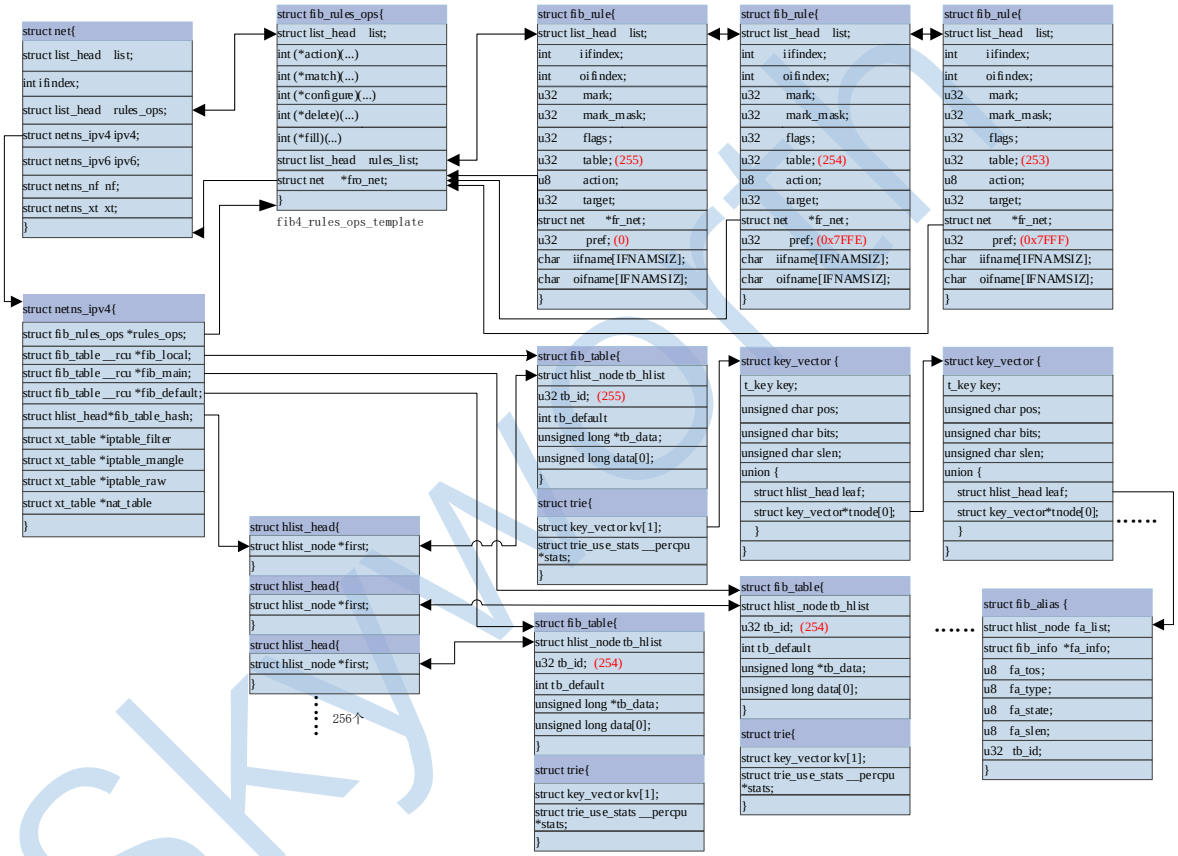
路由表

--陈绪勇
--2018.11.26

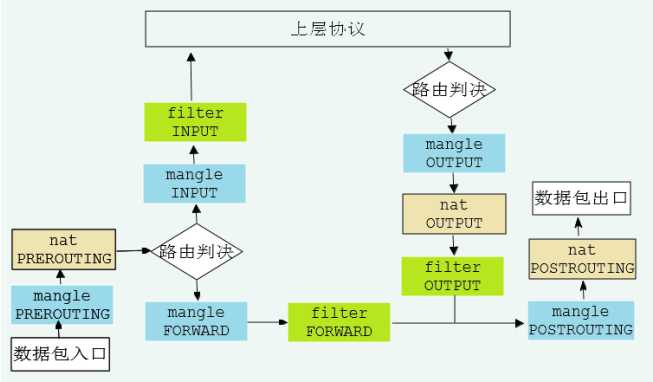
fib_table:记录 IP 转发信息的索引表,
fib_lookup:转发表检索过程。
路由表的作用，这里就不说了，如果不清楚，建议多了解了解相关的东西。
注意这里是基于 linux-linaro-lsk-v4.1 原生态内核来研究路由表，中兴微方案基于此内核，此内核的路由算法是基于 LC-Trie 树算法，而不是 hash 算法。
另外内核配置，CONFIG_IP_MULTIPLE_TABLES 开启，CONFIG_IP_MROUTE_MULTIPLE_TABLES 关闭。
下面是一张总的图，自己总结的，如果有纰漏，还请指出。



一． 路由信息查询

1.路由查询函数调用关系

首先从 netfilter 说起，下面是 netfilter 大概框架图:



数据包经过 PRE_ROUTING 后，会调用 ip_rcv_finish()，此函数作用是进行路由判决，决定数据包是发给上层，还是转发出去。
NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING, NULL, skb,dev, NULL,ip_rcv_finish);
设备向外发的路由判断这里就不讲述了。

```
static int ip_rcv_finish(struct sock *sk, struct sk_buff *skb)
{
	const struct iphdr *iph = ip_hdr(skb);
	struct rtable *rt;

	if (sysctl_ip_early_demux && !skb_dst(skb) && !skb->sk) {
		const struct net_protocol *ipprot;
		int protocol = iph->protocol;

		ipprot = rcu_dereference(inet_protos[protocol]);
		if (ipprot && ipprot->early_demux) {
			ipprot->early_demux(skb);
			/* must reload iph, skb->head might have changed */
			iph = ip_hdr(skb);
		}
	}
	/*
	 *	Initialise the virtual path cache for the packet. It describes
	 *	how the packet travels inside Linux networking.
	 */
	if (!skb_dst(skb)) {
		int err = ip_route_input_noref(skb, iph->daddr, iph->saddr,
					       iph->tos, skb->dev);
		if (unlikely(err)) {
			if (err == -EXDEV)
				NET_INC_STATS_BH(dev_net(skb->dev),
						 LINUX_MIB_IPRPFILTER);
			goto drop;
		}
	}

#ifdef CONFIG_IP_ROUTE_CLASSID
	if (unlikely(skb_dst(skb)->tclassid)) {
		struct ip_rt_acct *st = this_cpu_ptr(ip_rt_acct);
		u32 idx = skb_dst(skb)->tclassid;
		st[idx&0xFF].o_packets++;
		st[idx&0xFF].o_bytes += skb->len;
		st[(idx>>16)&0xFF].i_packets++;
		st[(idx>>16)&0xFF].i_bytes += skb->len;
	}
#endif

	if (iph->ihl > 5 && ip_rcv_options(skb))
		goto drop;

	rt = skb_rtable(skb);
	if (rt->rt_type == RTN_MULTICAST) {
		IP_UPD_PO_STATS_BH(dev_net(rt->dst.dev), IPSTATS_MIB_INMCAST,
				skb->len);
```

```
    } else if (rt->rt_type == RTN_BROADCAST)
        IP_UPD_PO_STATS_BH(dev_net(rt->dst.dev), IPSTATS_MIB_INBCAST,
                skb->len);

    return dst_input(skb);

drop:
    kfree_skb(skb);
    return NET_RX_DROP;
}
```

ip_rcv_finish()--->ip_route_input_noref()--->ip_route_input_slow
重点在于 ip_route_input_slow 函数:

```
ip_route_input_slow()
{
        res.fi = NULL;
        fl4.flowi4_oif = 0;
        fl4.flowi4_iif = dev->ifindex;
        fl4.flowi4_mark = skb->mark;
        fl4.flowi4_tos = tos;
        fl4.flowi4_scope = RT_SCOPE_UNIVERSE;
        fl4.daddr = daddr;
        fl4.saddr = saddr;
        err = fib_lookup(net, &fl4, &res);//查询路由。
        下面根据 res 来进行路由判决，即判断是发往本地(接收给应用层)，还是转发出去(发给 wan 侧设备或者 lan 侧设备)。
        如果 res.type == RTN_LOCAL 则执行下面代码:
        {
                rth->dst.input= ip_local_deliver;    //发往本地。
                rth->dst.output= ip_rt_bug;
        }
        如果是 res.type == RTN_UNICAST
        则执行下面代码:
        {
                ip_mkroute_input(skb, &res, &fl4, in_dev, daddr, saddr, tos);    //转发出去。
        }
}

ip_mkroute_input---> __mkroute_input
__mkroute_input 会调用:
{
        ......
        rth->dst.input = ip_forward;
        rth->dst.output = ip_output;
        ......
        skb_dst_set(skb, &rth->dst);
}

注意 skb_dst_set 函数:
static inline void skb_dst_set(struct sk_buff *skb, struct dst_entry *dst)
{
    skb->_skb_refdst = (unsigned long)dst;
}
```

如果网本地发送，则后面会调用 ip_local_deliver 函数。

2.fib_lookup 函数

fib_lookup()主要是进行路由查询，将报文的头部信息和路由表进行对比，如果匹配到后，将对应的路由放到 res 里面，后面根据 res 来决定数据包的去向(是收进来发给上层应用，还是转发出去)。

下面我们来看是如何进行路由查询的。

```
static inline int fib_lookup(struct net *net, struct flowi4 *flp,struct fib_result *res)
{
    struct fib_table *tb;
    int err;
    if (net->ipv4.fib_has_custom_rules)
    {
        return __fib_lookup(net, flp, res);
```

```
        }
        rcu_read_lock();
        res->tclassid = 0;
        for (err = 0; !err; err = -ENETUNREACH) {
                tb = rcu_dereference_rtnl(net->ipv4.fib_main);
                if(tb)
                if (tb && !fib_table_lookup(tb, flp, res, FIB_LOOKUP_NOREF))
                        break;
                tb = rcu_dereference_rtnl(net->ipv4.fib_default);
                if(tb)
                if (tb && !fib_table_lookup(tb, flp, res, FIB_LOOKUP_NOREF))
                        break;
        }
        rcu_read_unlock();
        return err;
}
```

3.__fib_lookup

这里首先要谈两个：fib4_rules_ops_template 和 ipmr_rules_ops_template，

他们都是通过 **fib_rules_register()**函数注册到 net.rules_ops 的 list 中，但是 CONFIG_IP_MROUTE_MULTIPLE_TABLES 关闭，所以 ipmr_rules_ops_template 就不会注册。因此 struct fib_rules_ops 在内核中只剩下一个：fib4_rules_ops_template，事情就简单很多了。

net.rules_ops 的 list 元素是一个双向链表，fib4_rules_ops_template 注册到了 net.rules_ops 的 list 中，同时执行操作：net->ipv4.rules_ops = fib4_rules_ops_template，这里 net->ipv4.rules_ops 和 net.rules_ops 两个不要混淆。

fib_rules_register()我们后面会讨论到。

```
int __fib_lookup(struct net *net, struct flowi4 *flp, struct fib_result *res)
{
        struct fib_lookup_arg arg = {
                .result = res,
                .flags = FIB_LOOKUP_NOREF,
        };
        int err;
        err = fib_rules_lookup(net->ipv4.rules_ops, flowi4_to_flowi(flp), 0, &arg);
#ifdef CONFIG_IP_ROUTE_CLASSID
        if (arg.rule)
                res->tclassid = ((struct fib4_rule *)arg.rule)->tclassid;
        else
                res->tclassid = 0;
#endif

        if (err == -ESRCH)
                err = -ENETUNREACH;

        return err;
}


int fib_rules_lookup(struct fib_rules_ops *ops, struct flowi *fl,
                     int flags, struct fib_lookup_arg *arg)
{
        struct fib_rule *rule;
        int err;

        rcu_read_lock();

          //list_for_each_entry_rcu 访问 net->ipv4.rules_ops 的 rules_list 的双向链表
          /*net->ipv4.rules_ops->rules_list 双向链表是从那里来的？？？就是 fib4_rules_ops_template.rules_list 怎么来的请看后面 fib4_rules_ops_template 的注册*/
        list_for_each_entry_rcu(rule, &ops->rules_list, list) {
jumped:
                if (!fib_rule_match(rule, ops, fl, flags))
```

```
                continue;

        if (rule->action == FR_ACT_GOTO) {
                struct fib_rule *target;

                target = rcu_dereference(rule->ctarget);
                if (target == NULL) {
                        continue;
                } else {
                        rule = target;
                        goto jumped;
                }
        } else if (rule->action == FR_ACT_NOP)
                continue;
        else
                err = ops->action(rule, fl, flags, arg);

        if (!err && ops->suppress && ops->suppress(rule, arg))
                continue;

        if (err != -EAGAIN) {
                if ((arg->flags & FIB_LOOKUP_NOREF) ||
                        likely(atomic_inc_not_zero(&rule->refcnt))) {
                        arg->rule = rule;
                        goto out;
                }
                break;
        }
    }

    err = -ESRCH;
out:
    rcu_read_unlock();

    return err;
}
```

fib4_rules_ops_template 的 match 对应 fib4_rule_match（）函数

```
static int fib4_rule_match(struct fib_rule *rule, struct flowi *fl, int flags)
{
        struct fib4_rule *r = (struct fib4_rule *) rule;
        struct flowi4 *fl4 = &fl->u.ip4;
        __be32 daddr = fl4->daddr;
        __be32 saddr = fl4->saddr;

        if (((saddr ^ r->src) & r->srcmask) ||
            ((daddr ^ r->dst) & r->dstmask))
                return 0;

        if (r->tos && (r->tos != fl4->flowi4_tos))
                return 0;

        return 1;
}

static int fib_rule_match(struct fib_rule *rule, struct fib_rules_ops *ops,
                          struct flowi *fl, int flags)
{
    int ret = 0;

    if (rule->iifindex && (rule->iifindex != fl->flowi_iif))
        goto out;

    if (rule->oifindex && (rule->oifindex != fl->flowi_oif))
        goto out;
```

```
    if ((rule->mark ^ fl->flowi_mark) & rule->mark_mask)
        goto out;

    //ops->match 就是 fib4_rules_ops_template 对应的 fib4_rule_match
    ret = ops->match(rule, fl, flags);
out:
    return (rule->flags & FIB_RULE_INVERT) ? !ret : ret;
}
```

4.fib_table_lookup

fib_table_lookup()是路由表查询的关键，中兴微内核路由采用 LC-trie 算法。这里就是 LC-trie 路由算法的查询部分，如果感兴趣可以详细研究。

```
int fib_table_lookup(struct fib_table *tb, const struct flowi4 *flp,
                     struct fib_result *res, int fib_flags)
{
    struct trie *t = (struct trie *) tb->tb_data;
#ifdef CONFIG_IP_FIB_TRIE_STATS
    struct trie_use_stats __percpu *stats = t->stats;
#endif
    const t_key key = ntohl(flp->daddr);
    struct key_vector *n, *pn;
    struct fib_alias *fa;
    unsigned long index;
    t_key cindex;

    pn = t->kv;
    cindex = 0;

    //相当于 n=pn;
    n = get_child_rcu(pn, cindex);
    if (!n)
        return -EAGAIN;

#ifdef CONFIG_IP_FIB_TRIE_STATS
    this_cpu_inc(stats->gets);
#endif

    /* Step 1: Travel to the longest prefix match in the trie */
    for (;;) {

        //即(key ^ n->key) >> n->pos
        index = get_cindex(key, n);

        /* This bit of code is a bit tricky but it combines multiple
         * checks into a single check.  The prefix consists of the
         * prefix plus zeros for the "bits" in the prefix. The index
         * is the difference between the key and this value.  From
         * this we can actually derive several pieces of data.
         *   if (index >= (1ul << bits))
         *     we have a mismatch in skip bits and failed
         *   else
         *     we know the value is cindex
         *
         * This check is safe even if bits == KEYLENGTH due to the
         * fact that we can only allocate a node with 32 bits if a
         * long is greater than 32 bits.
         */
        if (index >= (1ul << n->bits))
            break;

        /* we have found a leaf. Prefixes have already been compared */
```

```
		if (IS_LEAF(n))
			goto found;

		/* only record pn and cindex if we are going to be chopping
		 * bits later.  Otherwise we are just wasting cycles.
		 */
		if (n->slen > n->pos) {
			pn = n;
			cindex = index;
		}

		n = get_child_rcu(n, index);
		if (unlikely(!n))
			goto backtrace;
	}

	/* Step 2: Sort out leaves and begin backtracing for longest prefix */
	for (;;) {
		/* record the pointer where our next node pointer is stored */
		struct key_vector __rcu **cptr = n->tnode;

		/* This test verifies that none of the bits that differ
		 * between the key and the prefix exist in the region of
		 * the lsb and higher in the prefix.
		 */
		if (unlikely(prefix_mismatch(key, n)) || (n->slen == n->pos))
			goto backtrace;

		/* exit out and process leaf */
		if (unlikely(IS_LEAF(n)))
			break;

		/* Don't bother recording parent info.  Since we are in
		 * prefix match mode we will have to come back to wherever
		 * we started this traversal anyway
		 */

		while ((n = rcu_dereference(*cptr)) == NULL) {
backtrace:
#ifdef CONFIG_IP_FIB_TRIE_STATS
			if (!n)
				this_cpu_inc(stats->null_node_hit);
#endif
			/* If we are at cindex 0 there are no more bits for
			 * us to strip at this level so we must ascend back
			 * up one level to see if there are any more bits to
			 * be stripped there.
			 */
			while (!cindex) {
				t_key pkey = pn->key;

				/* If we don't have a parent then there is
				 * nothing for us to do as we do not have any
				 * further nodes to parse.
				 */
				if (IS_TRIE(pn))
					return -EAGAIN;
#ifdef CONFIG_IP_FIB_TRIE_STATS
				this_cpu_inc(stats->backtrack);
#endif
				/* Get Child's index */
				pn = node_parent_rcu(pn);
				cindex = get_index(pkey, pn);
			}

			/* strip the least significant bit from the cindex */
			cindex &= cindex - 1;
```

```
                /* grab pointer for next child node */
                cptr = &pn->tnode[cindex];
            }
        }

found:
        /* this line carries forward the xor from earlier in the function */
        index = key ^ n->key;

        /* Step 3: Process the leaf, if that fails fall back to backtracing */
        hlist_for_each_entry_rcu(fa, &n->leaf, fa_list) {
            struct fib_info *fi = fa->fa_info;
            int nhsel, err;

            if ((index >= (1ul << fa->fa_slen)) &&
                ((BITS_PER_LONG > KEYLENGTH) || (fa->fa_slen != KEYLENGTH)))
                continue;
            if (fa->fa_tos && fa->fa_tos != flp->flowi4_tos)
                continue;
            if (fi->fib_dead)
                continue;
            if (fa->fa_info->fib_scope < flp->flowi4_scope)
                continue;
            fib_alias_accessed(fa);
            err = fib_props[fa->fa_type].error;
            if (unlikely(err < 0)) {
#ifdef CONFIG_IP_FIB_TRIE_STATS
                this_cpu_inc(stats->semantic_match_passed);
#endif
                return err;
            }
            if (fi->fib_flags & RTNH_F_DEAD)
                continue;
            for (nhsel = 0; nhsel < fi->fib_nhs; nhsel++) {
                const struct fib_nh *nh = &fi->fib_nh[nhsel];

                if (nh->nh_flags & RTNH_F_DEAD)
                    continue;
                if (flp->flowi4_oif && flp->flowi4_oif != nh->nh_oif)
                    continue;

                if (!(fib_flags & FIB_LOOKUP_NOREF))
                    atomic_inc(&fi->fib_clntref);

                res->prefixlen = KEYLENGTH - fa->fa_slen;
                res->nh_sel = nhsel;
                res->type = fa->fa_type;
                res->scope = fi->fib_scope;
                res->fi = fi;
                res->table = tb;
                res->fa_head = &n->leaf;
#ifdef CONFIG_IP_FIB_TRIE_STATS
                this_cpu_inc(stats->semantic_match_passed);
#endif
                return err;
            }
        }
#ifdef CONFIG_IP_FIB_TRIE_STATS
        this_cpu_inc(stats->semantic_match_miss);
#endif
        goto backtrace;
}
```

fib_table_lookup()函数处理完，路由查询信息就查到了。fib_lookup 就处理完毕返回 res，接下来根据 res 来做路由判决，前面已经描述了。

二．路由表的初始化

这部分主要讲述路由表的初始化，重点是 fib4_rules_ops_template 是如何注册的，它的注册过程创建了路由表。

1.ip_rt_init

```
int __init ip_rt_init(void)
{
        ip_fib_init();
}

void __init ip_fib_init(void)
{
    //下面四个注册函数都很重要。
    rtnl_register(PF_INET, RTM_NEWROUTE, inet_rtm_newroute, NULL, NULL);
    rtnl_register(PF_INET, RTM_DELROUTE, inet_rtm_delroute, NULL, NULL);
    rtnl_register(PF_INET, RTM_GETROUTE, NULL, inet_dump_fib, NULL);
    register_pernet_subsys(&fib_net_ops);
    register_netdevice_notifier(&fib_netdev_notifier);
    register_inetaddr_notifier(&fib_inetaddr_notifier);
    fib_trie_init();
}
```

下面讲解：register_pernet_subsys(&fib_net_ops); 其余三个注册函数在第三节中讲解。
其核心东西就是：
list_add_tail(&ops->list, list);
==>
list_add_tail(&fib_net_ops.list, pernet_list);

```
static LIST_HEAD(pernet_list);
static struct list_head *first_device = &pernet_list;


int register_pernet_subsys(struct pernet_operations *ops)
{
    int error;
    mutex_lock(&net_mutex);
    error =  register_pernet_operations(first_device, ops);
    mutex_unlock(&net_mutex);
    return error;
}

static int register_pernet_operations(struct list_head *list,
                      struct pernet_operations *ops)
{
        ......
    error = __register_pernet_operations(list, ops);
    ......
}
 static int __register_pernet_operations(struct list_head *list,
                                struct pernet_operations *ops)
{
    struct net *net;
    int error;
    LIST_HEAD(net_exit_list);

    list_add_tail(&ops->list, list);
    if (ops->init || (ops->id && ops->size)) {
        for_each_net(net) {
            error = ops_init(ops, net);
            if (error)
                goto out_undo;
```

```
                list_add_tail(&net->exit_list, &net_exit_list);
            }
        }
        return 0;

out_undo:
        /* If I have an error cleanup all namespaces I initialized */
        list_del(&ops->list);
        ops_exit_list(ops, &net_exit_list);
        ops_free_list(ops, &net_exit_list);
        return error;
}
```

2.fib_net_ops

```
static struct pernet_operations fib_net_ops = {
        .init = fib_net_init,
        .exit = fib_net_exit,
};

static int __net_init fib_net_init(struct net *net)
{
        int error;

#ifdef CONFIG_IP_ROUTE_CLASSID
        net->ipv4.fib_num_tclassid_users = 0;
#endif
        error = ip_fib_net_init(net);
        if (error < 0)
                goto out;
        error = nl_fib_lookup_init(net); // 没研究，先不管吧。
        if (error < 0)
                goto out_nlfl;
        error = fib_proc_init(net);
        if (error < 0)
                goto out_proc;
out:
        return error;

out_proc:
        nl_fib_lookup_exit(net);
out_nlfl:
        ip_fib_net_exit(net);
        goto out;
}
```

3.ip_fib_net_init

```
#ifdef CONFIG_IP_MULTIPLE_TABLES
#define FIB_TABLE_HASHSZ 256
#else
#define FIB_TABLE_HASHSZ 2
#endif
```

关键点：初始化 fib_table_hash，FIB_TABLE_HASHSZ 值位 256，刚好对应 256 个路由表。可通过 ip rule add/del 添加删除路由规则。

```
net->ipv4.fib_table_hash = kzalloc(size, GFP_KERNEL);
static int __net_init ip_fib_net_init(struct net *net)
```

```
{
	int err;
	size_t size = sizeof(struct hlist_head) * FIB_TABLE_HASHSZ;

	/* Avoid false sharing : Use at least a full cache line */
	size = max_t(size_t, size, L1_CACHE_BYTES);

	net->ipv4.fib_table_hash = kzalloc(size, GFP_KERNEL);
	if (!net->ipv4.fib_table_hash)
		return -ENOMEM;

	err = fib4_rules_init(net);
	if (err < 0)
		goto fail;
	return 0;
fail:
	kfree(net->ipv4.fib_table_hash);
	return err;
}
```

4.初始化 255 和 254 表

struct netns_ipv4 结构体

```
struct netns_ipv4 {
#ifdef CONFIG_SYSCTL
	struct ctl_table_header	*forw_hdr;
	struct ctl_table_header	*frags_hdr;
	struct ctl_table_header	*ipv4_hdr;
	struct ctl_table_header *route_hdr;
	struct ctl_table_header *xfrm4_hdr;
#endif
	struct ipv4_devconf	*devconf_all;
	struct ipv4_devconf	*devconf_dflt;
#ifdef CONFIG_IP_MULTIPLE_TABLES
	struct fib_rules_ops	*rules_ops;
	bool			fib_has_custom_rules;
	struct fib_table __rcu	*fib_local;
	struct fib_table __rcu	*fib_main;
	struct fib_table __rcu	*fib_default;
#endif
	struct hlist_head	*fib_table_hash;
	bool			fib_offload_disabled;

#ifdef CONFIG_NETFILTER
	struct xt_table		*iptable_filter;
	struct xt_table		*iptable_mangle;
	struct xt_table		*iptable_raw;
	struct xt_table		*arptable_filter;
#ifdef CONFIG_SECURITY
	struct xt_table		*iptable_security;
#endif
	struct xt_table		*nat_table;
#endif
```

```
        struct local_ports ip_local_ports;
#ifdef CONFIG_IP_MROUTE
#ifndef CONFIG_IP_MROUTE_MULTIPLE_TABLES
        struct mr_table         *mrt;
#else
        struct list_head        mr_tables;
        struct fib_rules_ops    *mr_rules_ops;
#endif
#endif

}
```

struct net 结构体：

```
struct net {
        atomic_t                passive;        /* To decided when the network namespace should be freed.*/
        atomic_t                count;/* To decided when the network namespace should be shut down.*/

        struct list_head        list;           /* list of network namespaces */
        struct list_head        cleanup_list;   /* namespaces on death row */
        struct list_head        exit_list;      /* Use only net_mutex */

        struct idr              netns_ids;
        struct ns_common        ns;
        struct proc_dir_entry   *proc_net;
        struct proc_dir_entry   *proc_net_stat;

        struct sock             *rtnl;                  /* rtnetlink socket */
        struct sock             *genl_sock;
        struct list_head        dev_base_head;
        struct hlist_head       *dev_name_head;
        struct hlist_head       *dev_index_head;
        unsigned int            dev_base_seq;   /* protected by rtnl_mutex */
        int                     ifindex;

        /* core fib_rules */
        struct list_head        rules_ops;
        struct net_device       *loopback_dev;          /* The loopback */

        struct netns_mib        mib;
        struct netns_packet     packet;
        struct netns_ipv4       ipv4;
        struct netns_ipv6       ipv6;
        struct netns_nf                 nf;
        struct netns_xt                 xt;
}


enum rt_class_t {
        RT_TABLE_UNSPEC=0,
/* User defined values */
        RT_TABLE_COMPAT=252,
        RT_TABLE_DEFAULT=253,
        RT_TABLE_MAIN=254,
        RT_TABLE_LOCAL=255,
        RT_TABLE_MAX=0xFFFFFFFF
};
```

fib4_rules_init。将三张表加到 net->ipv4.fib_table_hash 中。

fib4_rules_init

```
int __net_init fib4_rules_init(struct net *net)
{
	int err;
	struct fib_rules_ops *ops;

	ops = fib_rules_register(&fib4_rules_ops_template, net);
	if (IS_ERR(ops))
		return PTR_ERR(ops);

	err = fib_default_rules_init(ops);
	if (err < 0)
		goto fail;
	net->ipv4.rules_ops = ops;
	net->ipv4.fib_has_custom_rules = false;
	return 0;
fail:
	/* also cleans all rules already added */
	fib_rules_unregister(ops);
	return err;
}
```

fib4_rules_ops_template

```
static const struct fib_rules_ops __net_initconst fib4_rules_ops_template = {
	.family		= AF_INET,
	.rule_size	= sizeof(struct fib4_rule),
	.addr_size	= sizeof(u32),
	.action		= fib4_rule_action,
	.suppress	= fib4_rule_suppress,
	.match		= fib4_rule_match,
	.configure	= fib4_rule_configure,
	.delete		= fib4_rule_delete,
	.compare	= fib4_rule_compare,
	.fill		= fib4_rule_fill,
	.default_pref	= fib_default_rule_pref,
	.nlmsg_payload	= fib4_rule_nlmsg_payload,
	.flush_cache	= fib4_rule_flush_cache,
	.nlgroup	= RTNLGRP_IPV4_RULE,
	.policy		= fib4_rule_policy,
	.owner		= THIS_MODULE,
};


static int fib4_rule_configure(struct fib_rule *rule, struct sk_buff *skb,
			       struct fib_rule_hdr *frh,
			       struct nlattr **tb)
{
......
net->ipv4.fib_has_custom_rules = true; //请注意这里的 fib_has_custom_rules
......
}
```

fib_rules_register

fib_rules_register(&fib4_rules_ops_template, net)将 fib4_rules_ops_template 注册到 net.rules_ops 的 list 中

```
struct fib_rules_ops * fib_rules_register(const struct fib_rules_ops *tmpl, struct net *net)
{
	struct fib_rules_ops *ops;
	int err;
	ops = kmemdup(tmpl, sizeof(*ops), GFP_KERNEL);
	if (ops == NULL)
		return ERR_PTR(-ENOMEM);
	INIT_LIST_HEAD(&ops->rules_list);
	ops->fro_net = net;	//注意这里。
	err = __fib_rules_register(ops);
}

static int __fib_rules_register(struct fib_rules_ops *ops)
{
	int err = -EEXIST;
	struct fib_rules_ops *o;
	struct net *net;
	net = ops->fro_net;
	if (ops->rule_size < sizeof(struct fib_rule))
		return -EINVAL;
	if (ops->match == NULL || ops->configure == NULL ||
		ops->compare == NULL || ops->fill == NULL ||
		ops->action == NULL)
		return -EINVAL;
	spin_lock(&net->rules_mod_lock);
	list_for_each_entry(o, &net->rules_ops, list)
		if (ops->family == o->family)
			goto errout;

	list_add_tail_rcu(&ops->list, &net->rules_ops);

}


list_add_tail_rcu(&ops->list, &net->rules_ops);
==>
list_add_tail_rcu(&fib4_rules_ops_template.list, &net->rules_ops);
```

fib_rules_ops

```
struct fib_rules_ops {
	int			family;
	struct list_head	list;
	int			rule_size;
	int			addr_size;
	int			unresolved_rules;
	int			nr_goto_rules;
	int			(*action)(struct fib_rule *, struct flowi *, int, struct fib_lookup_arg *);
	bool			(*suppress)(struct fib_rule *,struct fib_lookup_arg *);
	int			(*match)(struct fib_rule *,struct flowi *, int);
	int			(*configure)(struct fib_rule *,struct sk_buff *,struct fib_rule_hdr *,struct nlattr **);
	int			(*delete)(struct fib_rule *);
	int			(*compare)(struct fib_rule *, struct fib_rule_hdr *, struct nlattr **);
	int			(*fill)(struct fib_rule *, struct sk_buff *,struct fib_rule_hdr *);
	u32			(*default_pref)(struct fib_rules_ops *ops);
	size_t			(*nlmsg_payload)(struct fib_rule *);

	/* Called after modifications to the rules set, must flush
	 * the route cache if one exists. */
```

```
        void                    (*flush_cache)(struct fib_rules_ops *ops);

        int                     nlgroup;
        const struct nla_policy     *policy;
        struct list_head        rules_list;
        struct module               *owner;
        struct net              *fro_net;
        struct rcu_head             rcu;
};
```

fib_default_rules_init

```
static int fib_default_rules_init(struct fib_rules_ops *ops)
{
    int err;

    err = fib_default_rule_add(ops, 0, RT_TABLE_LOCAL, 0);
    if (err < 0)
        return err;
    err = fib_default_rule_add(ops, 0x7FFE, RT_TABLE_MAIN, 0);
    if (err < 0)
        return err;
    err = fib_default_rule_add(ops, 0x7FFF, RT_TABLE_DEFAULT, 0);
    if (err < 0)
        return err;
    return 0;
}


int fib_default_rule_add(struct fib_rules_ops *ops,
                 u32 pref, u32 table, u32 flags)
{
    struct fib_rule *r;

    r = kzalloc(ops->rule_size, GFP_KERNEL);
    if (r == NULL)
        return -ENOMEM;

    atomic_set(&r->refcnt, 1);
    r->action = FR_ACT_TO_TBL;
    r->pref = pref;
    r->table = table;
    r->flags = flags;
    r->fr_net = ops->fro_net;
    //这里请注意，在 fib_rules_lookup()中有调用到：ops->action(rule, fl, flags, arg); 实际上调用的是 fib4_rules_ops_template 的
函数 action：  fib4_rule_action，在 fib4_rule_action 中：fib_get_table(rule->fr_net, rule->table); rule->fr_net 就是 net


    r->suppress_prefixlen = -1;
    r->suppress_ifgroup = -1;

    /* The lock is not required here, the list in unreacheable
     * at the moment this function is called */
    list_add_tail(&r->list, &ops->rules_list);
    return 0;
}
```

fib4_rules_ops_template.rules_list

fib_default_rule_add ---> list_add_tail(&r->list, &ops->rules_list);
====>
将 rule 加到：&fib4_rules_ops_template.rules_list

0x7FFE 32766
0x7FFF 32767

```
root@skyworth:~ # ip rule show
0:      from all lookup local
2:      from all fwmark 0xfe prohibit
20001:  from all fwmark 0xa lookup 10
20002:  from all fwmark 0xa lookup 10 prohibit
20008:  from all fwmark 0xb lookup 11
20009:  from all fwmark 0xb lookup 11 prohibit
20014:  from 192.168.65.91 lookup 12
20015:  from all fwmark 0xc lookup 12
20016:  from all fwmark 0xc lookup 12 prohibit
20021:  from 10.32.91.138 lookup 13
20022:  from all fwmark 0xd lookup 13
20023:  from all fwmark 0xd lookup 13 prohibit
32764:  from all fwmark 0xff lookup 252
32765:  from all fwmark 0xff lookup 252 prohibit
32766:  from all lookup main
32767:  from all lookup default
```

这里只是添加了三个 rule，其他 rule 怎么添加，后面会讲述。

添加路由信息

1.route 命令设置路由

使用 ip_rt_ioctl()只能操作 254 表的路由。route add xxxx。

```
int ip_rt_ioctl(struct net *net, unsigned int cmd, void __user *arg)
{
    struct fib_config cfg;
    struct rtentry rt;
    int err;

    switch (cmd) {
    case SIOCADDRT:        /* Add a route */
    case SIOCDELRT:        /* Delete a route */
        if (!ns_capable(net->user_ns, CAP_NET_ADMIN))
            return -EPERM;

        if (copy_from_user(&rt, arg, sizeof(rt)))
            return -EFAULT;

        rtnl_lock();
        err = rtentry_to_fib_config(net, cmd, &rt, &cfg);
        if (err == 0) {
            struct fib_table *tb;

            if (cmd == SIOCDELRT) {
                tb = fib_get_table(net, cfg.fc_table);
                if (tb)
                    err = fib_table_delete(tb, &cfg);
                else
                    err = -ESRCH;
            } else {
                tb = fib_new_table(net, cfg.fc_table);
                if (tb)
                    err = fib_table_insert(tb, &cfg);
```

```
                else
                    err = -ENOBUFS;
            }

            /* allocated by rtentry_to_fib_config() */
            kfree(cfg.fc_mx);
        }
        rtnl_unlock();
        return err;
    }
    return -EINVAL;
}
```

struct rtentry

struct rtentry 就是用户 route 命令设置的内容。

```
struct rtentry {
    unsigned long   rt_pad1;
    struct sockaddr rt_dst;         //目的地址
    struct sockaddr rt_gateway; //网关地址
    struct sockaddr rt_genmask; //掩码
    unsigned short  rt_flags;     //
    short           rt_pad2;
    unsigned long   rt_pad3;
    void            *rt_pad4;
    short           rt_metric;  /* +1 for binary compatibility! */
    char __user *rt_dev;     //网卡接口名称
    unsigned long   rt_mtu;         /* per route MTU/Window */
#ifndef __KERNEL__
#define rt_mss  rt_mtu              /* Compatibility :-(    */
#endif
    unsigned long   rt_window;  /* Window clamping */
    unsigned short  rt_irtt;    /* Initial RTT */
};
```

struct fib_config

rtentry_to_fib_config 将 struct rtentry 转换 struct fib_config 结构。

```
struct fib_config {
    u8              fc_dst_len;
    u8              fc_tos;
    u8              fc_protocol;
    u8              fc_scope;
    u8              fc_type;
    /* 3 bytes unused */
    u32             fc_table;
    __be32              fc_dst;
    __be32              fc_gw;
    int             fc_oif;
    u32             fc_flags;
    u32             fc_priority;
    __be32              fc_prefsrc;
    struct nlattr       *fc_mx;
    struct rtnexthop    *fc_mp;
    int             fc_mx_len;
    int             fc_mp_len;
    u32             fc_flow;
    u32             fc_nlflags;
    struct nl_info      fc_nlinfo;
};
```

fib_table_insert---> fib_create_info

struct fib_config *cfg;
fi = fib_create_info(cfg);

2.ip 命令设置路由

ip rule add/del 设置 rule 规则

```
static int __init fib_rules_init(void)
{
	int err;
	rtnl_register(PF_UNSPEC, RTM_NEWRULE, fib_nl_newrule, NULL, NULL);
	rtnl_register(PF_UNSPEC, RTM_DELRULE, fib_nl_delrule, NULL, NULL);
	rtnl_register(PF_UNSPEC, RTM_GETRULE, NULL, fib_nl_dumprule, NULL);

	err = register_pernet_subsys(&fib_rules_net_ops);
	if (err < 0)
		goto fail;

	err = register_netdevice_notifier(&fib_rules_notifier);
	if (err < 0)
		goto fail_unregister;

	return 0;

fail_unregister:
	unregister_pernet_subsys(&fib_rules_net_ops);
fail:
	rtnl_unregister(PF_UNSPEC, RTM_NEWRULE);
	rtnl_unregister(PF_UNSPEC, RTM_DELRULE);
	rtnl_unregister(PF_UNSPEC, RTM_GETRULE);
	return err;
}
```

```
rtnl_register(PF_UNSPEC, RTM_NEWRULE, fib_nl_newrule, NULL, NULL);
rtnl_register(PF_UNSPEC, RTM_DELRULE, fib_nl_delrule, NULL, NULL);
rtnl_register(PF_UNSPEC, RTM_GETRULE, NULL, fib_nl_dumprule, NULL);
```

ip rule xxx 操作对应上面三个。

inet_rtm_newroute 添加路由

ip_fib_init()注册了下面三个函数。

```
rtnl_register(PF_INET, RTM_NEWROUTE, inet_rtm_newroute, NULL, NULL);
rtnl_register(PF_INET, RTM_DELROUTE, inet_rtm_delroute, NULL, NULL);
rtnl_register(PF_INET, RTM_GETROUTE, NULL, inet_dump_fib, NULL);
```

ip route add xxxx 在内核中调用 inet_rtm_newroute()来添加路由

```
static int inet_rtm_newroute(struct sk_buff *skb, struct nlmsghdr *nlh)
{
	struct net *net = sock_net(skb->sk);
	struct fib_config cfg;
	struct fib_table *tb;
	int err;

	err = rtm_to_fib_config(net, skb, nlh, &cfg);
	if (err < 0)
		goto errout;

	tb = fib_new_table(net, cfg.fc_table);
	if (!tb) {
		err = -ENOBUFS;
		goto errout;
	}
	err = fib_table_insert(tb, &cfg);
errout:
	return err;
}
```

rtm_to_fib_confi()将 ip route xxx 的命令内容转换成 struct fib_config 结构体。

inet_rtm_delroute 删除路由

```
static int inet_rtm_delroute(struct sk_buff *skb, struct nlmsghdr *nlh)
{
	struct net *net = sock_net(skb->sk);
	struct fib_config cfg;
	struct fib_table *tb;
	int err;

	err = rtm_to_fib_config(net, skb, nlh, &cfg);
	if (err < 0)
		goto errout;

	tb = fib_get_table(net, cfg.fc_table);
	if (!tb) {
		err = -ESRCH;
		goto errout;
	}

	err = fib_table_delete(tb, &cfg);
errout:
	return err;
}
```

inet_dump_fib 获取路由信息

inet_dump_fib 这个应该是查询路由用的。

```
static int inet_dump_fib(struct sk_buff *skb, struct netlink_callback *cb)
{
	struct net *net = sock_net(skb->sk);
	unsigned int h, s_h;
	unsigned int e = 0, s_e;
	struct fib_table *tb;
```

```
	struct hlist_head *head;
	int dumped = 0;

	if (nlmsg_len(cb->nlh) >= sizeof(struct rtmsg) &&
	    ((struct rtmsg *) nlmsg_data(cb->nlh))->rtm_flags & RTM_F_CLONED)
	    return skb->len;

	s_h = cb->args[0];
	s_e = cb->args[1];

	rcu_read_lock();
	for (h = s_h; h < FIB_TABLE_HASHSZ; h++, s_e = 0) {
	    e = 0;
	    head = &net->ipv4.fib_table_hash[h];
	    hlist_for_each_entry_rcu(tb, head, tb_hlist) {
	        if (e < s_e)
	            goto next;
	        if (dumped)
	            memset(&cb->args[2], 0, sizeof(cb->args) -
	                        2 * sizeof(cb->args[0]));
	        if (fib_table_dump(tb, skb, cb) < 0)
	            goto out;
	        dumped = 1;
next:
	        e++;
	    }
	}
out:
	rcu_read_unlock();

	cb->args[1] = e;
	cb->args[0] = h;

	return skb->len;
}
```

以上内容均属于自己总结的，如果有什么纰漏敬请指出。