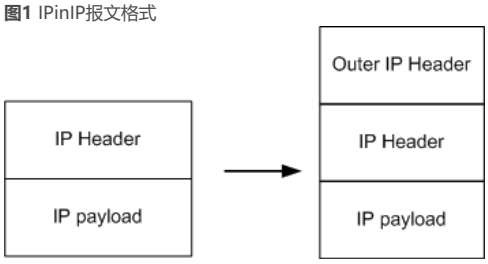
**Linux IPIP隧道协议**

**ipip协议为在ip协议报文的基础上继续封装ip报文，基于tun设备实现，是一种点对点的通讯技术。**

- **报文格式**

**图1** IPinIP报文格式




其中，内层IPv4头部和普通IPv4报文头部相同，IPv4报文头详细解释请参见IPv4报文格式。外层IPv4头部处理如下：

| 字段 | 含义 |
| --- | --- |
| Version | =4 |
| IHL | 指外层IP头部长度，以32比特为计算单位。 |
| TOS | 从内层IP头部复制。 |
| Total Length | 指整个IP负载的长度，包括外层IP头，内层IP头和IP负载。 |
| Identification, Flags, Fragment Offset | 这三个字段的含义与RFC791的定义相同。注意，如果内层IP头部的DF位置位，外层IP头部的DF位也必须置位。如果内层IP头的DF未置位，外层IP头部的DF位可以置位也可以不置位。 |
| Time to Live | 外层IP头部的TTL域设置为发送该数据包到隧道目的端的合适的值。 |
| Protocol | =4 |
| Header Checksum | 外层IP头部的校验字段。 |
| Source Address | 执行该IPinIP隧道头封装的隧道入口设备的IP地址。 |
| Destination Address | 执行该IPinIP隧道头解封装的隧道出口设备的IP地址。 |
| Options | 内层IP头部的任何选项字段通常不被复制到外层IP头部。隧道路径上的设备可以添加新的选项字字段。内层IP头部的安全选项字段的类型可能影响外层IP头部的安全选项字段的选择。 |

**报文示例**

**图2** IP in IP报文

昵称：游码树子
园龄：7年8个月
粉丝：0
关注：12
+加关注

< 2024年12月 >

| 日 | 一 | 二 | 三 | 四 | 五 | 六 |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| 22 | 23 | 24 | 25 | 26 | 27 | 28 |
| 29 | 30 | 31 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 | 10 | 11 |

搜索

找找看

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

我的标签

virt(1)
ParameterizedMessage(1)
MySQL(1)
AbstractConnPool(1)

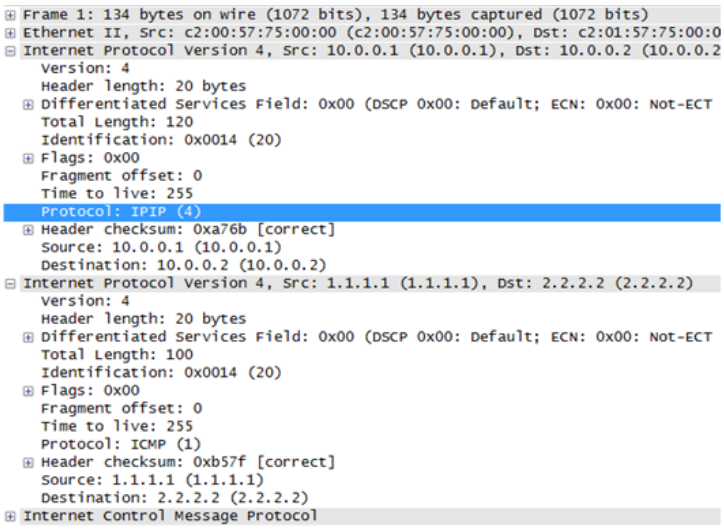
文章分类

Linux(1)
TCP/IP(1)
日志(2)

最新评论

1. Re:Mysql事件
实践出真知~

--游码树子




**参考标准**

| 标准 | 描述 |
| --- | --- |
| RFC 2003 | IP Encapsulation within IP |

## IP 隧道

Linux 原生支持多种三层隧道，其底层实现原理都是基于 tun 设备。我们可以通过命令 `ip tunnel help` 查看 IP 隧道的相关操作。

```
[root@localhost ~]# ip tunnel help
Usage: ip tunnel { add | change | del | show | prl | 6rd } [ NAME ]
          [ mode { ipip | gre | sit | isatap | vti } ] [ remote ADDR ] [ local ADDR ]
          [ [i|o]seq ] [ [i|o]key KEY ] [ [i|o]csum ]
          [ prl-default ADDR ] [ prl-nodefault ADDR ] [ prl-delete ADDR ]
          [ 6rd-prefix ADDR ] [ 6rd-relay_prefix ADDR ] [ 6rd-reset ]
          [ ttl TTL ] [ tos TOS ] [ [no]pmtudisc ] [ dev PHYS_DEV ]
Where: NAME := STRING
       ADDR := { IP_ADDRESS | any }
       TOS  := { STRING | 00..ff | inherit | inherit/STRING | inherit/00..ff }
       TTL  := { 1..255 | inherit }
       KEY  := { DOTTED_QUAD | NUMBER }
```

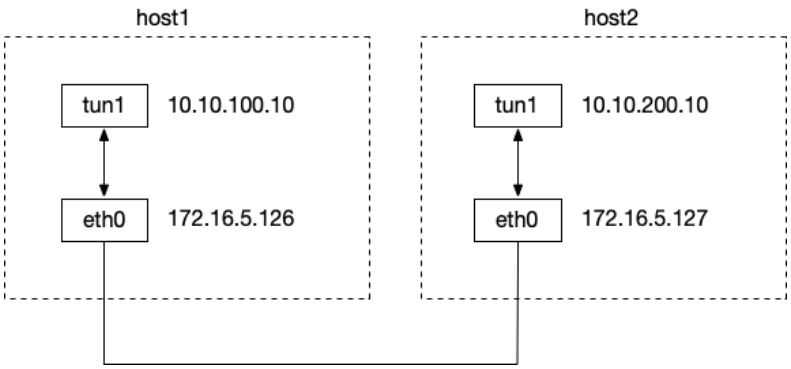
可以看到，Linux 原生一共支持 5 种 IP 隧道。

- `ipip` ：即 `IPv4 in IPv4` ，在 IPv4 报文的基础上再封装一个 IPv4 报文。
- `gre` ：即通用路由封装（ `Generic Routing Encapsulation` ），定义了在任意一种网络层协议上封装其他任意一种网络层协议的机制，IPv4 和 IPv6 都适用。
- `sit` ：和 `ipip` 类似，不同的是 `sit` 是用 IPv4 报文封装 IPv6 报文，即 `IPv6 over IPv4` 。
- `isatap` ：即站内自动隧道寻址协议（ `Intra-Site Automatic Tunnel Addressing Protocol` ），和 `sit` 类似，也是用于 IPv6 的隧道封装。
- `vti` ：即虚拟隧道接口（ `Virtual Tunnel Interface` ），是 cisco 提出的一种 `IPsec` 隧道技术。

## 实践IPIP隧道

ipip需要内核模块ipip的支持

```
#开启ipip协议模块
modprobe ipip
#查看ipip协议模块
lsmod |  grep ipip
ipip                   13465  0
tunnel4                13252  1 ipip
ip_tunnel              25163  1 ipip
```




## 实战

两台主机：172.16.5.126(host1)和172.16.5.127(host2)

在host1上创建tun1设备，执行如下命令：

```
# 用来创建tun1设备，并ipip协议的外层ip，目的ip为172.16.5.127， 源ip为172.16.5.126
ip tunnel add tun1 mode ipip remote 172.16.5.127  local 172.16.5.126
# 给tun1设备增加ip地址，并设置tun1设备的对端ip地址为10.10.200.10
ip addr add 10.10.100.10 peer 10.10.200.10 dev tun1
ip link  set tun1 up

$  ifconfig tun1
tun1: flags=209<up,pointopoint,running,noarp>  mtu 1480
        inet 10.10.100.10  netmask 255.255.255.255  destination 10.10.200.10
        tunnel   txqueuelen 1000  (IPIP Tunnel)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

# 增加一条路由，所有到达10.10.200.10的请求会经过设备tun1
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.16.7.253    0.0.0.0         UG    0      0        0 eth0
10.10.200.10    0.0.0.0         255.255.255.255 UH    0      0        0 tun1
172.16.0.0      0.0.0.0         255.255.248.0   U     0      0        0 eth0</up,pointopoint,runni
```

同样在host2上创建tun1设备：

```
ip tunnel add tun1 mode ipip remote 172.16.5.126  local 172.16.5.127
ip addr add 10.10.200.10 peer 10.10.100.10 dev tun1
ip link  set tun1 up
```

并分别在host1和host2上打开ip_forward功能

```
echo 1 >   /proc/sys/net/ipv4/ip_forward
```

在host1的tun1上抓包，可以看到正常的ping包。然后在host1上ping 10.10.200.10，可以ping通。

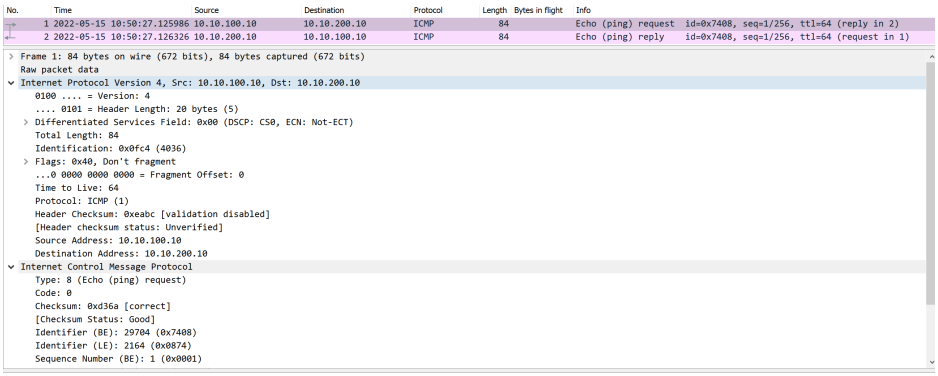
host1控制台

```
ping 10.10.200.10 -c 1
```

host1打开另外控制台抓包

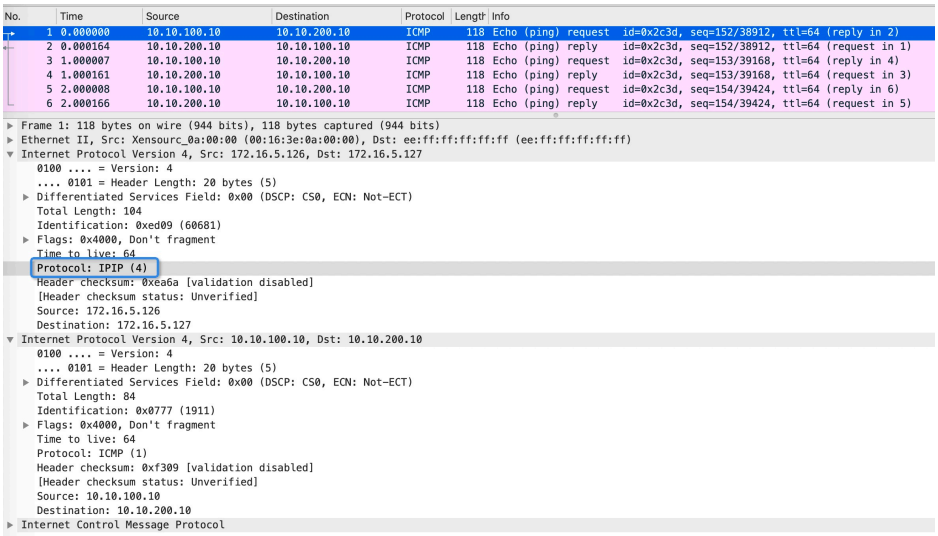
```
##在tun1虚拟网卡抓包
tcpdump -i tun1 -w tun1.pcap
```

tun1虚拟网卡看到是解封后的正常icmp包




在host1的eth1上抓包，可以看到已经是ipip的数据包了。

```
tcpdump -i ens160 host 10.10.0.0/16 -w ens160.pcap
```




ipip抓包

清理现场

分别在两台主机上执行

```
ip link delete tun1
```

# ref

**什么是 IP 隧道，Linux 怎么实现隧道通信？**

- ## Linux IPIP隧道实现

简单分析Linux（2.6.32版本）中的IPIP隧道的实现过程。

### 一. IPIP的初始化

Linux中的IPIP隧道文件主要分布在 `tunnel4.c` 和 `ipip.c` 文件中。因为是三层隧道，在IP报文中填充的三层协议自然就不能是常见的TCP和UDP，所以，Linux抽象了一个隧道层，位置就相当于传输层，主要的实现就是在 `tunnel4.c` 中。来看看他们的初始化：

抽象的隧道层和IPIP模块都是以注册模块的方式进行初始化

```
module_init(tunnel4_init);

module_init(ipip_init);
```

首先看隧道层的初始化，主要的工作就是注册隧道协议和对应的处理函数：

```
static int __init tunnel4_init( void )
{
    if (inet_add_protocol(&tunnel4_protocol, IPPROTO_IPIP)) {
        printk(KERN_ERR  "tunnel4 init: can't add protocol\n" );
        return -EAGAIN;
    }
```

```
#if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
    if (inet_add_protocol(&tunnel64_protocol, IPPROTO_IPV6)) {
        printk(KERN_ERR  "tunnel64 init: can't add protocol\n" );
        inet_del_protocol(&tunnel4_protocol, IPPROTO_IPIP);
        return -EAGAIN;
    }
#endif
    return 0;
}
```

`inet_add_protocol(&tunnel4_protocol, IPPROTO_IPIP)` 把IPIP隧道协议注册进 `inet_protos` 全局数组中，而 `inet_protos` 中的其他协议注册是在 `inet_init()` 中：

```
    if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
        printk(KERN_CRIT  "inet_init: Cannot add ICMP protocol\n" );
    if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
        printk(KERN_CRIT  "inet_init: Cannot add UDP protocol\n" );
    if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
        printk(KERN_CRIT  "inet_init: Cannot add TCP protocol\n" );
#ifdef CONFIG_IP_MULTICAST
    if (inet_add_protocol(&igmp_protocol, IPPROTO_IGMP) < 0)
        printk(KERN_CRIT  "inet_init: Cannot add IGMP protocol\n" );
#endif
```

看一下隧道层的处理函数：

```
static const struct net_protocol tunnel4_protocol = {
    .handler    =   tunnel4_rcv,
    .err_handler    =   tunnel4_err,
    .no_policy  =   1,
    .netns_ok   =   1,
};
```

这样注册完后，当接收到三层类型是 `IPPROTO_IPIP` 时，就会调用 `tunnel4_rcv` 进行下一步的处理。可以说在隧道层对隧道协议进行的注册，保证能够识别接收到隧道包。而对隧道包的处理则是在IPIP中完成的。

```
for (handler = tunnel4_handlers; handler; handler = handler->next)
        if (!handler->handler(skb))
            return 0;

icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PORT_UNREACH, 0);
```

在隧道层的处理函数中进一步调用注册的不同隧道协议的处理函数，分别处理。

接下来进一步看IPIP的初始化部分：

```
static int __init ipip_init( void )
{
    int err;

    printk(banner);

    if (xfrm4_tunnel_register(&ipip_handler, AF_INET)) {
        printk(KERN_INFO  "ipip init: can't register tunnel\n" );
        return -EAGAIN;
    }

    err = register_pernet_gen_device(&ipip_net_id, &ipip_net_ops);
    if (err)
        xfrm4_tunnel_deregister(&ipip_handler, AF_INET);

    return err;
}
```

IPIP模块初始化的部分也十分精简，主要就是两部分的工作，一个是注册协议相关的处理函数等；另一个是创建对应的虚拟设备。

首先是注册了IPIP对应的处理函数

```
static struct xfrm_tunnel ipip_handler = {
    .handler    =   ipip_rcv,
    .err_handler    =   ipip_err,
    .priority   =   1,
};
```

可以看到，从隧道层的处理函数进一步找到IPIP的处理函数后，IPIP报文就会最终进入ipip_rcv()处理，这部分在后面再详细说明。

再来看创建设备部分：

`register_pernet_gen_device()` -> `register_pernet_operations()` ,在其中，最后调用了操作集中的初始化函数

```
if (ops->init == NULL)
        return 0;
return ops->init(&init_net);
```

对应的操作函数集如下：

```
static struct pernet_operations ipip_net_ops = {
    .init = ipip_init_net,
    . exit = ipip_exit_net,
};
```

这样，就进入到 `ipip_init_net()` 中，终于看到创建设备咯

```
ipn->fb_tunnel_dev = alloc_netdev( sizeof ( struct ip_tunnel),
                      "tunl0" ,
                      ipip_tunnel_setup);

if (!ipn->fb_tunnel_dev) {
  err = -ENOMEM;
  goto err_alloc_dev;
}
```

在创建设备时，对设备还进行了初始化配置 `ipip_tunnel_setup()`

```
static void ipip_tunnel_setup( struct net_device *dev)
{
    dev->netdev_ops     = &ipip_netdev_ops;
    dev->destructor     = free_netdev;

    dev->type       = ARPHRD_TUNNEL;
    dev->hard_header_len    = LL_MAX_HEADER +  sizeof ( struct iphdr);
    dev->mtu        = ETH_DATA_LEN -  sizeof ( struct iphdr);
    dev->flags      = IFF_NOARP;
    dev->iflink     = 0;
    dev->addr_len       = 4;
    dev->features       |= NETIF_F_NETNS_LOCAL;
    dev->priv_flags     &= ~IFF_XMIT_DST_RELEASE;
}
```

这里看到有设备的操作集 `dev->netdev_ops = &ipip_netdev_ops;` ，通过这个，我们能知道这个设备都能进行哪些操作：

```
static const struct net_device_ops ipip_netdev_ops = {
    .ndo_uninit = ipip_tunnel_uninit,
    .ndo_start_xmit = ipip_tunnel_xmit,
    .ndo_do_ioctl   = ipip_tunnel_ioctl,
    .ndo_change_mtu = ipip_tunnel_change_mtu,

};
```

可以看出设备最后的发送函数就是 `ipip_tunnel_xmit()`

之后在 `ipip_fb_tunnel_init` ()中对IPIP隧道进行了参数的设置，包括名字，协议号什么的。最后就注册这个新创建的设备吧

```
if ((err = register_netdev(ipn->fb_tunnel_dev)))
        goto err_reg_dev;
```

这样整个的初始化过程就做完了，下面简单分析一下发送和接收的过程。

## 二. IPIP的接收

我们之前说到过，对应从网卡收上来的报文，过完链路层后就会到 `ip_rcv()` 中，大概是这样的路线：

`ip_rcv()` -> `ip_rcv_finish()` -> `ip_local_deliver()` -> `ip_local_deliver_finish()` ，最终会在其中看到

```
ret = ipprot->handler(skb);
if (ret < 0) {
  protocol = -ret;
  goto resubmit;
}
```

调用注册的协议的处理函数，也就是最终会调到 `tunnel4_rcv()` -> `ipip_rcv()` 。

```
if ((tunnel = ipip_tunnel_lookup(dev_net(skb->dev),
                    iph->saddr, iph->daddr)) != NULL) {  /* 查找对应的tunnel */
        if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
            read_unlock(&ipip_lock);
            kfree_skb(skb);
            return 0;
        }

        secpath_reset(skb);

        skb->mac_header = skb->network_header;  /* 修改报文的mac头指向网络层开始，为了下面使用netif_r
        skb_reset_network_header(skb);
        skb->protocol = htons(ETH_P_IP);
        skb->pkt_type = PACKET_HOST;   /* 填充报文信息 */

        tunnel->dev->stats.rx_packets++;
        tunnel->dev->stats.rx_bytes += skb->len;
        skb->dev = tunnel->dev;
        skb_dst_drop(skb);
        nf_reset(skb);
        ipip_ecn_decapsulate(iph, skb);
        netif_rx(skb);   /* 传递给上层协议栈 */
        read_unlock(&ipip_lock);
        return 0;
    }
```

## 三. IPIP的发送

在初始化的时候，我们看到IPIP报文的发送时通过 `ipip_tunnel_xmit()` 函数进行的。在发送时，要给原有的IP报文头前添加新的IP头，我们略过这个函数的前面的路由处理的部分，直接看关键的添加报文头的地方：

```
max_headroom = (LL_RESERVED_SPACE(tdev)+ sizeof ( struct iphdr));

    if (skb_headroom(skb) < max_headroom || skb_shared(skb) ||
        (skb_cloned(skb) && !skb_clone_writable(skb, 0))) {
        struct sk_buff *new_skb = skb_realloc_headroom(skb, max_headroom); /* 为新的报文头分配空间 *
        if (!new_skb) {
            ip_rt_put(rt);
            stats->tx_dropped++;
            dev_kfree_skb(skb);
            return NETDEV_TX_OK;
        }
        if (skb->sk)
            skb_set_owner_w(new_skb, skb->sk);
        dev_kfree_skb(skb);
        skb = new_skb;
        old_iph = ip_hdr(skb);
    }

    skb->transport_header = skb->network_header;  /* 重新设置传输层的头位置 */
    skb_push(skb,  sizeof ( struct iphdr));
    skb_reset_network_header(skb);
    memset (&(IPCB(skb)->opt), 0,  sizeof (IPCB(skb)->opt));
    IPCB(skb)->flags &= ~(IPSKB_XFRM_TUNNEL_SIZE | IPSKB_XFRM_TRANSFORMED |
                  IPSKB_REROUTED);
    skb_dst_drop(skb);
    skb_dst_set(skb, &rt->u.dst);

    /*
     *  Push down and install the IPIP header.
     */

    /* 设置新的IP头字段 */
    iph             =   ip_hdr(skb);
    iph->version        =   4;
    iph->ihl        =    sizeof ( struct iphdr)>>2;
    iph->frag_off       =   df;
    iph->protocol       =   IPPROTO_IPIP;
    iph->tos        =   INET_ECN_encapsulate(tos, old_iph->tos);
```

```
39      iph->daddr      =   rt->rt_dst;
40      iph->saddr      =   rt->rt_src;
41
42      if ((iph->ttl = tiph->ttl) == 0)
43          iph->ttl    =   old_iph->ttl;
```

最后调用 `IPTUNNEL_XMIT()` 宏发送出去。

分类: TCP/IP

好文要顶 关注我 收藏该文 微信分享


游码树子
粉丝 - 0 关注 - 12
+加关注

0 0

升级成为会员

posted on 2022-05-15 10:24 游码树子 阅读(2249) 评论(0) 编辑 收藏 举报

刷新评论 刷新页面 返回顶部

发表评论 升级成为园子VIP会员

编辑 预览 B

支持 Markdown

自动补全

提交评论 退出 订阅评论

[Ctrl+Enter快捷键提交]

【推荐】100%开源！大型工业跨平台软件C++源码提供，建模，组态！

【推荐】抖音旗下AI助手豆包，你的智能百科全书，全免费不限次数

【推荐】轻量又高性能的 SSH 工具 IShell：AI 加持，快人一步


**编辑推荐:**
· C#委托的前世今生
· Java日志记录几种实现方案
· RocketMQ系列2：领域模型和技术概念
· 技术项目文档书写规范指南
· .NET Core 锁(Lock)底层原理浅谈


**阅读排行:**
· C#委托的前世今生

· 关于服务器挖矿处理思路
· .NET静态代码编织——肉夹馍（Rougamo）5.0
· 一个.NET开源、免费、功能强大的 PDF 处理工具
· 全网最详细的Spring入门教程

Copyright © 2024 游码树子
Powered by .NET 9.0 on Kubernetes Powered By博客园