skb 函数的包装函数，它会在请求分配的大小上增加 NET_SKB_PAD（64）字节的空间防止网络层增长头部空间而又重新分配 sk_buff，它的分配要求使用原子操作（GFP_ATOMIC），这是因为它是在中断处理函数中被调用的。

sk_buff 的释放

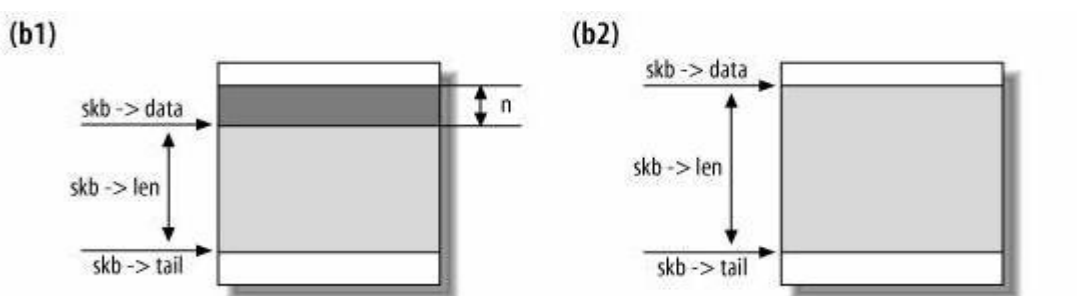
- 1 **kfree_skb(*skb)** 与 alloc_skb() 配对使用。用于释放 skb 指向的 sk_buff 管理结构和其数据缓冲区（主缓存区，分散/聚集 I/O 缓存区，skb_shinfo(SKB)->frag_list 链表的缓存区）。该函数只有在 skb->user 为 1 的情况下才真正释放（没有人引用这个结构）。否则，它只是简单的减少 skb->users。
- 2 **dev_kfree_skb(*skb)** 与 dev_alloc_skb() 配对使用。dev_kfree_skb 仅仅是一个简单的宏，它什么都不做，只简单的调用 kfree_skb()。

sk_buff 的数据缓存区管理

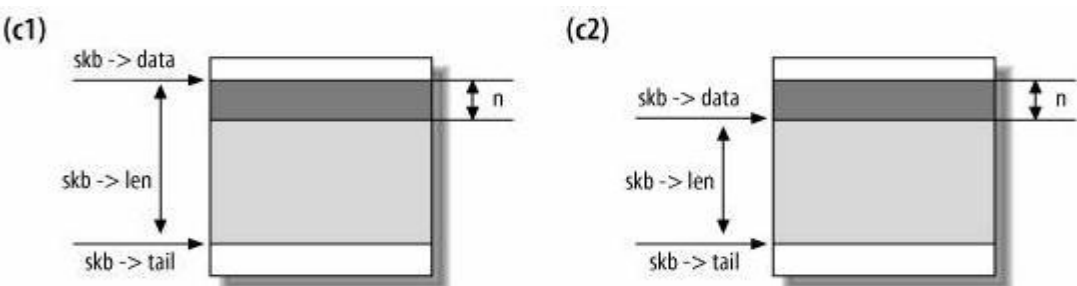
- 1 sk_buff 处理代码提供了标准的机制用于在 skb 数据上增加和删除协议头和尾。这种代码安全地操纵 sk_buff 中的 data、tail 和 len 字段。处理程序包括：
 - 1.1 **skb_reserve(skb, len)** 把 data 和 tail 指针都从数据区域起始向结尾移动，并不改变 len。用于在缓冲区头部预留一定空间，它通常被用来在缓冲区中插入协议头或者在某个边界上对齐。这个函数通常在分配缓冲区之后就调用，此时的 data 和 tail 指针还是指向同一地方。



- 1.2 **skb_push(skb, len)** 用于添加数据到 skb 主缓存区头部。它将 skb->data 向前移动 len 长度，并增加 skb->len 的长度 (skb->len += len)，返回移动后的 data 指针位置。它会检查 data 是否溢出主缓存区空间 (skb->data < skb->head)。但它不考虑分散/聚集 I/O 缓存区和 skb_shinfo(SKB)->frag_list 列表缓存区（也不需要考虑这两个缓存区）。

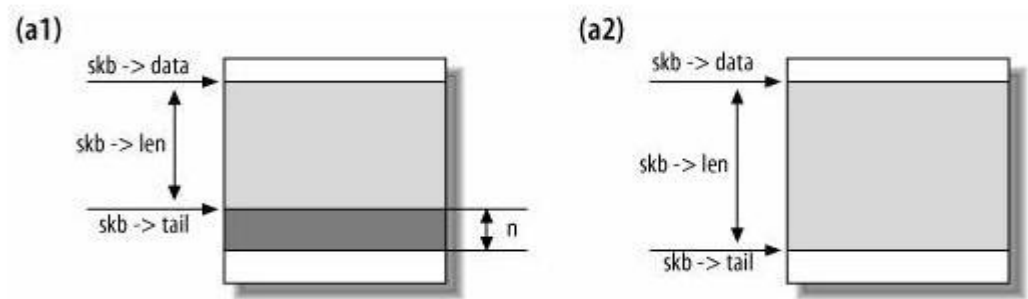


- 1.3 **skb_pull(skb, len)** 用于从 skb 主缓存区头部删除数据。它将 skb->data 向后移动 len 长度，并减少 skb->len 的长度 (skb->len -= len)，返回移动后的 data 指针位置。它会检查移动长度 len 是否超过 skb->len 长度 (len > skb->len)。但它不考虑分散/聚集 I/O 缓存区和 skb_shinfo(SKB)->frag_list 列表缓存区。（与 skb_push() 相反）



- 1.4 **pskb_pull(skb, len)** 用于从 skb 主缓存区头部删除数据。它将 skb->data 向后移动 len 长度，并减少 skb->len 的长度 (skb->len -= len)，返回移动后的 data 指针位置。如果主缓存区中数据达不到 len 长度，则调用 _pskb_pull_tail 函数重新构建一个更大的主缓存区，并从分散/聚集 I/O 缓存区和 skb_shinfo(SKB)->frag_list 链表缓存区中拷贝数据以满足 len 长度（被移动的数据会从相应的缓存区中删除）。它会对 len 是否超过 skb->len 长度做检查 (len > skb->len)。

- 1.5 `pskb_may_pull(skb, len)` 用于检查主缓存区中数据长度是否满足 len，如果不满足，则调用__pskb_pull_tail 函数重新构建一个更大的主缓存区，并从分散/聚集 I/O 缓存区和 skb_shinfo(SKB)->frag_list 链表缓存区中拷贝数据以满足 len 长度（被移动的数据会从相应的缓存区中删除）。但它不改变任何 skb 的指针（例如 skb->data）。
- 1.6 `skb_put(skb, len)` 用于添加数据到 skb 主缓存区尾部。它将 skb->tail 向后移动 len 长度，并增加 skb->len 的长度（skb->len += len），返回移动前的 tail 指针位置。它会检查 tail 是否溢出主缓存区空间（skb->tail > skb->end）。但它不考虑分散/聚集 I/O 缓存区和 skb_shinfo(SKB)->frag_list 列表缓存区。

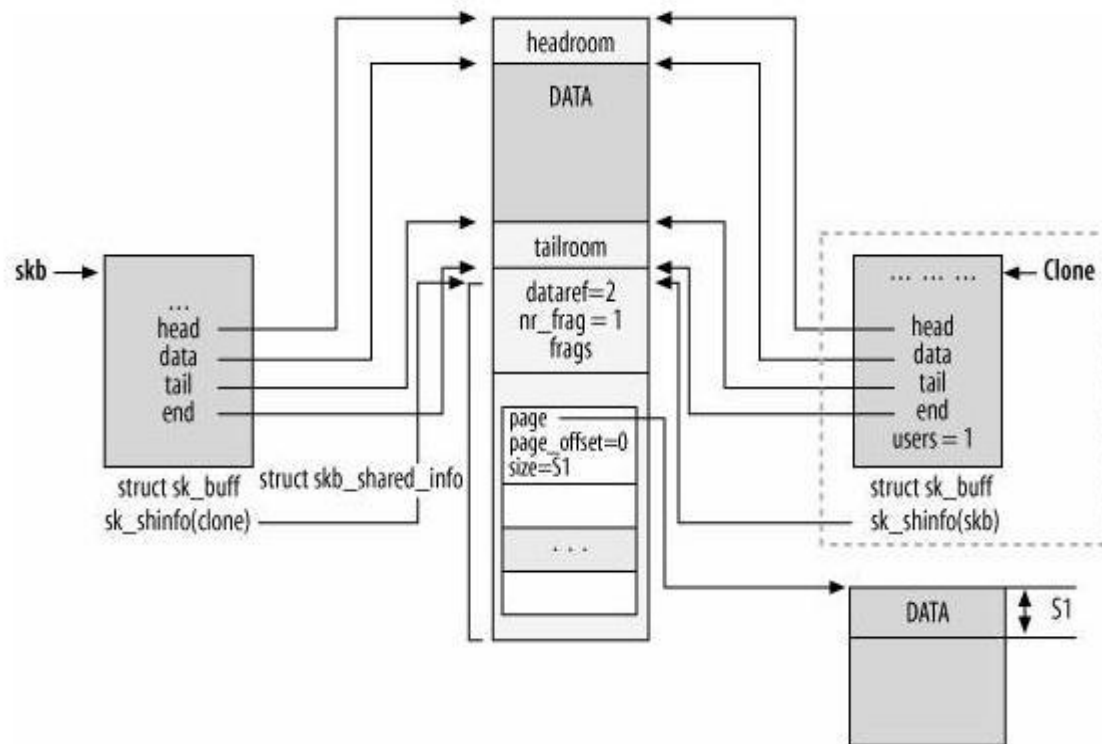


- 1.7 `skb_trim(skb, len)` 用于将 skb 有效载荷数据长度减小为 len(释放末尾数据)，它将 tail 指针向前移动来释放末尾数据，然后修改 skb 的数据长度（skb->len = len），无返回值。它会检查 len 是否大于数据包有效载荷长度（skb->len），如果大于将不做任何处理。（与 `skb_put()` 相反）
- 1.8 `pskb_trim(skb, len)` 用于将 skb 有效载荷数据长度减小为 len(释放末尾数据)，它首先释放 skb_shinfo(SKB)->frag_list 链表缓存区数据，其次释放分散/聚集 I/O 缓存区数据，最后释放主缓存数据(主缓存数据通过修改 tail 指针来达到释放目的)，直到满足要求为止（skb->len = len）（注意，只有释放到主缓存区数据（len < skb_headlen(skb)）才会修改 skb->tail 指针）。然后修改 skb 的数据长度（skb->len = len），成功则返回 0，失败返回其它值。它会检查 len 是否大于数据包有效载荷长度（skb->len），如果大于将不做任何处理。
- 1.9 `skb_headroom(skb)` 返回 skb 主缓存区中数据之前的空闲空间，即介于 skb->head 和 skb->data 之间的可用空间。
- 1.10 `skb_tailroom(skb)` 返回 skb 主缓存区中数据之后的空闲空间，即介于 skb->tail 和 skb->end 之间的可用空间。但是如果 skb 包含分散/聚集 I/O 缓存数据或 frag_list 链表缓存数据（skb->data_len > 0），则返回 0。
- 1.11 `skb_headlen(skb)` 返回主缓冲区中的数据量，即不计算分散/聚集 I/O 缓存区数据，也不考虑 frag_list 链表缓存区数据。
- 1.12 `skb_pagelen(skb)` 返回主缓冲区数据加上分散/聚集 I/O 缓存区数据长度，但是，不考虑任何 frag_list 链表缓存区数据。
- 1.13 `skb_is_nonlinear(skb)` skb 如果是非线性的（包含分散/聚集 I/O 缓存区数据或 frag_list 链表缓存区数据），则返回值大于 0。否则，返回 0，表明是线性的（只有主缓存区数据）。
- 1.14 `skb_linearize(skb)` skb 如果是非线性的(包含分散/聚集 I/O 缓存区数据或 frag_list 链表缓存区数据)，则调用__pskb_pull_tail() 重新构建一个足够大（skb->len）的主缓存区，并从分散/聚集 I/O 缓存区和 skb_shinfo(SKB)->frag_list 链表缓存区中拷贝全部数据到主缓存区，并释放掉这两个缓存区（skb->data_len = 0）。成功返回 0，失败返回非 0。

sk_buff 的克隆和拷贝

- 1 当同一个缓存区需要由不同消费者个别处理时，那些消费者可能需要修改 sk_buff 描述符的内容（指向协议报头的 `transport_header` 和 `network_header`），但内核不需要完全拷贝 sk_buff 结构和相关联的数据缓存区。相反，为了提高效率，内核可以克隆（clone）原始值，也就是拷贝 sk_buff 结构，然后对数据缓存区使用引用计数，以免过早释放共享的数据块。使用 sk_buff 克隆的一个例子是，当一个输入数据包需要传递给多个接收者时，如协议处理例程和一个或多个网络分流器。
- 1.1 `skb_clone(skb, gfp_mask)` 用于克隆一个 skb 的 sk_buff 结构，克隆出得 sk_buff 没有链接到任何链表，而且也没有引用套接字的拥有者。skb->cloned 字段在克隆的和原有的 sk_buff 中都置为 1，而克隆的 skb->users 也置为 1。但是，对包含数据的缓冲区的引用数目（skb_shinfo(skb)->dataref）则会递增（因为从现在起，又有一个 sk_buff 数据结构指向了该区）。如下图所示：

Figure 2-9. skb_clone function



- 1.2 `skb_cloned(skb)` 用于测试 `skb` 是否被克隆。如果是 (`skb->cloned` 为 1 并且数据缓冲区引用计数 `skb_shinfo(skb)->dataref!=1`)，则返回 1，否则返回 0。
 - 1.3 `skb_shared(skb)` 用于测试 `skb` 是否被多个实例共享。如果是 (`skb->user != 1`) 则返回 1，否则返回 0。
 - 1.4 `skb_share_check(skb, gfp_mask)` 用于检查 `skb` 是否被共享（通过 `skb_shared()` 进行检查），如果是则克隆一个新的 `sk_buff`，并返回它的指针。如果不是，则返回原 `skb` 指针。
- 2 当一个 `sk_buff` 被克隆时，这个 `sk_buff` 数据缓冲区的内容就不能被修改。这就意味着，访问该数据缓冲区的代码就不需要加锁。然而，当函数不仅需要修改 `sk_buff` 结构的内容，而且也需要修改缓冲区数据时，就必须连数据缓冲区一起克隆。在这种情况下，有下面两个选择：
- 2.1 `pskb_copy(skb, gfp_mask)` 当只需修改介于 `skb->head` 和 `skb->end` 之间的数据内容时，可以使用 `pskb_copy` 只克隆该区域。函数返回克隆出的新 `sk_buff` 指针。这个新 `sk_buff` 主缓存区长度等于原 `skb` 主缓存区长度，原 `skb` 主缓存区数据也被拷贝到新缓存区中（**注意：它仅拷贝 `skb->data` 到 `skb->tail` 之间的数据到新缓冲区，而 `skb->data` 到 `skb->head` 之间的数据不被拷贝。例如 `skb->data` 当前指向 3 层头部，那么 2 层头部将不被拷贝到新缓冲区**）。并对分散/聚集 I/O 缓存区中 `page` 页面和 `skb_shinfo(SKB)->frag_list` 链表中所有 `sk_buff` 的引用计数加 1。

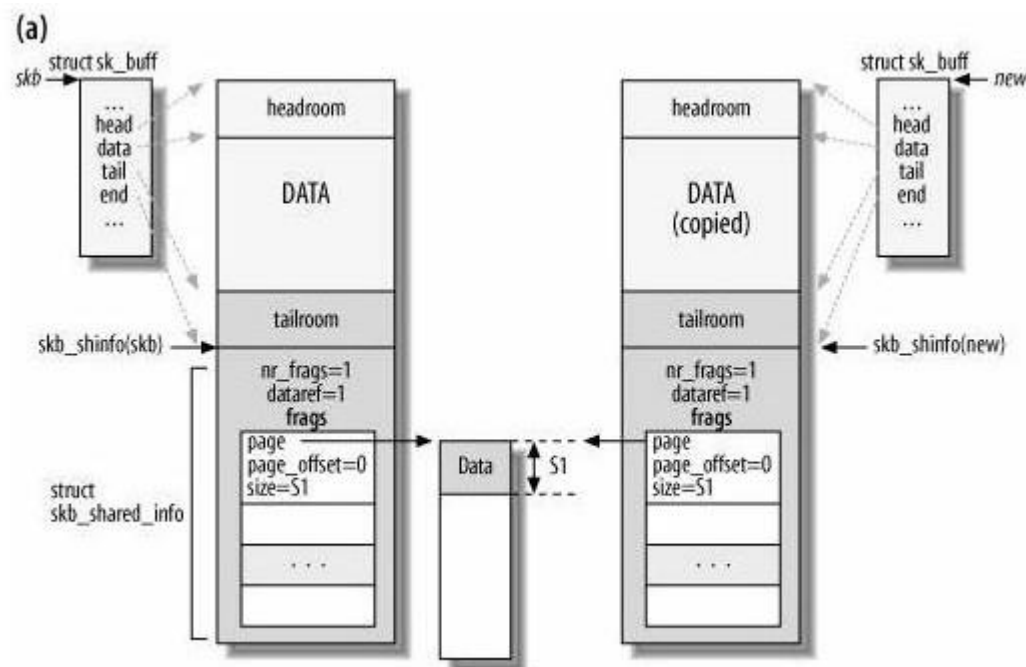
- 1.3 `skb_shared(skb)` 用于测试 `skb` 是否被多个实例共享。如果是 (`skb->user != 1`) 则返回 1, 否则返回 0。

- 1.4 `skb_share_check(skb, gfp_mask)` 用于检查 `skb` 是否被共享（通过 `skb_shared()` 进行检查），如果是则克隆一个新的 `sk_buff`，并返回它的指针。如果不是，则返回原 `skb` 指针。

- 2 当一个 `sk_buff` 被克隆时，这个 `sk_buff` 数据缓冲区的内容就不能被修改。这意味着，访问该数据缓冲区的代码就不需要加锁。然而，当函数不仅需要修改 `sk_buff` 结构的内容，而且也需要修改缓冲区数据时，就必须连数据缓冲区一起克隆。在这种情况下，有下面两个选择：

- 2.1 `pskb_copy(skb, gfp_mask)` 当只需修改介于 `skb->head` 和 `skb->end` 之间的数据内容时，可以使用 `pskb_copy` 只克隆该区域。函数返回克隆出的新 `sk_buff` 指针。这个新 `sk_buff` 主缓存区长度等于原 `skb` 主缓存区长度，原 `skb` 主缓存区数据也被拷贝到新缓存区中（注意：它仅拷贝 `skb->data` 到 `skb->tail` 之间的数据到新缓冲区，而 `skb->data` 到 `skb->head` 之间的数据不被拷贝。例如 `skb->data` 当前指向 3 层头部，那么 2 层头部将不被拷贝到新缓冲区）。并对分散/聚集 I/O 缓存区中 `page` 页面和 `skb_shinfo(SKB)->frag_list` 链表中所有 `sk_buff` 的引用计数加 1。

(a) pskb_copy function



- 2.2 `skb_copy(skb, GFP_MASK)` 当同时需要修改分散/聚集 I/O 缓存区数据和 `skb_shinfo(SKB)->frag_list` 链表的缓存区数据时, 就必须使用 `skb_copy()` 函数克隆。函数返回克隆出的新 `sk_buff` 指针。这个新 `sk_buff` 的主缓存区长度能包含所有数据区数据 (`skb->end - skb->head + skb->data_len`), 它将 `skb` 的分散/聚集 I/O 缓存区和 `skb_shinfo(SKB)->frag_list` 链表缓冲区数据都拷贝到新分配的主缓存区中。

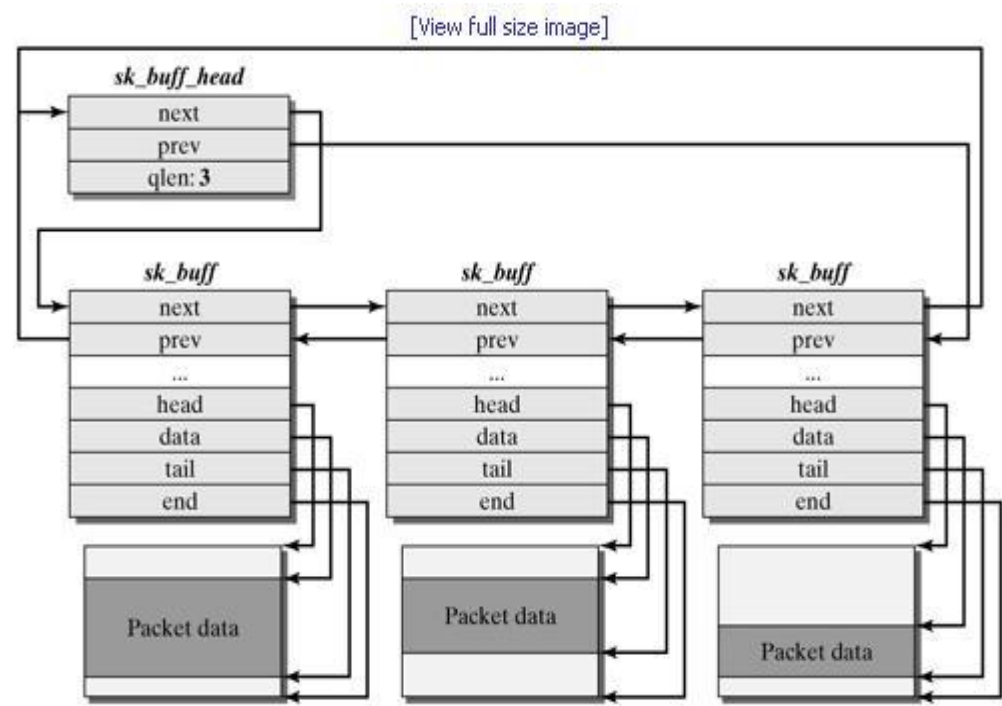
- 3 在决定克隆或者拷贝一个 `sk_buff` 缓冲区时，每个子系统的程序员无法预测其他内核组件（或其子系统）是否需要该缓冲区里的原始数据。内核是非常模块化的，而且以非常动态而无法预测的方式改变，所以，每个子系统都不知道其他子系统将如何操作这个缓冲区。因此，每个子系统的程序员只需记录其对该缓冲区所做的任何修改，而且在修改任何东西之前先做个拷贝，以免内核其它部分需要原有信息。

sk_buff 的链表管理

- 1 一个 `sk_buff` 可以链接到任何一个队列中，例如 TCP 发送队列中。同时，内核为了加快 `sk_buff` 的分配与释放，也专门为此分配了一个 `sk_head_pool` 缓存队列。该队列作用是将 `sk_buff` 管理结构缓存起来，当调用 `alloc_skb()` 分配一个 `sk_buff` 结构时，会首先到这个队列中提取，如果这个队列为空，则才调用 `kmem_cache_alloc` 从内存 `cache` 中分配一个 `sk_buff` 结构。当调用 `kfree_skb()` 释放一个 `sk_buff` 结构时，仅仅是将这个结构放入 `sk_head_pool` 缓存队列中。但这个队列也有长度限制，大

小保存在 `sysctl_hot_list_len`（128）中。如果队列中的个数超过了这个值，则调用 `kmem_cache_free` 将 `sk_buff` 释放到内存 `cache` 中。

1.1 下面是队列管理相关的函数，这些函数都是原子操作，它们必须先获取 `sk_buff_head` 中的自旋锁，然后才能访问队列中的元素。否则，它们有可能被其它异步的添加或删除操作打断，比如在定时器中调用的函数，这将导致链表出现错误而使得系统崩溃。



1.1.1 `skb_queue_head_init()` 作用是初始化 `sk_buff_head` 结构，创建一个空队列。

1.1.2 `skb_queue_head()` 和 `skb_queue_tail()` 作用是把一个缓冲区加入队列的头部或尾部。

1.1.3 `skb_dequeue()` 和 `skb_dequeue_tail()` 作用是从队列的头部或尾部取下一个缓冲区。

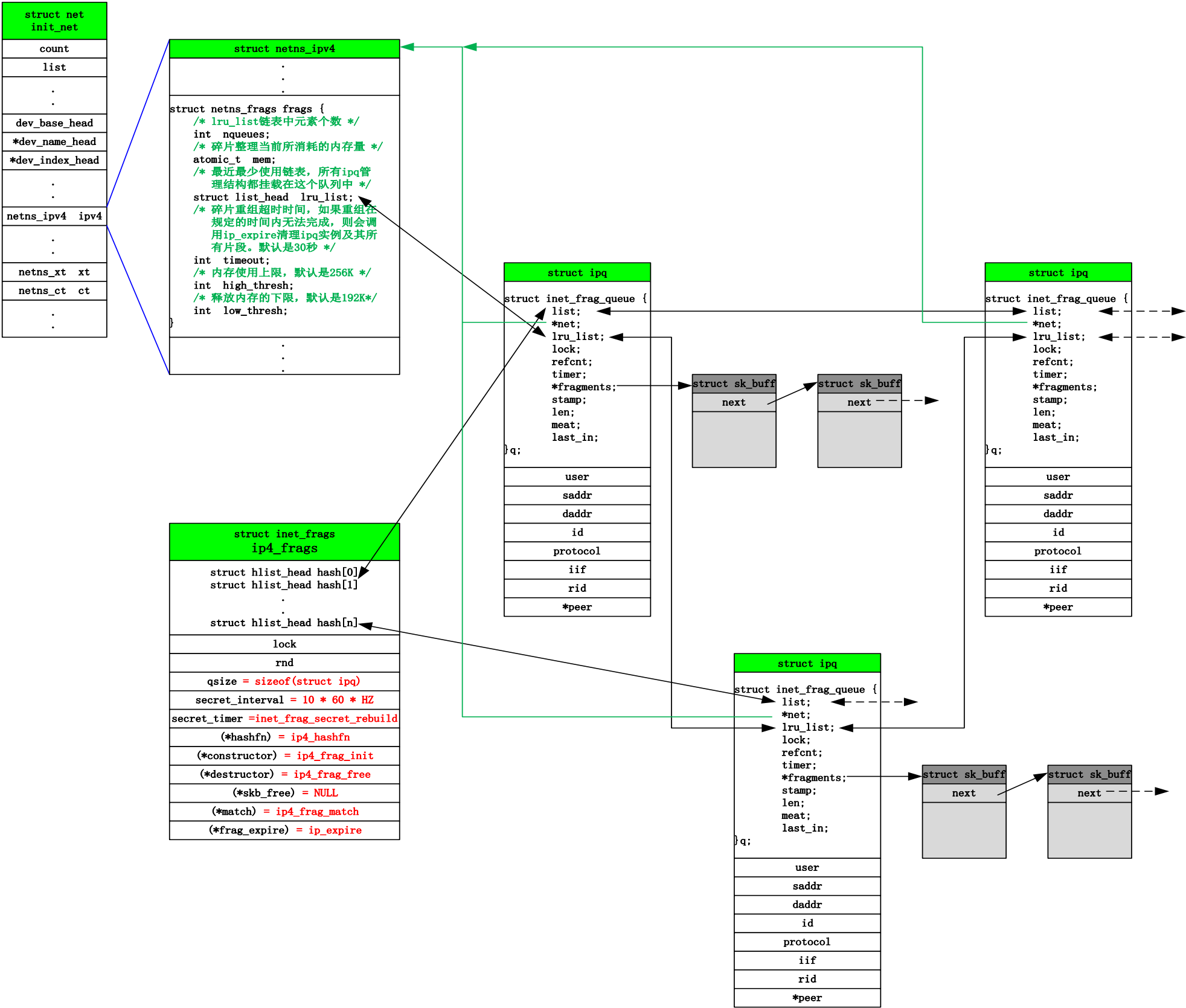
1.1.4 `skb_queue_purge()` 作用是清空一个队列。

1.1.5 `skb_queue_walk()` 按顺序遍历队列中的每一个元素。这个遍历不能用于删除元素。

1.1.6 `skb_queue_walk()` 按顺序遍历队列中的每一个元素。这个遍历可用于删除元素。

sk_buff 的分片与重组（IP 数据报文）

- 1 无论是分片后的 IP 数据报文还是重组后的 IP 数据报文，其表现形式都是上面所总结的管理形式。
- 2 对于分片功能（`ip_fragment`），其接收到一个 IP 数据报文，然后直接将其切割成多个 IP 片段（根据 `PMTU`）发送出去。
- 3 对于重组功能（`ip_defrag`），其先将接收到的 IP 片段缓存起来，当一个 IP 数据报文的所有片段都收集齐后，再将其重组成一个 IP 数据报文并交给上层处理。由于重组功能需要缓存 IP 片段，所以它创建了如下图所示的管理结构：



- 3.1 每个 ipq 代表一个 IP 数据报文，它通过 fragments 链接每个接收到的片段（每个片段都由一个 sk_buff 管理），这些片段都是按 offset 进行排序的。
- 3.2 每个 ipq 都有一个超时定时器（默认是 30 秒），如果定时器超时还未收齐所有片段，则销毁这个 ipq 上接收到的所有片段，并返回 ICMP 通告信息。

帧的接收

内核为驱动提供的接收接口

1 NAPI 接口

- 1.1 __napi_schedule() 将 napi_struct 结构的数据挂入 softdata.poll_list 队列中。
- 1.2 napi_schedule_prep() 检查当前设备是否支持 NAPI 已经是否已经被挂入 softdata.poll_list 队列中处理了。
- 1.3 napi_schedule() 是一个包裹函数，它首先调用 napi_schedule_prep()函数进行检查，检查通过后后调用__napi_schedule()函数进行操作。
- 1.4 使用 NAPI 接口的驱动必须自己提供 poll()虚函数，用于处理数据包的接收。
- 1.5 使用 NAPI 接口的驱动必须维护自己的接收队列，并由自己的 poll()函数处理该队列的数据。

2 支持非 NAPI 接口

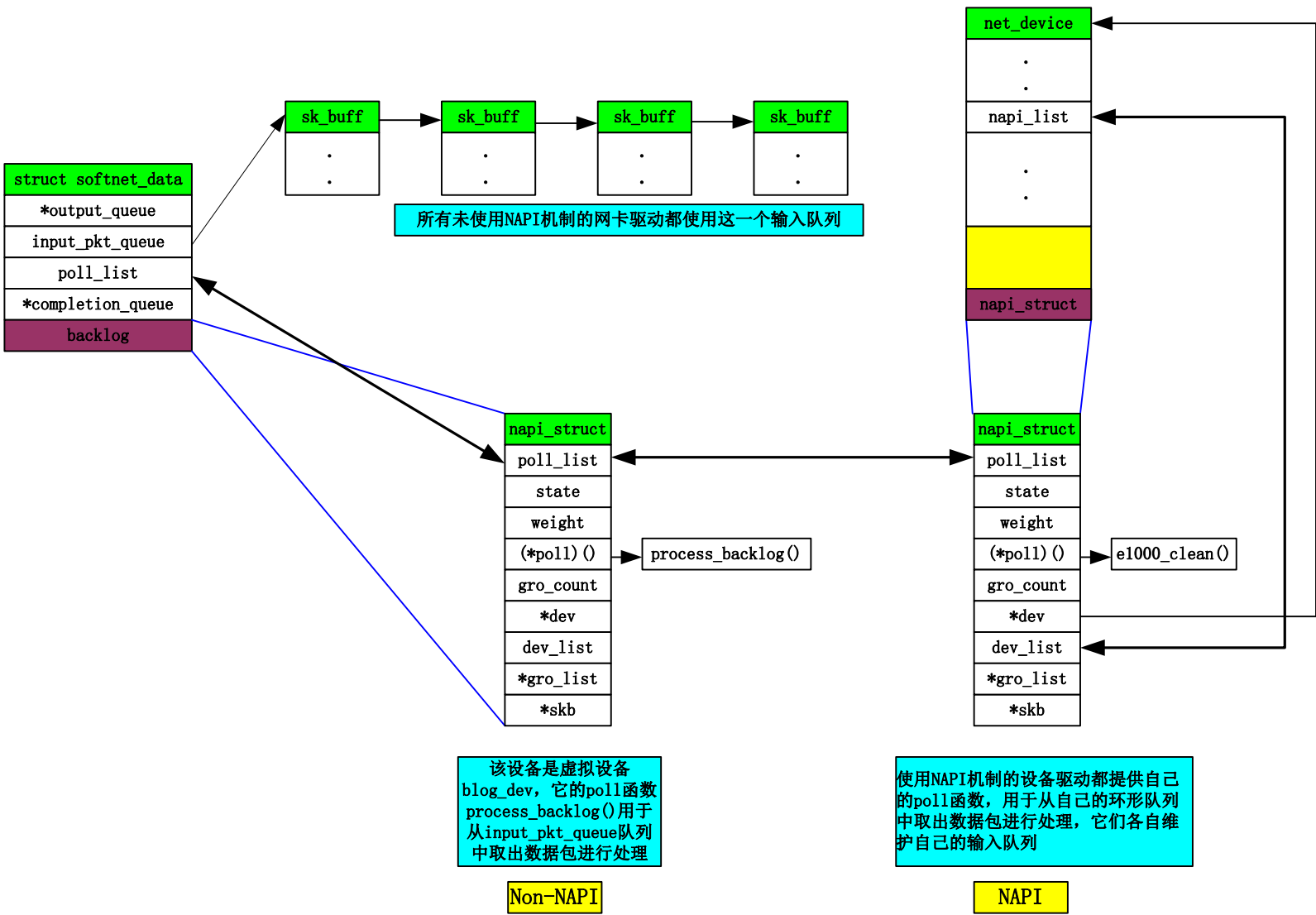
- 2.1 netif_rx() 它将数据包挂入 softdata.input_pkt_queue 队列中，然后用 softdata.backlog 作为自己的 napi 结构挂入 softdata.poll_list 队列中进行处理。
- 2.2 使用非 NAPI 的驱动使用通用的 process_backlog()接收接口。
- 2.3 使用非 NAPI 的驱动共享 softdata.input_pkt_queue 接收队列。

3 最终，无论是驱动自己提供的 poll()虚函数，还是公共 process_backlog()虚函数，都要调用统一的协议接收接口 netif_receive_skb()。

驱动如何接收数据包

- 1 网卡产生中断后，会调用相应的中断处理函数（驱动中设置的）。在中断处理函数中，都会调用上面介绍的接口函数，将数据包挂入软中断队列中，并通告软中断。
- 2 软中断调用 net_rx_action()函数处理 softdata.poll_list 队列中的所有接收项，它调用每一项的 poll()函数对各自接收队列中的数据包进行处理。（每个 poll()函数最终都会调用统一协议接收函数 netif_receive_skb()处理数据包）

接收队列的组织形式



帧的发送

内核为驱动提供的发送接口

- 1
- 内核为上层协议（第三层协议 IP、ARP 等等）提供了统一的发包接口 `dev_queue_xmit()` 函数，该函数用来发送一个 `sk_buff` 数据包。这个要发送的 `sk_buff` 必须包含所有必要的信息，例如输出设备、下一跳、链路层地址（MAC）等。
- 2
- dev_queue_xmit()函数会导致驱动程序传输函数 hard_start_xmit()通过以下两种途径之一执行：

2.1

有队列设备，衔接至流量控制（Qos 层），这是通过 `qdisc_run()` 函数实现的。

2.1.1

对于有队列设备，首先通过 `enqueue()` 将帧排入队列中，然后通过 `dequeue()` 从队列中取出一个帧，之后将这个取出的帧传输出去。如果出错（如设备关闭了队列），则可通过 `requeue()` 把之前取出的帧放回队列。

2.1.2

对于有队列设备，当传输时间过长（如超过一个 `jiffies`）时，通过 `__netif_schedule()` 将该队列放入发送软中断输出队列 `softnet_data.output_queue` 中，并启动软中断输出队列，等到下次软中断中发送。

2.2

无队列设备，直接启用 `hard_start_xmit()`，这是为那些不使用流量控制基础架构的设备（也就是虚拟设备）所做的。对于无队列的设备，每当一个帧传输时，就会立刻被传递出去。但是，因为没有队列可以让帧重新排入，如果有任何地方出错（如设备关闭了队列），帧就会被丢弃，没有第二次机会。

2.2.1

对于无队列设备，由于它没有队列，所以它也就不能通过 `__netif_schedule()` 来启动软中断发送。

驱动程序对发送控制的接口

- 1
- `netif_start_queue()` 开启设备的传输。
- 2
- `netif_stop_queue()` 关闭设备的传输。
- 3
- `__netif_schedule()` 设备调度以准备传输。即将该设备队列放入软中断输出队列中，并启动软中断发送。
- 4
- `netif_wake_queue()` 开启设备的传输，并启动软中断的输出队列。它相当于调用了 `netif_start_queue()` 和 `__netif_schedule()`。
- 5
- `dev_kfree_skb_irq()` 将已发送完的数据包放入发送软中断的 `softnet_data.completion_queue` 队列中，并开启发送软中断。

发送软中断作用和它的队列组织形式

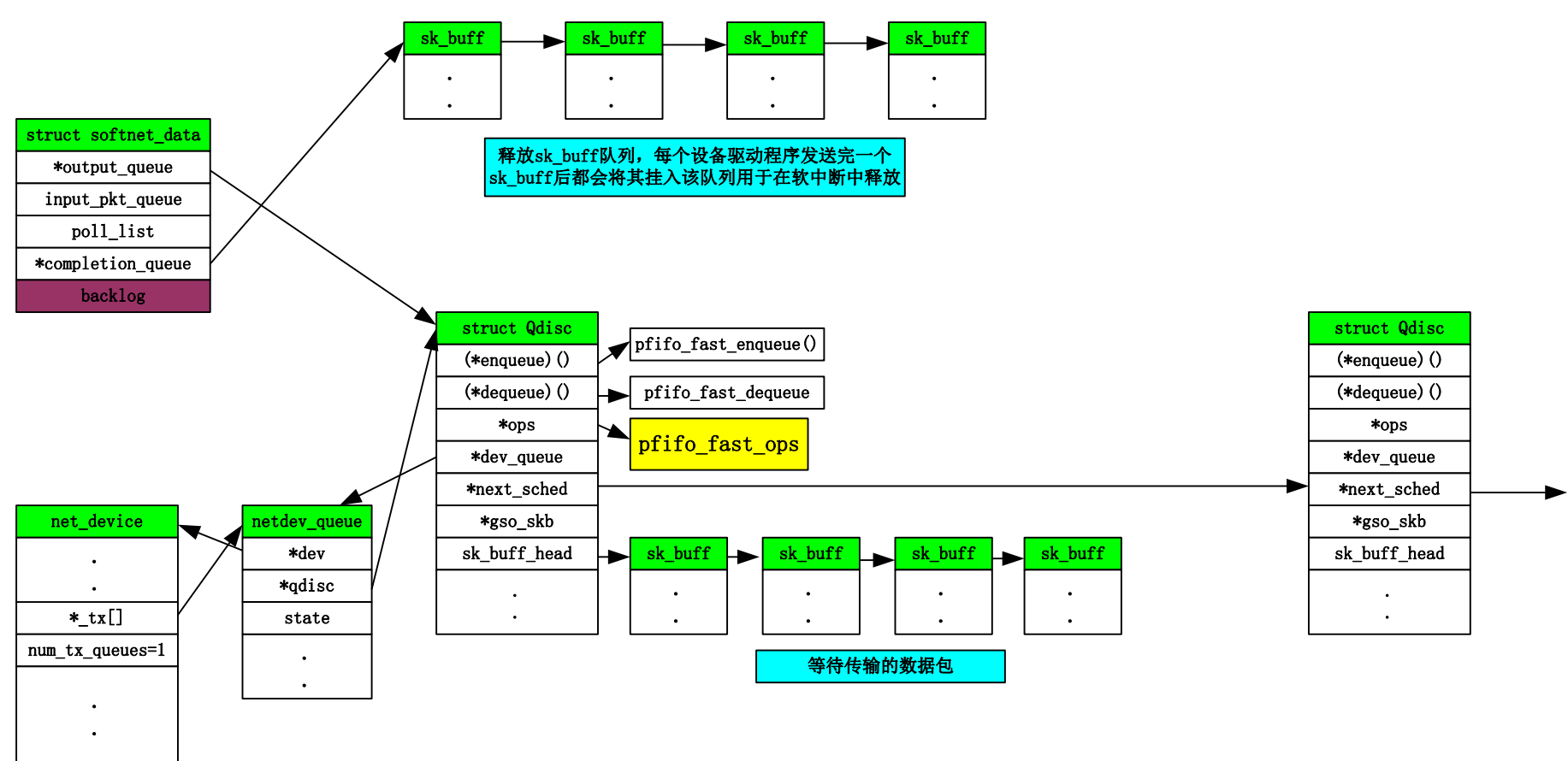
- 1
- net_tx_action() 发送软中断处理函数。此函数可由设备在两种情况下以 `raise_softirq_irqoff(NET_TX_SOFTIRQ)` 予以触发，以完成两项主要任务：

1.1

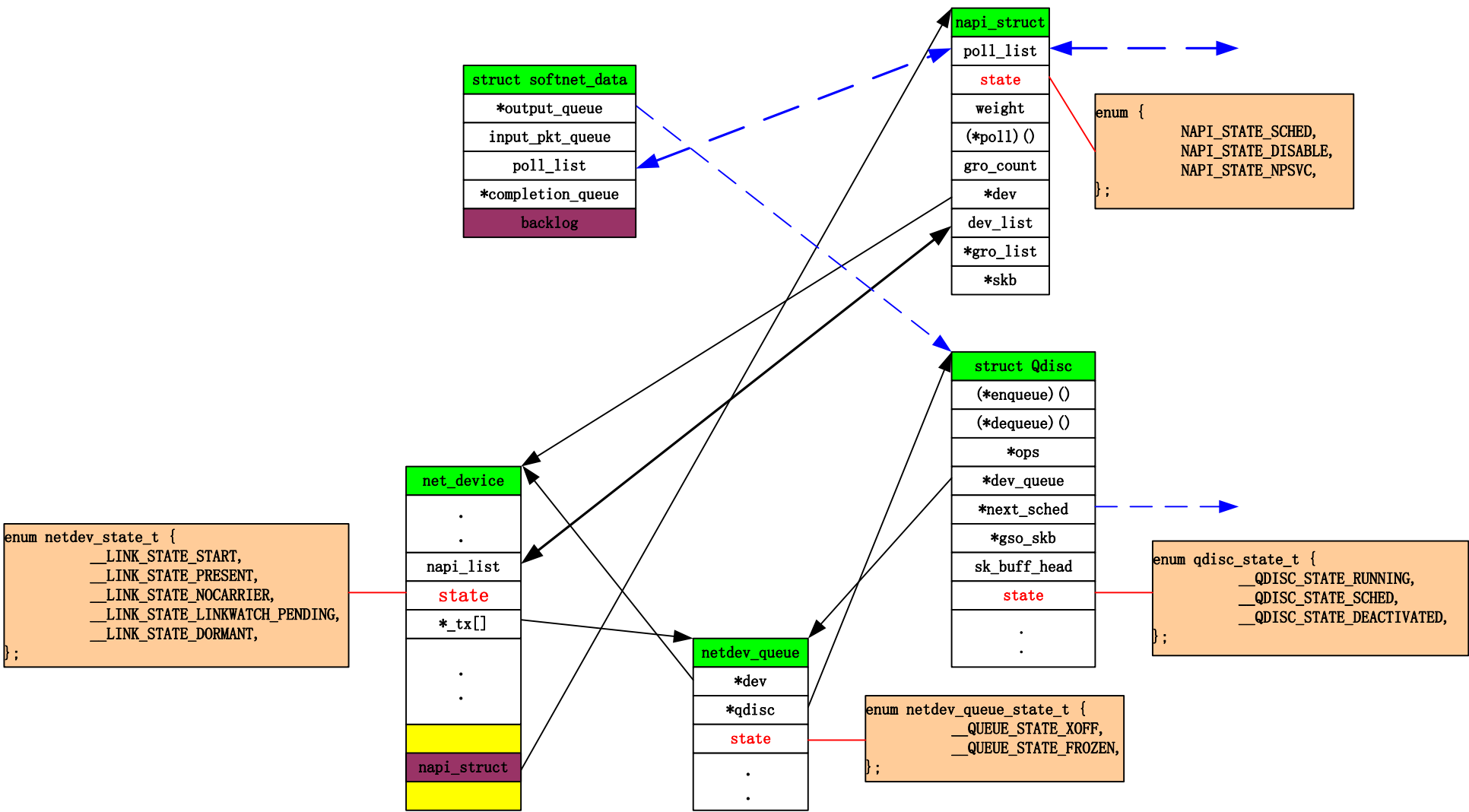
此函数要将输出队列 `softnet_data.output_queue` 中所有符合条件的帧都传送出去。

1.2

释放 `softnet_data.completion_queue` 中的帧。（驱动程序中调用 `dev_kfree_skb_irq()` 函数将发送完的帧放入 `softnet_data.completion_queue` 队列中，并启动发送软中断）
- 2
- 软中断发送队列的组织形式

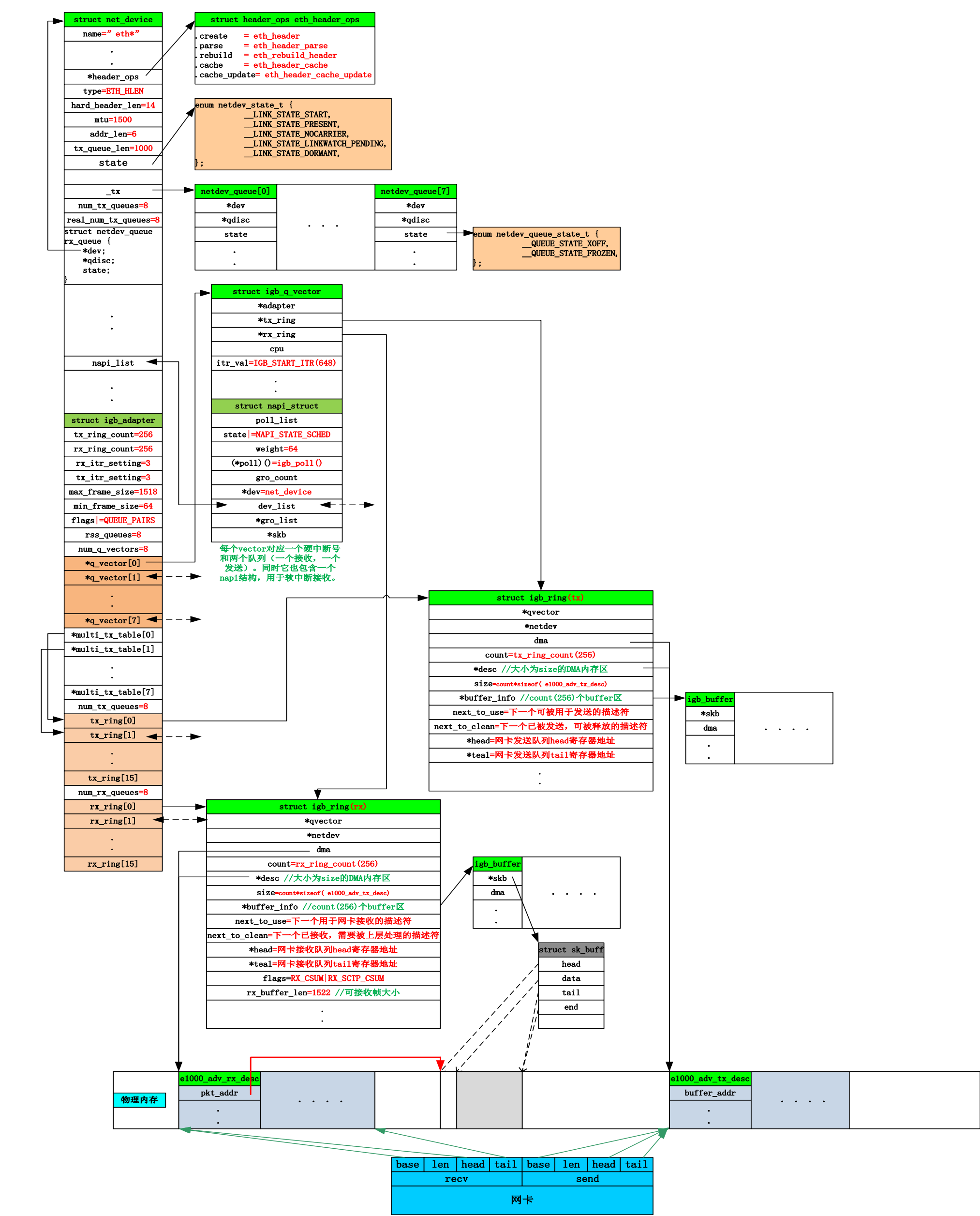


与设备、发送、接收相关的状态



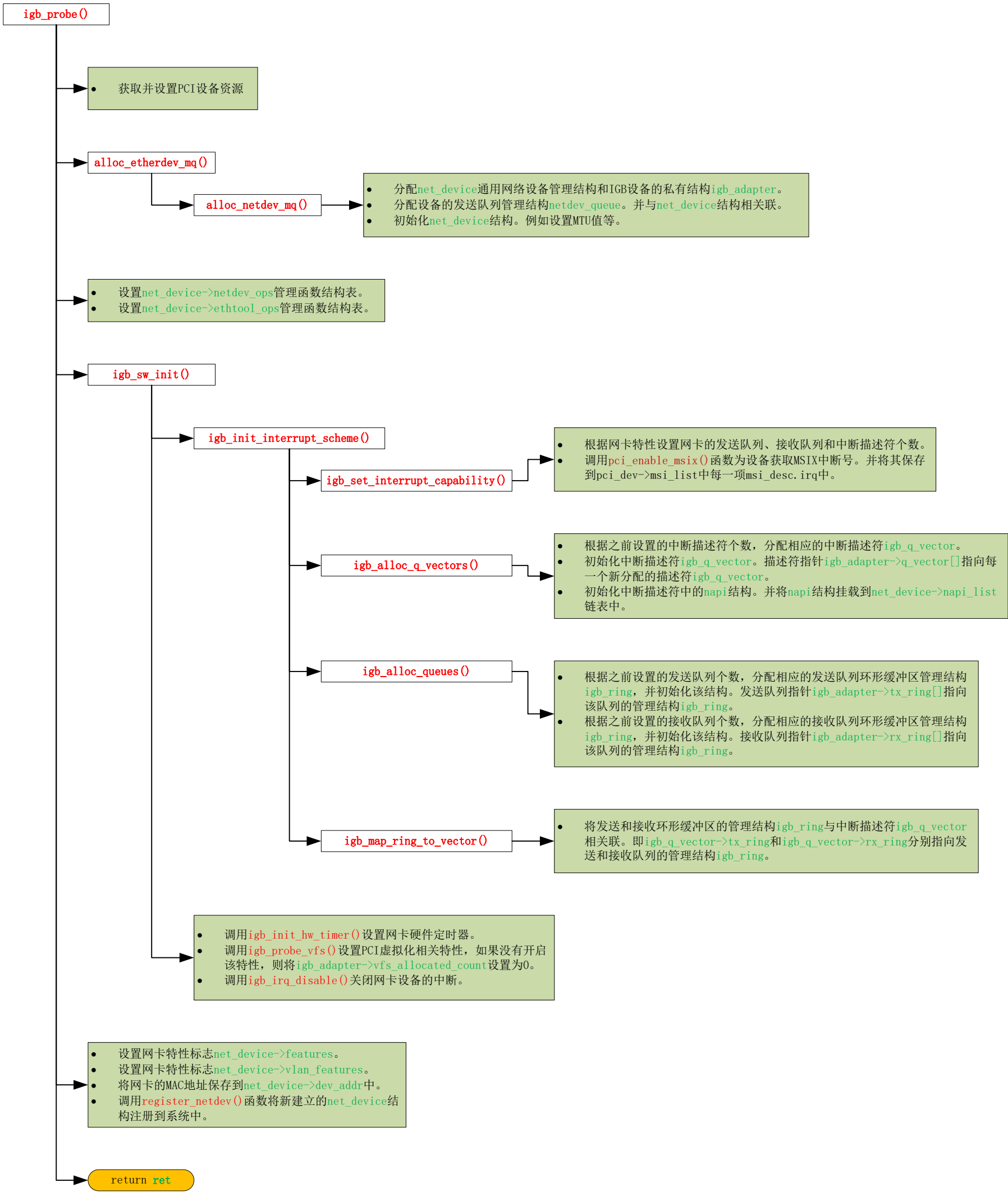
Intel 网卡驱动（igb）举例

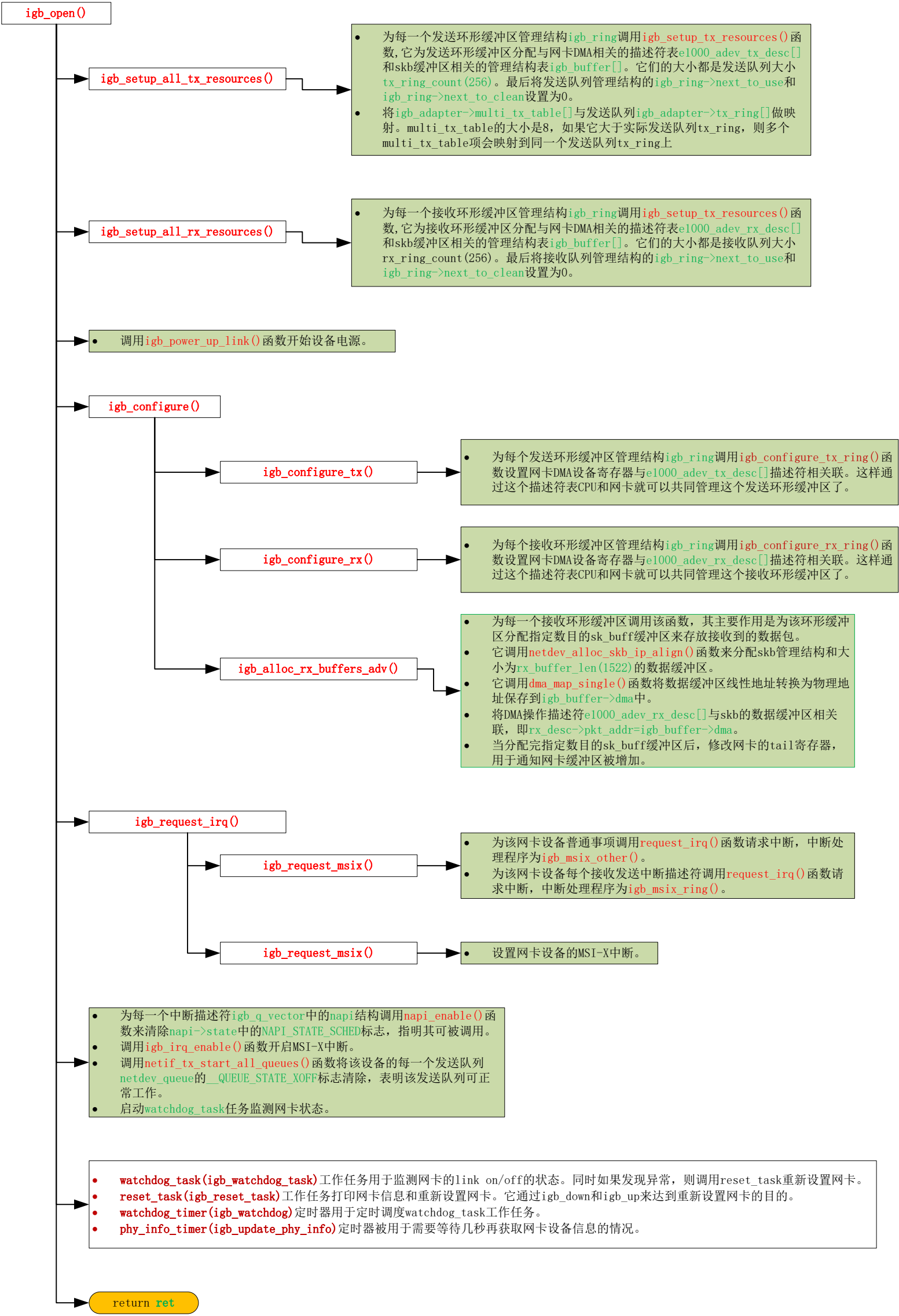
初始化后的全图结构图



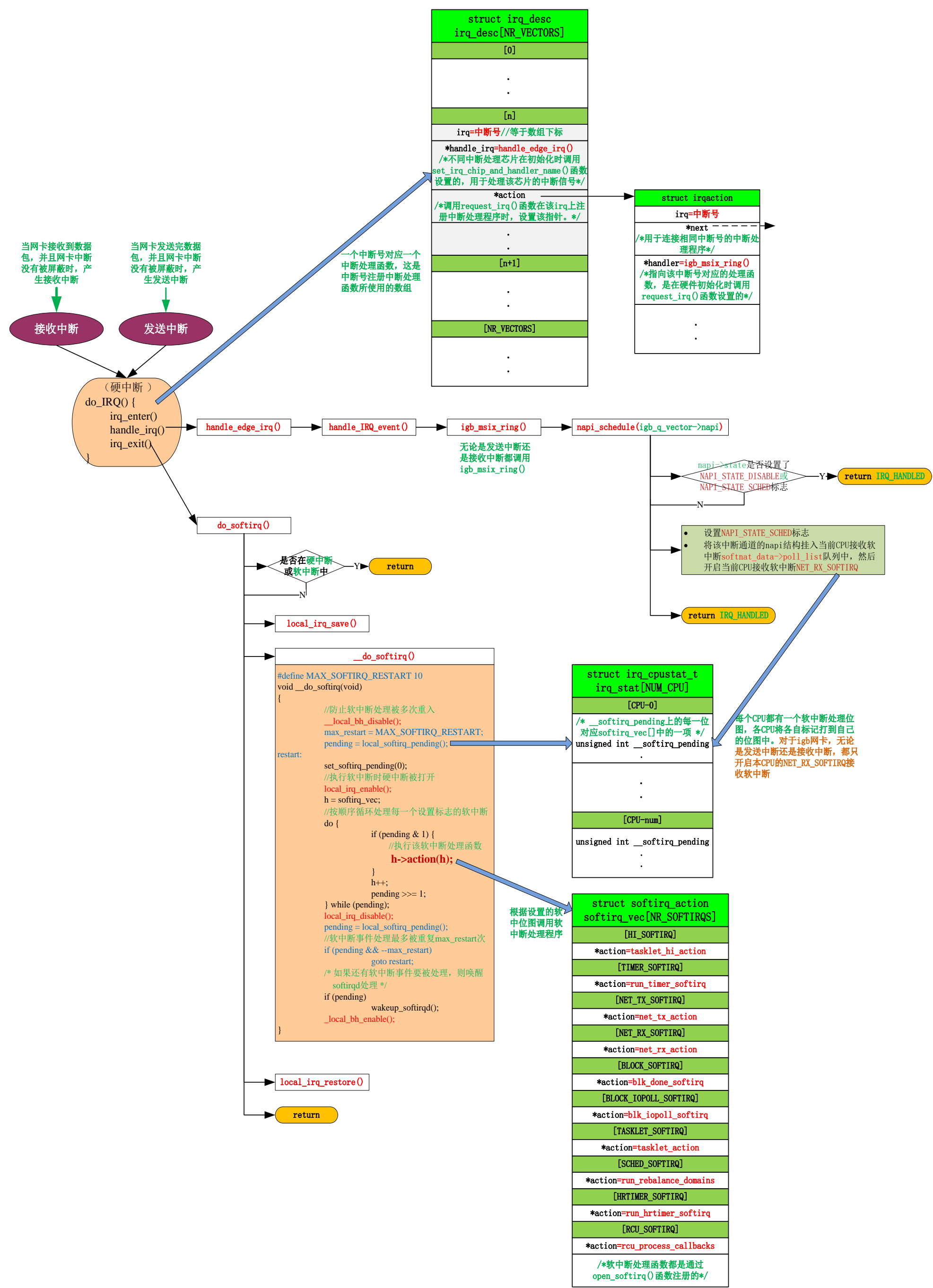
- 1.1 在一个网络设备 net_device 中可以有多中断通道，每个中断通道由 igb_q_vector 结构描述。
- 1.2 每个中断通道对应一个硬件中断号，同时它也包含一个 napi_struct 结构，poll 指针指向 igb_poll() 函数，它即处理接收中断，又处理发送中断。
- 1.3 每个中断通道都有自己的接收与发送队列，都由 igb_poll 函数进行处理。

初始化的处理流程图





当中断产生时的处理流程



由于我们介绍的是 igb 网卡处理流程，所以下面介绍接收 net_rx_action()和发送 net_tx_action()软中断的处理逻辑。

```
static void net_rx_action(struct softirq_action *h)
{
    struct softnet_data *sd = &__get_cpu_var(softnet_data); //只处理本CPU的softnet_data项
    unsigned long time_limit = jiffies + 2;
    //在整个软中断接收函数中，最多处理netdev_budget(300)个数据包
    int budget = netdev_budget; //300

    while (!list_empty(&sd->poll_list)) {
        struct napi_struct *n;
        int work, weight;

        /*如果处理的数据包数超过了budget(300) 或超过了2 jiffies, 则重新开启接收软中断, 并返回。*/
        if (unlikely(budget <= 0 || time_after(jiffies, time_limit))) {
            __raise_softirq_irqoff(NET_RX_SOFTIRQ);
            goto out;
        }

        n = list_first_entry(&sd->poll_list, struct napi_struct, poll_list);
        //每一个poll函数最多处理weight(64)个数据包
        weight = n->weight; //64
        work = n->poll(n, weight);
        budget = budget - work;

        /*如果poll函数处理数据包数等于weight, 我们就认为它还有数据包要被处理, 所以我们将其放入poll_list队列尾部, 它将被继续处理。*/
        if (work == weight)
            list_move_tail(&n->poll_list, &sd->poll_list);
    }
out:
    return;
}

static void net_tx_action(struct softirq_action *h)
{
    struct softnet_data *sd = &__get_cpu_var(softnet_data); //只处理本CPU的softnet_data项
    /*
     * 释放由dev_kfree_skb_irq()函数放入的skb缓冲区,
     * 当代码在软中断中无法释放skb时, 就会将skb放入该队列中。
     */
    if (sd->completion_queue) {
        struct sk_buff *clist;

        clist = sd->completion_queue;
        sd->completion_queue = NULL;

        while (clist) {
            struct sk_buff *skb = clist;
            clist = clist->next;

            __kfree_skb(skb);
        }

        if (sd->output_queue) {
            struct Qdisc *head;

            head = sd->output_queue;
            sd->output_queue = NULL;
            sd->output_queue_tailp = &sd->output_queue;

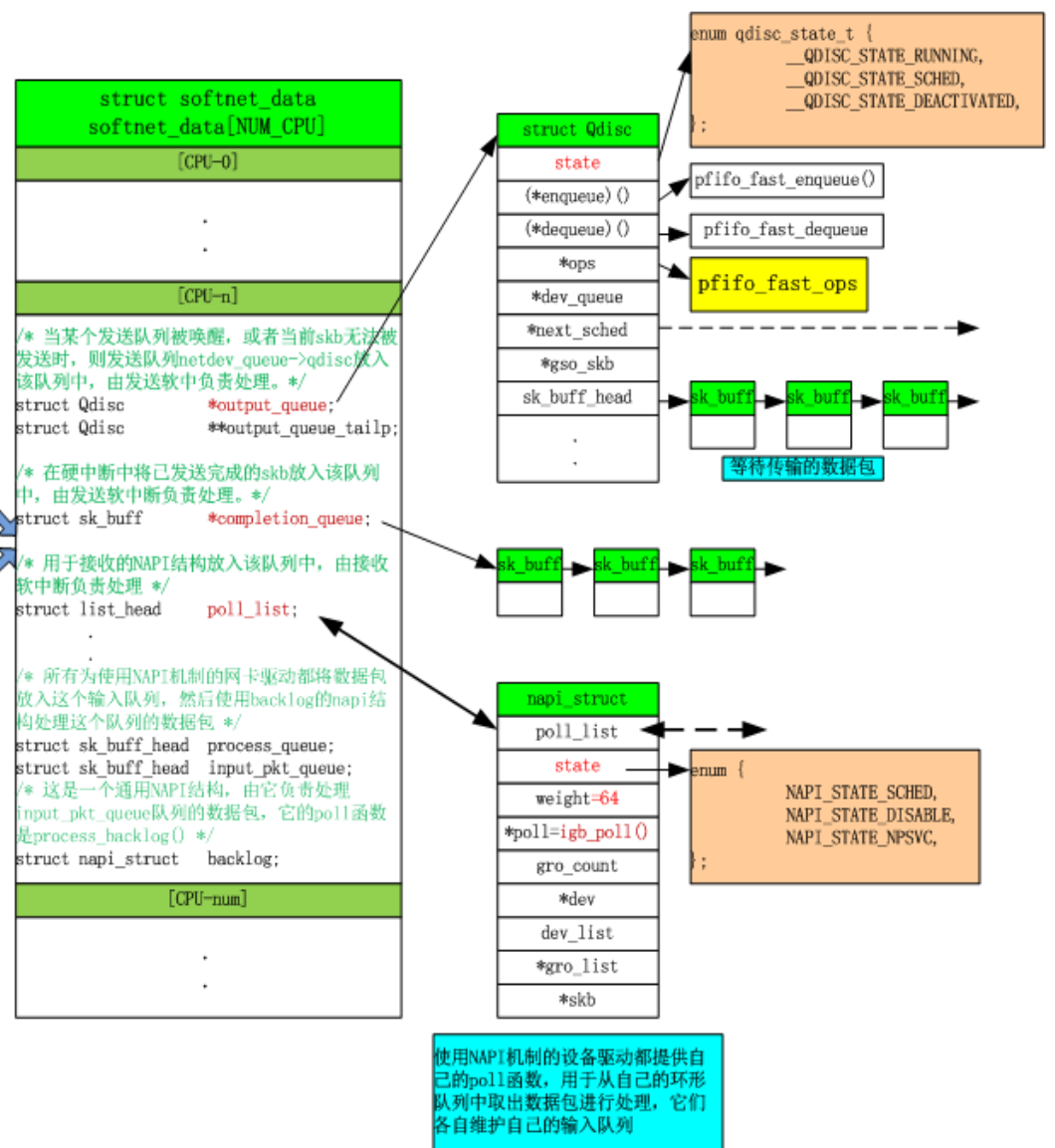
            while (head) {
                struct Qdisc *q = head;
                spinlock_t *root_lock;

                head = head->next_sched;

                root_lock = qdisc_lock(q);
                if (spin_trylock(root_lock)) {
                    smp_mb__before_clear_bit();
                    clear_bit(__QDISC_STATE_SCHED, &q->state);
                    qdisc_run(q);
                    spin_unlock(root_lock);
                } else {
                    if (!test_bit(__QDISC_STATE_DEACTIVATED, &q->state)) {
                        __netif_reschedule(q);
                    } else {
                        smp_mb__before_clear_bit();
                        clear_bit(__QDISC_STATE_SCHED, &q->state);
                    }
                }
            }
        }
    }
}
```

net_rx_action() 函数只处理poll链表中的每一个napi项。

net_tx_action() 函数释放 completion_queue队列中的skb, 并处理output_queue中的qdisc队列。



igb_poll(weight(64))

是否有发送队列?

igb_q_vector->tx_ring

Y

igb_clean_tx_irq()

是否有接收队列?

igb_q_vector->rx_ring

Y

igb_clean_rx_irq_adv()

```
static bool igb_clean_rx_irq_adv(struct igb_q_vector *q_vector, int *work_done, int budget)
{
    struct igb_ring *rx_ring = q_vector->rx_ring;
    struct net_device *netdev = rx_ring->netdev;
    struct device *dev = rx_ring->dev;
    int cleaned_count = 0;

    /* 获取下一个将要被处理的buffer、描述符和描述符状态，
     * 每一个buffer和描述符是一一对应的。*/
    i = rx_ring->next_to_clean;
    buffer_info = &rx_ring->buffer_info[i];
    rx_desc = E1000_RX_DESC_ADV(*rx_ring, i);
    staterr = le32_to_cpu(rx_desc->wb.upper.status_error);

    while (staterr & E1000_RXD_STAT_DD) { //当前描述符已接收了数据
        //如果处理了指定数目(budget)的数据包，则退出
        if (*work_done >= budget)
            break;
        (*work_done)++;

        //获取接收到的skb
        skb = buffer_info->skb;
        buffer_info->skb = NULL;

        i++;
        if (i == rx_ring->count)
            i = 0;
        cleaned_count++;

        //做校验和处理
        igb_rx_checksum_adv(rx_ring, staterr, skb);
        //获取skb->protocol和skb->pkt_type值
        skb->protocol = eth_type_trans(skb, netdev);
        skb_record_rx_queue(skb, rx_ring->queue_index);
        //获取vlan标识符
        vlan_tag = ((staterr & E1000_RXD_STAT_VP) ?
            le16_to_cpu(rx_desc->wb.upper.vlan) : 0);

        //调用上层函数处理该数据包
        igb_receive_skb(q_vector, skb, vlan_tag);

        /* 如果处理的数据包数达到了IGB_RX_BUFFER_WRITE(16),
         * 则为接收缓冲区分配空间，用于接收数据包。*/
        if (cleaned_count >= IGB_RX_BUFFER_WRITE) {
            igb_alloc_rx_buffers_adv(rx_ring, cleaned_count);
            cleaned_count = 0;
        }

        rx_desc = E1000_RX_DESC_ADV(*rx_ring, i);
        buffer_info = &rx_ring->buffer_info[i];
        staterr = le32_to_cpu(rx_desc->wb.upper.status_error);
    }

    rx_ring->next_to_clean = i;
    //为已处理的数据包再次分配缓冲区，用于接收数据包。
    cleaned_count = igb_desc_unused(rx_ring);
    if (cleaned_count)
        igb_alloc_rx_buffers_adv(rx_ring, cleaned_count);

    return cleaned_count;
}
```

发送与接收都已完成?

return work_done

- 调用napi_complete()函数将它的napi结构从softnet_data->poll_list链表中取下
- 调用igb_ring_irq_enable()函数开启网卡中断。

- 对当前中断所绑定发送队列igb_q_vector->tx_ring中已发送完成的skb调用dev_kfree_skb_any(skb)函数进行释放。但最多释放tx_ring->count(256)是个skb，为防止饿死其它软中断处理程序。
- dev_kfree_skb_any(skb)函数发现当前CPU如果是在硬中断中，则将skb放入当前CPU软中断softnet_data->completion_queue中，并开启当前CPU的发送软中断NET_TX_SOFTIRQ
- 如果发送队列已被停止(__QUEUE_STATE_XOFF)，则调用netif_wake_subqueue()函数将发送队列netdev_queue->qdisc放入当前CPU软中断softnet_data->output_queue中，并开启当前CPU的发送软中断NET_TX_SOFTIRQ

netif_receive_skb()

dev_queue_xmit()

dev->netdev_ops->ndo_start_xmit()=igb_xmit_frame_adv()

igb_xmit_frame_adv()

设备是否被关闭?

Y

dev_kfree_skb_any()

return NETDEV_TX_OK

N

skb->len <= 0

Y

dev_kfree_skb_any()

return NETDEV_TX_OK

N

获取发送队列。

r_idx = skb->queue_mapping & (IGB_ABS_MAX_TX_QUEUES - 1);
tx_ring = adapter->multi_tx_table[r_idx];

igb_xmit_frame_ring_adv()

igb_maybe_stop_tx()

剩余发送空间小于size(4)

Y

return -EBUSY

N

return 0

igb_tx_map_adv()

igb_tx_queue_advr()

igb_maybe_stop_tx()

剩余发送空间小于size(4)

Y

return -EBUSY

N

return 0

return NETDEV_TX_OK

对数据包进行协议栈的处理，例如：
ip_rcv()->ip_forward()
->ip_output()
最后通过指定设备发送数据包，函数dev_queue_xmit()

用数据包指定设备
skb->dev的队列
qdisc发送该数据包

dev->netdev_ops->ndo_start_xmit()=igb_xmit_frame_adv()

igb_xmit_frame_adv()

设备是否被关闭?

Y

dev_kfree_skb_any()

return NETDEV_TX_OK

N

skb->len <= 0

Y

dev_kfree_skb_any()

return NETDEV_TX_OK

N

获取发送队列。

r_idx = skb->queue_mapping & (IGB_ABS_MAX_TX_QUEUES - 1);
tx_ring = adapter->multi_tx_table[r_idx];

igb_xmit_frame_ring_adv()

igb_maybe_stop_tx()

剩余发送空间小于size(4)

Y

return -EBUSY

N

return 0

igb_tx_map_adv()

igb_tx_queue_advr()

igb_maybe_stop_tx()

剩余发送空间小于size(4)

Y

return -EBUSY

N

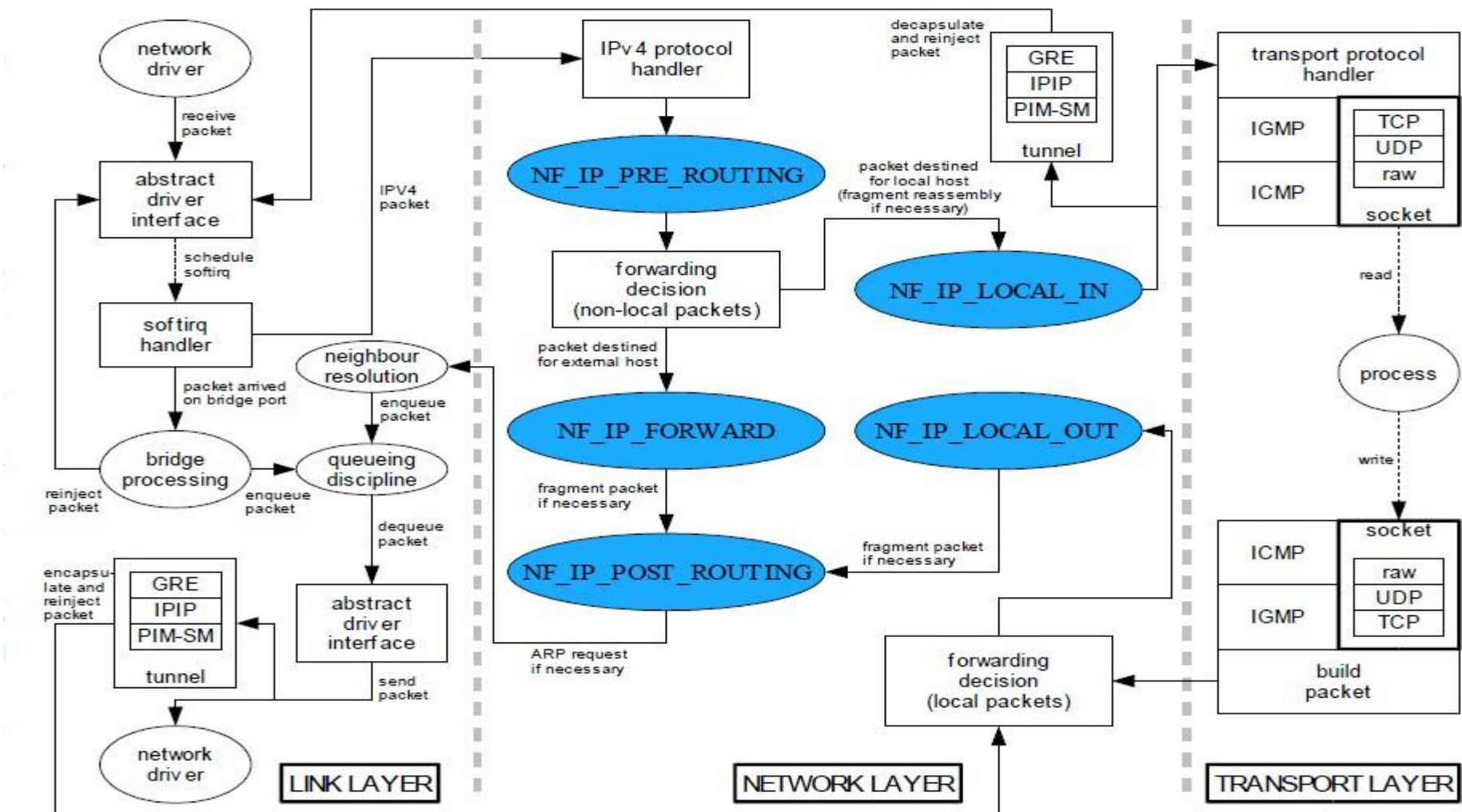
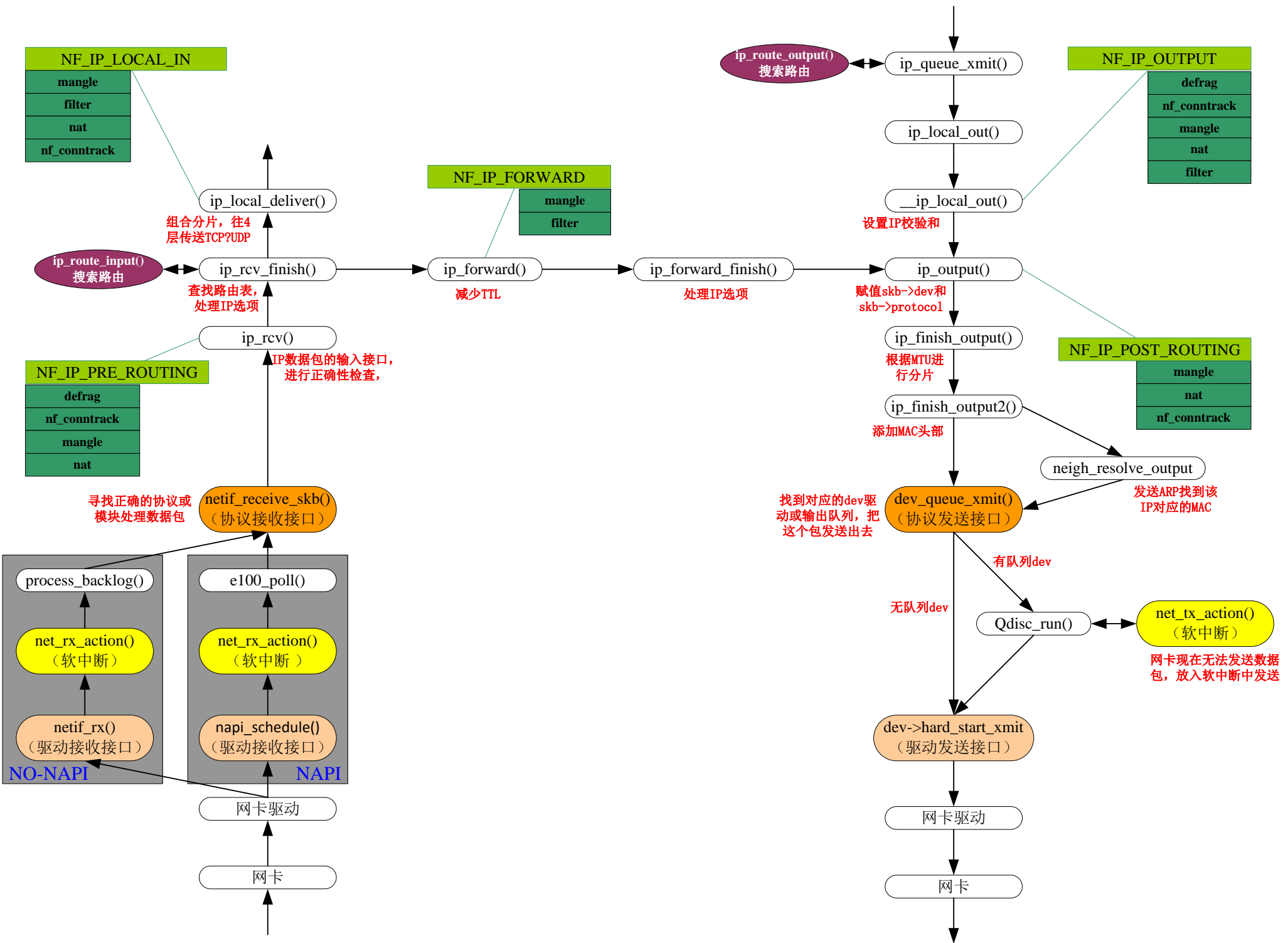
return 0

return NETDEV_TX_OK

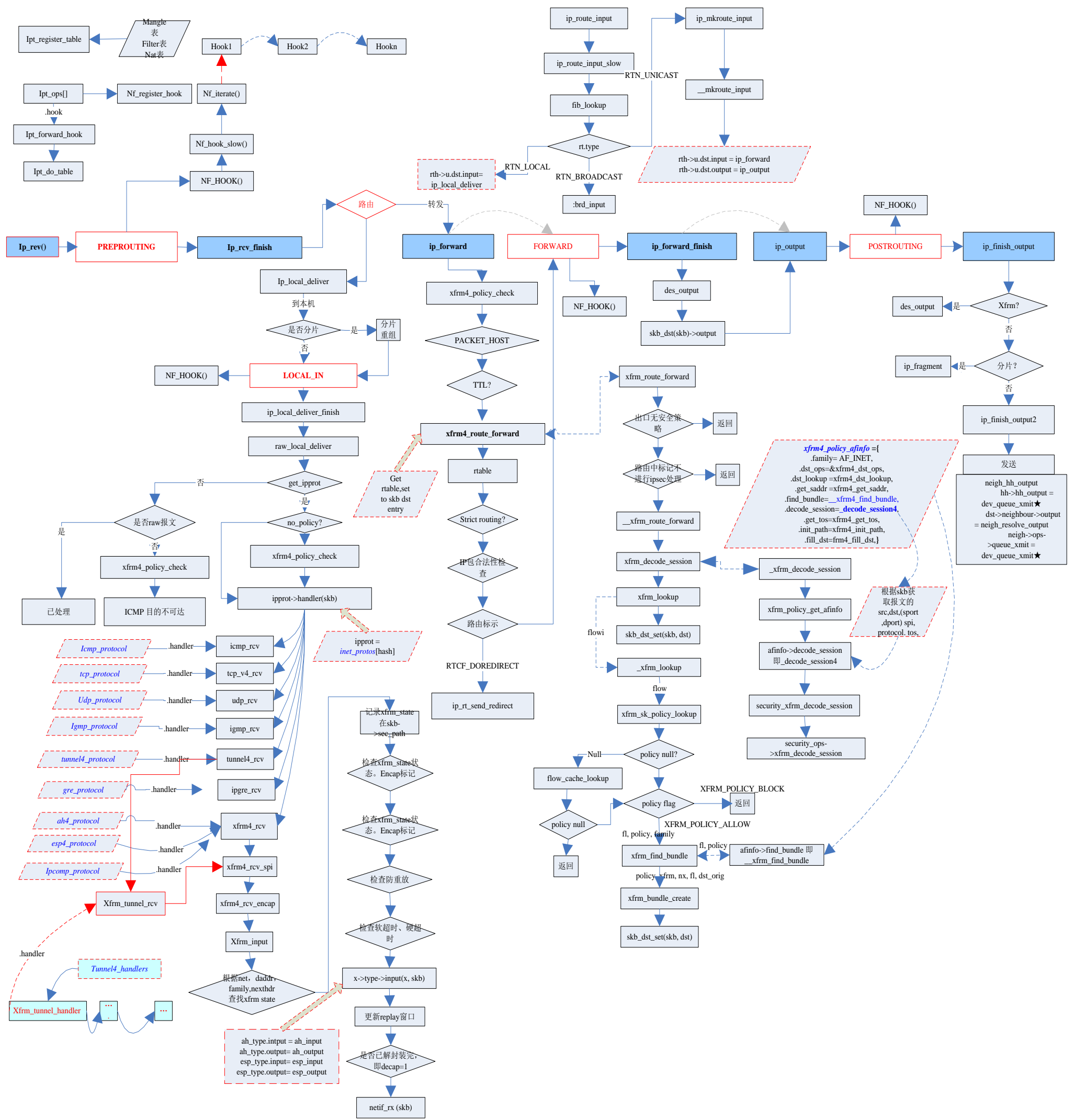
- 调用netif_stop_subqueue()函数设置__QUEUE_STATE_XOFF标志。
- 如果发送空间又被其它CPU释放，则调用netif_wake_subqueue()函数清除__QUEUE_STATE_XOFF标志，并设置_QDISC_STATE_SCHED标志，然后将该发送队列qdisc放入softnet_data->output_queue队列中。
- return -EBUSY

- 调用netif_stop_subqueue()函数设置__QUEUE_STATE_XOFF标志。
- 如果发送空间又被其它CPU释放，则调用netif_wake_subqueue()函数清除__QUEUE_STATE_XOFF标志，并设置_QDISC_STATE_SCHED标志，然后将该发送队列qdisc放入softnet_data->output_queue队列中。
- return -EBUSY

IPv4 接收与转发协议栈流程图

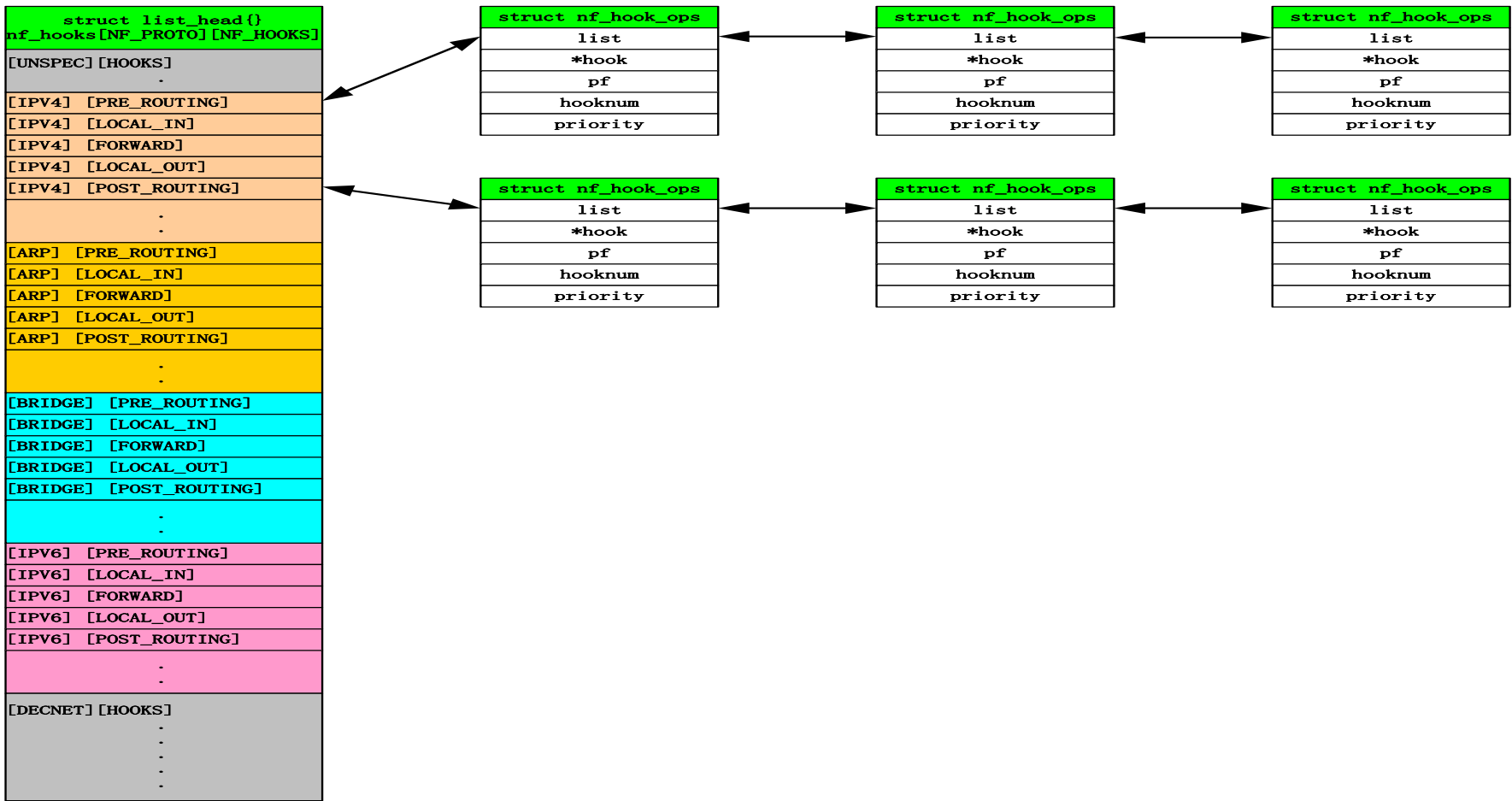


Linux2.6 网络协议栈处理流程图



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。

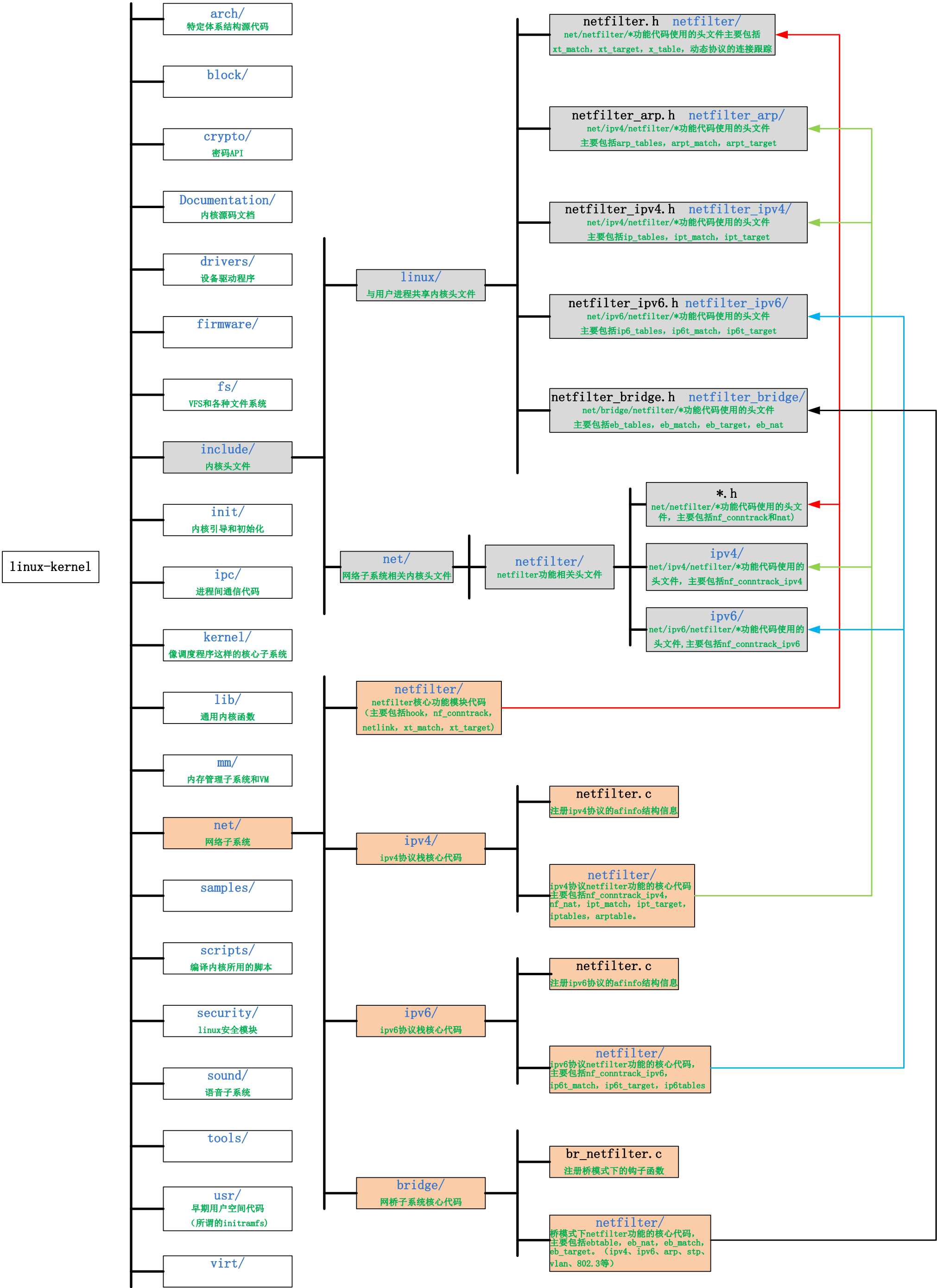
nf_log 子功能

- 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

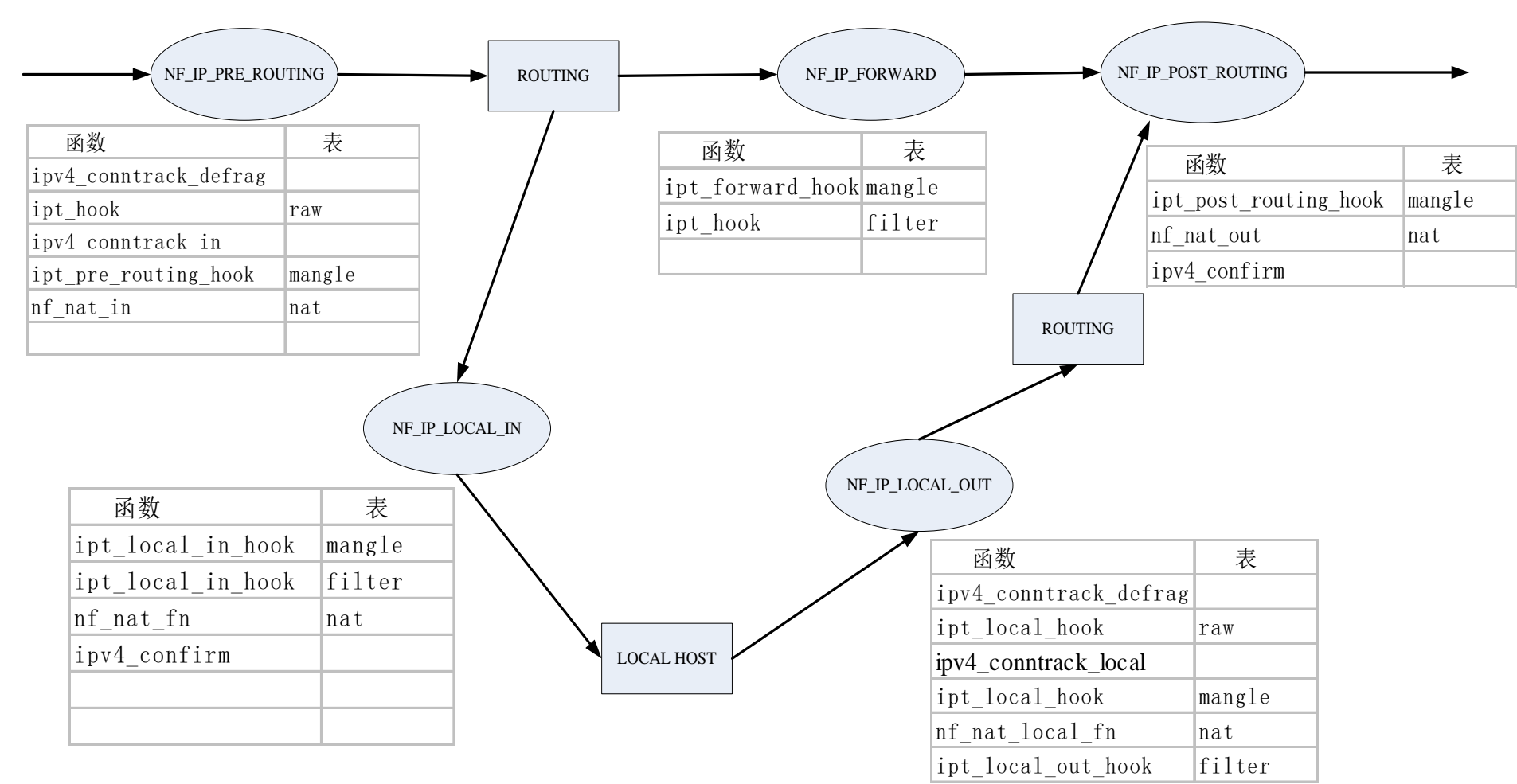
- 1
- nfnetlink 是建立在 netlink 基础上的一个与应用层进行通信的接口。它采用了 netlink attributes 接口，该接口使用 TLV<Type, Length, Value>（类型，长度，值）三元组来描述在传输中每一个数据单元，保证了发送方和接收方都以同样的方式来解释传输的数据。这样该接口就可以传输任何数据了。
- 2
- nfnetlink 利用 netlink 接口创建一个类型为 NETLINK_NETFILTER 的内核 socket 接口。它接收应用层传输数据的接口是 nfnetlink_rcv()，nfnetlink_rcv()根据 nlmsg_hdr->nlmsg_type 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_contrack

nf_contrack 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_contrack_helper 创建的期待子连接 nf_contrack_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_contrack_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_contrack_expect 项的 HASH 桶 */
    struct hlist_nulls_head unconfirmed; /* 对于一个链接的第一个包，在 init_contrack()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_contrack 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（contrack -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_contrack 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_contrack_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 桶是否使用 vmalloc()进行分配的 */
    int expect_vmalloc;                  /* 存储期待子连接 nf_contrack_expect 项的 HASH 桶是否使用 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_contrack ct_general;      /* 连接跟踪的引用计数 */
    spinlock_t lock;
    /* These are my tuples; original and reply */
    struct nf_contrack_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_contrack_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_CONTRACK_MARK
    u_int32_t mark;
#endif

#ifdef CONFIG_NF_CONTRACK_SECMARK
    u_int32_t secmark;
#endif

    /* Storage reserved for other modules: */
    union nf_contrack_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。

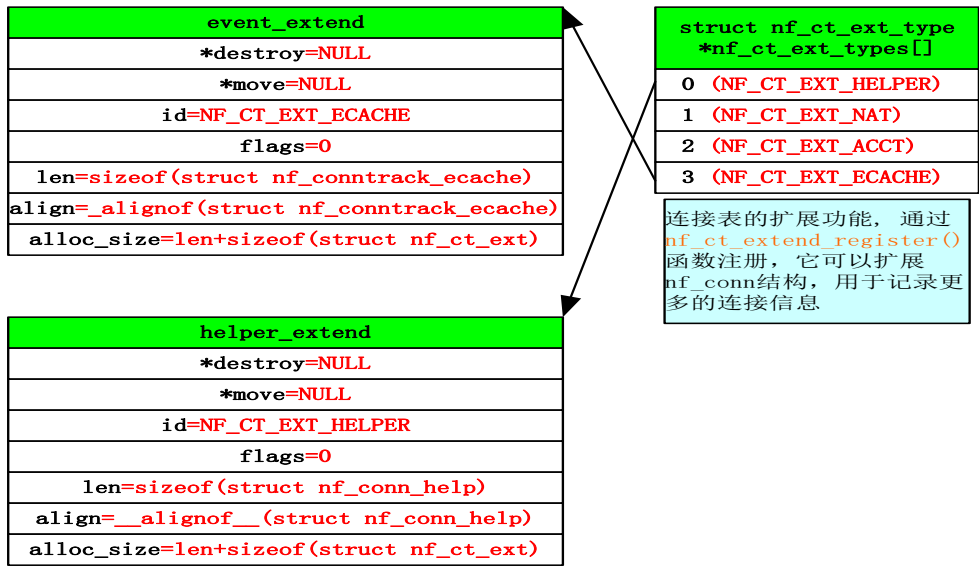
5.2 nf_nat_lock

- 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。

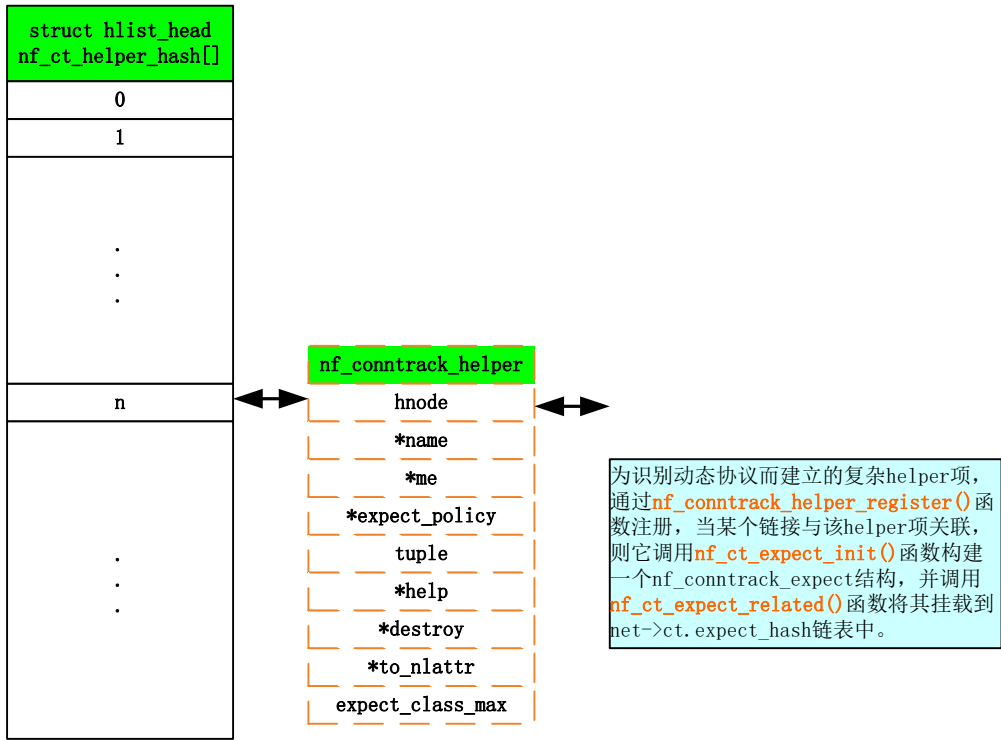
5.3 ct->lock

- 5.3.1 修改某个 ct 的数据时使用该 ct 自己的锁。

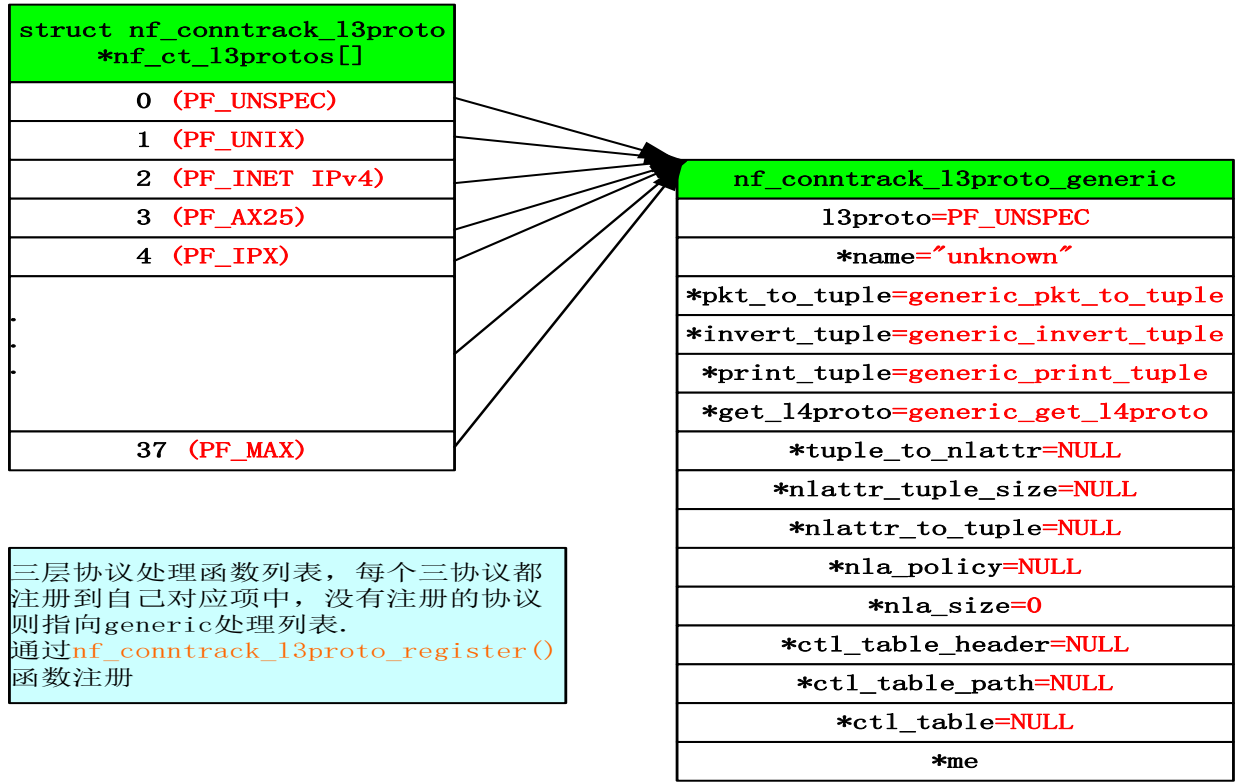
6 扩展连接跟踪结构（nf_conn）利用 nf_conntrack_extend.c 文件中的 nf_ct_extend_register(struct nf_ct_ext_type *type)和 nf_ct_extend_unregister(struct nf_ct_ext_type *type)进行扩展，并修改连接跟踪相应代码来利用这部分扩展功能。



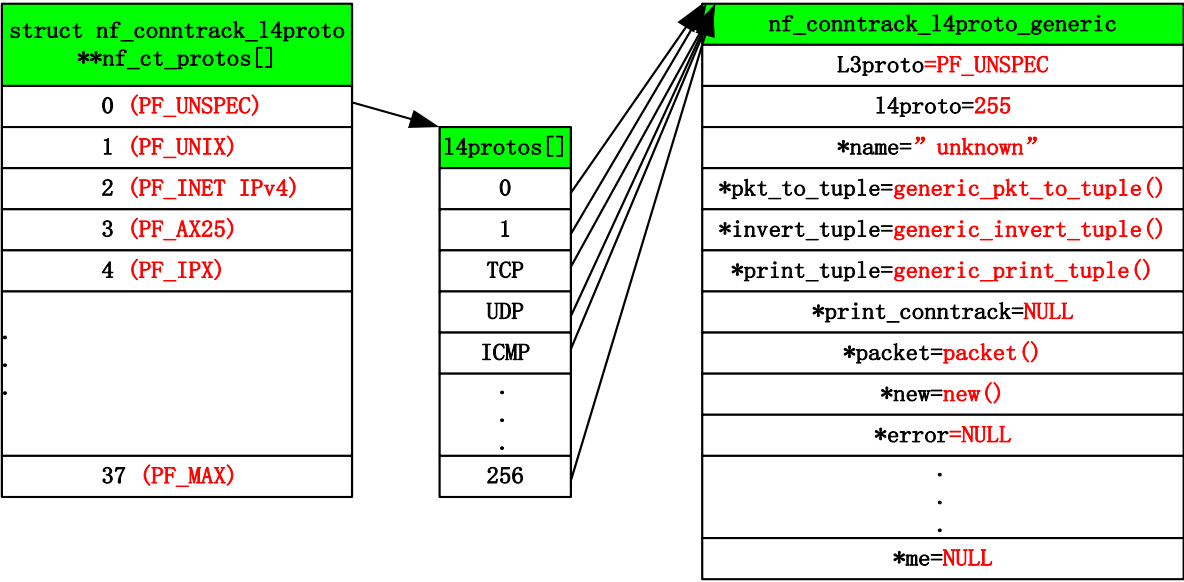
7 处理一个连接的子连接协议，利用 nf_conntrack_helper.c 文件中的 nf_conntrack_helper_register(struct nf_conntrack_helper *me)来注册 nf_conntrack_helper 结构，和 nf_conntrack_expect.c 文件中的 nf_ct_expect_related_report(struct nf_conntrack_expect *expect, u32 pid, int report)来注册 nf_conntrack_expect 结构。



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。



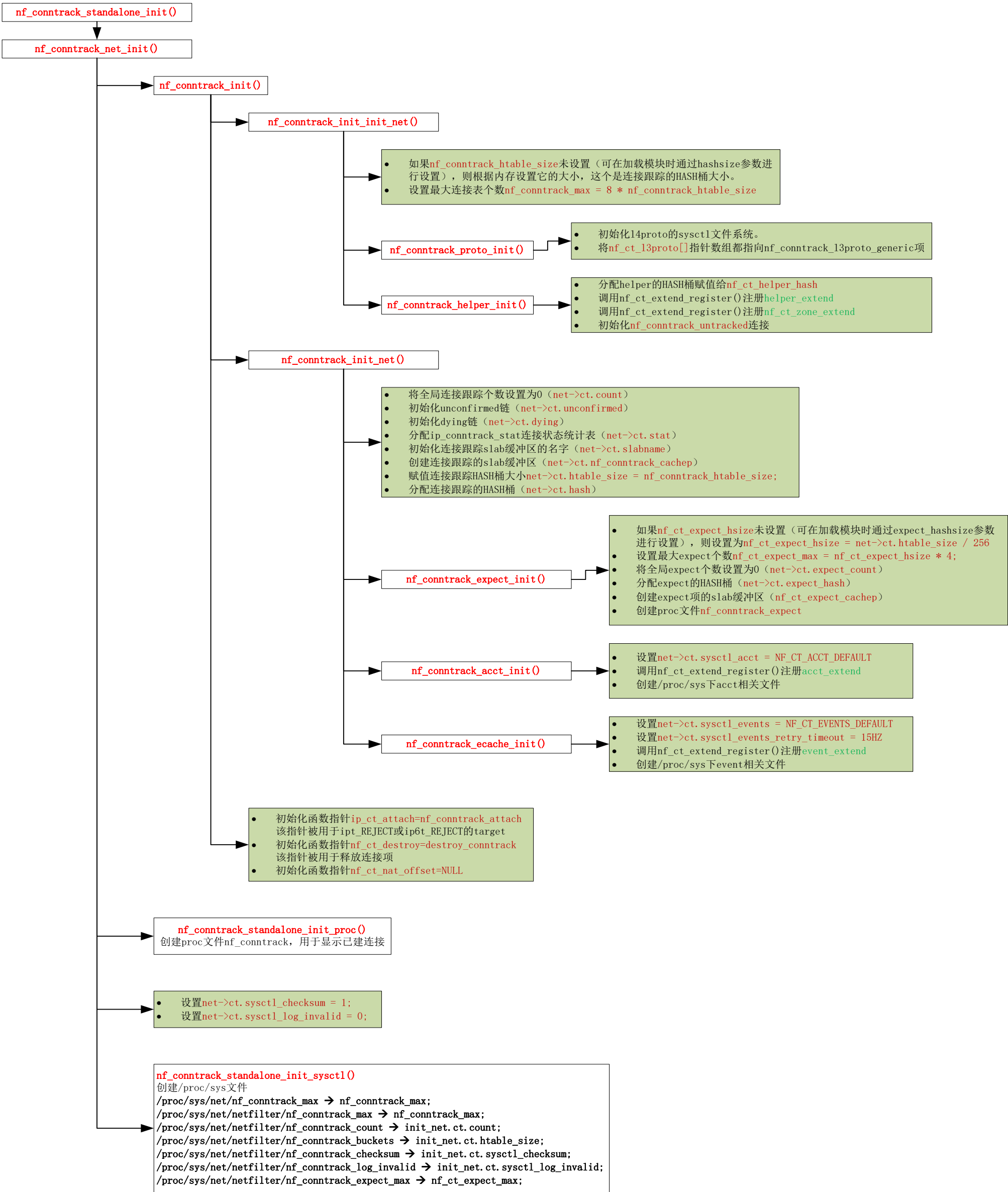
9 四层协议（TCP/UDP）利用 nf_conntrack_proto.c 文件中的 nf_conntrack_l4proto_register(struct nf_conntrack_l4proto *l4proto)和 nf_conntrack_l4proto_unregister(struct nf_conntrack_l4proto *l4proto)在 nf_ct_protos[]数组中注册自己的四层协议处理函数。



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

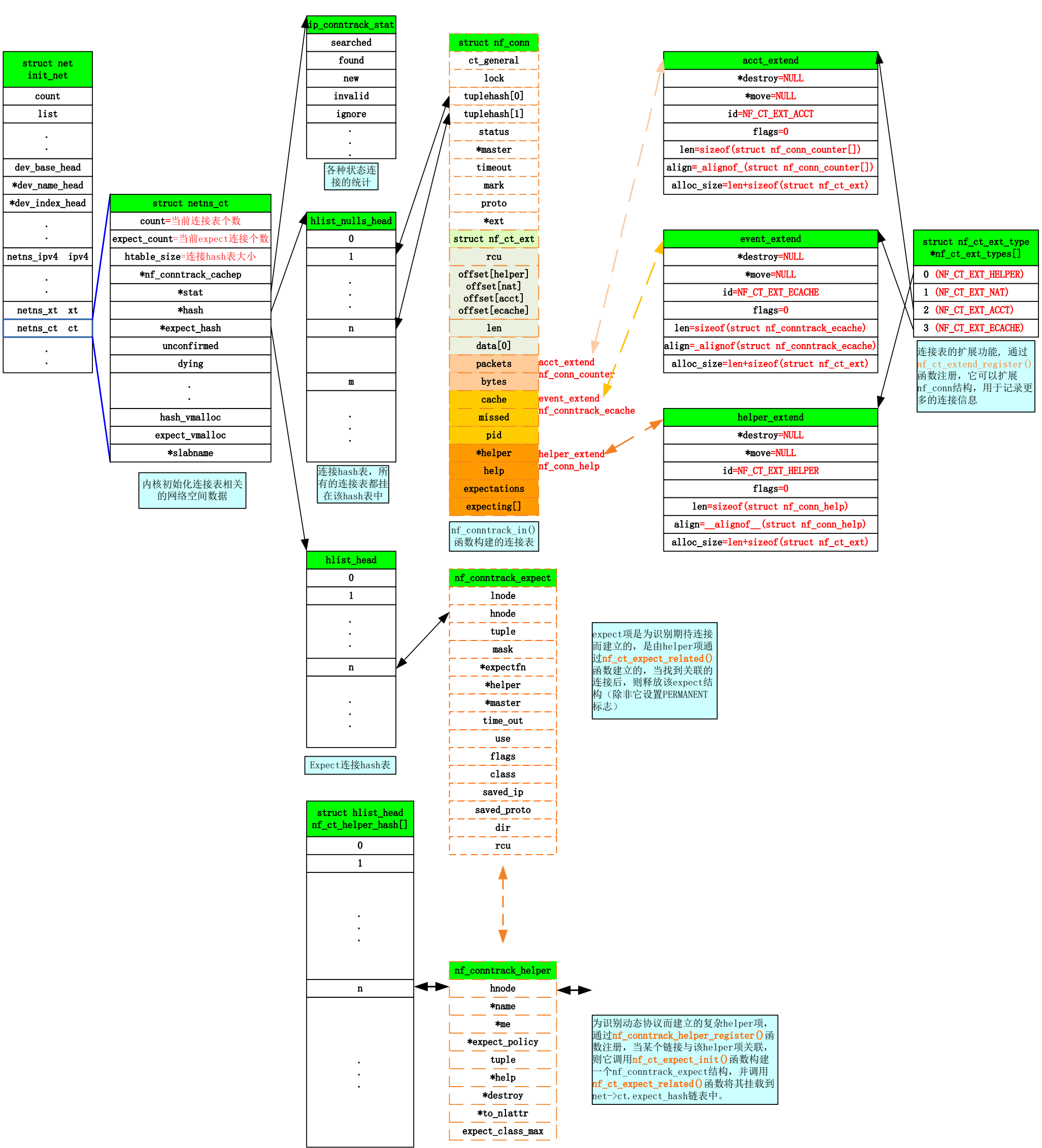
10 建立连接跟踪结构（nf_conn）利用 nf_conntrack_core.c 文件中的 nf_conntrack_in()函数进行构建的。nf_conntrack_core.c 文件中还包括其它相应的处理函数。

nf_contrack 的初始化

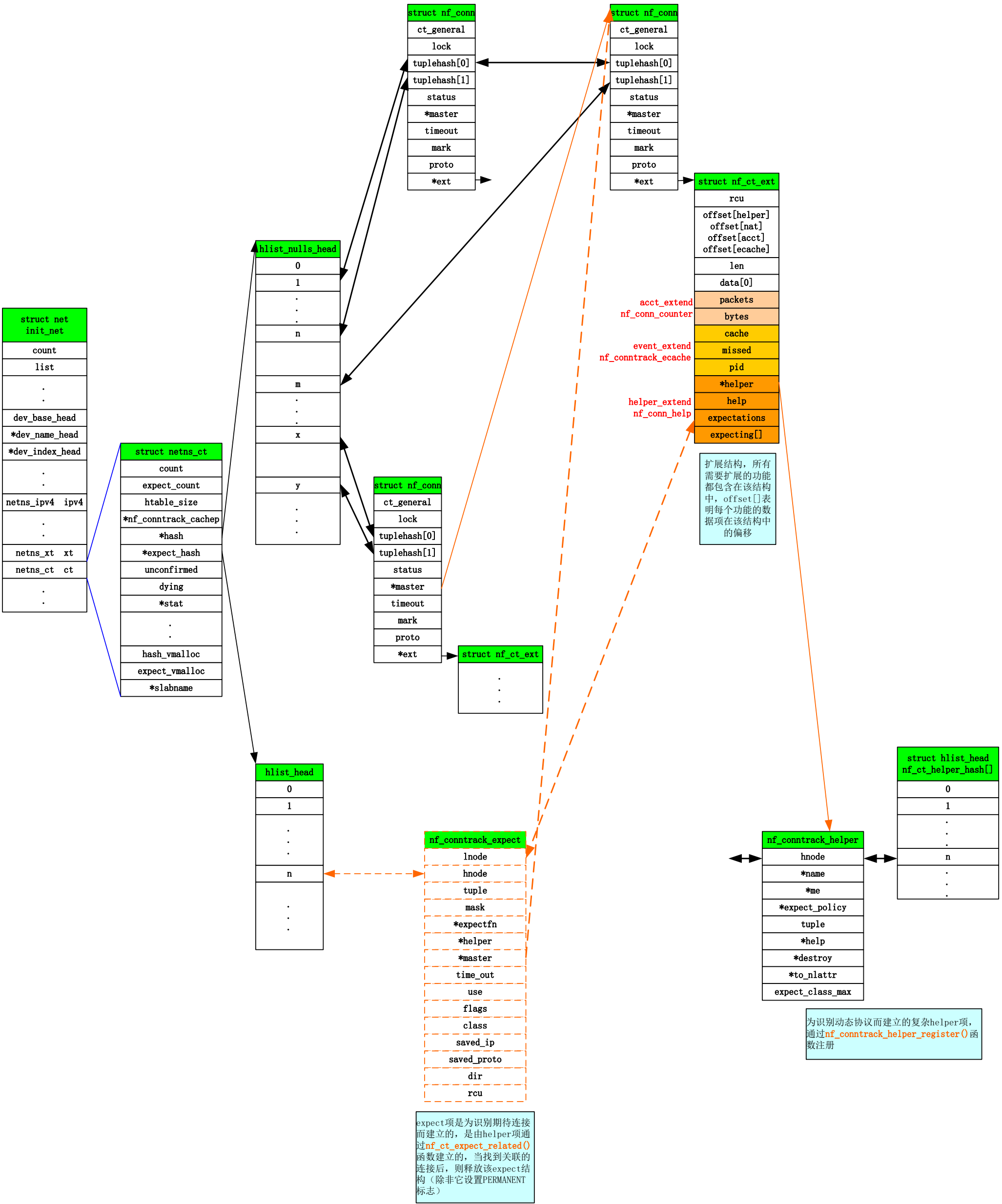


- nf_contrack 的初始化，就是初始化上面提到的这些资源，它在内核启动时调用 nf_contrack_standalone_init()函数进行初始化的。
- 初始化完成后，构建出如下面连接表表现形式中的图所示，只是还没有创建 nf_conn、nf_contrack_expect、nf_contrack_helper 项。
- ct_hash、expect_hash、helper_hash 这三个 HASH 桶大小在初始化时就已确定，后面不能再更改。其中 ct_hash、expect_hash 可在加载 nf_contrack.ko 模块时通过参数 hashsize 和 expect_hashsize 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)。
- 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



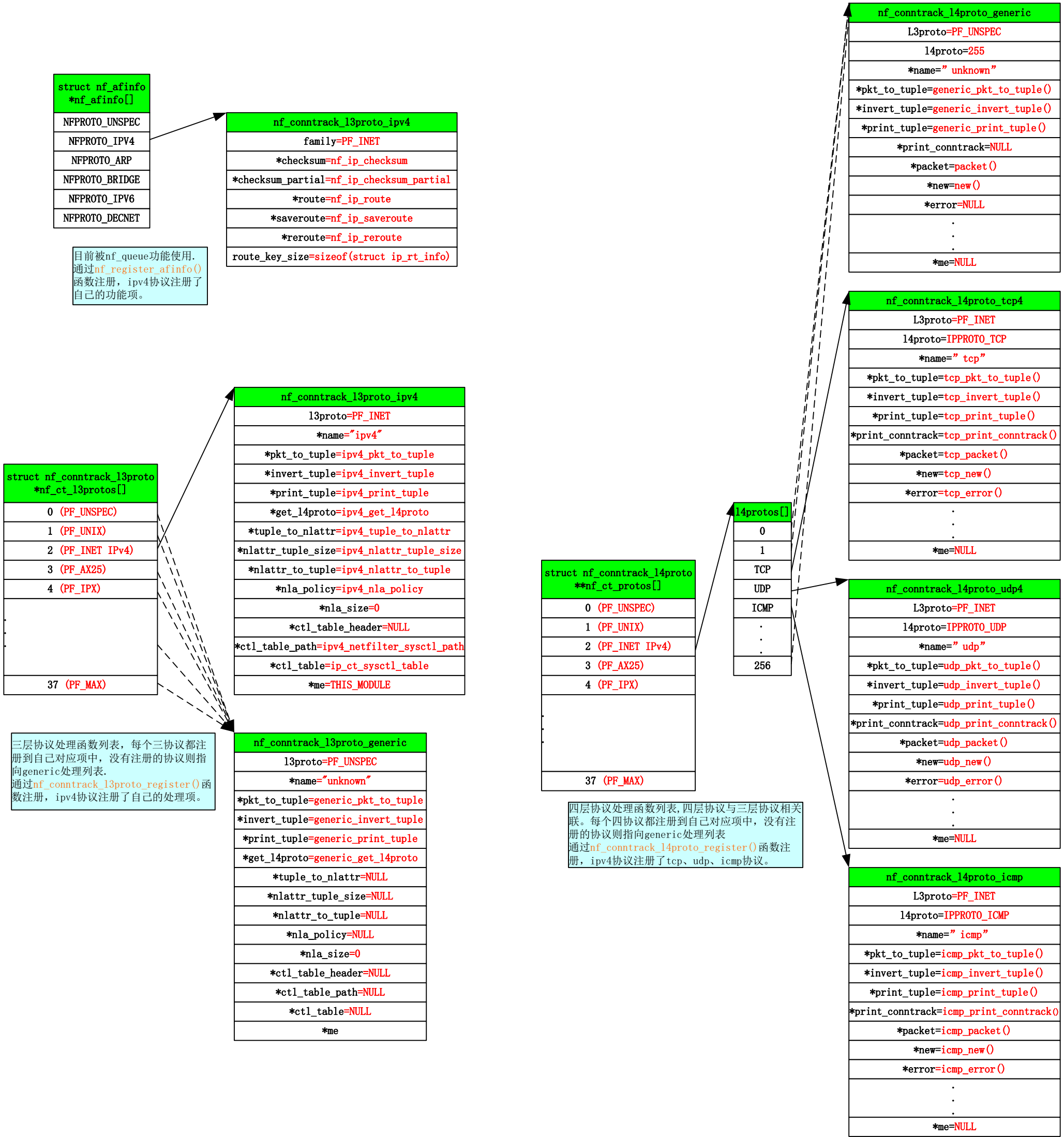
连接表和动态协议的表现形式



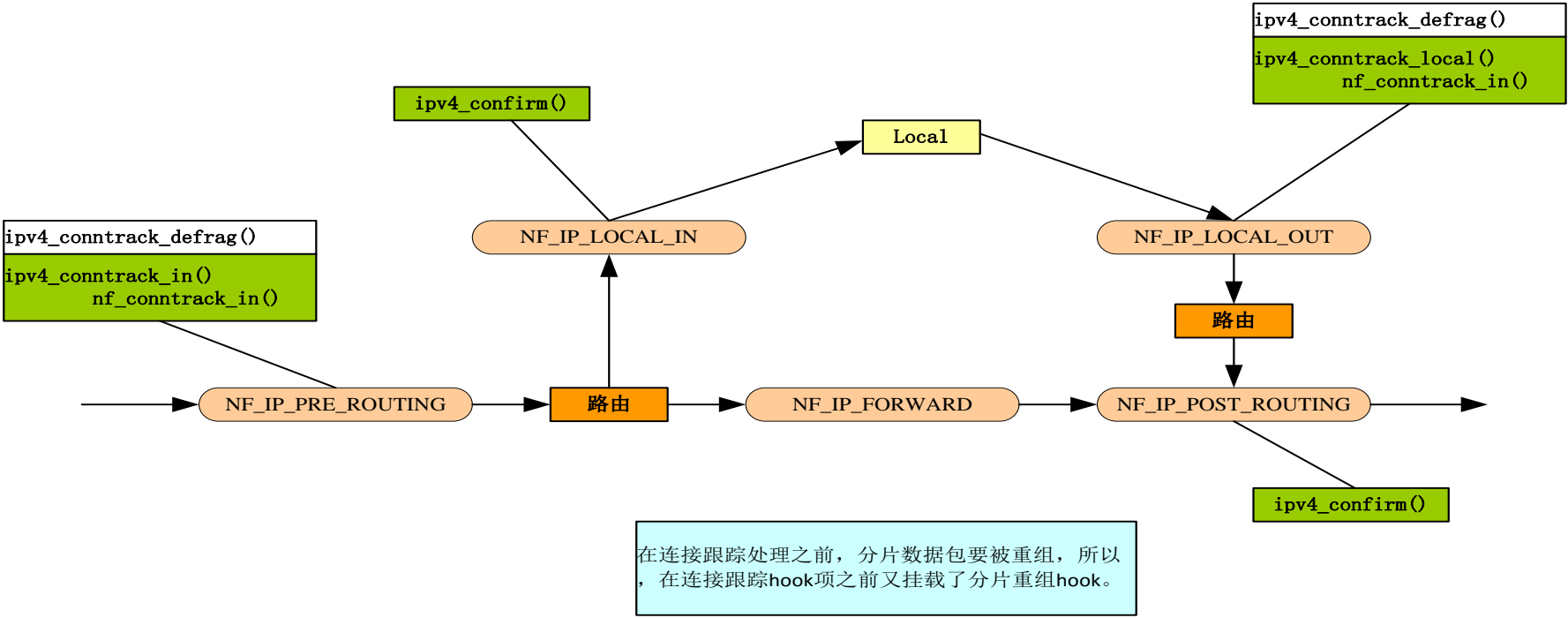
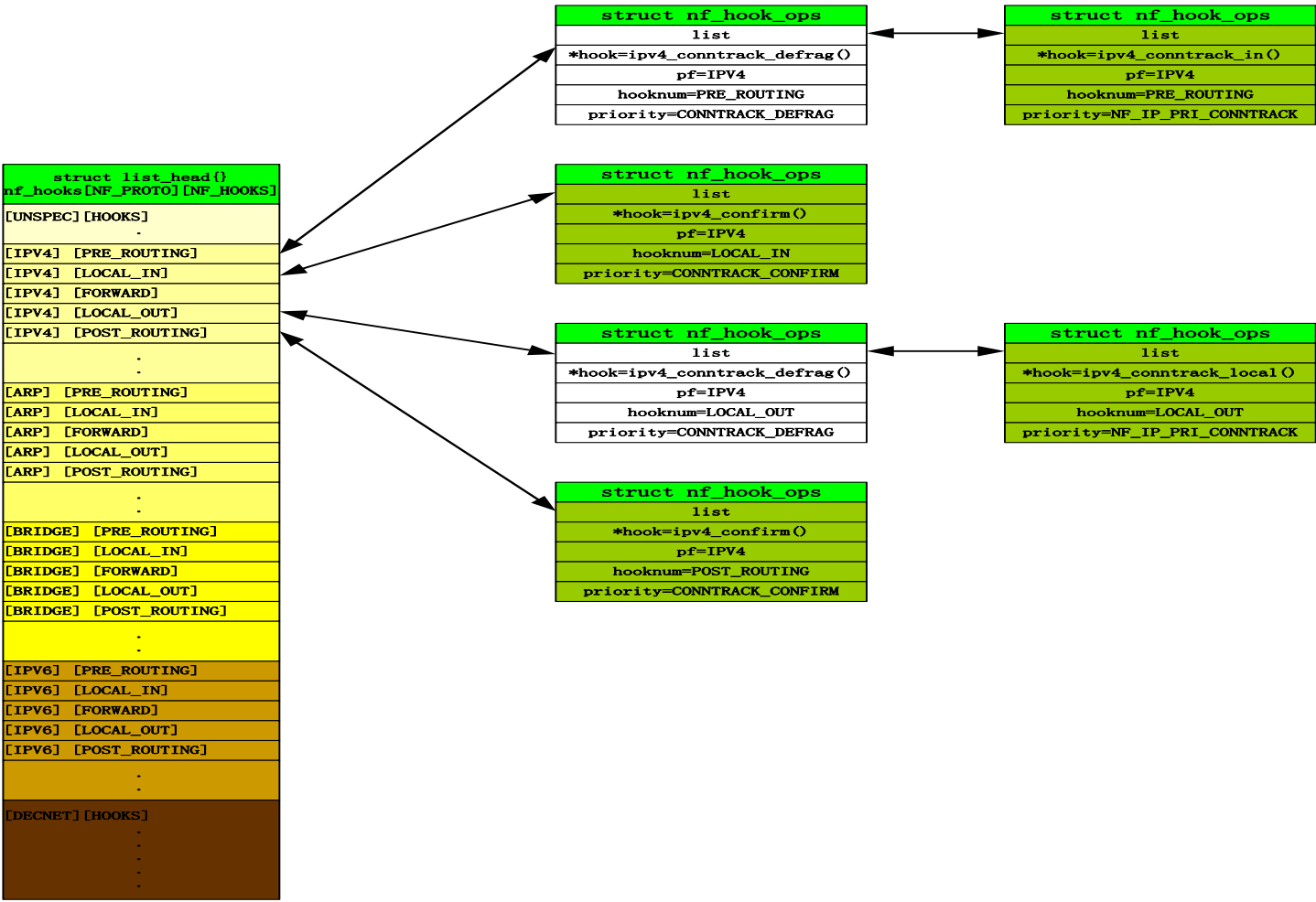
IPv4 利用 nf_contrack 进行链接跟踪

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。



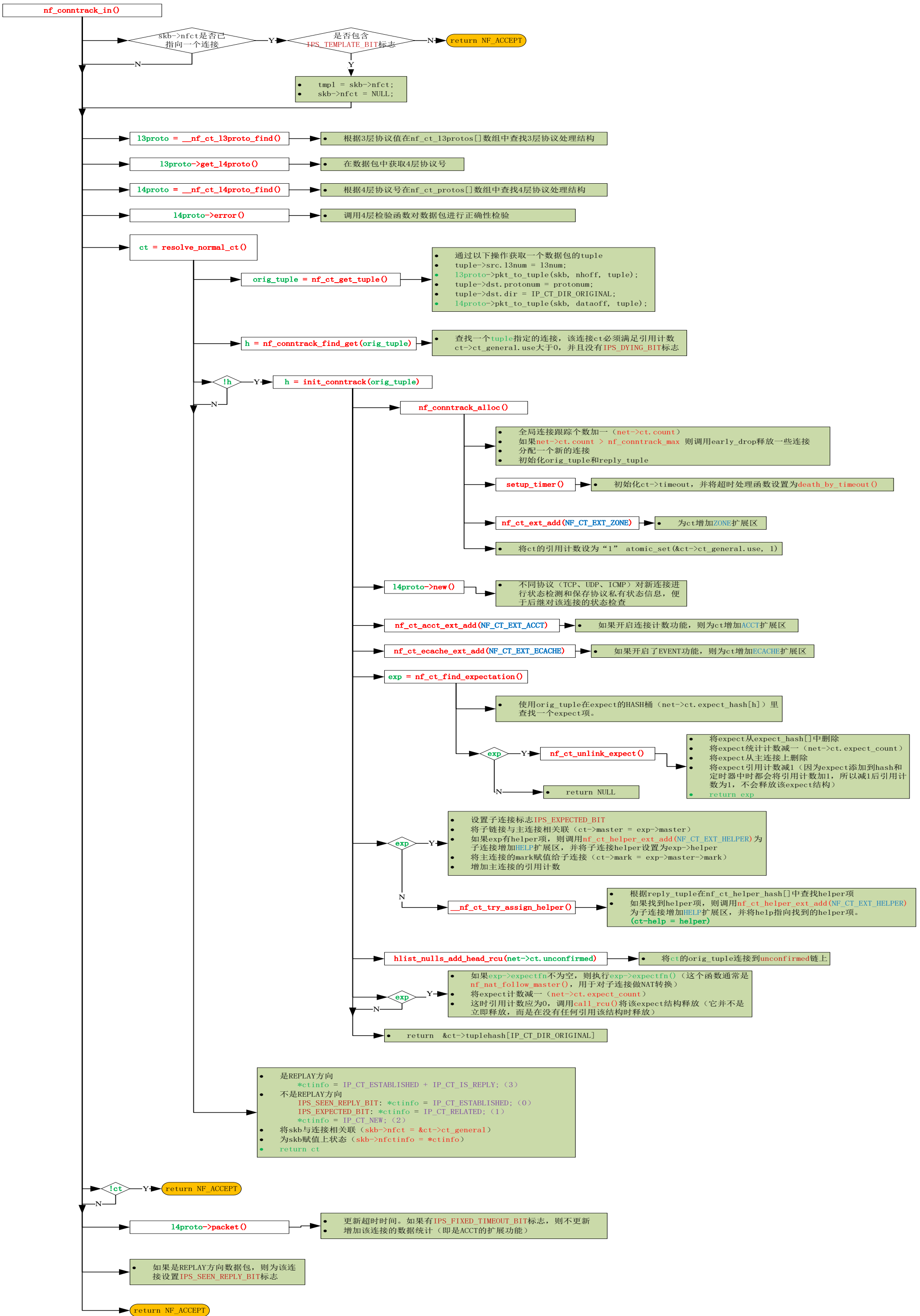
2 在 netfilter 中利用 nf_register_hook(struct nf_hook_ops *reg)、nf_unregister_hook(struct nf_hook_ops *reg)函数注册自己的钩子项，调用 nf_contrack_in()函数来建立相应连接。



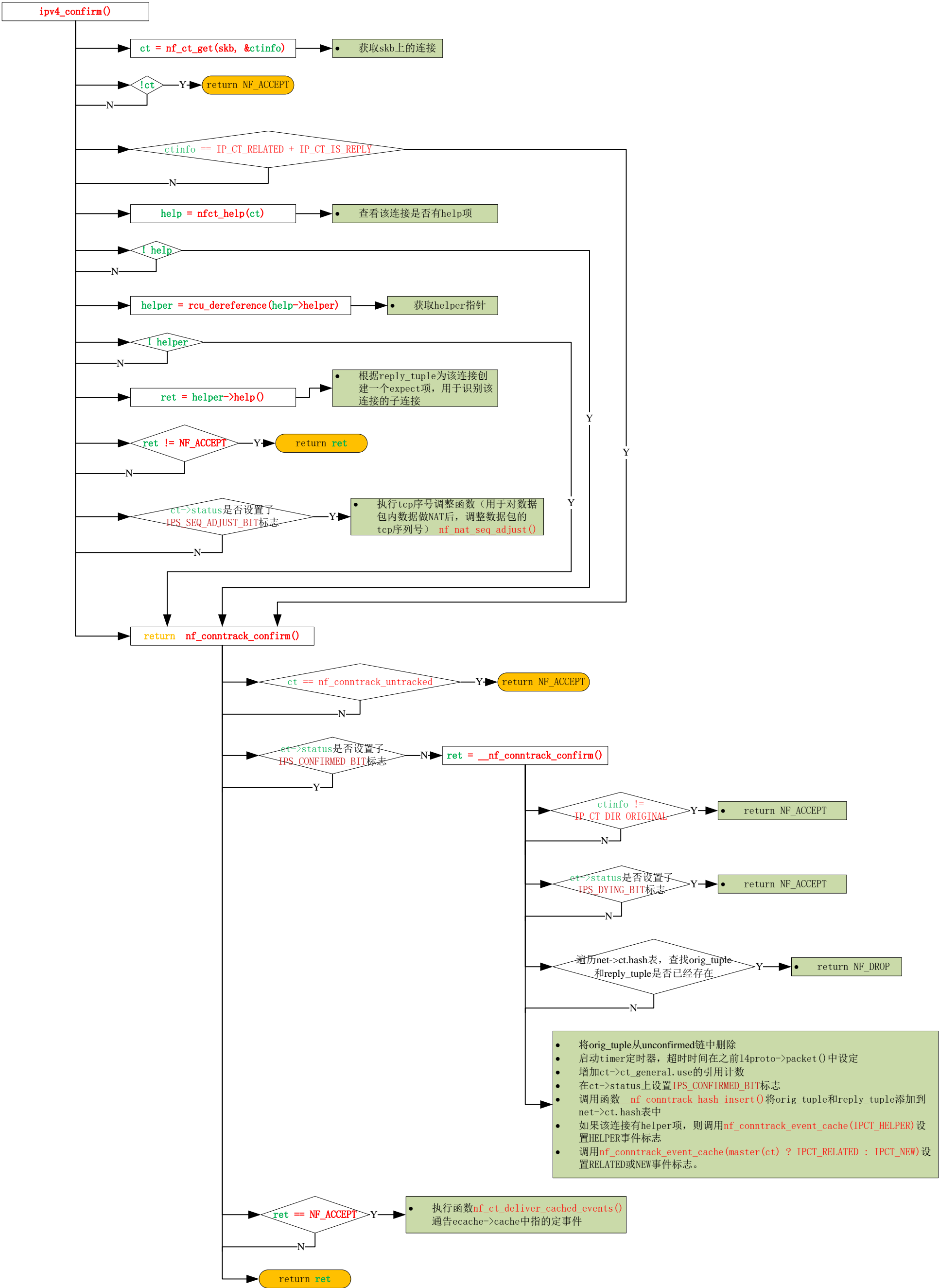
- 3 这样数据包就会经过 `ipv4` 注册的钩子项，并调用 `nf_contrack_in()`函数建立连接表项，连接表项中的 `tuple` 由 `ipv4` 注册的 3/4 层协议处理函数构建。
 - 3.1 `ipv4_contrack_in()` 挂载在 `NF_IP_PRE_ROUTING` 点上。该函数主要功能是创建链接，即创建 `struct nf_conn` 结构，同时填充 `struct nf_conn` 中的一些必要的信息，例如链接状态、引用计数、helper 结构等。
 - 3.2 `ipv4_confirm()` 挂载在 `NF_IP_POST_ROUTING` 和 `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

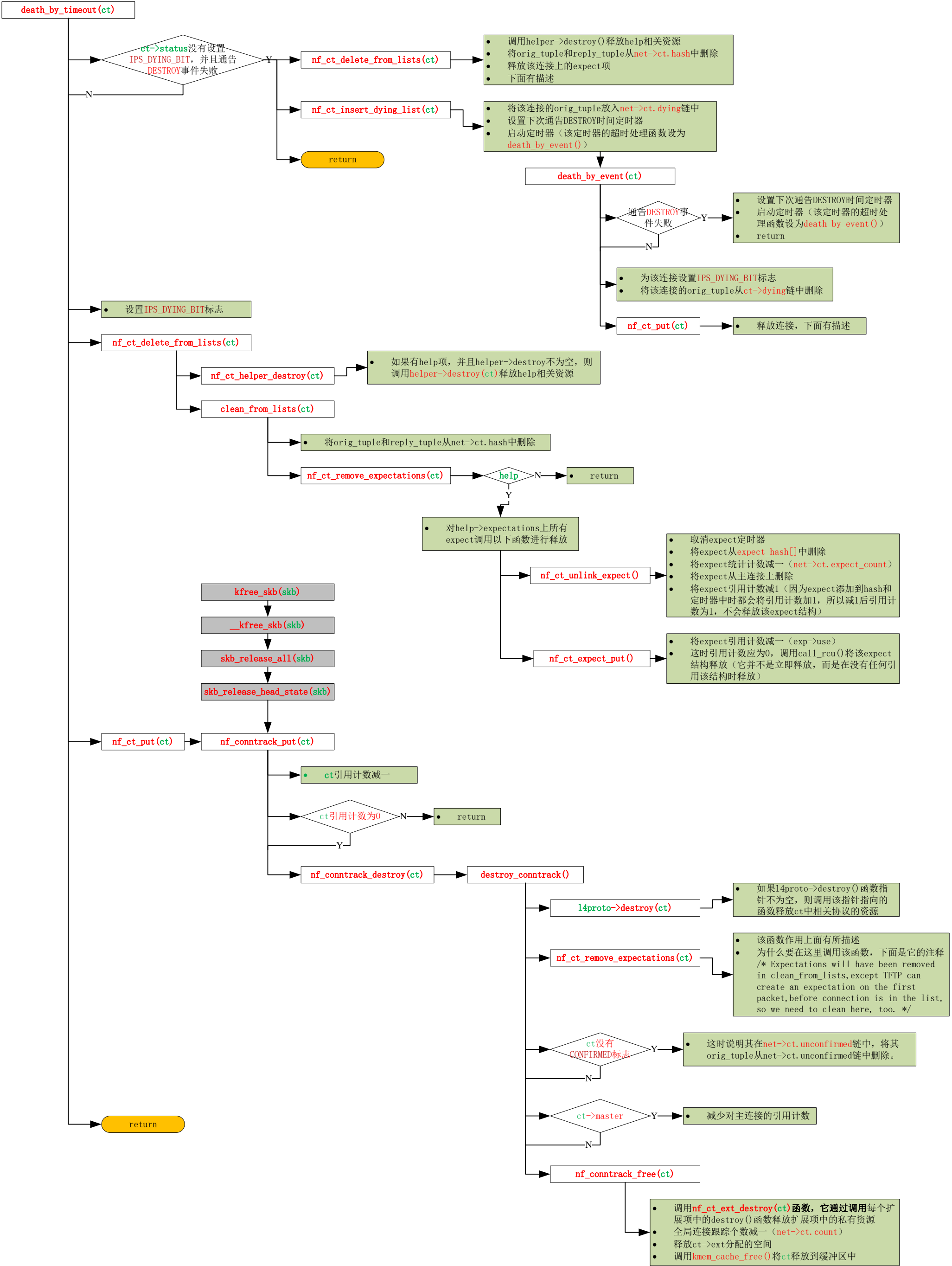
- 1 如上图所示，连链接跟踪主要由三个函数来完成，即 `ipv4_contrack_in()`，`ipv4_confirm()`与 `ipv4_contrack_local()`。其中 `ipv4_contrack_in()`与 `ipv4_contrack_local()` 都是通过调用函数 `nf_contrack_in()`来实现的，所以下面我们主要关注 `nf_contrack_in()`与 `ipv4_confirm()`这两个函数。`nf_contrack_in()`函数主要完成创建链接、添加链接的扩展结构(例如 `helper`, `acct` 结构)、设置链接状态等。`ipv4_confirm()`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。



ipv4_confirm



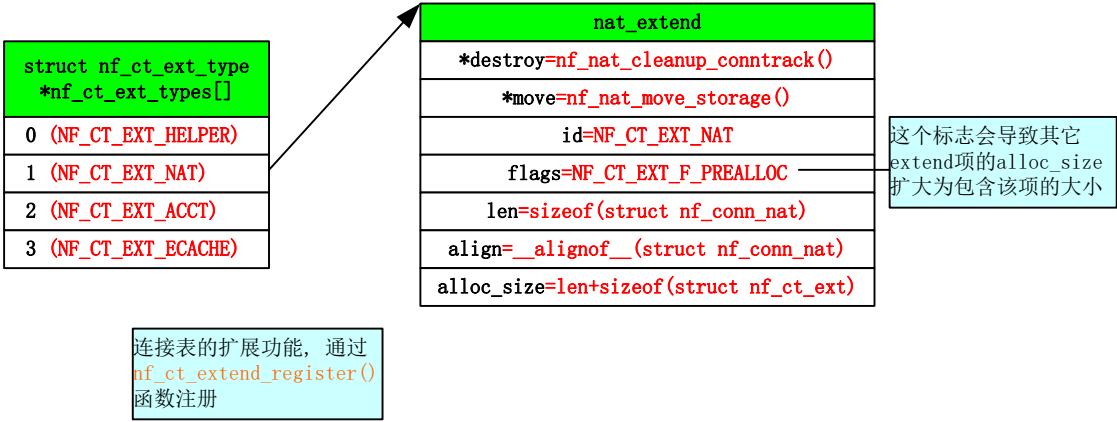
death_by_timeout



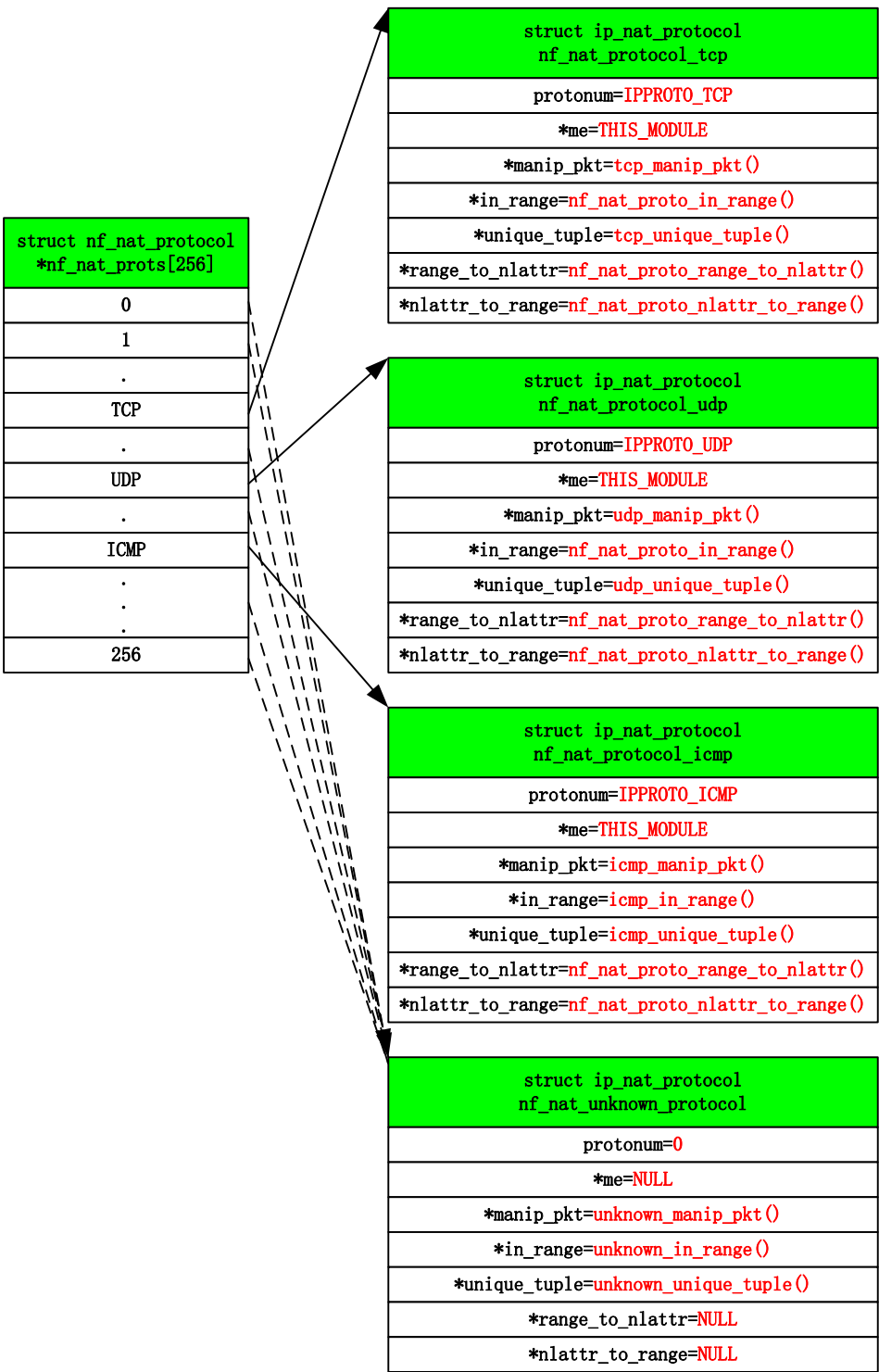
IPv4 利用 nf_conntrack 进行 NAT 转换

IPv4-NAT 初始化的资源

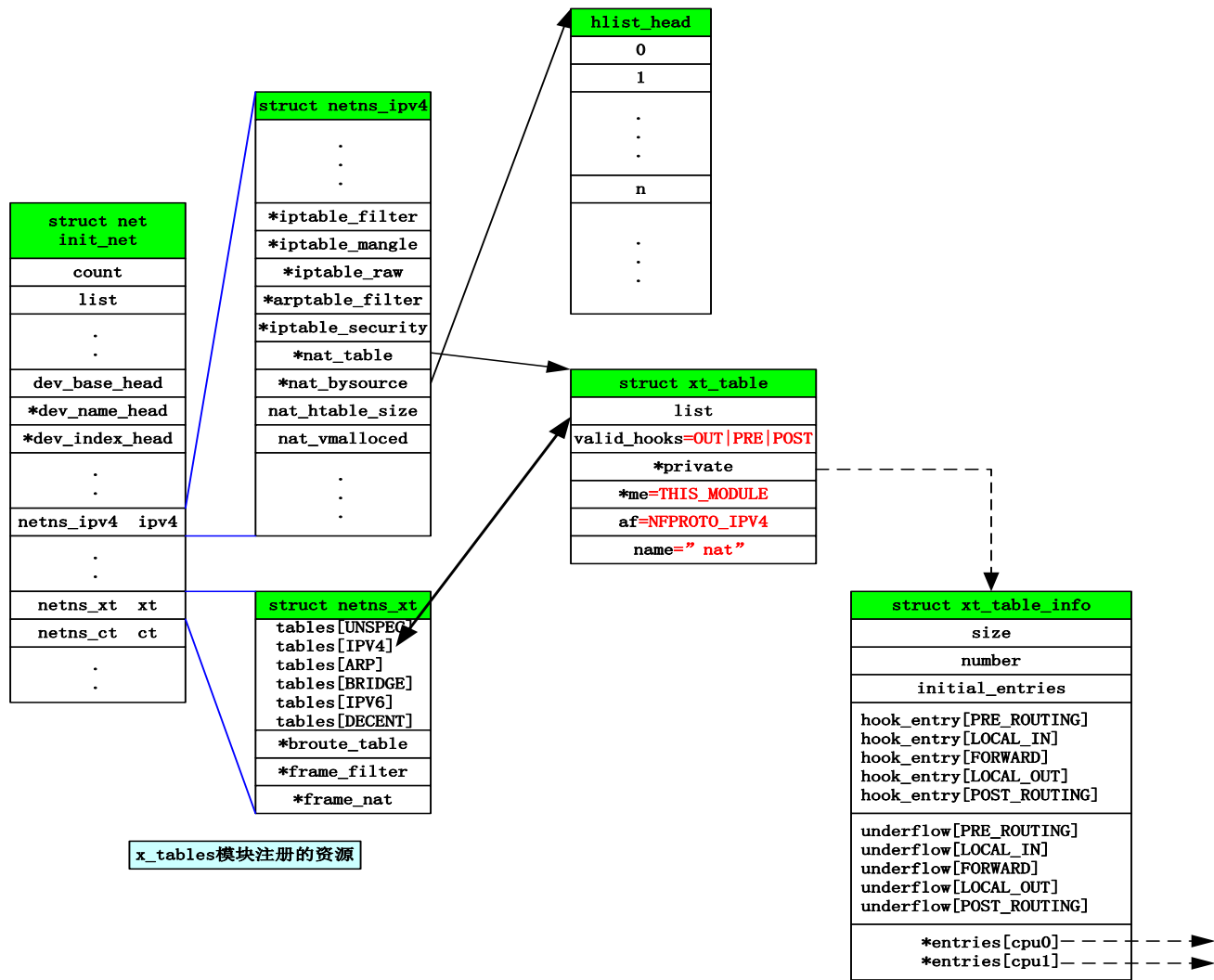
- 1 NAT 功能的连接跟踪部分初始化，通过函数 nf_nat_init()
 - 1.1 调用 nf_ct_extend_register() 注册一个连接跟踪的扩展功能。



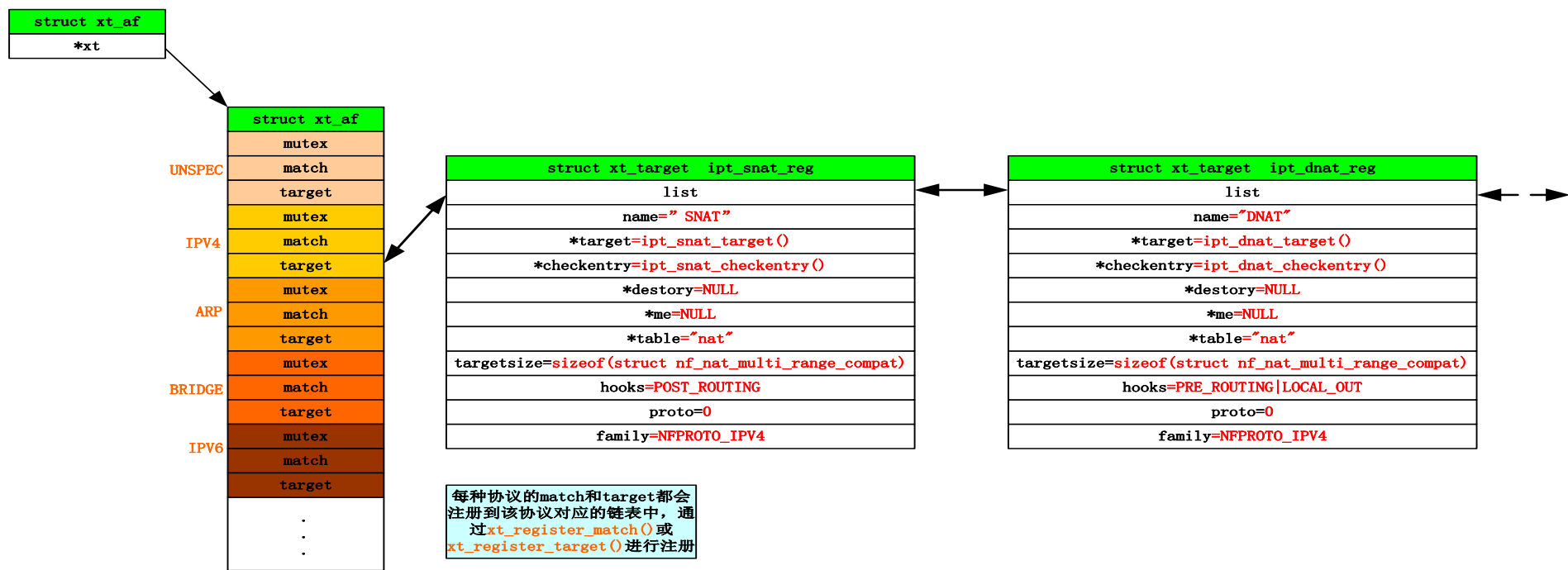
- 1.2 调用 register_pernet_subsys() --> nf_nat_net_init() 创建 net->ipv4.nat_bysource 的 HASH 表，大小等于 net->ct.htable_size。
 - 1.3 初始化 nf_nat_protos[] 数组，为 TCP、UDP、ICMP 协议指定专用处理结构，其它协议都指向默认处理结构。



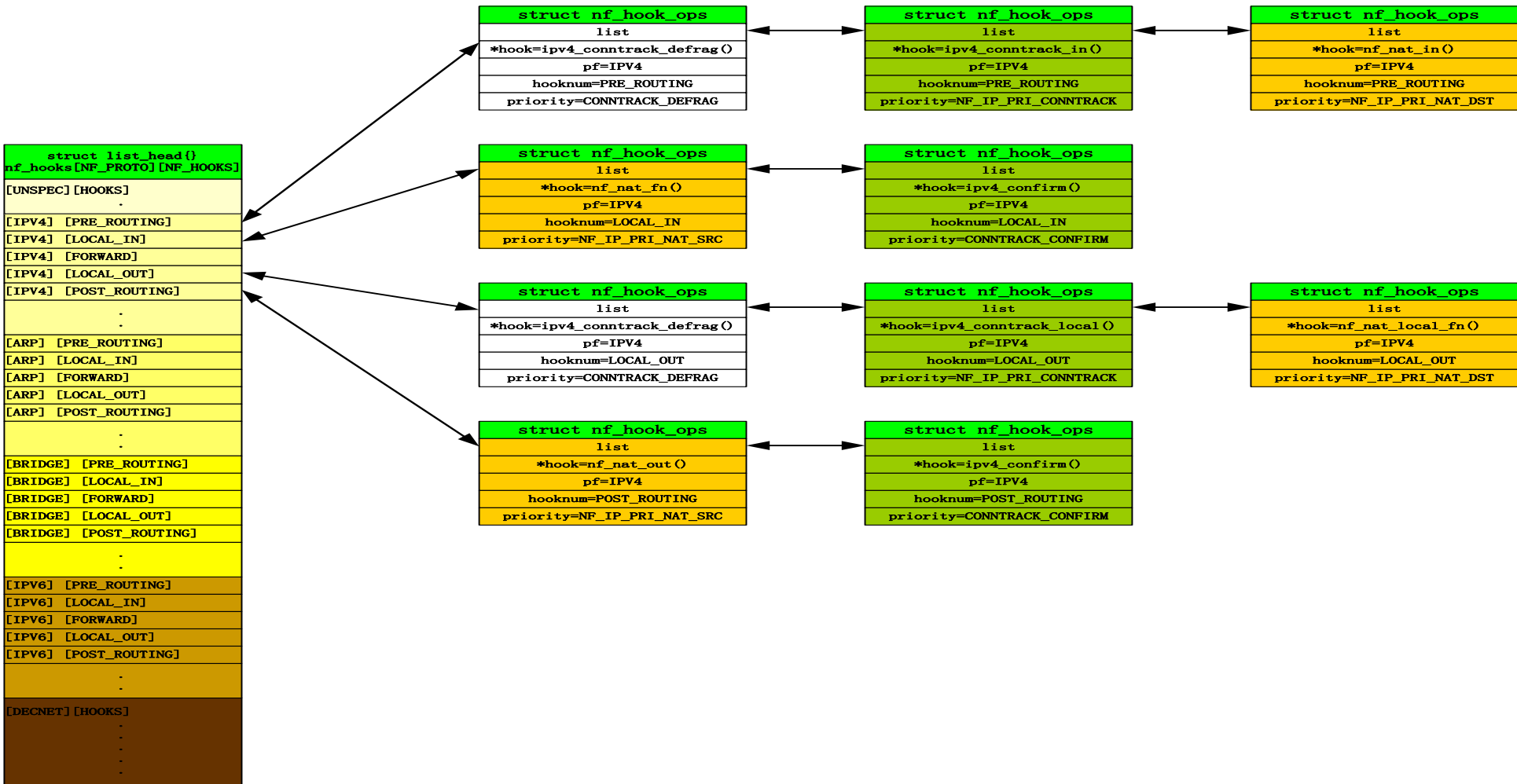
- 1.4 为 nf_conntrack_untracked 连接设置 IPS_NAT_DONE_MASK 标志。
 - 1.5 将 NAT 模块的全局变量 l3proto 指向 IPV4 协议的 nf_conntrack_l3proto 结构。
 - 1.6 设置全局指针 nf_nat_seq_adjust_hook 指向 nf_nat_seq_adjust() 函数。
 - 1.7 设置全局指针 nfnetlink_parse_nat_setup_hook 指向 nfnetlink_parse_nat_setup() 函数。
 - 1.8 设置全局指针 nf_ct_nat_offset 指向 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）

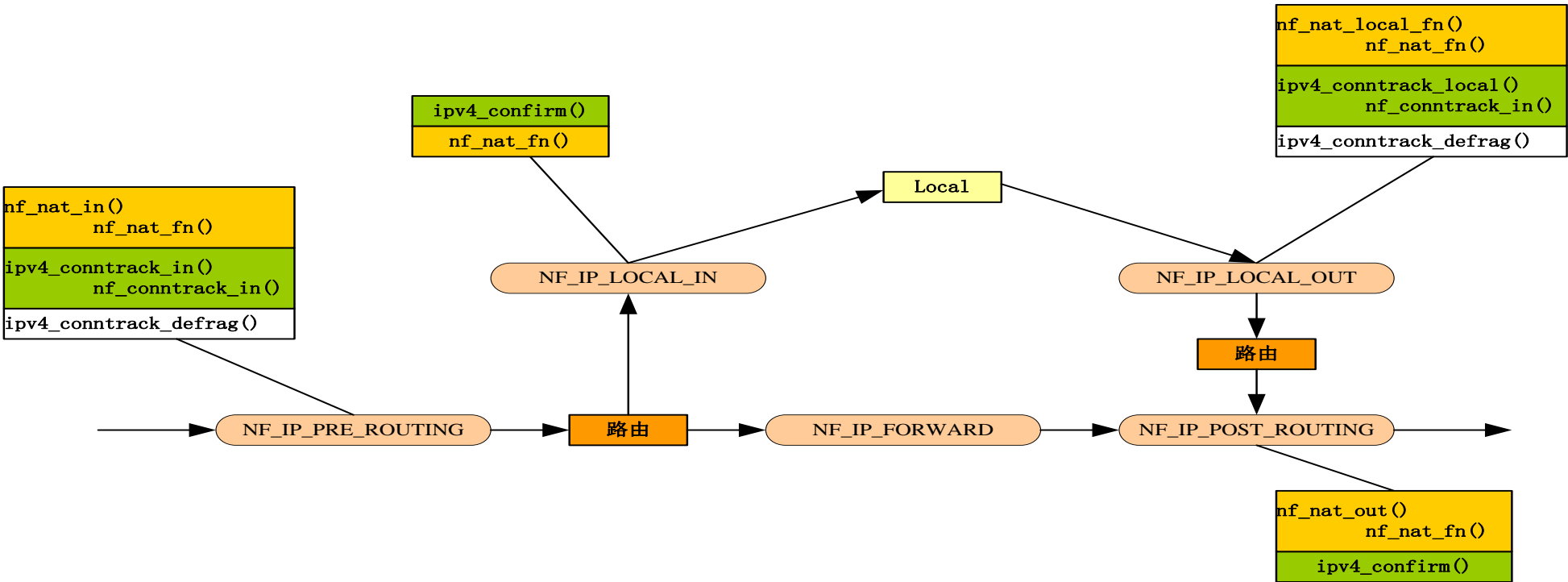


2.2 调用 nf_nat_rule_init() 注册 SNAT target 和 DNAT target（通过 xt_register_target()函数）



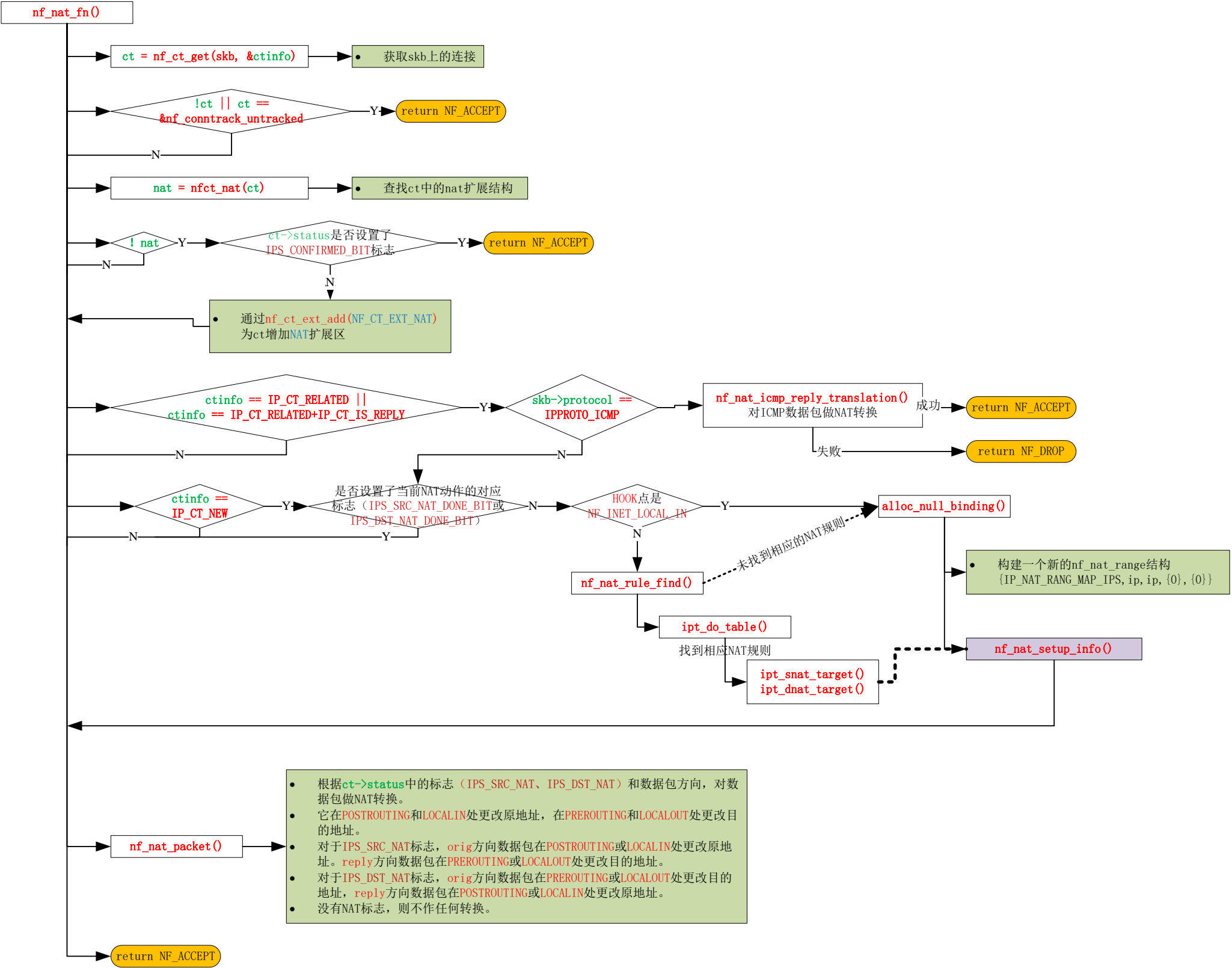
2.3 调用 nf_register_hooks() 挂载 NAT 的 HOOK 函数



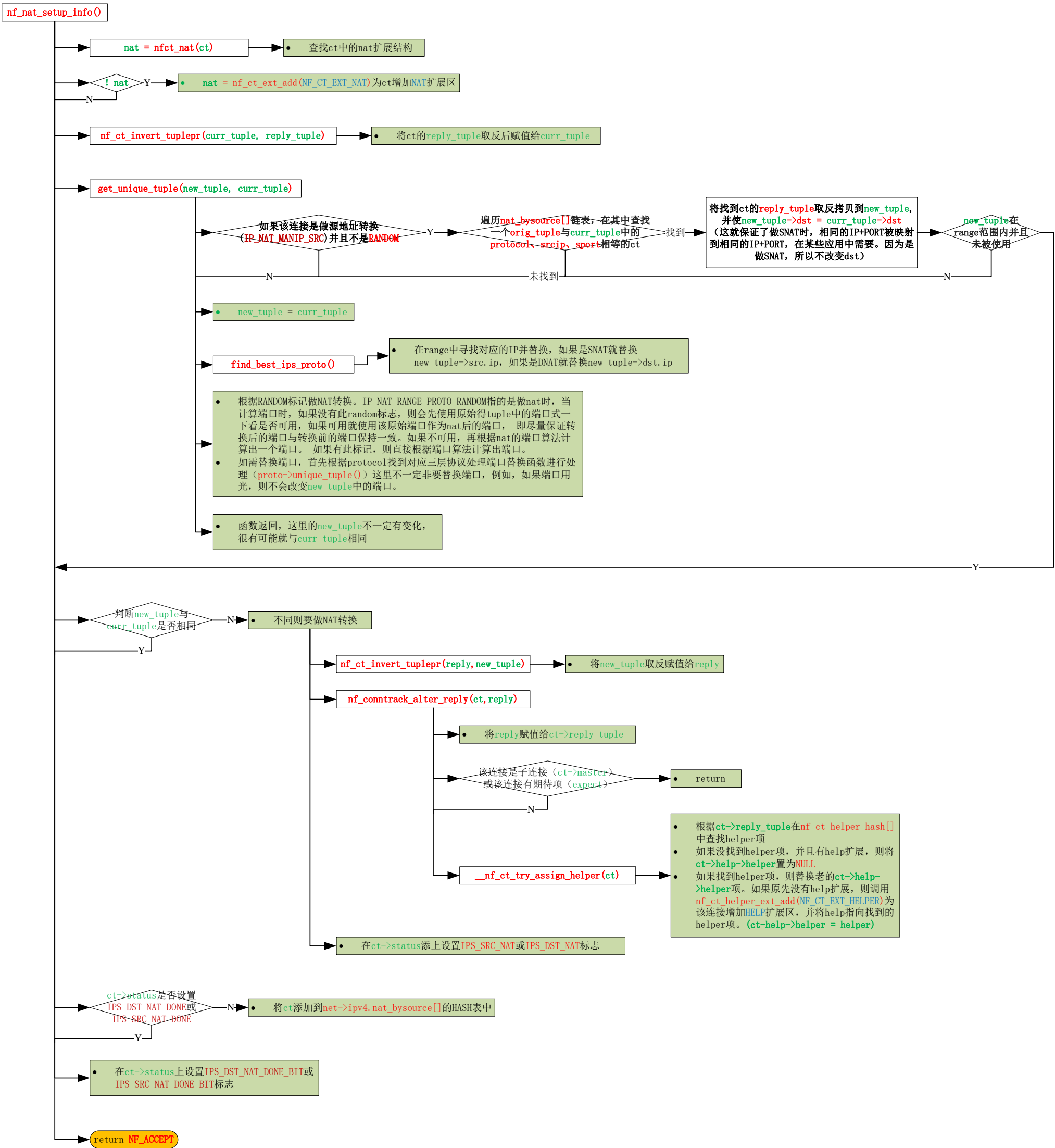


IPv4-NAT 处理流程

nf_nat_fn



nf_nat_setup_info

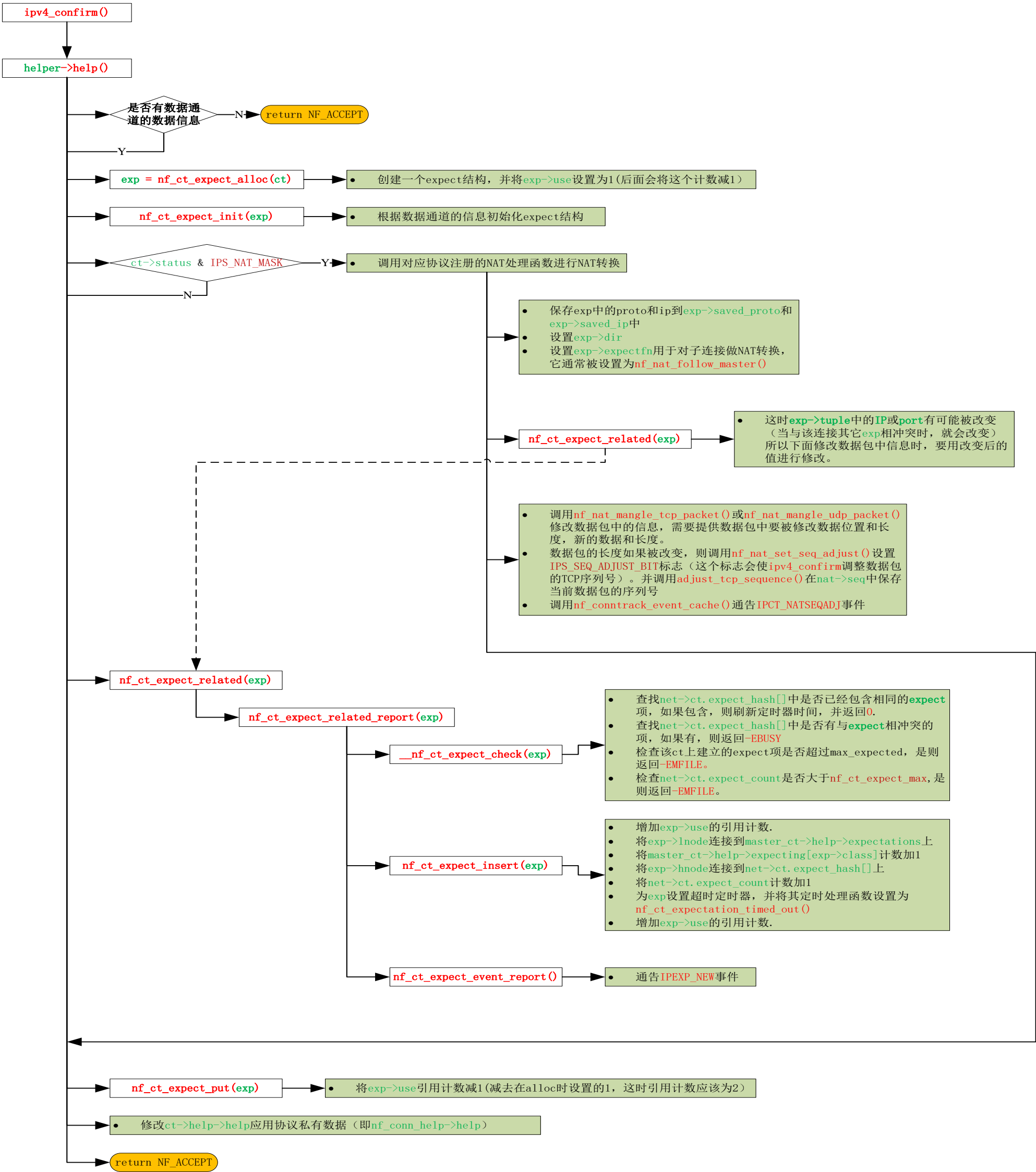


- 1
- 每个 ct 在第一个包就会做好 snat 与 dnat, nat 的信息全放在 reply tuple 中, orig tuple 不会被改变。一旦第一个包建立好 nat 信息后, 后续再也不会修改 tuple 内容了。
- 2
- orig tuple 中的地址信息与 reply tuple 中的地址信息就是原始数据包的信息。例如对 A->B 数据包同时做 snat 与 dnat, PREROUTING 处 B 被 dnat 到 D, POSTROUTING 处 A 被 snat 到 C。则 ct 的内容是: A->B | D->C, A->B 说明了 orig 方向上数据包刚到达墙时的地址内容, 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。

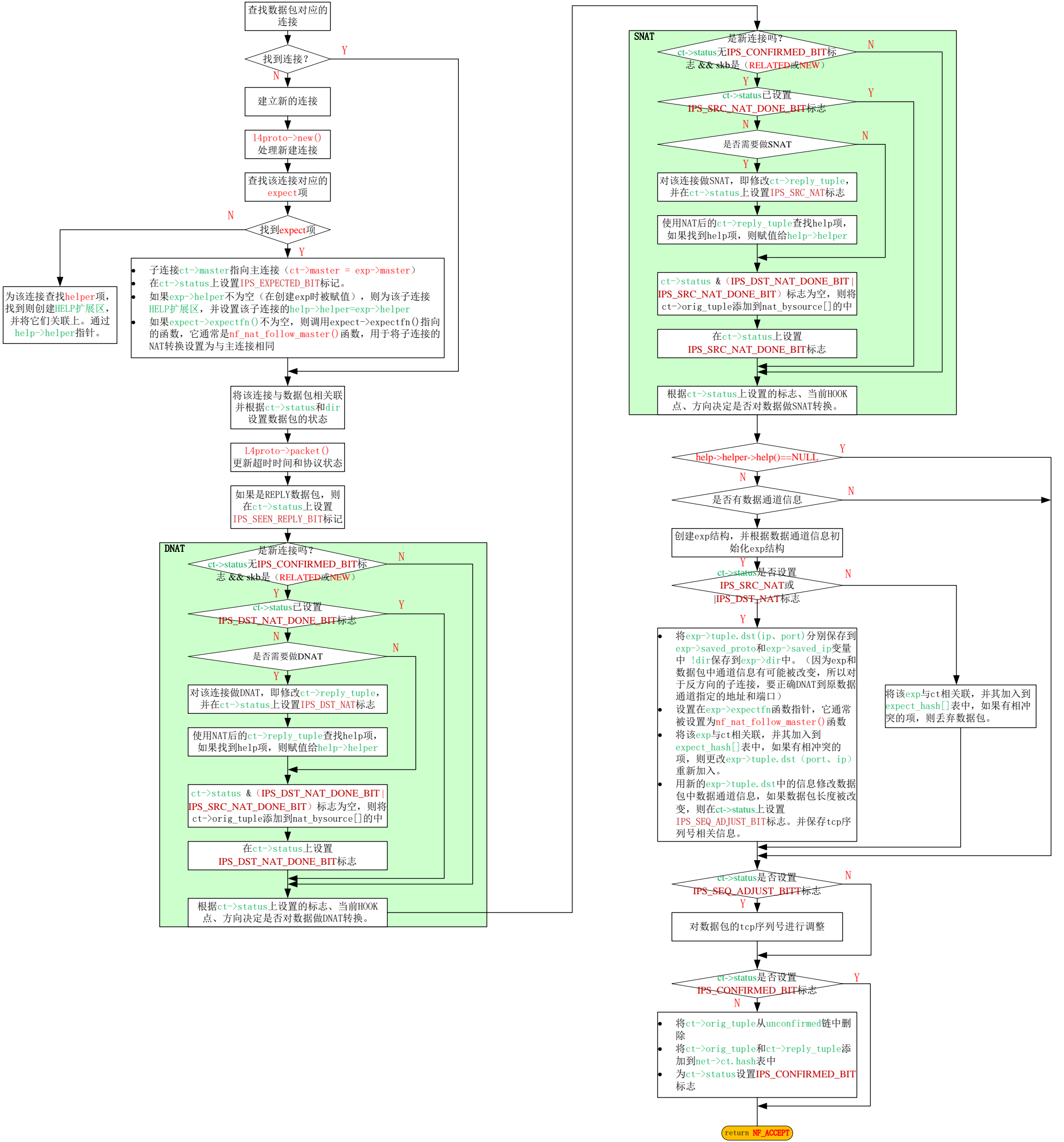
- 6 第一个包之后，ct 的两个方向的 tuple 内容就固定了，所有的 nat 操作都必须在第一个包就完成。所以 daddr = &ct->tuplehash[!dir].tuple.dst.u3;会有这样的操作。
- 7 IPS_SRC_NAT 与 IPS_DST_NAT，如果被设置，表示经过了 NAT，并且 ct 中的 tuple 被做过 SNAT 或 DNAT。
- 8 数据包永远都是在 PREROUTING 链做目的地址和目的端口转换，在 POSTROUTING 链做原地址和原端口转换。是否要做 NAT 转换则要根数据包方向（dir）和 NAT 标志（IPS_SRC_NAT 或 IPS_DST_NAT）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->tupletuplehash[!dir].tuple 这样的操作。

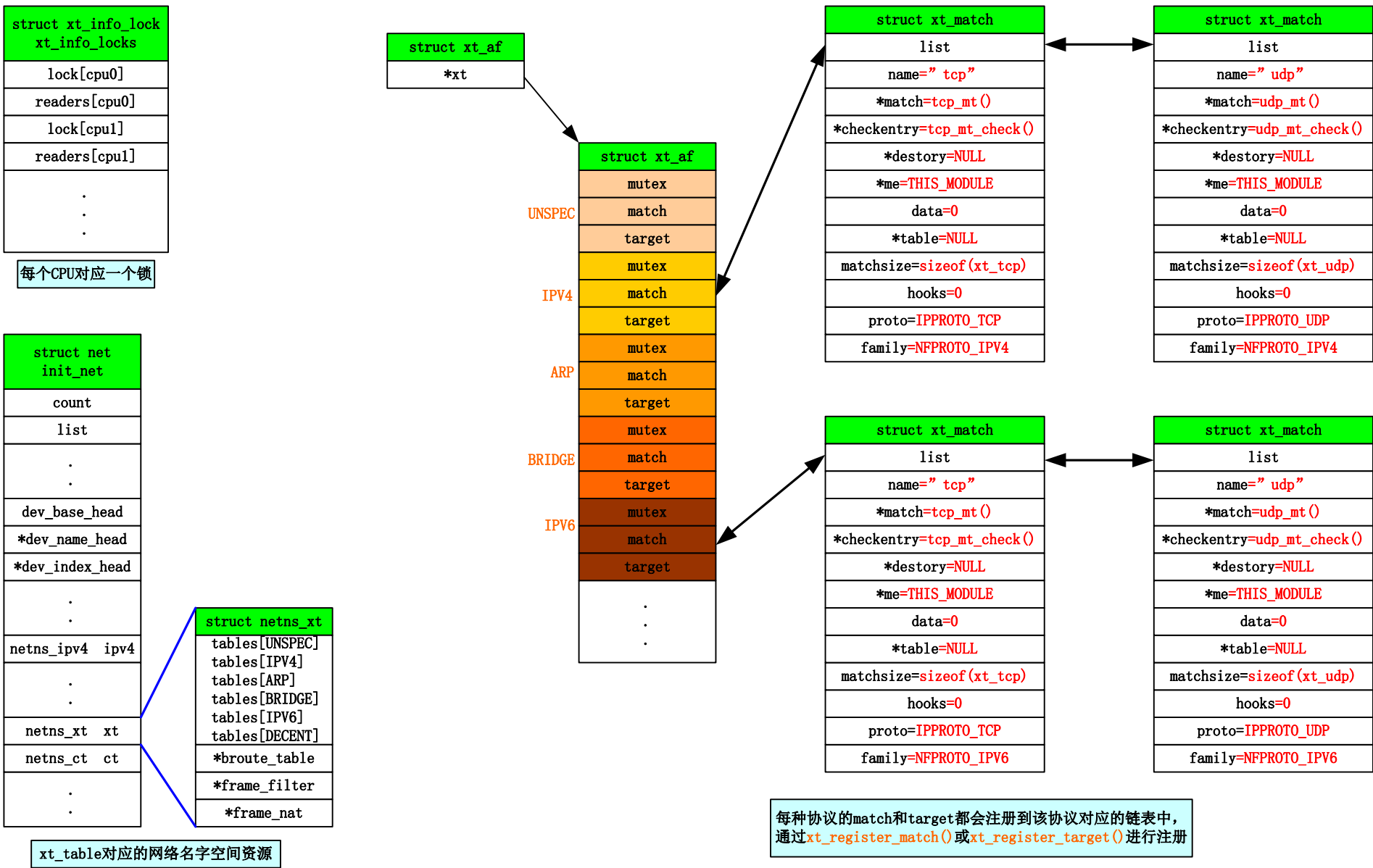
2 有子连接

- 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。

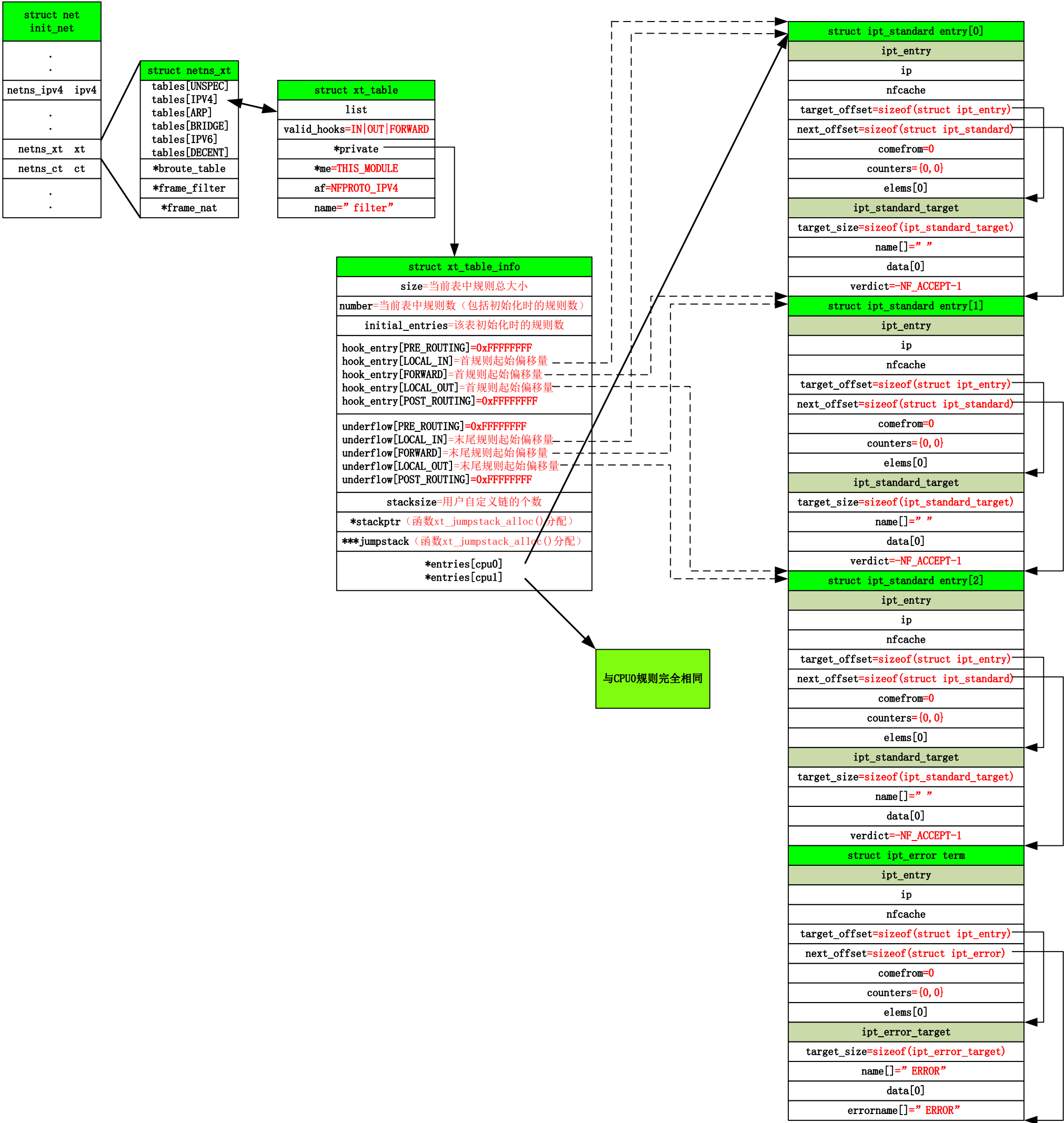
包过滤规则

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时配，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_unregisgter_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_regisgter_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。

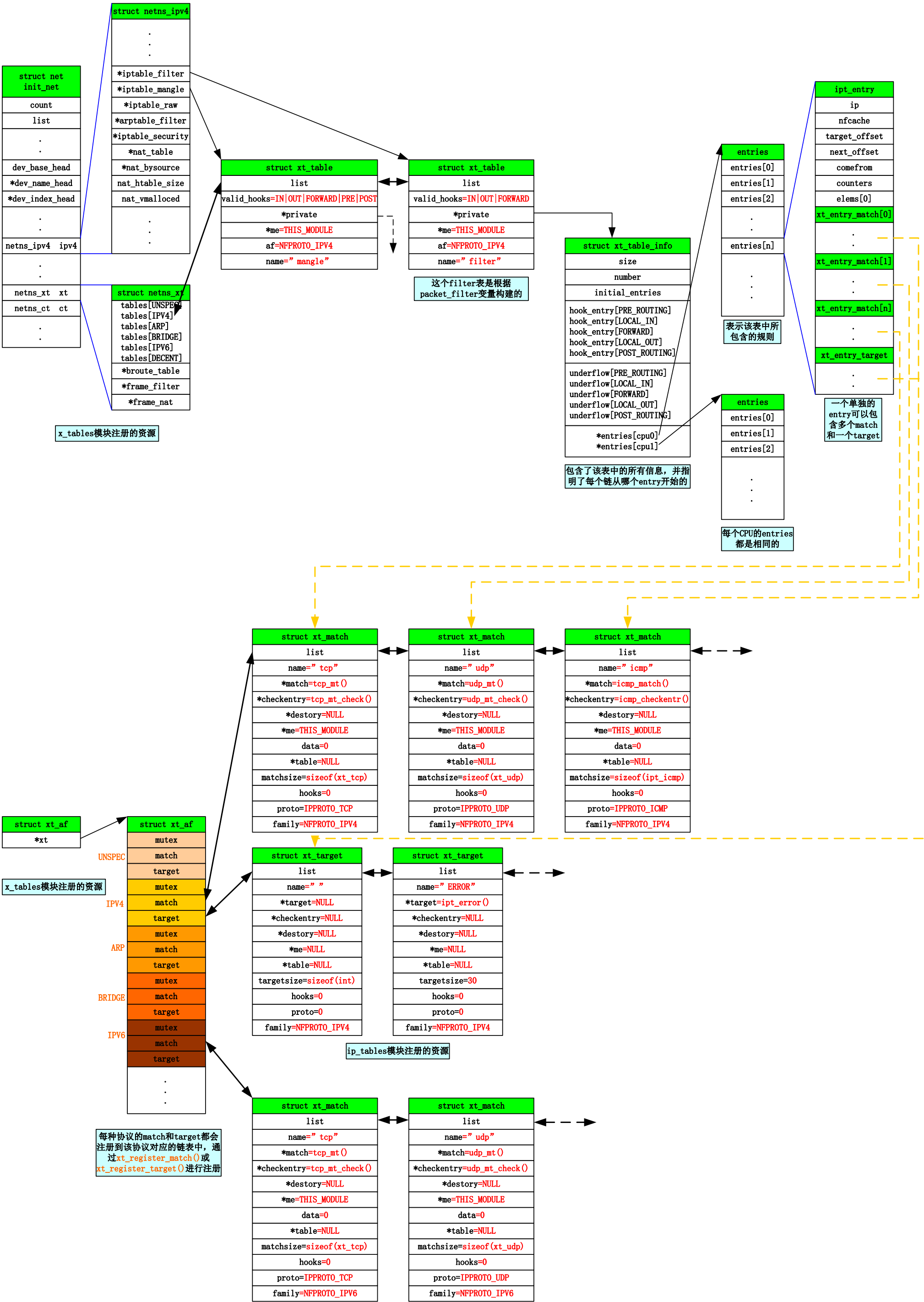
7.2 struct xt_table_info *xt_replace_table(struct xt_table *table, unsigned int num_counters, struct xt_table_info *newinfo, int *error)
为 newinfo 调用 xt_jumpstack_alloc(struct xt_table_info *i)初始化 stack 相关数据，然后使 table->private 指向 newinfo，并返回 oldinfo。

8 总结：

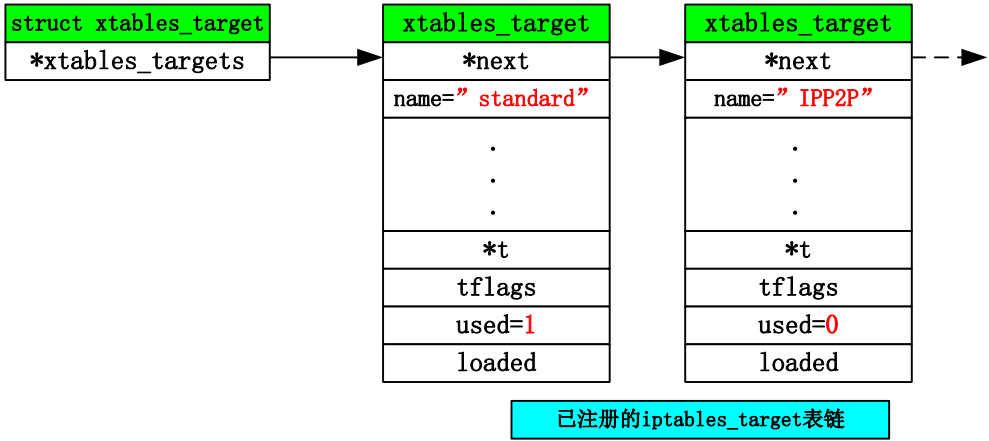
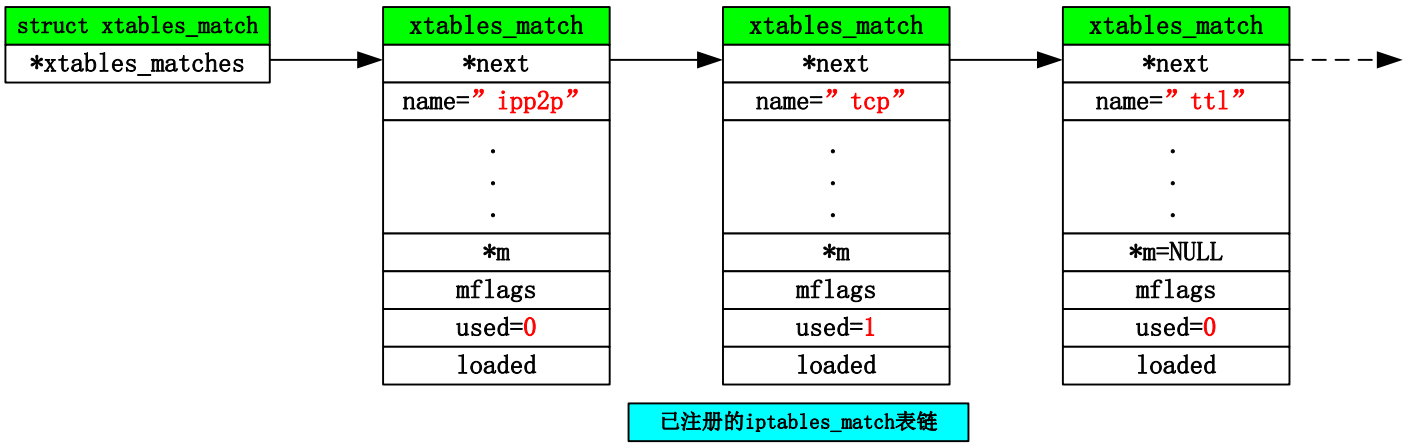
8.1 为每个 CPU 建立一个规则集的原因：是为了在更新规则计数时避免加写锁，每个 CPU 只更新自己规则集中的规则计数。用户上下文通过 write_lock_bh()来阻止本地 CPU 和其它 CPU 访问或修改这个规则集，这样它就可以读取或更新规则集了。

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



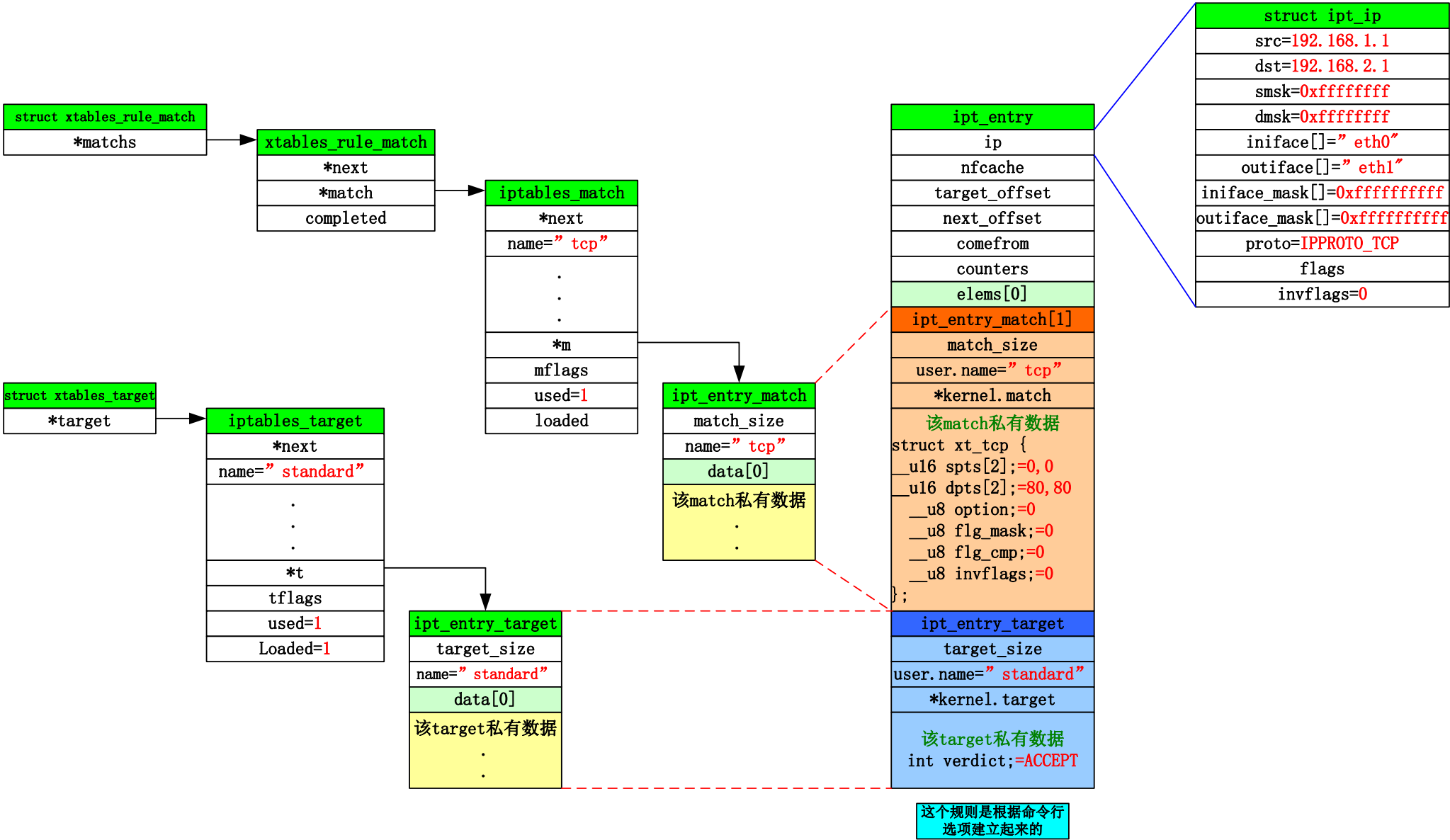
Iptable 用户态的资源



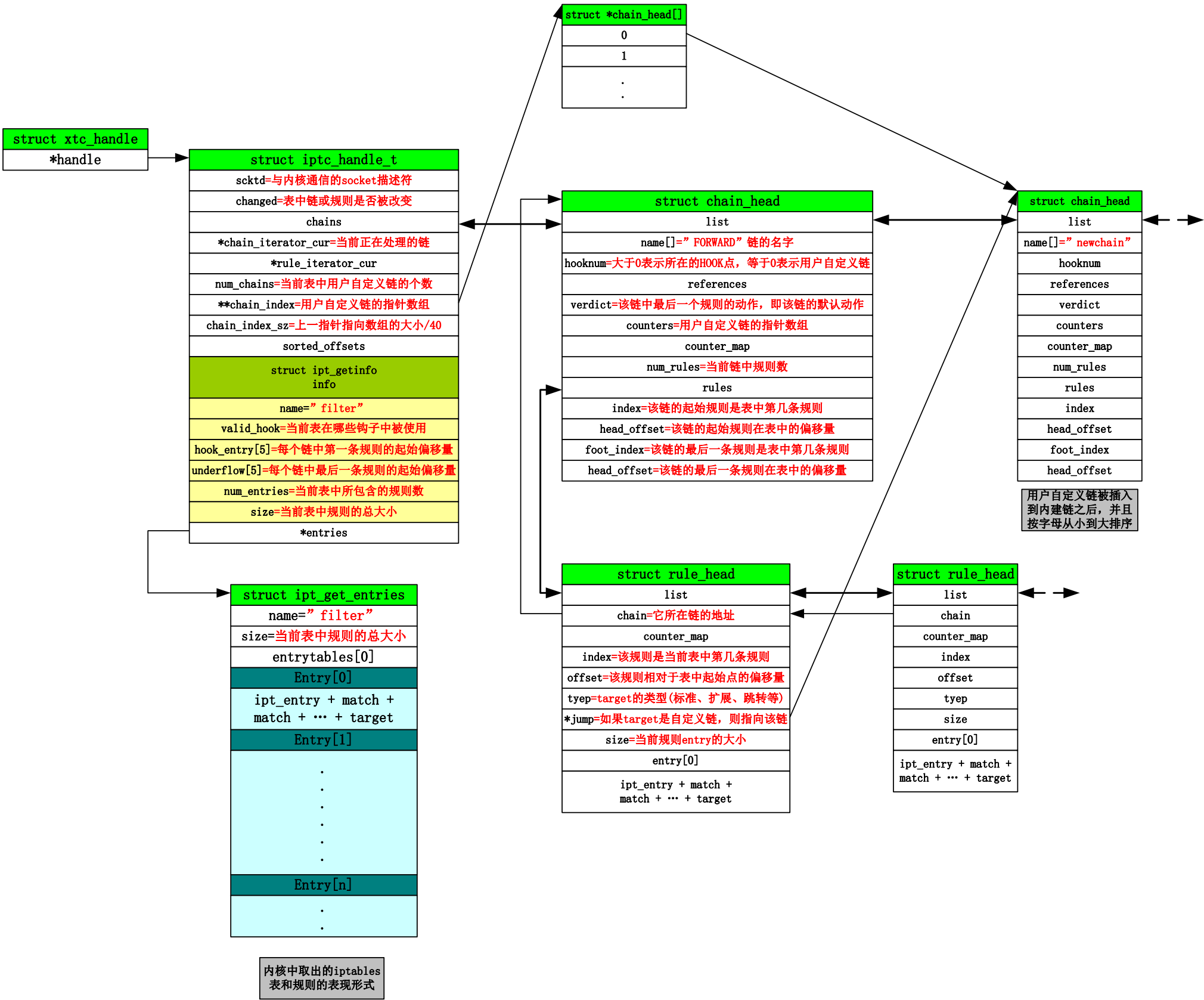
- 1. 用户态的 iptables 包含许多 match 和 target 资源，它们都以链表的形式进行组织。
- 2. 可以使用 xtables_register_match()和 xtables_register_target()函数扩展 match 与 target 资源。

一条 iptables 规则是如何组织的

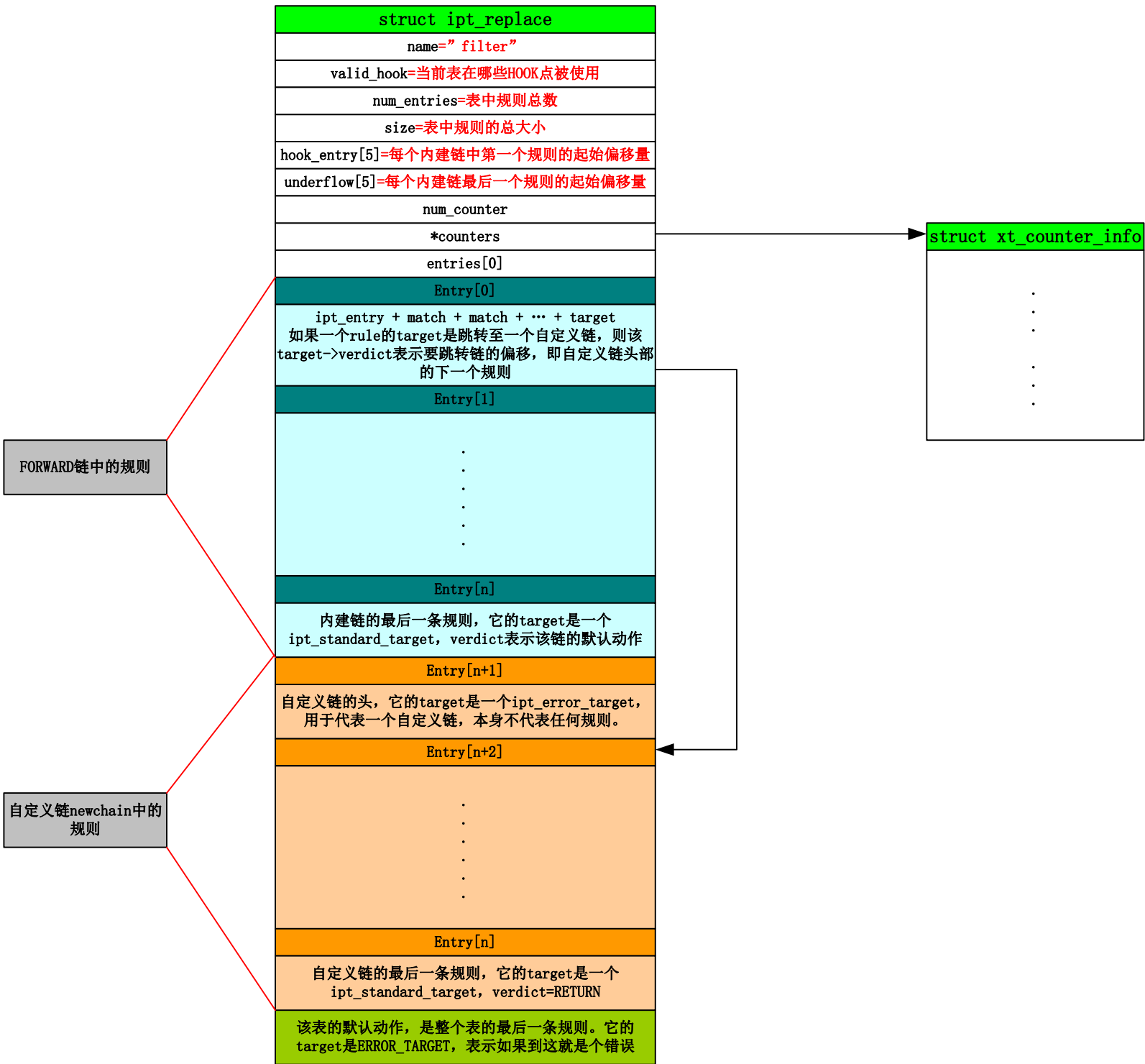
Iptables -t filter -A FORWARD -s 192.168.1.1 -d 192.168.2.1 -i eth0 -o eth1 -p tcp --dport 80 -j ACCEPT



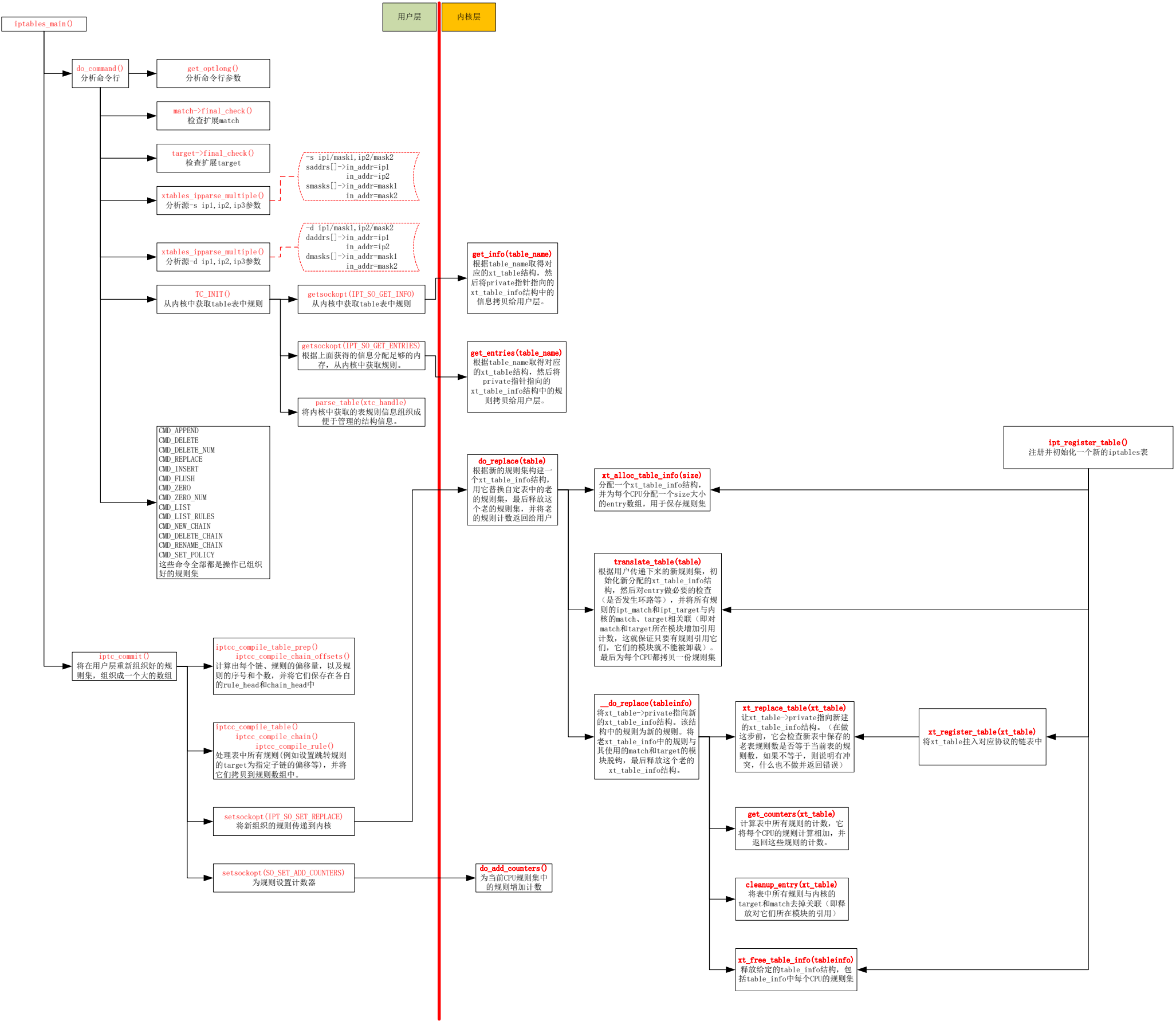
Iptables 对一个表中的规则是如何组织和操作的



Iptables 如何组织规则与内核通信



Iptable 的执行流程



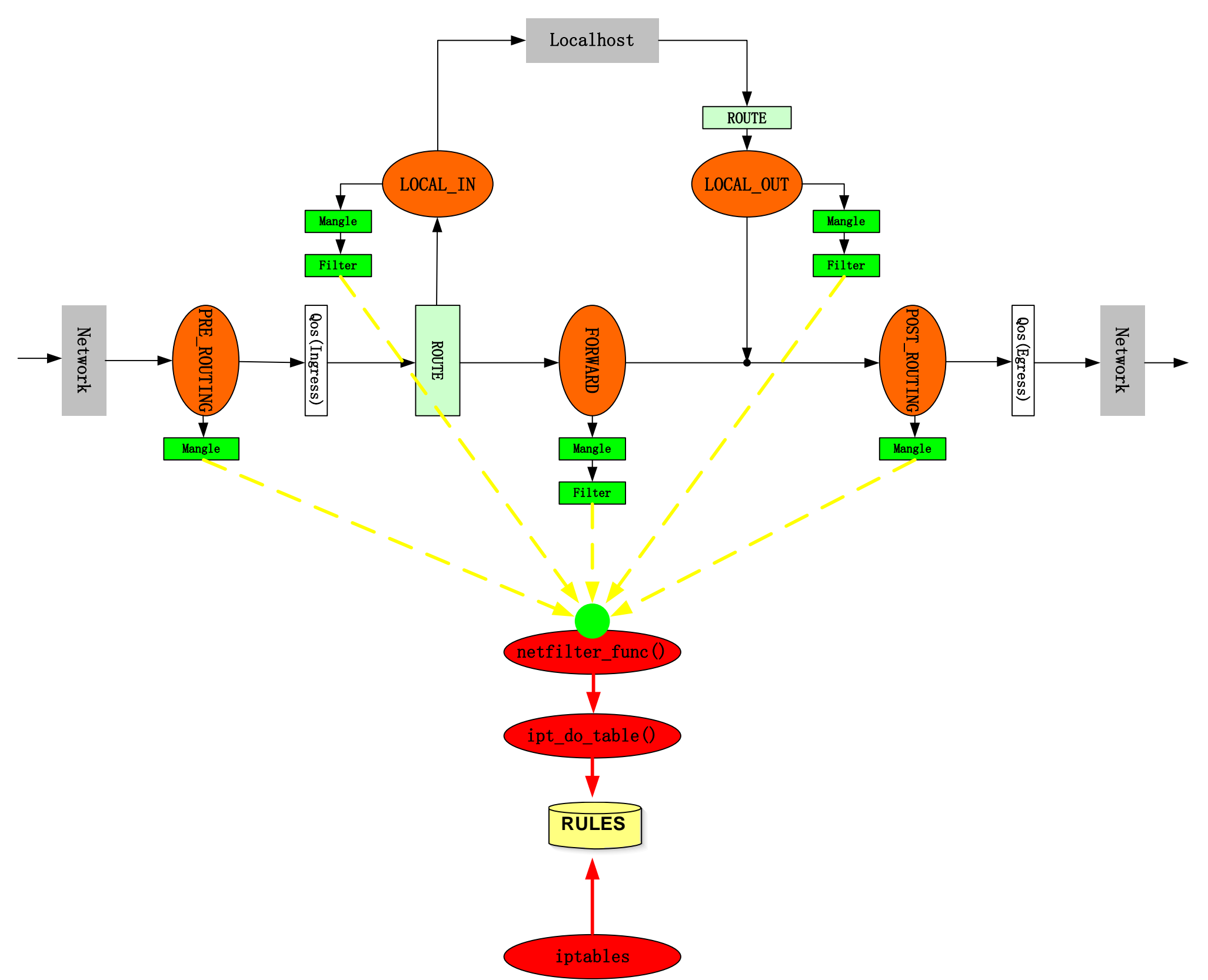
1 Iptables 包含以下 target

- 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_REPEAT-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 + data[] (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

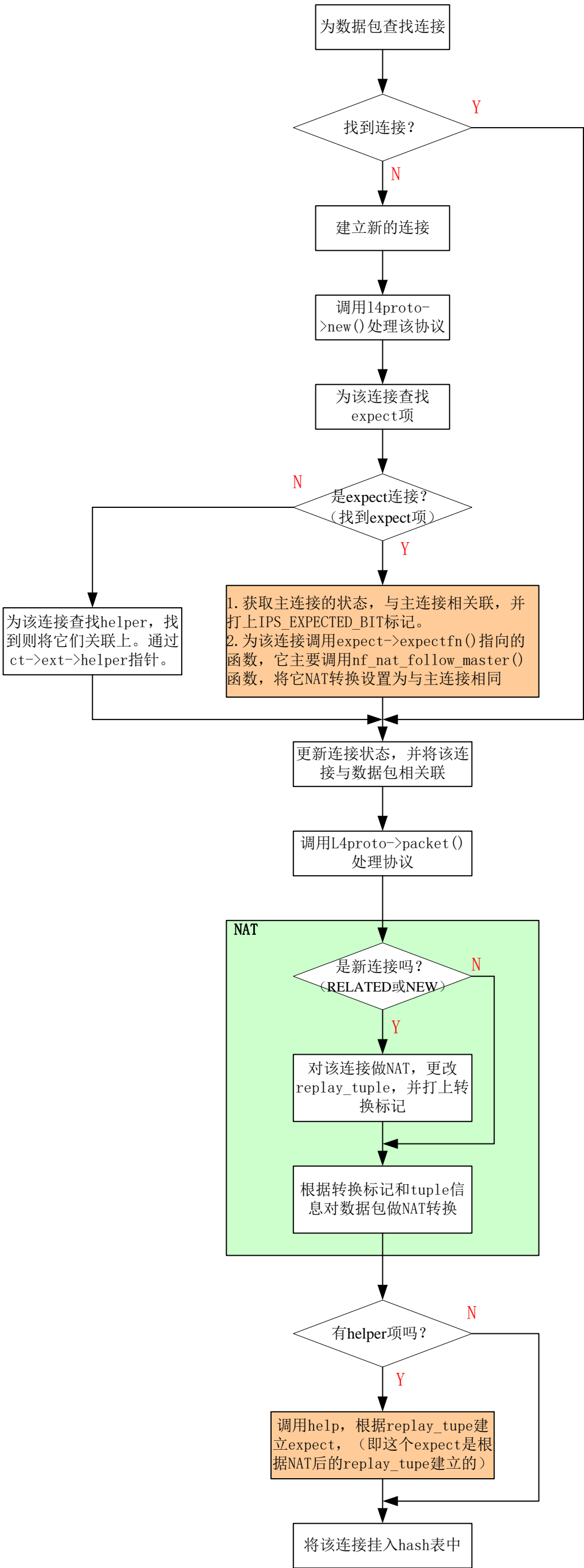
Iptables 的 hook 与规则



Iptables 的互斥

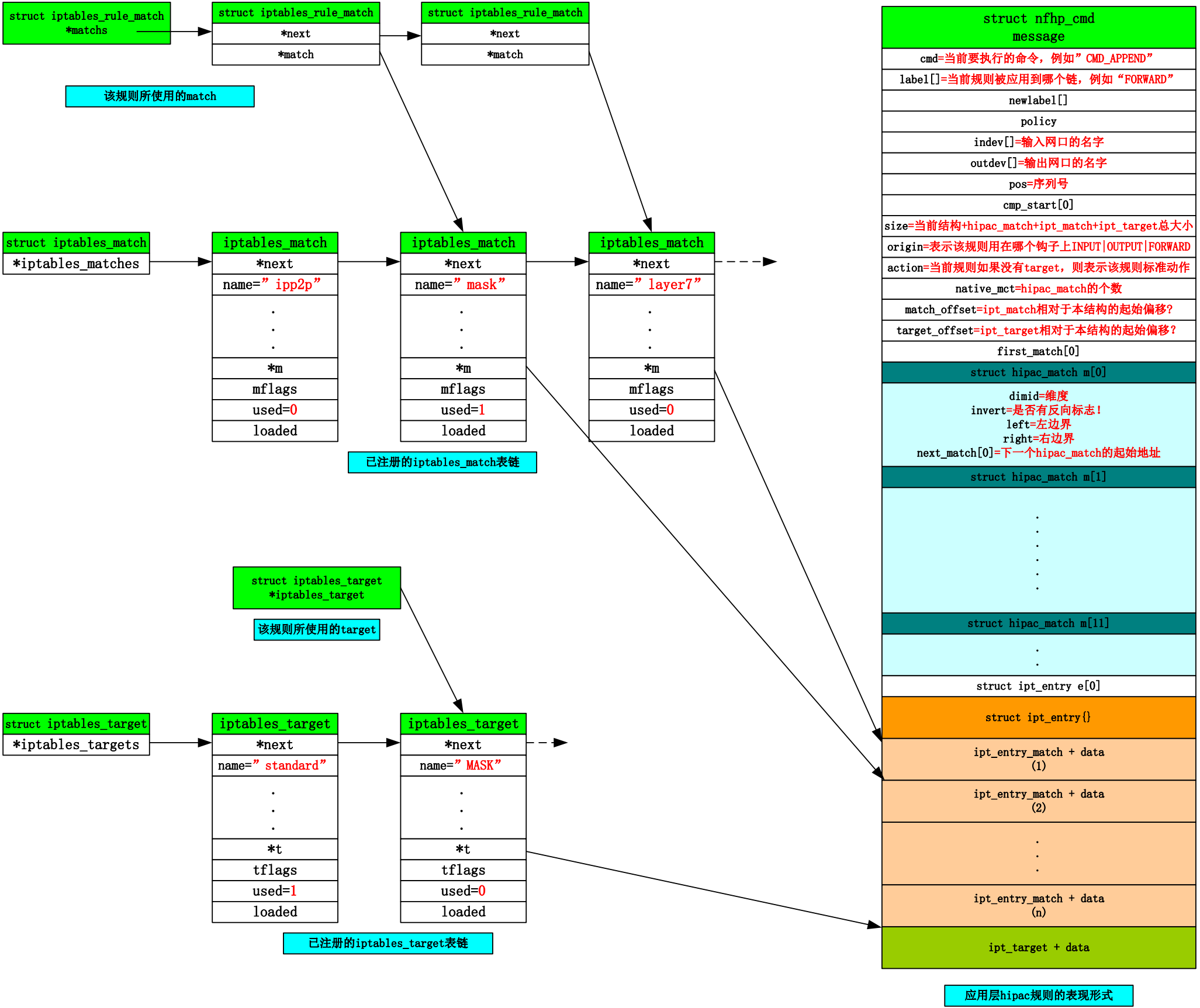
- 1 iptables 在应用层本身是没有互斥的，它是在内核层根据规则数进行判断的，执行流程如下：
 - 1.1 首先从内核中将老的规则取出，同时会将当前的规则数保存在 `ipt_replace->num_counters` 中。
 - 1.2 修改规则，并将新的规则下发到内核。
 - 1.3 内核 `xt_replace_table()` 函数在用新规则替换当前规则之前，先将新规则的 `ipt_replace->num_counter`（之前保存过的）与当前规则数对比，如果不相等则报错。
 - 1.4 这就保证了多个用户使用 `iptables` 添加规则时，不会相互覆盖。但如果第一个用户使用 `iptables -R` 替换一个规则（没有使规则数改变），而另一个用户同时在添加、删除或修改一个规则，就有可能导致相互覆盖。

连接表的建立、动态协议的识别、NAT 转换之间的处理流程



NF-hipac

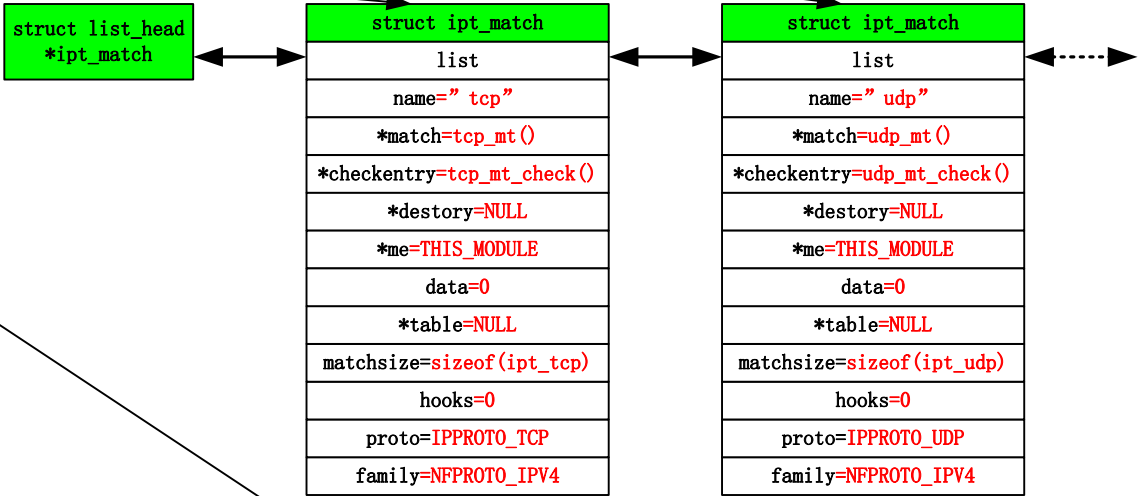
NF-hipac 用户态规则的组织形式



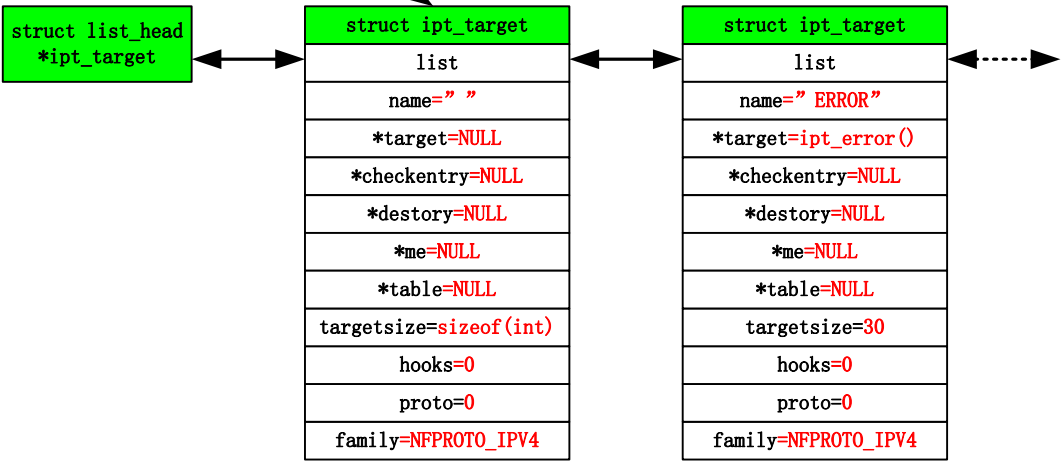
NF-hipac 内核态规则的组织形式

struct chian_rule
head
*dtr
pos=序列号
size=hipac_rule+hipac_match+ipt_match+ipt_target
origin=表示该规则用在哪个钩子上INPUT OUTPUT FORWARD
action=当前规则如果没有target，则表示该规则标准动作
native_mct=hipac_match的个数
match_offset=ipt_match相对于本结构的起始偏移
target_offset=ipt_target相对于本结构的起始偏移?
first_match[0]
struct hipac_match m[0]
dimid=维度 invert=是否有反向标志! left=左边界 right=右边界 next_match[0]=下一个hipac_match的起始地址
struct hipac_match m[1]
.
.
.
.
.
struct hipac_match m[n]
.
.
struct ipt_entry{
ipt_entry_match + data (1)
ipt_entry_match + data (2)
ipt_entry_match + data (n)
ipt_target + data

内核的hipac规则的表现形式

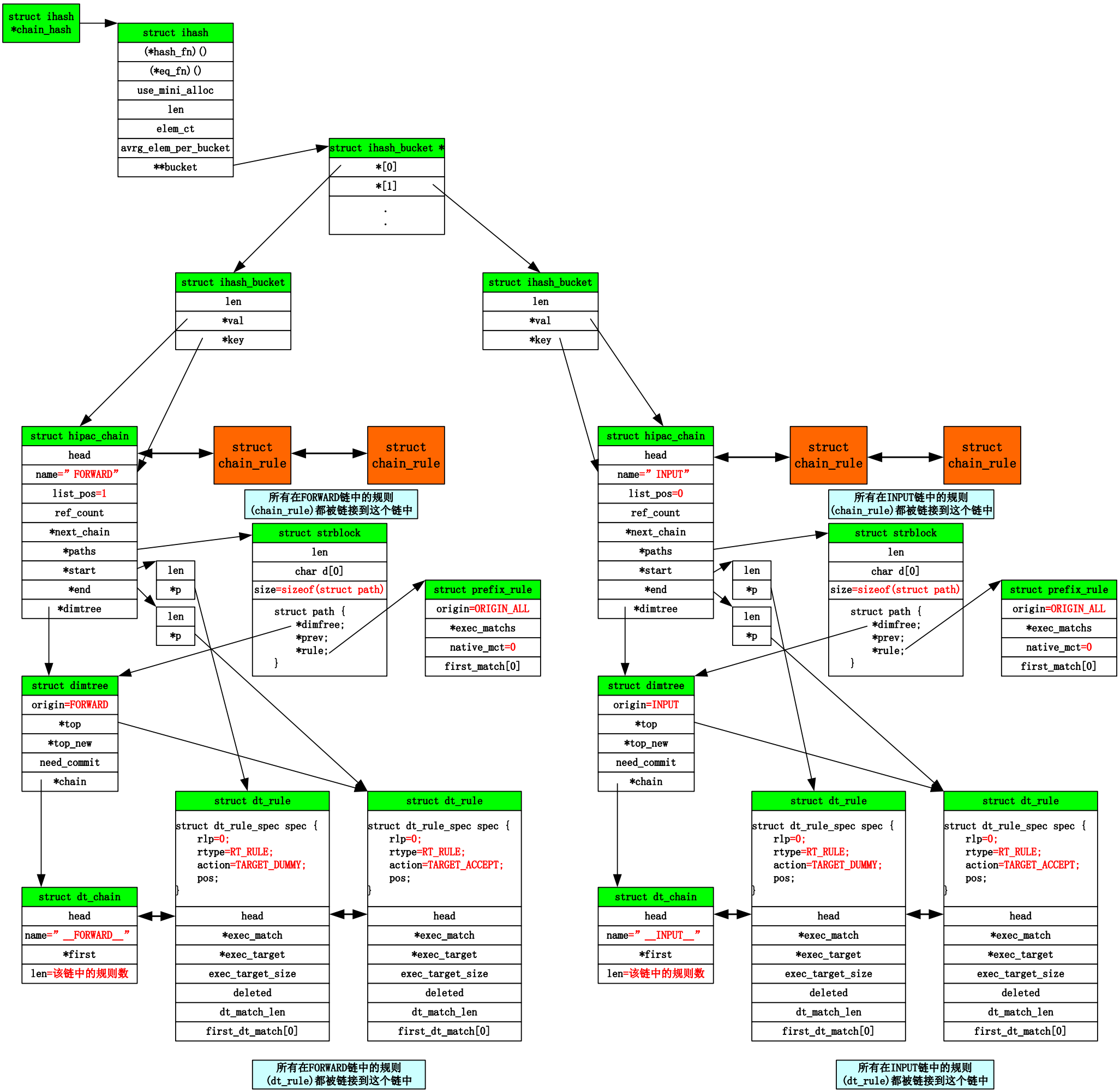


内核中注册的ipt_match链表

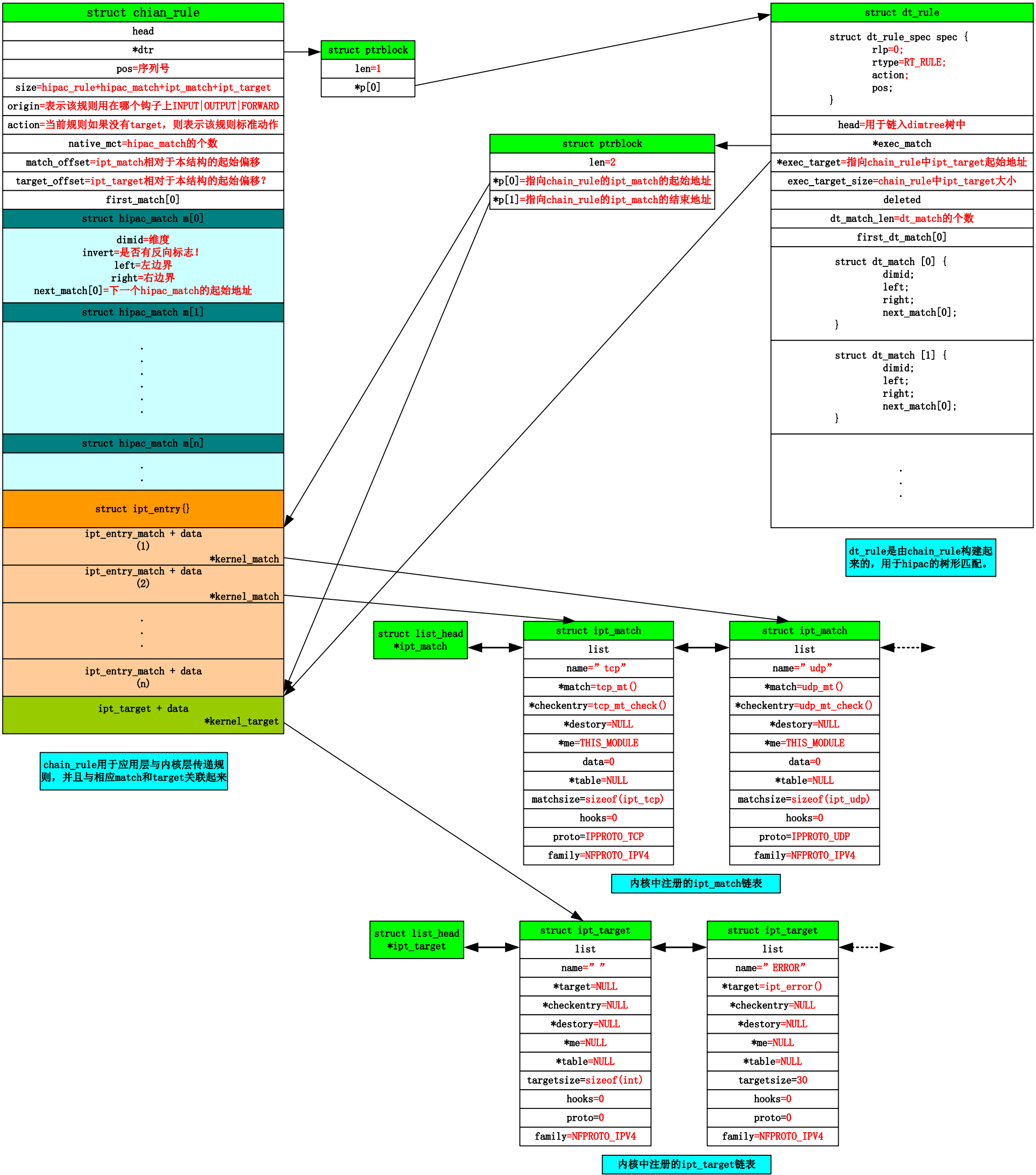


内核中注册的ipt_target链表

NF-hipac 内核初始化后组织形式

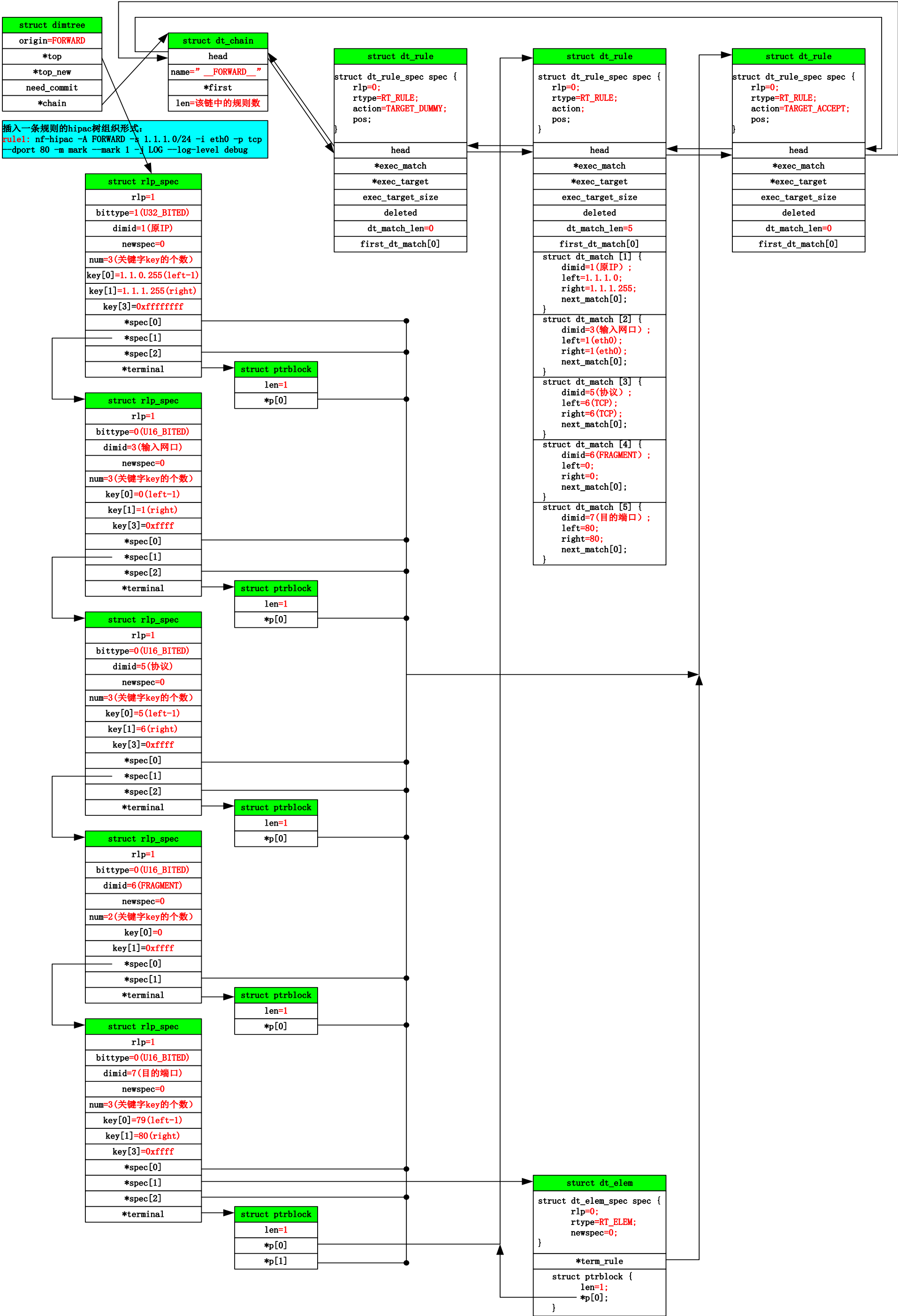


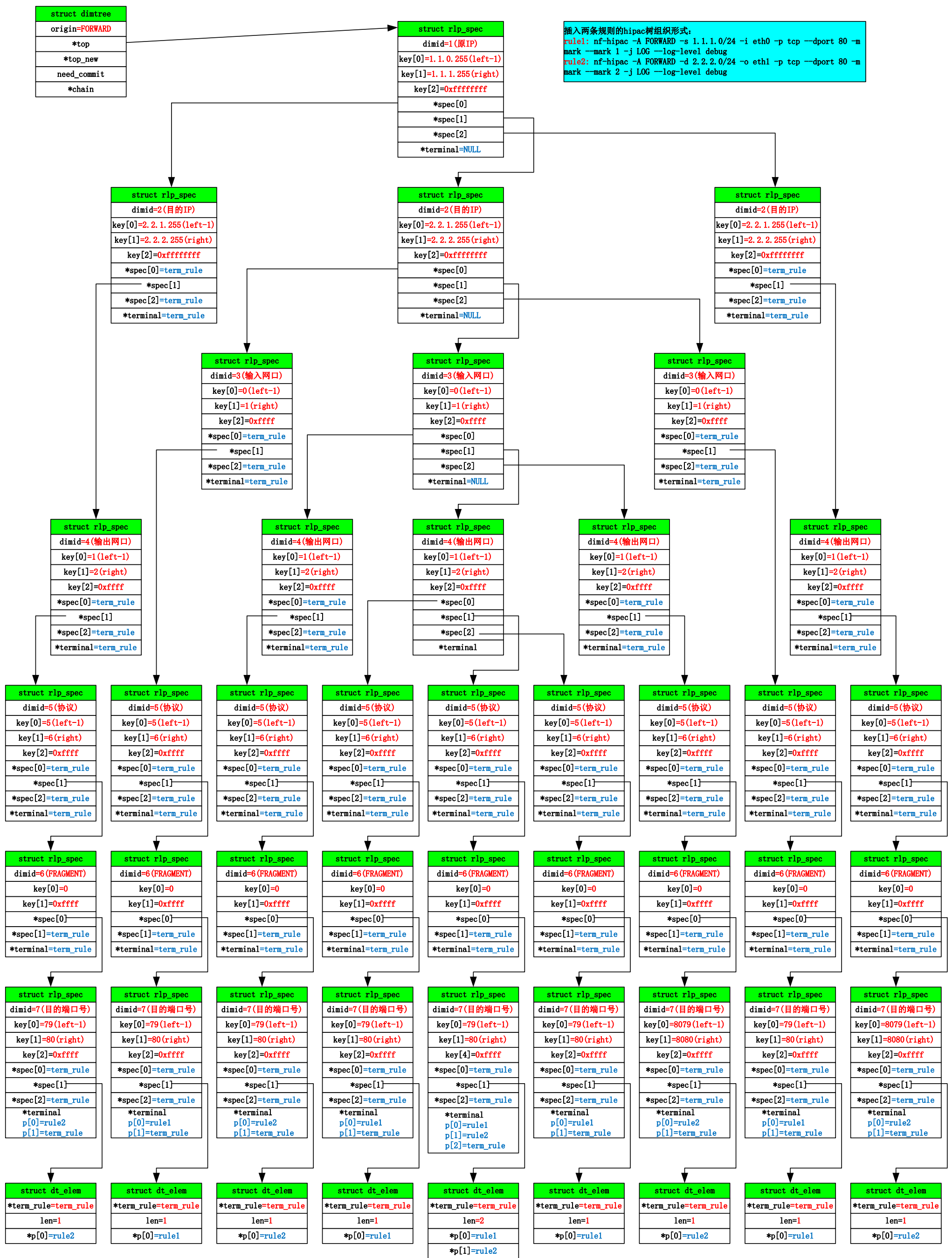
dt_rule 与 chain_rule 之间的关系（dt_rule 是由 chain_rule 构建出来的）



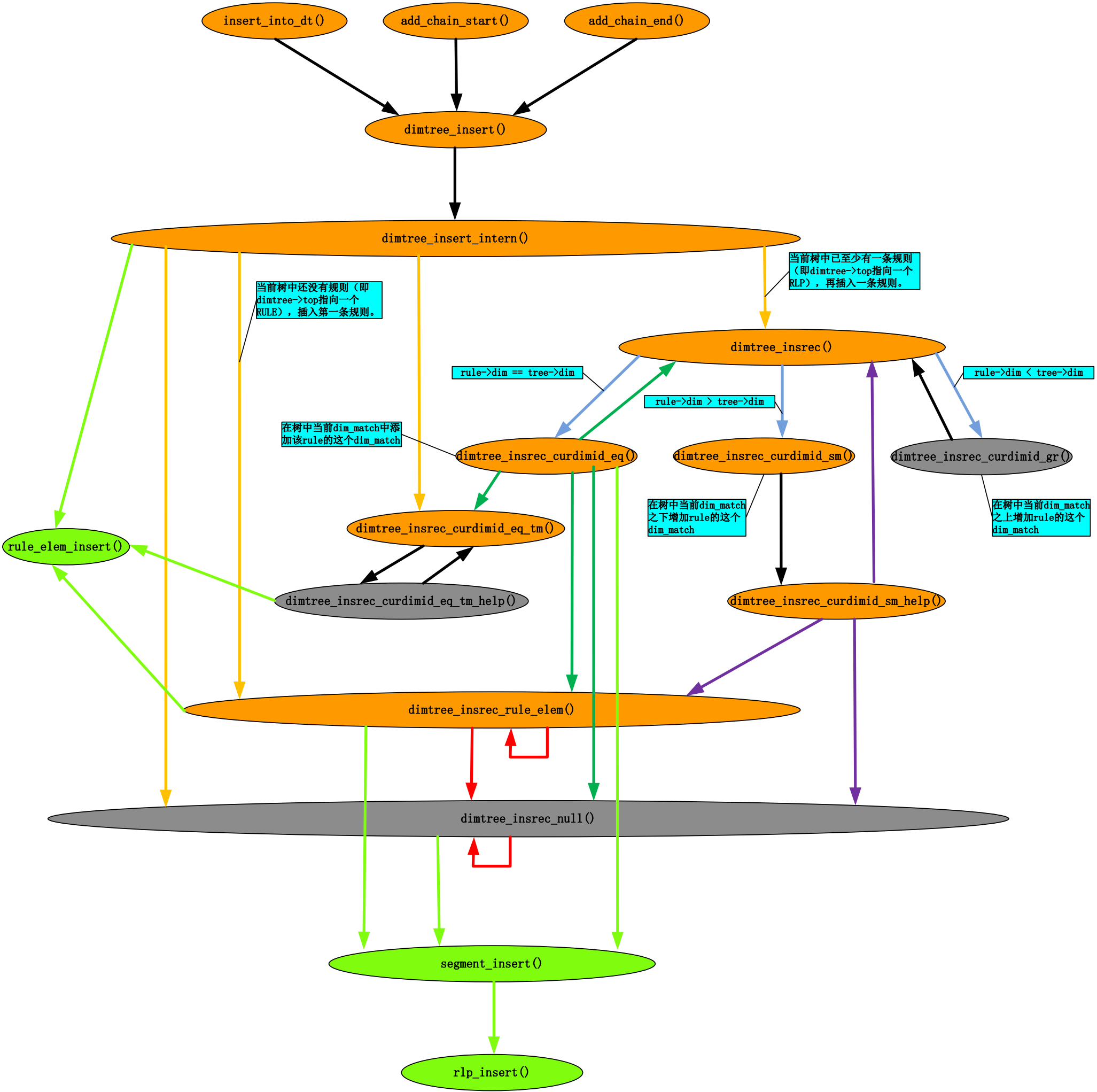
chain_rule用于应用层与内核层传递规则，并且与相应match和target关联起来

NF-hipac 是如何将规则组织成树形结构的





NF-hipac 添加规则的执行流程图



用户空间与内核的接口

procfs (/proc 文件系统)

此接口允许内核以文件的形式向用户空间输出内部信息。（《深入理解 linux 网络技术内幕》第三章、《linux 设备驱动程序》P86）

sysctl (/proc/sys 文件系统)

此接口允许用户空间读取或修改内核变量的值。用户在/proc/sys 下看到的一个文件，实际上是一个内核变量。/proc/sys 中的文件和目录都是以 ctl_table 结构定义的。该结构的注册和注销是通过 kernel/sysctl.c 中定义的 register_sysctl_table 和 unregister_sysctl_table 函数完成的。（《深入理解 linux 网络技术内幕》第三章）

sysfs (/sys 文件系统)

此接口以非常干净而有组织的方式输出很多信息，sysfs 提供了 kobject 对象层次结构的视图，帮助用户以一个简单文件系统的方式来观察系统中各种设备的拓扑结构，它代替了先前处于/proc 下的设备相关文件。它与 kobject 子系统紧密地结合在一起，因此内核开发者不需要直接使用它，而是内核的各个子系统使用它。用户要想使用 sysfs 读取和设置内核参数，仅需装载 sysfs 就可以通过文件操作应用来读取和设置内核通过 sysfs 开放给用户的各个参数。模块作为一个 kobject 也被出口到 sysfs，模块参数则是作为模块属性出口的，内核实现者为模块的使用提供了更灵活的方式，允许用户设置模块参数在 sysfs 的可见性并允许用户在编写模块时设置这些参数在 sysfs 下的访问权限，然后用户就可以通过 sysfs 来查看和设置模块参数，从而使得用户能在模块运行时控制模块行为。

对于模块而言，声明为 static 的变量都可以通过命令行来设置，但要想在 sysfs 下可见，必须通过宏 module_param 来显式声明，该宏有三个参数，第一个为参数名，即已经定义的变量名，第二个参数则为变量类型，可用的类型有 byte, short, ushort, int, uint, long, ulong, charp 和 bool 或 invbool, 分别对应于 c 类型 char, short, unsigned short, int, unsigned int, long, unsigned long, char * 和 int，用户也可以自定义类型 XXX（如果用户自己定义了 param_get_XXX，param_set_XXX 和 param_check_XXX）。该宏的第三个参数用于指定访问权限，如果为 0，该参数将不出现在 sysfs 文件系统中，允许的访问权限为 S_IRUSR， S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH 和 S_IWOTH 的组合，它们分别对应于用户读，用户写，用户组读，用户组写，其他用户读和其他用户写，因此用文件的访问权限设置是一致的。

（《linux 设备驱动程序》第十四章、《linux 内核设计与实现》第十七章）

ioctl (文件或套接字系统调用)

ioctl（输入/输出控制）系统调用操作的对象是一个文件，通常用于实现特殊设备所需但标准文件系统没有提供的操作。也可以将 socket 系统调用返回的套接字描述符传给 ioctl，而这也是网络代码使用 ioctl 的方式。此接口由老一代命令所用，如 ifconfig 和 route 等。（《深入理解 linux 网络技术内幕》第三章、《linux 设备驱动程序》第六章）

socket (套接字系统调用)

打开一个网络 socket 后可以使用 set/getsockopt(2)可实现用户空间与内核的通信，本质和 ioctl 差不多，区别在于 set/getsockopt 不用新建设备，直接利用系统已有的 socket 类型就可以进行，可用 setsockopt 函数向内核写数据，用 getsockopt 向内核读数据。（iptables 使用该接口与内核通信）

netlink (套接字系统调用)

这是网络应用程序与内核通信最新的首选机制。iproute2 包中的大多数命令都使用此接口。netlink 相对于其它用户--内核接口的有点之一，就是内核可以启动传输，而不只是仅限于相应用户空间请求而返回信息。netlink 是一种在内核与用户应用间进行双向数据传输的非常好的方式，用户态应用使用标准的 socket API（socket(), bind(), sendmsg(), recvmsg() 和 close()）就可以使用 netlink 提供的强大功能。内核态则需要在 linux/netlink.h 头文件中增加新的 netlink 协议，并通过 netlink_kernel_create()为新增加的 netlink 协议创建处理函数。netlink 的功能之一是传送单播和多播消息，目的终端点地址可以是一个进程的 PID、一个多播群组 ID 或两者的结合，而特殊 PID 值 0 代表的就是内核。（NFhipac 使用该接口与内核通信）（《深入理解 linux 网络技术内幕》第三章）

debugfs (调试内核文件系统)

relayfs (快速转发数据文件系统)

用户与内核接口的对比

通信接口	通信方式	限制	备注
procfs (/proc 文件系统)	可使用标准操作文件命令（cat、echo）等通过该文件系统与内核进行通信	与内核通信数据不能大于一页内存。使用 seq_file 接口后可以从内核输出大于 1 页内存数据	
sysctl (/proc/sys 文件系统)	可使用标准操作文件命令（cat、echo）等通过该文件系统与内核进行通信	其文件关联的必须是一个简单的内核变量或数据结构	
sysfs (/sys 文件系统)	可使用标准操作文件命令（cat、echo）等通过该文件系统与内核进行通信	它与 kobject 子系统紧密地结合在一起，因此内核开发者不需要直接使用它，而是内核的各个子系统使用它	由于模块作为一个 kobject 被出口到 sysfs，所以 sysfs 可用于操作模块参数
ioctl (文件或网络套接字系统调用)	需要开发单独命令使用该系统调用与内核进行通信。	必须建立一个设备文件或用在网络套接字上	
socket (网络套接字系统调用)	需要开发单独命令使用该系统调用与内核进行通信。	只能用在网络套接字上	
netlink (网络套接字系统调用)	需要开发单独命令使用该系统调用与内	只能用在网络套接字上	

	核进行通信。		
--	--------	--	--

conntrack 连接管理

netlink 介绍

- 1
- netlink_kernel_create(int protocol, void (*input)(struct sk_buff *skb), ...) 内核层用于创建 protocol 指定 netlink 协议（包含有 NETLINK_ROUTE、NETLINK_NETFILTER 等）的 socket 的接口，同时将 input 指针保存到该 socket 中（sk->netlink_rcv），并将该 socket 存储在 nl_table[]全局数组中。该 socket 用于接收用户层指定 netlink 协议的数据。当用户建立一个 protocol 指定协议的 socket，并通过该 socket 向内核发送数据时，netlink 将调用*input 指定的函数处理该数据。
- 2
- netlink_create(struct net *net, struct socket *sock, int protocol, int kern) 应用层创建一个 netlink 的 socket 时，会调用该接口将 sock 与 protocol 指定的内核 netlink 模块相关联。
- 3
- netlink_bind(struct socket *sock, struct sockaddr *addr, int addr_len) 应用层调用 bind()函数时，内核会调用该接口将 sock 存储到 nl_table[]全局数组中，位置由 addr 中的 pid 指定。
- 4
- netlink_lookup(int protocol, u32 pid) 根据 protocol 和 pid 值在 nl_table[]全局数组中查找对应的 socket。pid = 0 时查找的是内核创建的 socket，pid > 0 时查找的是用户创建的 socket。
- 5
- netlink_rcv_skb(struct sk_buff *skb, int (*cb)(struct sk_buff *, struct nlmsghdr *)) 一个封装接口，用于检查应用层传递给内核层数据的合法性，然后调用 cb 指向的回调函数处理该数据，并且对需要返回 ACK 的请求调用 netlink_ack()进行应答。它被在 netlink 上注册的其它协议使用，如 nfnetlink。
- 6
- netlink_dump(struct sock *sk) 调用 sk 上注册的回调项中 cb->dump 指向的函数，构建发往应用层的数据。如果回调函数没有数据可返回，则向应用层发送 NLMSG_DONE 结束标志，并调用回调项中 cb->done 指向的函数，然后注销 sk 上的这个回调项。
- 7
- netlink_dump_start(struct sock *ssk, struct sk_buff *skb, struct nlmsghdr *nlh, int (*dump)(), int (*done)()) 用于注册一个回调项。它调用 netlink_lookup()查找到接收端（目的端）的应用层 socket（通过 protocol 和 NETLINK_CB(skb).pid），然后在该 socket 上注册一个回调项，该回调项的 cb->dump 指向参数的*dump，cb->done 指向参数的*done。
- 8
- netlink_unicast_kernel(struct sock *sk, struct sk_buff *skb) 将数据发给 sk 指定的内核接收者，即它调用 nlk_sk(sk)-> netlink_rcv(skb)来接收该数据包。netlink_rcv 指定的函数是通过 netlink_kernel_create(protocol, *input())创建内核 socket 时 input 参数指定的函数。不同协议有不同的处理函数。
- 9
- netlink_unicast(struct sock *ssk, struct sk_buff *skb, u32 pid, int nonblock) 它调用 netlink_lookup(ssk->protocol, pid)查找目的端的 socket，如果目的端 socket 是内核创建的，则调用 netlink_unicast_kernel()将该数据发送给内核处理函数。如果目的端 socket 是应用层创建的，则将数据挂到该 socket 接收队列尾部。
- 10
- netlink_sendmsg(struct kiocb *kiocb, struct socket *sock, struct msghdr *msg, size_t len) 应用层调用 sendto()函数向内核发送数据时，则内核调用该接口。它根据 sock 绑定的协议值和数据包中指定的 pid 调用 netlink_unicast 将数据包发送给对应的 socket。
- 11
- netlink_recvmsg(struct kiocb *kiocb, struct socket *sock, struct msghdr *msg, size_t len, int flags) 应用层调用 recvfrom()函数从内核接收数据时，则内核调用该接口。它从 sock 指定的 socket 接收队列中取下一个数据包，并将该数据包数据传递给应用层。如果该 socket 有回调项，则调用 netlink_dump()调用该回调项。该回调项是之前通过 netlink_dump_start()注册的，用于返回一个请求需要的数据（在一个数据包中无法包含全部内容）。

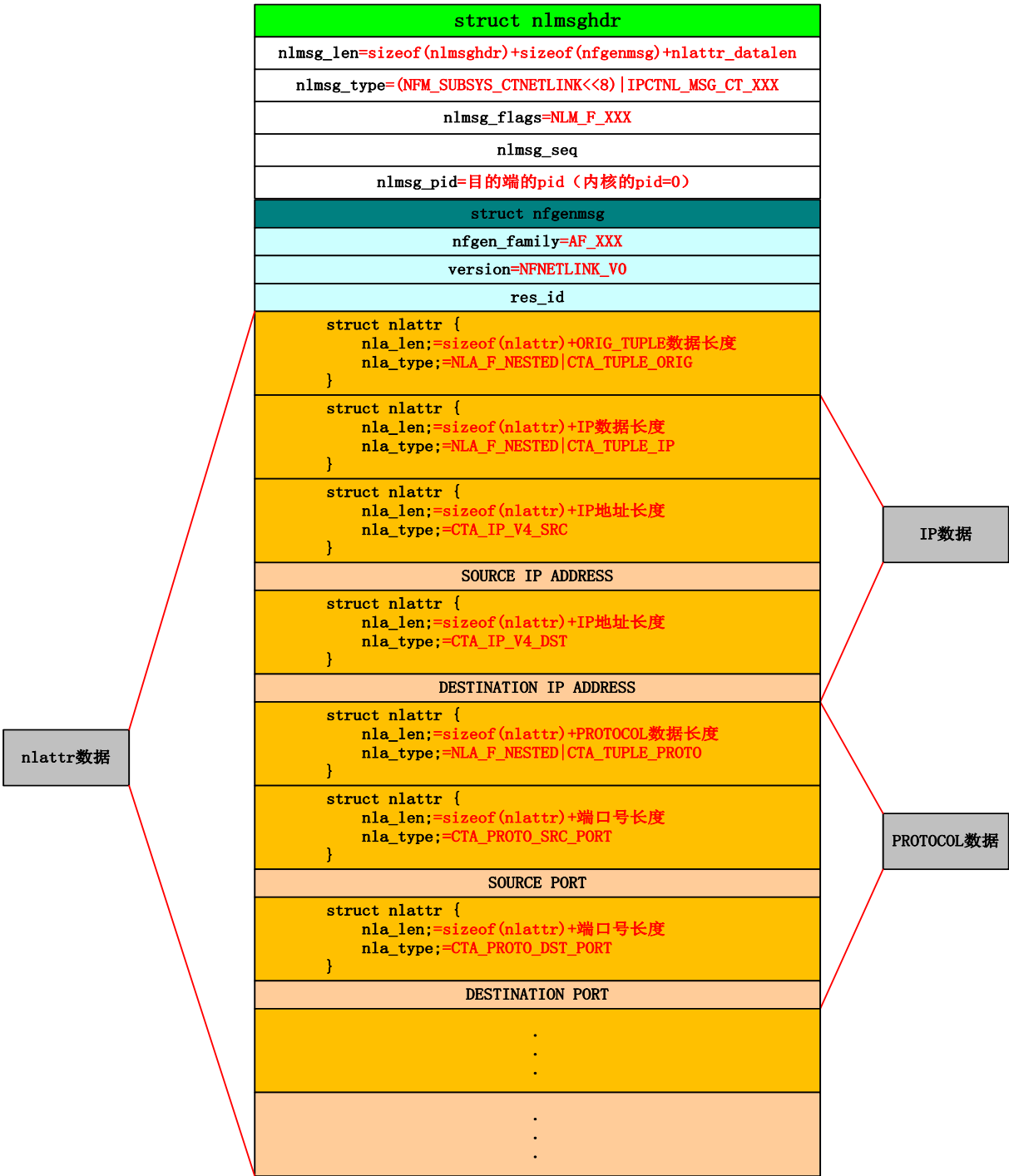
nfnetlink 介绍

- 1
- nfnetlink 是建立在 netlink 基础上的，它通过 netlink_kernel_creat()创建一个 NETLINK_NETFILTER 协议 socket，用于接收所有协议为 NETLINK_NETFILTER 的数据。它的接收函数为 nfnetlink_rcv()。
- 2
- nfnetlink_rcv(struct sk_buff *skb) 调用 netlink_rcv_skb(skb, &nfnetlink_rcv_msg)来检验数据包，并给定回调函数 nfnetlink_rcv_msg()进行后续处理。
- 3
- nfnetlink_rcv_msg(struct sk_buff *skb, struct nlmsghdr *nlh) 首先调用 ss = nfnetlink_get_subsys(nlh->nlmsg_type)从 subsys_table[]全局数组中获取子系统（通过 nfnetlink_subsys_register()函数注册的），该子系统由 netlink 数据包头部的 nlmsg_type 字段高 8 位指定。然后调用 nc = nfnetlink_find_client(nlh->nlmsg_type, ss)从该子系统中获得子项，该子项由 netlink 数据包头部的 nlmsg_type 字段低 8 位指定。最后调用子项中的处理函数处理 skb 数据。
- 4
- nfnetlink_subsys_register(const struct nfnetlink_subsystem *n) 用于注册一个 nfnetlink 子系统，该子系统被保存在 subsys_table[]数组中。
- 5
- nfnetlink 与 netlink 不同之处在于它采用了 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。

nf_conntrack_netlink 介绍

- 1
- nf_conntrack_netlink 是建立在 nfnetlink 基础上，它通过 NETLINK_NETFILTER 协议的 socket 与应用层进行通信。它调用 nfnetlink_subsys_register()在 nfnetlink 上注册了两个子系统 NFNL_SUBSYS_CTNETLINK 和 NFNL_SUBSYS_CTNETLINK_EXP。用于与连接有关的请求。
- 2
- conntrack -L [table] [options] 用户指定参数是可选的。它从内核获取所有连接（NFCT_Q_DUMP），然后与指定参数信息进行匹配，相同则打印。如果未指定参数信息，则打印全部连接信息。
- 3
- conntrack -G [table] parameters 用户最少需要指定 ORIGINAL 方向的五元组参数。内核使用该五元组构建一个 tuple，通过 nf_ct_tuplehash_to_ctrack(tuple)接口获取该连接（一个 tuple 仅对应一个连接），并将其上传给应用层。用户可以指定更多参数，这些参数的匹配在应用层进行。
- 4
- conntrack -D [table] parameters 用户指定的参数是可选的。它从内核获取所有连接（NFCT_Q_DUMP），然后与指定参数信息进行匹配，相同则向内核发送信息删除该连接，然后将其信息打印出来。如果未指定参数，则将获取的所有连接全部删除，并打印它们。
- 5
- conntrack -F [table] 向内核发送命令（NFCT_Q_FLUSH）删除全部连接。
- 6
- [table]: conntrack, expect 命令 conntrack 可以操作两张 hash 表（conntrack 和 expect），conntrack 表中的信息是连接跟踪信息，其每项数据是 nf_conn 结构。expect 表中信息是识别期待连接信息，其每项数据是 nf_conntrack_expect。

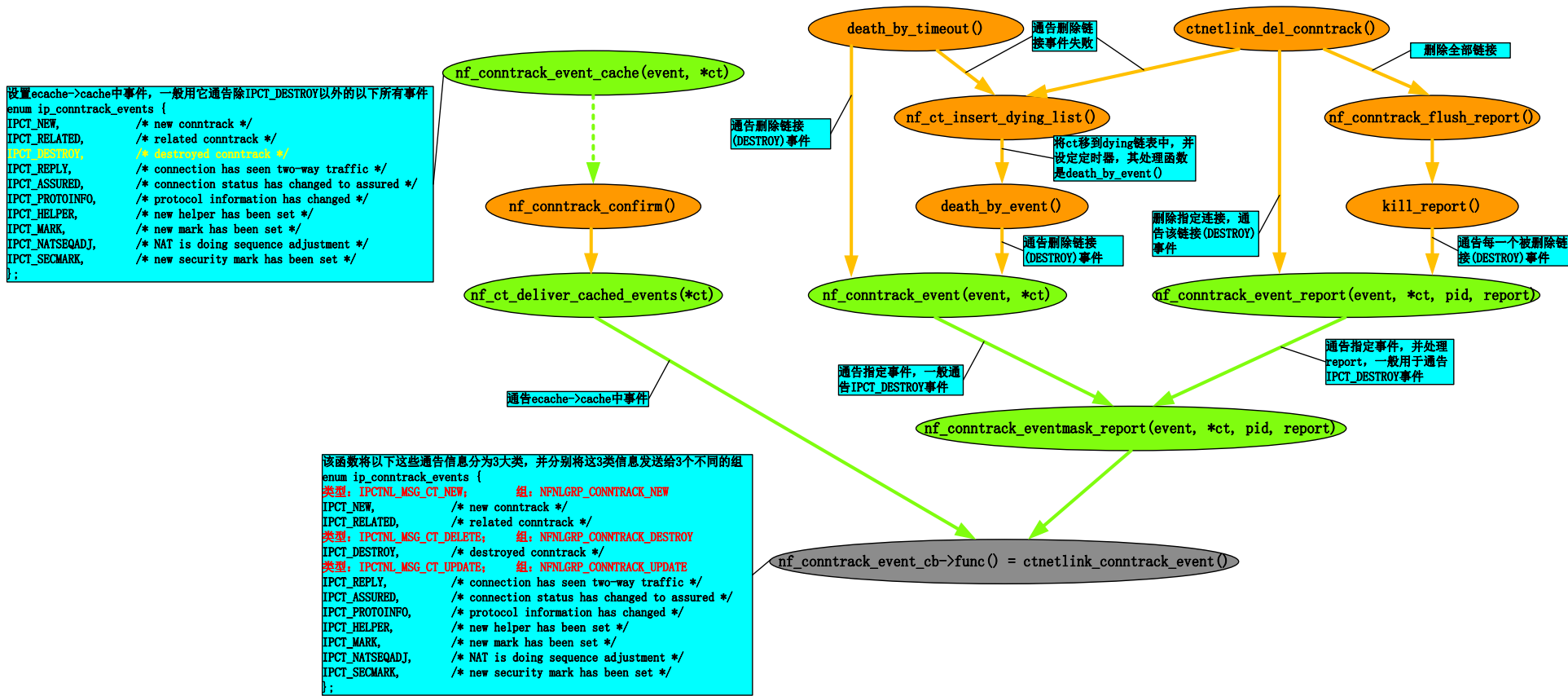
- 7
- conntrack -C [table] 获取当前连接的个数。如果 table 是 conntrack 表，则直接从/proc/sys/net/netfilter/nf_conntrack_count 文件中读取。如果 table 是 expect 表，则从内核获取所有 expect 项（NFCT_Q_DUMP），然后进行统计。
- 8
- conntrack -S 显示统计信息，它通过读取/proc/net/stat/nf_conntrack 文件来获取统计信息。
- 9
- 应用层与内核层通信数据的形式



10 conntrack 应用层创建 ct_handle 和注册回调函数结构图

12 conntrack -E [table] [options] 获取内核通告数据，它可指定获取 NEW、UPDATE、DESTROY 三类事件的数据。

- 12.1 应用层首先建立一个 nfnetlink 的 socket，并将该 socket 绑定到指定的事件（NEW、UPDATE、DESTROY）组中。其次注册一个回调函数 event_cb()，它根据用户指定的 options 对数据进行过滤，并将匹配的数据打印出来。最后调用 nfctCatch()等待内核发送的通告数据，并调用回调函数进行处理。
- 12.2 内核层通告 nf_conntrack_register_notifier()注册了一个连接跟踪通告函数 ctnetlink_conntrack_event()，通过全局指针 nf_conntrack_event_cb 指定。连接跟踪代码通过接口 nf_conntrack_event()、nf_conntrack_event_report()、nf_ct_deliver_cached_events()通告 ct 所发生的事件。这些接口最终会调用 ctnetlink_conntrack_event()将不同事件的 ct 归为不同事件组（NEW、UPDATE、DESTROY），并将该 ct 发送给绑定在这些事件组的 socket。
- 12.3 内核中通告连接事件的点



IPv6

IPv6 地址间的关系

1. 接口 MAC 地址-----FE80::[EUI-64]/64（**链路本地地址根据接口 MAC 地址自动生成**）-----FF02::0001:FF[24bit 映射]/104（被请求节点组播地址）-----33:33:FF:[24bit 映射]（组播 MAC 地址）
2. 2000::/3（全球单播地址）-----FF20::0001:FF[24bit 映射]/104（被请求节点组播地址）-----33:33:FF:[24bit 映射]（组播 MAC 地址）
3. FC00::/7（全球唯一本地地址）-----FF20::0001:FF[24bit 映射]/104（被请求节点组播地址）-----33:33:FF:[24bit 映射]（组播 MAC 地址）
4. FF00::/8（众所周知组播地址）-----33:33:[32bit 映射]（组播 MAC 地址）

备注

以太网帧的限制

- 1
- 以太网对传输的数据帧大小被限制为 `64 < len < 1518`，即数据帧最大不超过 `1518` 字节，最小不得小于 `64` 字节。由于以太网协议头部长度是 `14` 字节，以太网校验和（CRC）长度是 `4` 字节，所以以太网内部包含数据大小被限制为 `46 < len < 1500`。所以我们发送的数据应该在这个范围内。
- 2
- 当我们发送的以太网帧小于 `64` 字节（即以太网内部数据小于 `46` 字节）时，以太网设备（网卡）驱动程序会自动将发送的数据填充到 `64` 字节长，并且以太网校验和以填充后数据做校验。接收端收到的以太网帧长度也将是 `64`（`skb->len=64`）字节。当上传给 `3` 层协议时，数据包长度将是 `46`（`skb->len=46`）字节（以太网帧长度 `64` 字节 - 以太网协议头部长度 `14` 字节 - 以太网校验和长度 `4` 字节 = 数据长度 `46` 字节）。但这些数据中包含填充数据，所以我们不能以 `skb->len` 长度判断有效数据长度，应该以三层协议头部中 `len` 字段（如果有的话）表示的长度来确定有效数据长度。
- 3
- 当我们发送的以太网帧大于 `64` 字节，会如何处理？？？？？？？？？？

连接跟踪

- 1
- `expect` 结构并不增加主连接的引用计数，它只是被挂载在主连接的 `helper` 扩展的 `help->expectations` 链表中。不增加引用计数的原因是，如果 `expect` 关联的连接被删除，则该 `expect` 也会被删除。
- 2
- `ct->nat.bysource` 被加入 `nat_bysource[]`链表中，并不会增加 `ct` 的引用计数。
- 3
- `nat` 模块的注销，是先将 `HOOK` 函数注销掉（在 `iptable_nat` 模块中），这样就保证新建的连接不会再进入 `nf_nat_setup_info()`函数和 `nf_nat_fn()`函数中创建 `nat` 扩展。然后再搜索所有的 `ct`（在 `net->ct.hash` 和 `net->ct.unconfirmed` 链表中搜索），将它的 `nat` 扩展字段全部置为 `0`，并清除 `ct->status` 上的 `NAT` 标记，这样就去掉了 `ct` 与 `nat` 模块之间的关联。最后释放 `nat` 资源。
- 4
- `helper` 结构的注销（`nf_conntrack_helper_unregister`）是先将 `helper` 从全局 `HASH` 表（`nf_ct_helper_hash[]`）删除，然后调用 `synchronize_rcu()`函数等待所有读者都完成读端临界区后（这时已不会有读者再从 `nf_ct_helper_hash` 表中找到该 `helper` 结构），再搜索所有与该 `helper` 相关联的 `ct` 和 `expect`，解除它们之间的关联（通过 `nf_conntrack_lock` 锁进行保护,防止搜索过程中有新的 `ct` 或 `expect` 被插入），即完成了 `helper` 结构的注销。所以读者只需使用 `rcu_read_lock()`加锁,同时将找到的 `helper` 与 `ct` 或 `expect` 关联，并在调用 `rcu_read_unlock()`解锁之前保证 `ct` 和 `expect` 已被加入它们的全局链表（`ct` 要么在 `hash` 链表中，要么在 `unconfirmed` 链表中）中（因为 `nf_conntrack_helper_unregister()`会调用 `synchronize_rcu()`等待所有 `cpu` 都完成 `rcu_read` 临界区操作后，才执行将 `helper` 与 `ct` 和 `expect` 解除关联的操作的）。

性能优化

- 1
- 局部性原理（对硬件和软件系统的设计都有着极大的影响）
- 1.1
- 时间局部性：在一个具有良好时间局部性的程序中，被引用过一次的存储器位置，很可能在不远的将来再被程序多次引用。
- 1.2
- 空间局部性：在一个具有良好空间局部性的程序中，被引用过一次的存储器位置，很可能在不远的将来被程序引用其附近的一个存储器位置。
- 1.3
- 程序的目标就是利用时间局部性，使得频繁使用的字从 `L1` 中取出，还要利用空间局部性，使得尽可能多的字从一个 `L1` 高速缓存行中访问到。推荐下列技术：
- 1.3.1
- 将你的注意力集中在内部循环上，大部分计算和存储器访问都发生在这里。
- 1.3.2
- 通过按照数据对象存储在存储器中的顺序来读数据，从而使得你程序中的空间局部性最大。
- 1.3.3
- 一旦从存储器中读入了一个数据对象，就尽可能多的使用它，从而使得你程序中的时间局部性最大。
- 1.3.4
- 记住，不命中率只是确定你代码性能的一个因素（虽然是重要的）。存储器访问数量也扮演着重要角色，有时需要在两者之间做一下折中。

tcpdump

分别在 `netif_receive_skb()`和 `dev_hard_start_xmit()`中抓取数据包（通过在 `packet_all` 链注册一个接口抓取数据包）