

# Linux

## 内核源码剖析

### —— TCP/IP实现

下册

樊东东 莫澜 编著



机械工业出版社  
CHINA MACHINE PRESS

ISBN 978-7-111-32373-0

◎ 策划 车忱  
◎ 封面设计 旭洲企划  
刘吉维

# Linux 内核源码剖析

## — TCP/IP 实现

下册

本书详细论述了 Linux 内核 2.6.20 版本中 TCP/IP 的实现。书中给出了大量的源代码，通过对源代码的详细注释，帮助读者掌握 TCP/IP 的实现。本书根据协议栈层次，从驱动层逐步论述到传输层，包括驱动的实现、接口层的输入输出、IP 层的输入输出以及 IP 选项的处理、邻居子系统、路由、套接口及传输层等内容，全书基本涵盖了网络体系架构全部的知识。特别是 TCP，包括 TCP 连接的建立和终止、输入与输出，以及拥塞控制的实现。

- 套接口缓存
- 网络设备
- IP：网际协议
- ICMP：Internet 控制报文协议
- IP 组播
- IGMP：Internet 组管理协议
- 邻居子系统
- 路由表
- 套接口层
- TCP：传输控制协议
- TCP 连接的建立
- TCP 拥塞控制的实现
- TCP 的输出
- TCP 的输入
- TCP 连接的终止
- UDP：用户数据报

上架建议：操作系统 / Linux

地址：北京市百万庄大街22号  
电话服务  
社服务中心：(010)88361066  
销售一部：(010)68326294  
销售二部：(010)88379649  
读者服务部：(010)68993821

邮政编码：100037  
网络服务  
门户网：<http://www.cmpbook.com>  
教材网：<http://www.cmpedu.com>  
封面防伪标均为盗版

定价：142.00元(上、下册)

ISBN 978-7-111-32373-0



9 787111 323730 >

# Linux 内核源码剖析

## ——TCP/IP 实现

下册

樊东东 莫 澜 编著



机械工业出版社

# 下册目录

|   |     |  |     |
|---|-----|--|-----|
| 第 20 章 路由缓存 .....                           | 551 | 20.9.7 通过写/proc 的 flush 文件 .....           | 585 |
| 20.1 系统参数 .....                             | 551 | 20.10 ICMP 重定向消息的处理 .....                  | 585 |
| 20.2 路由缓存的组织结构 .....                        | 552 | 20.11 ICMP 目的不可达, 需要分片<br>消息的处理 .....      | 588 |
| 20.2.1 rtable 结构 .....                      | 552 | 第 21 章 路由策略 .....                          | 590 |
| 20.2.2 flowi 结构 .....                       | 555 | 21.1 路由策略组织结构 .....                        | 590 |
| 20.2.3 dst_entry 结构 .....                   | 556 | 21.1.1 fib_rules_ops 结构 .....              | 590 |
| 20.2.4 dst_ops 结构 .....                     | 559 | 21.1.2 fib_rule 结构 .....                   | 592 |
| 20.3 初始化 .....                              | 561 | 21.1.3 fib4_rule 结构 .....                  | 594 |
| 20.4 创建路由缓存项 .....                          | 563 | 21.2 三个默认路由策略 .....                        | 595 |
| 20.4.1 创建输入路由缓存项 .....                      | 563 | 21.3 IPv4 协议族的 fib_rules_ops<br>结构实例 ..... | 595 |
| 20.4.2 创建输出路由缓存项 .....                      | 565 | 21.3.1 fib4_rule_action() .....            | 595 |
| 20.5 添加路由由表项到缓存中:<br>rt_intern_hash() ..... | 568 | 21.3.2 fib4_rule_match() .....             | 596 |
| 20.6 输入路由由缓存查询:<br>ip_route_input() .....   | 571 | 21.3.3 fib4_rule_configure() .....         | 596 |
| 20.7 输出路由由缓存查询 .....                        | 573 | 21.3.4 fib4_rule_compare() .....           | 598 |
| 20.7.1 ip_route_output_key() .....          | 573 | 21.3.5 fib4_rule_fill() .....              | 598 |
| 20.7.2 __ip_route_output_key() .....        | 573 | 21.3.6 fib4_rule_default_pref() .....      | 599 |
| 20.8 垃圾回收 .....                             | 575 | 21.4 netlink 接口 .....                      | 599 |
| 20.8.1 路由缓存项的过期 .....                       | 575 | 21.4.1 netlink 路由策略消息结构 .....              | 599 |
| 20.8.2 判断缓存路由表项是否<br>可被删除 .....             | 575 | 21.4.2 fib_nl_newrule() .....              | 600 |
| 20.8.3 同步清理 .....                           | 576 | 21.4.3 fib_nl_delrule() .....              | 602 |
| 20.8.4 异步清理 .....                           | 580 | 21.5 受网络设备状态改变的影响 .....                    | 604 |
| 20.8.5 路由缓存项的释放 .....                       | 582 | 21.6 策略路由的查找 .....                         | 604 |
| 20.9 刷新缓存 .....                             | 582 | 第 22 章 套接口层 .....                          | 606 |
| 20.9.1 通过定时器定时刷新 .....                      | 584 | 22.1 socket 结构 .....                       | 607 |
| 20.9.2 网络设备的硬件地址发生<br>改变 .....              | 584 | 22.2 proto_ops 结构 .....                    | 608 |
| 20.9.3 网络设备状态发生变化 .....                     | 584 | 22.3 套接口文件系统 .....                         | 610 |
| 20.9.4 给设备添加或删除一个<br>IP 地址 .....            | 584 | 22.3.1 套接口文件系统类型 .....                     | 610 |
| 20.9.5 全局转发状态或设备的转发<br>状态发生变化 .....         | 584 | 22.3.2 套接口文件系统超级块操作<br>接口 .....            | 610 |
| 20.9.6 一条路由被删除 .....                        | 585 | 22.3.3 套接口文件的 inode .....                  | 611 |
|   |     | 22.3.4 sock_alloc_inode() .....            | 611 |
|   |     | 22.3.5 sock_destroy_inode() .....          | 612 |
|   |     | 22.4 套接口文件 .....                           | 612 |

|   |            |   |            |
|---|------------|---|------------|
| 22.4.1 套接口文件与套接口的绑定                         | 612        | 25.2 传输描述块结构  | 662        |
| 22.4.2 根据文件描述符获取套接口                         | 614        | 25.2.1 sock_common 结构                                     | 662        |
| 22.5 进程、文件描述符和套接口                           | 615        | 25.2.2 sock 结构  | 663        |
| 22.6 套接口层的系统初始化                             | 616        | 25.2.3 inet_sock 结构                                       | 670        |
| 22.7 套接口系统调用                                | 617        | 25.3 proto 结构   | 674        |
| 22.7.1 套接口系统调用入口                            | 617        | 25.3.1 proto 实例组织结构                                       | 677        |
| 22.7.2 socket 系统调用                          | 621        | 25.3.2 proto_register()                                   | 677        |
| 22.7.3 bind 系统调用                            | 629        | 25.3.3 proto_unregister()                                 | 679        |
| 22.7.4 listen 系统调用                          | 632        | 25.4 传输控制块的内存管理   | 680        |
| 22.7.5 accept 系统调用                          | 633        | 25.4.1 传输控制块的分配和释放  | 680        |
| 22.7.6 connect 系统调用                         | 635        | 25.4.2 普通的发送缓存区的分配  | 682        |
| 22.7.7 shutdown 系统调用                        | 636        | 25.4.3 发送缓存的分配与释放   | 685        |
| 22.7.8 close 系统调用                           | 638        | 25.4.4 接收缓存的分配与释放   | 686        |
| 22.7.9 select 系统调用的实现                       | 640        | 25.4.5 辅助缓存的分配与释放   | 688        |
| <b>第 23 章 套接口 I/O</b>                       | <b>641</b> | 25.5 异步 IO 机制   | 688        |
| 23.1 输出/输入数据的组织                             | 641        | 25.5.1 sk_wake_async()                                    | 689        |
| 23.1.1 msghdr 结构                            | 641        | 25.5.2 sock_def_wakeup()                                  | 690        |
| 23.1.2 verify_iovec()                       | 643        | 25.5.3 sock_def_error_report()                            | 690        |
| 23.1.3 memcpy_toiovec()                     | 644        | 25.5.4 sock_def_readable()                                | 691        |
| 23.1.4 memcpy_fromiovec()                   | 644        | 25.5.5 sock_def_write_space()和<br>sk_stream_write_space() | 691        |
| 23.1.5 memcpy_fromiovecend()                | 644        | 25.5.6 sk_send_sigurg()                                   | 692        |
| 23.1.6 csum_partial_copy_<br>fromiovecend() | 644        | 25.5.7 接收到 FIN 段后通知进程                                     | 692        |
| 23.2 输出系统调用                                 | 644        | 25.5.8 sock_fasync()                                      | 693        |
| 23.2.1 sock_sendmsg()                       | 644        | 25.6 传输控制块的同步锁  | 694        |
| 23.2.2 sendto 系统调用                          | 645        | 25.6.1 socket_lock_t 结构                                   | 694        |
| 23.2.3 send 系统调用                            | 646        | 25.6.2 控制用户进程和下半部间<br>同步锁                                 | 695        |
| 23.2.4 sendmsg 系统调用                         | 646        | 25.6.3 控制下半部间同步锁  | 698        |
| 23.3 输入系统调用                                 | 649        | <b>第 26 章 TCP: 传输控制协议</b>                                 | <b>699</b> |
| <b>第 24 章 套接口选项</b>                         | <b>650</b> | 26.1 系统参数   | 699        |
| 24.1 setsockopt 系统调用                        | 650        | 26.2 TCP 的 inet_protosw 实例                                | 705        |
| 24.2 ioctl 系统调用                             | 655        | 26.3 TCP 的 net_protocol 结构                                | 705        |
| 24.2.1 ioctl 在文件系统内的调用过程                    | 655        | 26.4 TCP 传输控制块  | 706        |
| 24.2.2 套接口文件 ioctl 调用接口的<br>实现              | 655        | 26.4.1 inet_connection_sock 结构                            | 706        |
| 24.2.3 套接口层的实现                              | 658        | 26.4.2 inet_connection_sock_af_ops<br>结构                  | 710        |
| 24.3 getsockname 系统调用                       | 659        | 26.4.3 tcp_sock 结构  | 711        |
| 24.4 getpeername 系统调用                       | 660        | 26.4.4 tcp_options_received 结构                            | 721        |
| <b>第 25 章 传输控制块</b>                         | <b>661</b> | 26.4.5 tcp_skb_cb 结构                                      | 723        |
| 25.1 系统参数                                   | 662        |   |            |

|  |            |                                    |            |
|--|------------|------------------------------------|------------|
| 26.5 TCP 的 proto 结构和 proto_ops 结构的实例 ..... | 725        | 27.7 FIN_WAIT_2 定时器 .....          | 764        |
| 26.6 TCP 状态迁移图 .....                       | 725        | 27.7.1 FIN_WAIT_2 定时器处理函数 .....    | 765        |
| 26.7 TCP 首部 .....                          | 726        | 27.7.2 激活 FIN_WAIT_2 定时器 .....     | 765        |
| 26.8 TCP 校验和 .....                         | 727        | 27.8 TIME_WAIT 定时器 .....           | 766        |
| 26.8.1 输入 TCP 段的校验和检测 .....                | 728        | <b>第 28 章 TCP 连接的建立 .....</b>      | <b>767</b> |
| 26.8.2 输出 TCP 段校验和的计算 .....                | 729        | 28.1 服务端建立连接过程 .....               | 767        |
| 26.9 TCP 的初始化 .....                        | 729        | 28.2 连接相关的数据结构 .....               | 770        |
| 26.10 TCP 传输控制块的管理 .....                   | 731        | 28.2.1 request_sock_queue 结构 ..... | 770        |
| 26.10.1 inet_hashinfo 结构 .....             | 732        | 28.2.2 listen_sock 结构 .....        | 771        |
| 26.10.2 管理除 LISTEN 状态之外的 TCP 传输控制块 .....   | 733        | 28.2.3 tcp_request_sock 结构 .....   | 771        |
| 26.10.3 管理 LISTEN 状态的 TCP 传输控制块 .....      | 734        | 28.2.4 request_sock_ops 结构 .....   | 774        |
| 26.11 TCP 层的套接口选项 .....                    | 735        | 28.3 bind 系统调用的实现 .....            | 775        |
| 26.12 TCP 的 ioctl .....                    | 736        | 28.3.1 bind 端口散列表 .....            | 775        |
| 26.13 TCP 传输控制块的初始化 .....                  | 737        | 28.3.2 传输接口层的实现 .....              | 775        |
| 26.14 TCP 的差错处理 .....                      | 737        | 28.4 listen 系统调用的实现 .....          | 779        |
| 26.15 TCP 传输控制块层的缓存管理 .....                | 741        | 28.4.1 inet_listen() .....         | 779        |
| 26.15.1 缓存管理的算法 .....                      | 741        | 28.4.2 实现侦听:                       |            |
| 26.15.2 发送缓存的管理 .....                      | 744        | inet_csk_listen_start() .....      | 780        |
| 26.15.3 接收缓存的管理 .....                      | 745        | 28.4.3 分配连接请求块散列表:                 |            |
| <b>第 27 章 TCP 的定时器 .....</b>               | <b>746</b> | reqsk_queue_alloc() .....          | 781        |
| 27.1 初始化 .....                             | 746        | 28.5 accept 系统调用的实现 .....          | 782        |
| 27.2 连接建立定时器 .....                         | 747        | 28.5.1 套接口层的实现:                    |            |
| 27.2.1 连接建立定时器处理函数 .....                   | 747        | inet_accept() .....                | 782        |
| 27.2.2 连接建立定时器的激活 .....                    | 751        | 28.5.2 传输接口层的实现:                   |            |
| 27.3 重传定时器 .....                           | 751        | inet_csk_accept() .....            | 783        |
| 27.3.1 重传定时器处理函数 .....                     | 751        | 28.6 被动打开 .....                    | 785        |
| 27.3.2 重传定时器的激活 .....                      | 756        | 28.6.1 SYN cookies .....           | 785        |
| 27.4 延迟确认定时器 .....                         | 756        | 28.6.2 第一次握手: 接收 SYN 段 .....       | 786        |
| 27.4.1 延时确认定时器的处理函数 .....                  | 756        | 28.6.3 第二次握手:                      |            |
| 27.4.2 延时确认定时器的激活 .....                    | 758        | 发送 SYN+ACK 段 .....                 | 793        |
| 27.5 持续定时器 .....                           | 758        | 28.6.4 第三次握手: 接收 ACK 段 .....       | 798        |
| 27.5.1 持续定时器处理函数 .....                     | 758        | 28.7 connect 系统调用的实现 .....         | 813        |
| 27.5.2 激活持续定时器 .....                       | 762        | 28.7.1 套接口层的实现:                    |            |
| 27.6 保活定时器 .....                           | 763        | inet_stream_connect() .....        | 813        |
| 27.6.1 保活定时器处理函数 .....                     | 763        | 28.7.2 传输接口层的实现 .....              | 815        |
| 27.6.2 激活保活定时器 .....                       | 764        | 28.8 主动打开 .....                    | 816        |
|  |            | 28.8.1 第一次握手: 发送 SYN 段 .....       | 816        |
|  |            | 28.8.2 第二次握手:                      |            |
|  |            | 接收 SYN+ACK 段 .....                 | 823        |
|  |            | 28.8.3 第三次握手: 发送 ACK 段 .....       | 828        |

|                                    |     |                              |     |
|------------------------------------|-----|------------------------------|-----|
| 28.9 同时打开 .....                    | 828 | 29.9.4 选取某种拥塞控制算法: tcp_set_  |     |
| 28.9.1 SYN_SENT 状态接收 SYN 段 ..      | 828 | congestion_control() .....   | 861 |
| 28.9.2 SYN_RECV 状态接收               |     | 29.9.5 Linux 支持的拥塞控制算法 ..... | 862 |
| SYN+ACK 段 .....                    | 830 | <b>第 30 章 TCP 的输出</b> .....  | 864 |
| <b>第 29 章 TCP 拥塞控制的实现</b> .....    | 831 | 30.1 引言 .....                | 864 |
| 29.1 拥塞控制引擎 .....                  | 831 | 30.2 最大段长度 (MSS) .....       | 867 |
| 29.2 拥塞控制状态机 .....                 | 832 | 30.3 sendmsg 系统调用在 TCP 中的    |     |
| 29.2.1 Open 状态 .....               | 833 | 实现 .....                     | 870 |
| 29.2.2 Disorder 状态 .....           | 833 | 30.3.1 分割 TCP 段 .....        | 871 |
| 29.2.3 CWR 状态 .....                | 833 | 30.3.2 套接口层的实现 .....         | 871 |
| 29.2.4 Recovery 状态 .....           | 834 | 30.3.3 传输接口层的实现 .....        | 871 |
| 29.2.5 Loss 状态 .....               | 834 | 30.4 对 TCP 选项的处理 .....       | 889 |
| 29.3 拥塞窗口调整撤销 .....                | 836 | 30.4.1 构建 SYN 段的选项 .....     | 889 |
| 29.3.1 撤销拥塞窗口的检测 .....             | 837 | 30.4.2 构建非 SYN 段的选项 .....    | 892 |
| 29.3.2 tcp_undo_cwr() .....        | 837 | 30.5 Nagle 算法 .....          | 893 |
| 29.3.3 从 Disorder 拥塞状态撤销 .....     | 838 | 30.6 ACK 的接收 .....           | 894 |
| 29.3.4 从 Recovery 状态撤销 .....       | 838 | 30.6.1 tcp_ack() .....       | 894 |
| 29.3.5 从 Recovery 拥塞状态撤销 .....     | 839 | 30.6.2 发送窗口的更新 .....         | 899 |
| 29.3.6 从 Loss 拥塞状态撤销 .....         | 839 | 30.6.3 根据 SACK 选项标记重传        |     |
| 29.4 显式拥塞通知 .....                  | 840 | 队列中段的记分牌 .....               | 900 |
| 29.4.1 IP 对 ECN 的支持 .....          | 841 | 30.6.4 重传队列中已经确认段的删除 .....   | 910 |
| 29.4.2 TCP 对 ECN 的支持 .....         | 841 | 30.7 往返时间测量和 RTO 的计算 ..      | 913 |
| 29.5 拥塞控制状态的处理及转换 ..               | 843 | 30.8 路径 MTU 发现 .....         | 915 |
| 29.5.1 拥塞控制状态的处理:                  |     | 30.8.1 路径 MTU 发现原理 .....     | 915 |
| tcp_fastretrans_alert() .....      | 843 | 30.8.2 路径 MTU 发现时的黑洞 .....   | 916 |
| 29.5.2 拥塞避免 .....                  | 852 | 30.8.3 有关数据结构的初始化 .....      | 916 |
| 29.6 拥塞窗口的检测:                      |     | 30.8.4 创建路径 MTU 发现 TCP 段并    |     |
| tcp_cwnd_test() .....              | 852 | 发送 .....                     | 916 |
| 29.7 F-RTO 算法 .....                | 853 | 30.8.5 路径 MTU 发现失败后处理 .....  | 920 |
| 29.7.1 进入 F-RTO 算法处理阶段 .....       | 853 | 30.8.6 处理需要分片 ICMP 目的        |     |
| 29.7.2 进行 F-RTO 算法处理 .....         | 855 | 不可达报文 .....                  | 920 |
| 29.8 拥塞窗口的检验 .....                 | 857 | 30.8.7 更新当前有效的 MSS .....     | 921 |
| 29.8.1 tcp_event_data_sent() ..... | 857 | 30.8.8 路径 MTU 发现成功后处理 .....  | 922 |
| 29.8.2 tcp_cwnd_validate() .....   | 858 | 30.9 TCP 重传接口 .....          | 922 |
| 29.9 支持多拥塞控制算法的机制 ..               | 859 | <b>第 31 章 TCP 的输入</b> .....  | 926 |
| 29.9.1 接口 .....                    | 859 | 31.1 引言 .....                | 926 |
| 29.9.2 注册拥塞控制算法: tcp_register_     |     | 31.2 TCP 接收的总入口 .....        | 927 |
| congestion_control() .....         | 861 | 31.2.1 接收到 prequeue 队列 ..... | 930 |
| 29.9.3 注销拥塞控制算法: tcp_unregister_   |     | 31.2.2 有效 TCP 段的处理 .....     | 931 |
| congestion_control() .....         | 861 | 31.3 报文的过滤 .....             | 932 |



|   |   |     |                                |  |      |
|---|---|-----|--------------------------------|--|------|
| 31.3.1                                    | 过滤器的数据结构 .....                              | 933 | 31.10.1                        | 套接口层的实现 .....  | 980  |
| 31.3.2                                    | 安装过滤器 .....                                 | 935 | 31.10.2                        | 传输接口层的实现 .....   | 980  |
| 31.3.3                                    | 卸载过滤器 .....                                 | 937 | 31.11                          | sk_backlog_rcv 接口 .....                                      | 991  |
| 31.3.4                                    | 过滤执行 .....                                  | 938 | <b>第 32 章 TCP 连接的终止</b> .....  | <b>992</b>   |      |
| <b>31.4 ESTABLISHED 状态的接收</b> .....       | <b>938</b>                                  |     | 32.1                           | 连接终止过程 .....   | 993  |
| 31.4.1                                    | 首部预测 .....                                  | 939 | 32.1.1                         | 正常关闭 .....   | 993  |
| 31.4.2                                    | 接收处理无负荷的 ACK 段 .....                        | 941 | 32.1.2                         | 同时关闭 .....   | 994  |
| 31.4.3                                    | 执行快速路径 .....                                | 942 | 32.2                           | shutdown 传输接口层的实现 .....                                      | 994  |
| 31.4.4                                    | 执行慢速路径 .....                                | 945 | 32.2.1                         | tcp_shutdown() .....   | 994  |
| 31.4.5                                    | 数据从内核空间复制到用户空间 .....                        | 948 | 32.2.2                         | tcp_send_fin() .....   | 995  |
| 31.4.6                                    | 通过调节接收窗口进行流量控制 .....                        | 952 | 32.3                           | close 传输接口层的实现: tcp_close() .....                            | 995  |
| 31.4.7                                    | 确定是否需要发送 ACK 段 (用于接收的数据从内核空间复制到用户空间时) ..... | 956 | 32.4                           | 被动关闭: FIN 段的接收处理 .....                                       | 999  |
| <b>31.5 TCP 选项的处理</b> .....               | <b>957</b>                                  |     | 32.5                           | 主动关闭 .....   | 1002 |
| 31.5.1                                    | 慢速路径中快速解析 TCP 选项 .....                      | 957 | 32.5.1                         | timewait 控制块的数据结构 .....                                      | 1002 |
| 31.5.2                                    | 全面解析 TCP 选项 .....                           | 958 | 32.5.2                         | timewait 控制块取代 TCP 传输控制块 .....                               | 1006 |
| <b>31.6 慢速路径的数据处理</b> .....               | <b>961</b>                                  |     | 32.5.3                         | 启动 FIN_WAIT_2 或 TIME_WAIT 定时器 .....                          | 1008 |
| 31.6.1                                    | 接收处理预期的段 .....                              | 963 | 32.5.4                         | CLOSE_WAIT、LAST_ACK、FIN_WAIT1、FIN_WAIT2 与 CLOSING 状态处理 ..... | 1010 |
| 31.6.2                                    | 接收处理在接收窗口之外的段 .....                         | 965 | 32.5.5                         | FIN_WAIT2 和 TIME_WAIT 状态处理 .....                             | 1013 |
| 31.6.3                                    | 接收处理乱序的段 .....                              | 966 | 32.5.6                         | timewait 控制块的 2MSL 超时处理 .....                                | 1020 |
| 31.6.4                                    | tcp_ofo_queue() .....                       | 969 | <b>第 33 章 UDP: 用户数据报</b> ..... | <b>1023</b>  |      |
| <b>31.7 带外数据处理</b> .....                  | <b>970</b>                                  |     | 33.1                           | 引言 .....   | 1023 |
| 31.7.1                                    | 检测紧急指针 .....                                | 970 | 33.1.1                         | UDP 首部 .....   | 1023 |
| 31.7.2                                    | 读取带外数据 .....                                | 972 | 33.1.2                         | UDP 的输入与输出 .....   | 1024 |
| <b>31.8 SACK 信息</b> .....                 | <b>973</b>                                  |     | 33.2                           | UDP 的 inet_protosw 结构 .....                                  | 1024 |
| 31.8.1                                    | SACK 允许选项 .....                             | 973 | 33.3                           | UDP 的传输控制块 .....   | 1025 |
| 31.8.2                                    | SACK 选项 .....                               | 974 | 33.4                           | UDP 的 proto 结构和 proto_ops 结构的实例 .....                        | 1027 |
| 31.8.3                                    | SACK 的产生 .....                              | 974 | 33.5                           | UDP 的状态 .....  | 1027 |
| 31.8.4                                    | 发送方对 SACK 的响应 .....                         | 975 | 33.6                           | UDP 传输控制块的管理 .....   | 1027 |
| 31.8.5                                    | 实现 .....                                    | 975 | 33.7                           | bind 系统调用的实现 .....   | 1028 |
| <b>31.9 确认的发送</b> .....                   | <b>975</b>                                  |     |                                |  |      |
| 31.9.1                                    | 快速确认模式 .....                                | 976 |                                |  |      |
| 31.9.2                                    | 处理数据接收事件 .....                              | 977 |                                |  |      |
| 31.9.3                                    | 发送确认紧急程度和状态 .....                           | 978 |                                |  |      |
| 31.9.4                                    | 延迟或快速确认 .....                               | 979 |                                |  |      |
| <b>31.10 recvmsg 系统调用在 TCP 中的实现</b> ..... | <b>980</b>                                  |     |                                |  |      |



|         |                                 |      |                                |                               |      |
|---------|---------------------------------|------|--------------------------------|-------------------------------|------|
| 33.8    | UDP 套接口的关闭 .....                | 1031 | 33.14.1                        | udp_sendmsg() .....           | 1040 |
| 33.9    | connect 系统调用的实现 .....           | 1032 | 33.14.2                        | udp_push_pending_frames() ... | 1047 |
| 33.9.1  | udp_disconnect() .....          | 1033 | 33.15                          | UDP 的输入 .....                 | 1048 |
| 33.9.2  | ip4_datagram_connect() .....    | 1033 | 33.15.1                        | UDP 接收的入口:                    |      |
| 33.10   | select 系统调用的实现 .....            | 1034 | udp_rcv() .....                | 1048                          |      |
| 33.11   | UDP 的 ioctl .....               | 1037 | 33.15.2                        | UDP 组播数据报输入:                  |      |
| 33.12   | UDP 的套接口选项 .....                | 1037 | __udp4_lib_mcast_deliver() ... | 1052                          |      |
| 33.13   | UDP 校验和 .....                   | 1038 | 33.15.3                        | udp_queue_rcv_skb() .....     | 1053 |
| 33.13.1 | 输入 UDP 数据报校验和的<br>计算 .....      | 1038 | 33.16                          | recvmsg 系统调用的实现 ...           | 1055 |
| 33.13.2 | 输出 UDP 数据报校验和的<br>计算 .....      | 1039 | 33.17                          | UDP 的差错处理:                    |      |
| 33.14   | UDP 的输出: sendmsg 系统<br>调用 ..... | 1040 | udp_err() .....                | 1059                          |      |
|         |                                 |      | 33.18                          | 轻量级 UDP .....                 | 1061 |
|         |                                 |      | 参考文献                           | .....                         | 1063 |

## 第20章 路由缓存

路由缓存用于提高路由查找的命中率，减少路由表查找的时间。缓存的主要工作是存储使路由子系统能够找到报文目的地的信息，并通过一组函数向更高层提供该信息。路由缓存还提供了—些函数用于缓存的清理。路由缓存存储的信息为可应用于所有三层协议的路由缓存项，所以可以包含表示路由缓存项的任意数据结构。

路由缓存项的管理及查找会涉及以下文件：

- `include/net/route.h`，定义目的路由缓存项的部分结构、宏、函数等。
- `include/net/flow.h`，定义查询路由缓存的条件组合结构、宏和函数原型等。
- `include/net/dst.h`，定义目的路由缓存项的部分结构、宏、函数等。
- `net/ipv4/route.c`，实现路由缓存项的操作函数。

### 20.1 系统参数

系统参数如下：

(1) `flush`，控制路由缓存的刷新。当对参数进行写操作时，并不是简单地调整参数，而是当用户将 `n` 写入该文件时，会触发一个动作，`ipv4_sysctl_rtcache_flush()`被调用。将在 `n` 秒后对路由表缓存进行一次刷新。当写入一个负值到 `flush` 中时，内核调度在默认延迟 `min_delay` 秒之后进行一次刷新。

(2) `gc_elasticity`，在添加路由缓存项到缓存时，当路由缓存散列表桶链表长度超过此值时，便会释放最老的缓存项，参见 20.5 节。

(3) `gc_min_interval` 与 `gc_min_interval_ms`，用于控制路由缓存垃圾回收的频率和行为，`gc_min_interval_ms` 标识路由表垃圾回收的最小间隔，不得小于 0.5s。

`gc_min_interval` 已废弃，被 `gc_min_interval_ms` 取代。

(4) `gc_thresh`，用来控制路由缓存垃圾回收机制的频率和行为。当缓存中的路由条数超过此值时，开始回收垃圾。

(5) `max_delay`

(6) `min_delay`，分别是在用户调度进行刷新和内核实际刷新缓存之间的最短和最长时间间隔，`min_delay` 默认为 2s，而 `max_delay` 默认为 10s。

(7) `max_size`，路由缓存数量的最大值，该值在初始化路由模块时被设置。当缓存数量达到该值后，老的路由表项会被清除。

(8) `min_adv_mss`，用于初始化路由表项的度量值中的 MSS 值，取设备的 MTU 和 `min_adv_mss` 之间的较大值。

(9) `mtu_expires`，PMTU 缓存在路由缓存项度量值中的时间，超过该值后会过期，参见 20.8.1 节。默认值为 600s。

(10) `min_pmtu`，路径 MTU 发现协议可以为路由表项设置的最小 PMTU 值。

(11) `secret_interval`，定时刷新路由缓存的周期，每隔 `secret_interval` 秒会刷新一次，参见

20.9 节。

## 20.2 路由缓存的组织结构

如图 20-1 所示，路由缓存是用一张散列表来实现的，路由缓存散列表的类型是 `rt_hash_bucket` 结构，该结构只包含指向缓存元素链表的一个指针。缓存项的类型为 `rtable` 结构。

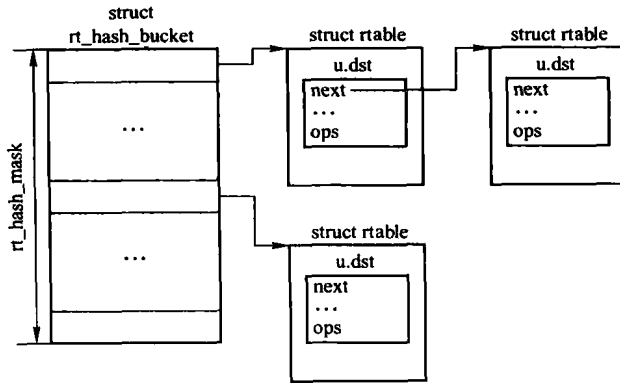


图 20-1 路由缓存结构

`dst_entry` 结构包含了缓存与邻居层的接口、transformers（诸如 `Ipsec`）以及路由缓存管理。

### 20.2.1 rtable 结构

IPv4 使用 `rtable` 结构来存储缓存内的路由表项。可以通过查看 `/proc/net/rtable` 文件，或者通过 `ip route list cache` 和 `route -C` 命令来列出路由缓存的内容。

```

51 struct rtable
52 {
53     union
54     {
55         struct dst_entry    dst;
56         struct rtable      *rt_next;
57     } u;
58
59     struct in_device      *idev;
60
61     unsigned              rt_flags;
62     __u16                  rt_type;
63     __u16                  rt_multipath_alg;
64
65     __be32                 rt_dst;    /* Path destination */
66     __be32                 rt_src;    /* Path source */
67     int                    rt_iif;
68
69     /* Info on neighbour */
70     __be32                 rt_gateway;
71
72     /* Cache lookup keys */
73     struct flowi            fl;

```

```

74
75  /* Miscellaneous cached information */
76  __be32      rt_spec_dst; /* RFC1122 specific destination */
77  struct inet_peer *peer; /* long-living peer info */
78 };

```

53 union {...} u

`dst_entry` 结构作为一部分嵌入到 `rtable` 结构中，而 `dst_entry` 结构中的第一个成员 `next` 就是用于链接分布在同一个散列桶内的 `rtable` 实例，为了便于访问 `next`，因此将 `dst` 和 `rt_next` 联合起来。虽然指针的名称不同，但它们所指向的内存位置是相同的。

59 struct in\_device \*idev

指向输出网络设备的 IPv4 协议族中的 IP 配置块。注意：对送往本地的输入报文的路由，输出网络设备设置为回环设备。

61 unsigned rt\_flags

用于标识路由表项的一些特性和标志，见表 20-1。

表 20-1 路由表项的标志

| rt_flags                           | 描述   |
|------------------------------------|--|
| RTCF_NOTIFY                        | 路由表项的所有变化通过 <code>netlink</code> 通知给感兴趣的用户空间应用程序，该选项还没有完全实现。利用诸如 <code>ip route</code> 等命令来设置该标志   |
| RTCF_REDIRECTED                    | 由接收到的 ICMP REDIRECT 消息作出响应而添加一条路由缓存项，参见 20.10 节  |
| RTCF_DOREDIRECT                    | 表示并不是最优路由。 <code>ip_forward()</code> 依据该标志和其他信息，决定是否需要发送 ICMP 重定向消息。例如，如果报文是基于源站的路由，就不应当生成 ICMP 重定向消息  |
| RTCF_DIRECTSRC                     | 不正确的源地址。ICMP 模块不会对来自此源地址的地址掩码请求消息作出回应。每当调用 <code>fib_validate_source()</code> 检查到接收报文的源地址通过一个本地作用范围 ( <code>RT_SCOPE_HOST</code> ) 的下一跳是可达时，就设置该标志 |
| RTCF_SNAT<br>RTCF_DNAT<br>RTCF_NAT | 已废除  |
| RTCF_BROADCAST                     | 路由的目的地址是一个广播地址   |
| RTCF_MULTICAST                     | 路由的目的地址是一个多播地址   |
| RTCF_LOCAL                         | 路由的目的地址是一个本地地址（即本地接口上配置的某个地址）。对本地广播地址和本地多播地址也设置该标志，参见 <code>ip_route_input_slow()</code> 和 <code>ip_route_input_mc()</code>                        |
| RTCF_REJECT                        | 未被使用。依据 IPROUTE2 软件包的 <code>ip rule</code> 命令的语法，在该命令中有一个关键字 <code>reject</code> ，但该关键字还未被接受   |
| RTCF_TPROXY                        | 未使用  |
| RTCF_DIRECTDST                     | 未使用  |
| RTCF_FAST                          | 已废除  |
| RTCF_MASQ                          | IPv4 已不使用  |

62 \_\_u16 rt\_type

路由表项的类型，见表 20-2。它间接定义了当路由查找匹配时应采取的动作。

表 20-2 路由类型

| rt_type    | 描述   |
|------------|--|
| RTN_UNSPEC | 定义一个未初始化的值。例如，当从路由表中删除一个表项时使用该值，这是因为删除操作不需要指定路由表项的类型 |

(续)

| rt_type   | 描述   |
|---|--|
| RTN_LOCAL   | 目的地址被配置为一个本地接口的地址  |
| RTN_UNICAST   | 该路由是一条到单播地址的直连或非直连（通过一个网关）路由。当用户通过 <code>ip route</code> 命令添加路由但没有指定其他路由类型时，路由类型默认设置为 <code>RTN_UNICAST</code> |
| RTN_MULTICAST   | 目的地址是一个多播地址  |
| RTN_BROADCAST   | 目的地址是一个广播地址。匹配的 <code>ingress</code> 报文以广播方式送往本地，匹配的 <code>egress</code> 报文以广播方式发送出去                           |
| RTN_ANYCAST   | 匹配的输入报文以广播方式送往本地，匹配的输入报文以单播发送出去。IPv4 没有该类型   |
| RTN_BLACKHOLE<br>RTN_UNREACHABLE<br>RTN_PROHIBIT<br>RTN_THROW | 这些值与特定的管理配置而不是与目的地址类型相关联   |
| RTN_NAT   | 已废弃  |
| RTN_XRESOLVE  | 有一个外部解析器来处理该路由，目前尚未实现该功能   |

63 `__u16 rt_multipath_alg`

标识多路径缓存算法，在创建路由表项时根据相关路由项的配置来设置。

65 `__be32 rt_dst`

66 `__be32 rt_src`

目的 IP 地址和源 IP 地址。

67 `int rt_iif`

输入网络设备标识，从输入网络设备的 `net_device` 数据结构中得到。对本地生成的流量（因此不是从任何接口上接收到的），该字段被设置为出设备的 `ifindex` 字段。对本地生成的报文，`fl` 中的 `iif` 字段被设置为 0。

70 `__be32 rt_gateway`

当目的主机为直连时，即在同一链路上，`rt_gateway` 表示目的地址。当需要通过一个网关到达目的地时，`rt_gateway` 被设置为路由项中的下一跳的网关。

73 `struct flowi fl`

用于缓存查找的搜索的条件组合，参见 20.2.2 节。

76 `__be32 rt_spec_dst`

首选源地址。

添加到路由缓存内的路由缓存项是单向的。但是在一些情况下，接收到报文可能触发一个动作，要求本地主机选择一个源 IP 地址，以便在向发送方回送报文时使用。这个地址，即首选源 IP 地址，必须与路由该输入报文的由路由缓存项保存在一起。首选源 IP 地址被保存在 `rt_spec_dst` 字段内，下面是使用该地址的两种情况：

1) 当一个主机接收到一个 ICMP 回显请求消息时（常用的 `ping` 命令），如果主机没有明确配置为不作出回应，则该主机返回一个 ICMP 回显应答消息。对该输入 ICMP 回显请求消息选择路由，路由项的 `rt_spec_dst` 被用作路由 ICMP 回显请求消息而进行路由查找的源地址。参见 14.6.2 节的 `icmp_reply()` 和 11.11.2 节的 `ip_send_reply()`。

2) 记录路由 IP 选项和时间戳 IP 选项要求途经主机的 IP 地址记录到选项中。

77 `struct inet_peer *peer`

指向与目的地址相关的对端信息块。

## 20.2.2 flowi 结构

利用 flowi 数据结构, 就可以根据诸如输入网络设备和输出网络设备、三层和四层协议报头中的参数等字段的组合对流量进行分类。它通常被用作路由查找的搜索条件组合, IPsec 策略的流量选择器以及其他高级用途。

```
13 struct flowi {
14     int    oif;
15     int    iif;
16     __u32  mark;
17
18     union {
19         struct {
20             __be32      daddr;
21             __be32      saddr;
22             __u8        tos;
23             __u8        scope;
24         } ip4_u;
25
26         struct {
27             struct in6_addr    daddr;
28             struct in6_addr    saddr;
29             __be32            flowlabel;
30         } ip6_u;
31
32         struct {
33             __le16      daddr;
34             __le16      saddr;
35             __u8        scope;
36         } dn_u;
37     } nl_u;
38 #define fld_dst      nl_u.dn_u.daddr
39 #define fld_src      nl_u.dn_u.saddr
40 #define fld_scope    nl_u.dn_u.scope
41 #define fl6_dst      nl_u.ip6_u.daddr
42 #define fl6_src      nl_u.ip6_u.saddr
43 #define fl6_flowlabel  nl_u.ip6_u.flowlabel
44 #define fl4_dst      nl_u.ip4_u.daddr
45 #define fl4_src      nl_u.ip4_u.saddr
46 #define fl4_tos      nl_u.ip4_u.tos
47 #define fl4_scope    nl_u.ip4_u.scope
48
49     __u8  proto;
50     __u8  flags;
51 #define FLOWI_FLAG_MULTIPATHOLDROUTE 0x01
52     union {
53         struct {
54             __be16  sport;
55             __be16  dport;
56         } ports;
57
58         struct {
```

```

59     __u8    type;
60     __u8    code;
61     } icmp;
62
63     struct {
64         __le16    sport;
65         __le16    dport;
66     } dnports;
67
68     __be32    spi;
69
70 #ifdef CONFIG_IPV6_MIP6
71     struct {
72         __u8    type;
73     } mht;
74 #endif
75     } uli_u;
76 #define fl_ip_sport    uli_u.ports.sport
77 #define fl_ip_dport    uli_u.ports.dport
78 #define fl_icmp_type    uli_u.icmp.type
79 #define fl_icmp_code    uli_u.icmp.code
80 #define fl_ipsec_spi    uli_u.spi
81 #ifdef CONFIG_IPV6_MIP6
82 #define fl_mh_type    uli_u.mht.type
83 #endif
84     __u32    secid;    /* used by xfrm; see secid.txt */
85 } __attribute__((__aligned__(BITS_PER_LONG/8)));

```

```
14 int oif
```

```
15 int iif
```

输出网络设备索引和输入网络设备索引。

```
18 union {...} nl_u
```

该联合对应三层，目前支持的协议为 IPv4、IPv6 和 DECnet。

这里应该说明一下 RTO\_ONLINK 标志，该标志通过 TOS 变量传递，但该变量与 IP 包头中的 TOS 域无关。此处只使用 TOS 域的一个无用的比特位。当该标志被设置时，表示目的地位于本地子网，所以无须路由查找。

```
49 __u8 proto
```

标识四层协议。

```
50 __u8 flags
```

该变量只定义了一个标志，FLOWI\_FLAG\_MULTIPATHOLDROUTE，它最初用于多路径代码，但现已废弃不再使用。

```
52 union {...} uli_u
```

该联合对应四层，目前支持的协议为 TCP、UDP、ICMP、DECnet 和 IPsec。

### 20.2.3 dst\_entry 结构

dst\_entry 结构被用于存储缓存路由项中独立于协议的信息。三层协议在另外的结构中存储本协议中更多的私有信息（例如，IPv4 使用 rtable 结构）。



```

38 struct dst_entry
39 {
40     struct dst_entry    *next;
41     atomic_t            __refcnt; /* client references */
42     int                 __use;
43     struct dst_entry    *child;
44     struct net_device   *dev;
45     short               error;
46     short               obsolete;
47     int                 flags;
48 #define DST_HOST        1
49 #define DST_NOXFRM      2
50 #define DST_NOPOLICY    4
51 #define DST_NOHASH      8
52 #define DST_BALANCED    0x10
53     unsigned long       lastuse;
54     unsigned long       expires;
55
56     unsigned short      header_len; /* more space at head required */
57     unsigned short      nfheader_len; /* more non-fragment space at head
    required */
58     unsigned short      trailer_len; /* space to reserve at tail */
59
60     u32                 metrics[RTAX_MAX];
61     struct dst_entry    *path;
62
63     unsigned long       rate_last; /* rate limiting for ICMP */
64     unsigned long       rate_tokens;
65
66     struct neighbour    *neighbour;
67     struct hh_cache     *hh;
68     struct xfrm_state   *xfrm;
69
70     int                 (*input)(struct sk_buff*);
71     int                 (*output)(struct sk_buff*);
72
73 #ifdef CONFIG_NET_CLS_ROUTE
74     __u32                tclassid;
75 #endif
76
77     struct dst_ops      *ops;
78     struct rcu_head     rcu_head;
79
80     char                info[0];
81 };

```

40 struct dst\_entry \*next

用于将分布在同一个散列表桶内的 dst\_entry 实例链接在一起。

41 atomic\_t refcnt

引用计数。

42 int \_\_use

该表项已经被使用的次数（即缓存查找返回该表项的次数）。

**注意：**不要将这个值与 rt\_cache\_stat[smp\_processor\_id( )].in\_hit 混淆：后者表示针对某个

CPU 的全局缓存命中次数。

```
43 struct dst_entry *child
56 unsigned short header_len
57 unsigned short nfheader_len
58 unsigned short trailer_len
61 struct dst_entry *path
68 struct xfrm_state *xfrm
```

这些字段与 IPsec 相关，本书不作论述。

```
44 struct net_device *dev
45 short error
```

当 `fib_lookup()` 查找失败时，错误码值会被保存在这个字段中，在之后 `ip_error()` 中使用该值来决定如何处理本次路由查找失败，即决定生成哪一类 ICMP 消息。

```
46 short obsolete
```

用于标识本 `dst_entry` 实例的可用状态，见表 20-3。

表 20-3 obsolete 取值

| obsolete | 描述                           |
|----------|------------------------------|
| 0 (默认值)  | 表示所在结构实例有效而且可以被使用            |
| 2        | 表示所在结构实例将被删除因而不能被使用          |
| -1       | 被 IPsec 和 IPv6 使用但不被 IPv4 使用 |

```
47 int flags
```

标志集合，见表 20-4。

表 20-4 flags 取值

| flags                                    | 描述                          |
|--|-----------------------------|
| DST_HOST                                 | 表示主机路由，即不是到网络或到一个广播/多播地址的路由 |
| DST_NOXFRM<br>DST_NOPOLICY<br>DST_NOHASH | 只用于 IPsec，本书不作论述            |

```
53 unsigned long lastuse
```

记录该表项最后一次被使用的时间戳。当缓存查找成功时更新该时间戳，垃圾回收程序使用该时间戳来决定最应该被释放表项。

```
54 unsigned long expires
```

表示该表项将过期的时间戳。

```
60 u32 metrics[RTAX_MAX]
```

多种度量值，TCP 中多处使用。

```
63 unsigned long rate_last
```

```
64 unsigned long rate_tokens
```

这两个字段被用于对两种类型的 ICMP 消息限速。