目　　录

前言

第一篇　TCP/IP协议详解

第1章　TCP/IP协议族 / 2

1.1　TCP/IP协议族体系结构以及主要协议 / 2
　1.1.1　数据链路层 / 2
　1.1.2　网络层 / 3
　1.1.3　传输层 / 4
　1.1.4　应用层 / 5
1.2　封装 / 6
1.3　分用 / 7
1.4　测试网络 / 8
1.5　ARP协议工作原理 / 9
　1.5.1　以太网ARP请求/应答报文详解 / 9
　1.5.2　ARP高速缓存的查看和修改 / 10
　1.5.3　使用tcpdump观察ARP通信过程 / 10
1.6　DNS工作原理 / 12
　1.6.1　DNS查询和应答报文详解 / 12
　1.6.2　Linux下访问DNS服务 / 14
　1.6.3　使用tcpdump观察DNS通信过程 / 15
1.7　socket和TCP/IP协议族的关系 / 16

第2章　IP协议详解 / 17

2.1　IP服务的特点 / 17
2.2　IPv4头部结构 / 18
　2.2.1　IPv4头部结构 / 18
　2.2.2　使用tcpdump观察IPv4头部结构 / 20
2.3　IP分片 / 21
2.4　IP路由 / 22
　2.4.1　IP模块工作流程 / 23
　2.4.2　路由机制 / 24
　2.4.3　路由表更新 / 25
2.5　IP转发 / 25
2.6　重定向 / 26
　2.6.1　ICMP重定向报文 / 26
　2.6.2　主机重定向实例 / 27
2.7　IPv6头部结构 / 27
　2.7.1　IPv6固定头部结构 / 28
　2.7.2　IPv6扩展头部 / 29

第3章　TCP协议详解 / 30

3.1　TCP服务的特点 / 30
3.2　TCP头部结构 / 32
　3.2.1　TCP固定头部结构 / 32
　3.2.2　TCP头部选项 / 33
　3.2.3　使用tcpdump观察TCP头部信息 / 35
3.3　TCP连接的建立和关闭 / 37
　3.3.1　使用tcpdump观察TCP连接的建立和关闭 / 37
　3.3.2　半关闭状态 / 39
　3.3.3　连接超时 / 39
3.4　TCP状态转移 / 40
　3.4.1　TCP状态转移总图 / 41
　3.4.2　TIME_WAIT状态 / 43
3.5　复位报文段 / 44
　3.5.1　访问不存在的端口 / 44
　3.5.2　异常终止连接 / 45

3.5.3 处理半打开连接 / 45
3.6 TCP交互数据流 / 46
3.7 TCP成块数据流 / 48
3.8 带外数据 / 50
3.9 TCP超时重传 / 51
3.10 拥塞控制 / 53
3.10.1 拥塞控制概述 / 53
3.10.2 慢启动和拥塞避免 / 54
3.10.3 快速重传和快速恢复 / 55

第4章 TCP/IP通信案例：访问Internet上的Web服务器 / 57

4.1 实例总图 / 57
4.2 部署代理服务器 / 58
4.2.1 HTTP代理服务器的工作原理 / 58
4.2.2 部署squid代理服务器 / 59
4.3 使用tcpdump抓取传输数据包 / 60
4.4 访问DNS服务器 / 62
4.5 本地名称查询 / 63
4.6 HTTP通信 / 64
4.6.1 HTTP请求 / 65
4.6.2 HTTP应答 / 66
4.7 实例总结 / 68

第二篇 深入解析高性能服务器编程

第5章 Linux网络编程基础API / 70

5.1 socket地址API / 70
5.1.1 主机字节序和网络字节序 / 70
5.1.2 通用socket地址 / 71
5.1.3 专用socket地址 / 72
5.1.4 IP地址转换函数 / 73
5.2 创建socket / 74
5.3 命名socket / 75
5.4 监听socket / 76

5.5 接受连接 / 78
5.6 发起连接 / 80
5.7 关闭连接 / 80
5.8 数据读写 / 81
5.8.1 TCP数据读写 / 81
5.8.2 UDP数据读写 / 85
5.8.3 通用数据读写函数 / 86
5.9 带外标记 / 87
5.10 地址信息函数 / 87
5.11 socket选项 / 87
5.11.1 SO_REUSEADDR选项 / 89
5.11.2 SO_RCVBUF和SO_SNDBUF选项 / 89
5.11.3 SO_RCVLOWAT和SO_SNDLOWAT选项 / 93
5.11.4 SO_LINGER选项 / 93
5.12 网络信息API / 94
5.12.1 gethostbyname和gethostbyaddr / 94
5.12.2 getservbyname和getservbyport / 95
5.12.3 getaddrinfo / 96
5.12.4 getnameinfo / 98

第6章 高级I/O函数 / 100

6.1 pipe函数 / 100
6.2 dup函数和dup2函数 / 101
6.3 readv函数和writev函数 / 103
6.4 sendfile函数 / 106
6.5 mmap函数和munmap函数 / 107
6.6 splice函数 / 108
6.7 tee函数 / 110
6.8 fcntl函数 / 112

第7章 Linux服务器程序规范 / 114

7.1 日志 / 114
7.1.1 Linux系统日志 / 114
7.1.2 syslog函数 / 115
7.2 用户信息 / 116

7.2.1 UID、EUID、GID和EGID / 116
7.2.2 切换用户 / 117
7.3 进程间关系 / 118
7.3.1 进程组 / 118
7.3.2 会话 / 118
7.3.3 用ps命令查看进程关系 / 119
7.4 系统资源限制 / 119
7.5 改变工作目录和根目录 / 120
7.6 服务器程序后台化 / 121

第8章 高性能服务器程序框架 / 123

8.1 服务器模型 / 123
8.1.1 C/S模型 / 123
8.1.2 P2P模型 / 124
8.2 服务器编程框架 / 125
8.3 I/O模型 / 126
8.4 两种高效的事件处理模式 / 127
8.4.1 Reactor模式 / 128
8.4.2 Proactor模式 / 128
8.4.3 模拟Proactor模式 / 129
8.5 两种高效的并发模式 / 130
8.5.1 半同步/半异步模式 / 131
8.5.2 领导者/追随者模式 / 134
8.6 有限状态机 / 136
8.7 提高服务器性能的其他建议 / 144
8.7.1 池 / 144
8.7.2 数据复制 / 145
8.7.3 上下文切换和锁 / 145

第9章 I/O复用 / 146

9.1 select系统调用 / 146
9.1.1 select API / 146
9.1.2 文件描述符就绪条件 / 148
9.1.3 处理带外数据 / 148

9.2 poll系统调用 / 150
9.3 epoll系列系统调用 / 151
9.3.1 内核事件表 / 151
9.3.2 epoll_wait函数 / 152
9.3.3 LT和ET模式 / 153
9.3.4 EPOLLONESHOT事件 / 157
9.4 三组I/O复用函数的比较 / 161
9.5 I/O复用的高级应用一：非阻塞connect / 162
9.6 I/O复用的高级应用二：聊天室程序 / 165
9.6.1 客户端 / 165
9.6.2 服务器 / 167
9.7 I/O复用的高级应用三：同时处理TCP和UDP服务 / 171
9.8 超级服务xinetd / 175
9.8.1 xinetd配置文件 / 175
9.8.2 xinetd工作流程 / 176

第10章 信号 / 178

10.1 Linux信号概述 / 178
10.1.1 发送信号 / 178
10.1.2 信号处理方式 / 179
10.1.3 Linux信号 / 179
10.1.4 中断系统调用 / 181
10.2 信号函数 / 181
10.2.1 signal系统调用 / 181
10.2.2 sigaction系统调用 / 181
10.3 信号集 / 182
10.3.1 信号集函数 / 182
10.3.2 进程信号掩码 / 183
10.3.3 被挂起的信号 / 183
10.4 统一事件源 / 184
10.5 网络编程相关信号 / 188
10.5.1 SIGHUP / 188
10.5.2 SIGPIPE / 189
10.5.3 SIGURG / 190

第11章　定时器 / 193

11.1　socket选项SO_RCVTIMEO和SO_SNDTIMEO / 193
11.2　SIGALRM信号 / 195
　11.2.1　基于升序链表的定时器 / 195
　11.2.2　处理非活动连接 / 200
11.3　I/O复用系统调用的超时参数 / 205
11.4　高性能定时器 / 206
　11.4.1　时间轮 / 206
　11.4.2　时间堆 / 211

第12章　高性能I/O框架库Libevent / 218

12.1　I/O框架库概述 / 218
12.2　Libevent源码分析 / 220
　12.2.1　一个实例 / 220
　12.2.2　源代码组织结构 / 222
　12.2.3　event结构体 / 224
　12.2.4　往注册事件队列中添加事件处理器 / 226
　12.2.5　往事件多路分发器中注册事件 / 230
　12.2.6　eventop结构体 / 233
　12.2.7　event_base结构体 / 235
　12.2.8　事件循环 / 236

第13章　多进程编程 / 239

13.1　fork系统调用 / 239
13.2　exec系列系统调用 / 240
13.3　处理僵尸进程 / 240
13.4　管道 / 241
13.5　信号量 / 243
　13.5.1　信号量原语 / 243
　13.5.2　semget系统调用 / 244
　13.5.3　semop系统调用 / 245
　13.5.4　semctl系统调用 / 247
　13.5.5　特殊键值IPC_PRIVATE / 249
13.6　共享内存 / 251

13.6.1 shmget系统调用 / 251
13.6.2 shmat和shmdt系统调用 / 252
13.6.3 shmctl系统调用 / 253
13.6.4 共享内存的POSIX方法 / 254
13.6.5 共享内存实例 / 254
13.7 消息队列 / 263
13.7.1 msgget系统调用 / 263
13.7.2 msgsnd系统调用 / 264
13.7.3 msgrcv系统调用 / 264
13.7.4 msgctl系统调用 / 265
13.8 IPC命令 / 266
13.9 在进程间传递文件描述符 / 267

第14章 多线程编程 / 269

14.1 Linux线程概述 / 269
14.1.1 线程模型 / 269
14.1.2 Linux线程库 / 270
14.2 创建线程和结束线程 / 271
14.3 线程属性 / 273
14.4 POSIX信号量 / 275
14.5 互斥锁 / 276
14.5.1 互斥锁基础API / 276
14.5.2 互斥锁属性 / 277
14.5.3 死锁举例 / 278
14.6 条件变量 / 279
14.7 线程同步机制包装类 / 280
14.8 多线程环境 / 282
14.8.1 可重入函数 / 282
14.8.2 线程和进程 / 283
14.8.3 线程和信号 / 284

第15章 进程池和线程池 / 287

15.1 进程池和线程池概述 / 287
15.2 处理多客户 / 288
15.3 半同步/半异步进程池实现 / 289

15.4 用进程池实现的简单CGI服务器 / 298
15.5 半同步/半反应堆线程池实现 / 301
15.6 用线程池实现的简单Web服务器 / 304
15.6.1 http_conn类 / 304
15.6.2 main函数 / 318

第三篇 高性能服务器优化与监测

第16章 服务器调制、调试和测试 / 324

16.1 最大文件描述符数 / 324
16.2 调整内核参数 / 325
16.2.1 /proc/sys/fs目录下的部分文件 / 325
16.2.2 /proc/sys/net目录下的部分文件 / 325
16.3 gdb调试 / 326
16.3.1 用gdb调试多进程程序 / 326
16.3.2 用gdb调试多线程程序 / 328
16.4 压力测试 / 329

第17章 系统监测工具 / 333

17.1 tcpdump / 333
17.2 lsof / 334
17.3 nc / 336
17.4 strace / 338
17.5 netstat / 341
17.6 vmstat / 342
17.7 ifstat / 344
17.8 mpstat / 344

参考文献 / 346

第一篇

TCP/IP 协议详解

第 1 章　TCP/IP 协议族

第 2 章　IP 协议详解

第 3 章　TCP 协议详解

第 4 章　TCP/IP 通信案例：访问 Internet 上的 Web 服务器

第 1 章　TCP/IP 协议族

现在 Internet（因特网）使用的主流协议族是 TCP/IP 协议族，它是一个分层、多协议的通信体系。本章简要讨论 TCP/IP 协议族各层包含的主要协议，以及它们之间是如何协作完成网络通信的。

TCP/IP 协议族包含众多协议，我们无法一一讨论。本书将在后续章节详细讨论 IP 协议和 TCP 协议，因为它们对编写网络应用程序具有最直接的影响。本章则简单介绍其中几个相关协议：ICMP 协议、ARP 协议和 DNS 协议，学习它们对于理解网络通信很有帮助。读者如果想要系统地学习网络协议，那么 RFC（Request For Comments，评论请求）文档无疑是首选资料。

1.1　TCP/IP 协议族体系结构以及主要协议

TCP/IP 协议族是一个四层协议系统，自底而上分别是数据链路层、网络层、传输层和应用层。每一层完成不同的功能，且通过若干协议来实现，上层协议使用下层协议提供的服务，如图 1-1 所示。

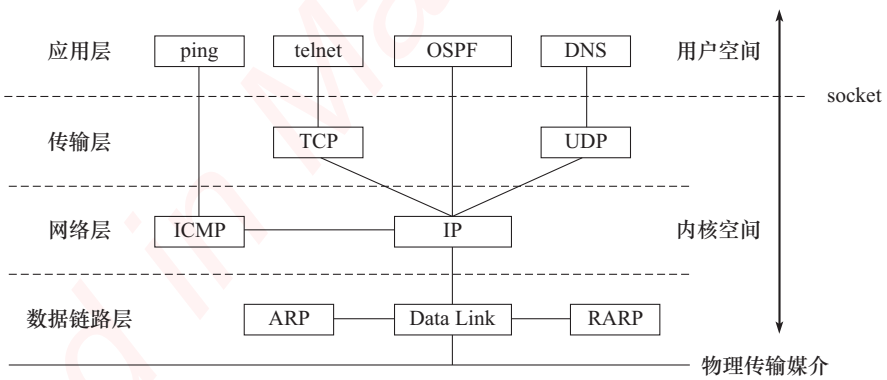

图 1-1　TCP/IP 协议族体系结构及主要协议

1.1.1　数据链路层

数据链路层实现了网卡接口的网络驱动程序，以处理数据在物理媒介（比如以太网、令牌环等）上的传输。不同的物理网络具有不同的电气特性，网络驱动程序隐藏了这些细节，为上层协议提供一个统一的接口。

数据链路层两个常用的协议是 ARP 协议（Address Resolve Protocol，地址解析协议）和

RARP 协议（Reverse Address Resolve Protocol，逆地址解析协议）。它们实现了 IP 地址和机器物理地址（通常是 MAC 地址，以太网、令牌环和 802.11 无线网络都使用 MAC 地址）之间的相互转换。

网络层使用 IP 地址寻址一台机器，而数据链路层使用物理地址寻址一台机器，因此网络层必须先将目标机器的 IP 地址转化成其物理地址，才能使用数据链路层提供的服务，这就是 ARP 协议的用途。RARP 协议仅用于网络上的某些无盘工作站。因为缺乏存储设备，无盘工作站无法记住自己的 IP 地址，但它们可以利用网卡上的物理地址来向网络管理者（服务器或网络管理软件）查询自身的 IP 地址。运行 RARP 服务的网络管理者通常存有该网络上所有机器的物理地址到 IP 地址的映射。

由于 ARP 协议很重要，所以我们将在后面章节专门讨论它。

1.1.2 网络层

网络层实现数据包的选路和转发。WAN（Wide Area Network，广域网）通常使用众多分级的路由器来连接分散的主机或 LAN（Local Area Network，局域网），因此，通信的两台主机一般不是直接相连的，而是通过多个中间节点（路由器）连接的。网络层的任务就是选择这些中间节点，以确定两台主机之间的通信路径。同时，网络层对上层协议隐藏了网络拓扑连接的细节，使得在传输层和网络应用程序看来，通信的双方是直接相连的。

网络层最核心的协议是 IP 协议（Internet Protocol，因特网协议）。IP 协议根据数据包的目的 IP 地址来决定如何投递它。如果数据包不能直接发送给目标主机，那么 IP 协议就为它寻找一个合适的下一跳（next hop）路由器，并将数据包交付给该路由器来转发。多次重复这一过程，数据包最终到达目标主机，或者由于发送失败而被丢弃。可见，IP 协议使用逐跳（hop by hop）的方式确定通信路径。我们将在第 2 章详细讨论 IP 协议。

网络层另外一个重要的协议是 ICMP 协议（Internet Control Message Protocol，因特网控制报文协议）。它是 IP 协议的重要补充，主要用于检测网络连接。ICMP 协议使用的报文格式如图 1-2 所示。

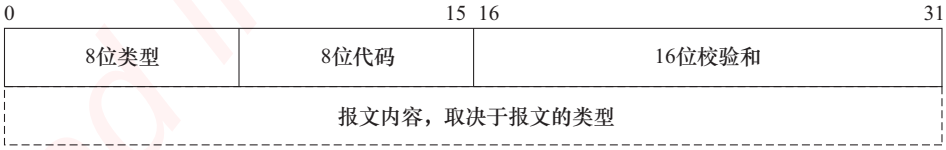

图 1-2　ICMP 报文格式

图 1-2 中，8 位类型字段用于区分报文类型。它将 ICMP 报文分为两大类：一类是差错报文，这类报文主要用来回应网络错误，比如目标不可到达（类型值为 3）和重定向（类型值为 5）；另一类是查询报文，这类报文用来查询网络信息，比如 ping 程序就是使用 ICMP 报文查看目标是否可到达（类型值为 8）的。有的 ICMP 报文还使用 8 位代码字段来进一步细分不同的条件。比如重定向报文使用代码值 0 表示对网络重定向，代码值 1 表示对主机重

定向。ICMP 报文使用 16 位校验和字段对整个报文（包括头部和内容部分）进行循环冗余校验（Cyclic Redundancy Check，CRC），以检验报文在传输过程中是否损坏。不同的 ICMP 报文类型具有不同的正文内容。我们将在第 2 章详细讨论主机重定向报文，其他 ICMP 报文格式请参考 ICMP 协议的标准文档 RFC 792。

需要指出的是，ICMP 协议并非严格意义上的网络层协议，因为它使用处于同一层的 IP 协议提供的服务（一般来说，上层协议使用下层协议提供的服务）。

1.1.3 传输层

传输层为两台主机上的应用程序提供端到端（end to end）的通信。与网络层使用的逐跳通信方式不同，传输层只关心通信的起始端和目的端，而不在乎数据包的中转过程。图 1-3 展示了传输层和网络层的这种区别。

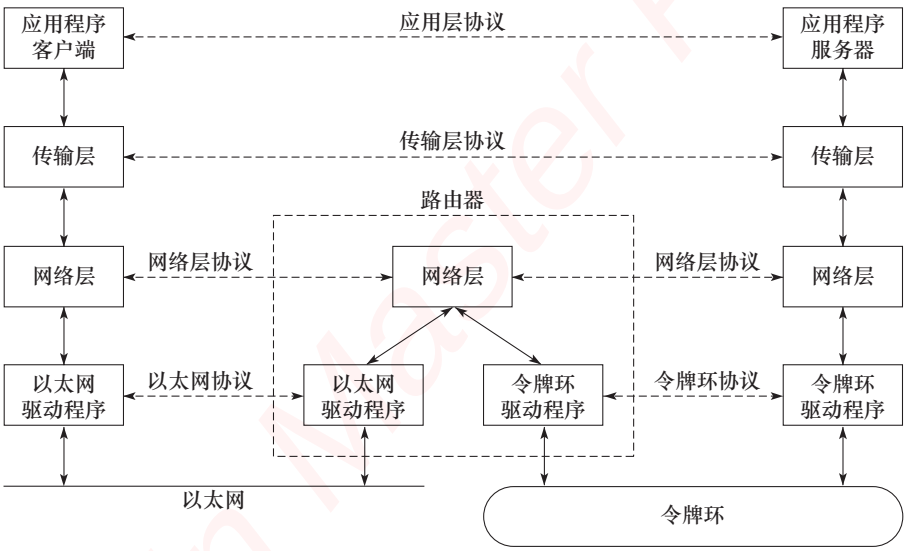

图 1-3 传输层和网络层的区别

图 1-3 中，垂直的实线箭头表示 TCP/IP 协议族各层之间的实体通信（数据包确实是沿着这些线路传递的），而水平的虚线箭头表示逻辑通信线路。该图中还附带描述了不同物理网络的连接方法。可见，数据链路层（驱动程序）封装了物理网络的电气细节；网络层封装了网络连接的细节；传输层则为应用程序封装了一条端到端的逻辑通信链路，它负责数据的收发、链路的超时重连等。

传输层协议主要有三个：TCP 协议、UDP 协议和 SCTP 协议。

TCP 协议（Transmission Control Protocol，传输控制协议）为应用层提供可靠的、面向连接的和基于流（stream）的服务。TCP 协议使用超时重传、数据确认等方式来确保数据包被正确地发送至目的端，因此 TCP 服务是可靠的。使用 TCP 协议通信的双方必须先建立 TCP 连接，并在内核中为该连接维持一些必要的数据结构，比如连接的状态、读写缓冲区，

以及诸多定时器等。当通信结束时，双方必须关闭连接以释放这些内核数据。TCP 服务是基于流的。基于流的数据没有边界（长度）限制，它源源不断地从通信的一端流入另一端。发送端可以逐个字节地向数据流中写入数据，接收端也可以逐个字节地将它们读出。

UDP 协议（User Datagram Protocol，用户数据报协议）则与 TCP 协议完全相反，它为应用层提供不可靠、无连接和基于数据报的服务。“不可靠”意味着 UDP 协议无法保证数据从发送端正确地传送到目的端。如果数据在中途丢失，或者目的端通过数据校验发现数据错误而将其丢弃，则 UDP 协议只是简单地通知应用程序发送失败。因此，使用 UDP 协议的应用程序通常要自己处理数据确认、超时重传等逻辑。UDP 协议是无连接的，即通信双方不保持一个长久的联系，因此应用程序每次发送数据都要明确指定接收端的地址（IP 地址等信息）。基于数据报的服务，是相对基于流的服务而言的。每个 UDP 数据报都有一个长度，接收端必须以该长度为最小单位将其所有内容一次性读出，否则数据将被截断。

SCTP 协议（Stream Control Transmission Protocol，流控制传输协议）是一种相对较新的传输层协议，它是为了在因特网上传输电话信号而设计的。本书不讨论 SCTP 协议，感兴趣的读者可参考其标准文档 RFC 2960。

我们将在第 3 章详细讨论 TCP 协议，并附带介绍 UDP 协议。

1.1.4 应用层

应用层负责处理应用程序的逻辑。数据链路层、网络层和传输层负责处理网络通信细节，这部分必须既稳定又高效，因此它们都在内核空间中实现，如图 1-1 所示。而应用层则在用户空间实现，因为它负责处理众多逻辑，比如文件传输、名称查询和网络管理等。如果应用层也在内核中实现，则会使内核变得非常庞大。当然，也有少数服务器程序是在内核中实现的，这样代码就无须在用户空间和内核空间来回切换（主要是数据的复制），极大地提高了工作效率。不过这种代码实现起来较复杂，不够灵活，且不便于移植。本书只讨论用户空间的网络编程。

应用层协议很多，图 1-1 仅列举了其中的几个：

ping 是应用程序，而不是协议，前面说过它利用 ICMP 报文检测网络连接，是调试网络环境的必备工具。

telnet 协议是一种远程登录协议，它使我们能在本地完成远程任务，本书后续章节将会多次使用 telnet 客户端登录到其他服务上。

OSPF（Open Shortest Path First，开放最短路径优先）协议是一种动态路由更新协议，用于路由器之间的通信，以告知对方各自的路由信息。

DNS（Domain Name Service，域名服务）协议提供机器域名到 IP 地址的转换，我们将在后面简要介绍 DNS 协议。

应用层协议（或程序）可能跳过传输层直接使用网络层提供的服务，比如 ping 程序和 OSPF 协议。应用层协议（或程序）通常既可以使用 TCP 服务，又可以使用 UDP 服务，比如 DNS 协议。我们可以通过 /etc/services 文件查看所有知名的应用层协议，以及它们都能使

用哪些传输层服务。

1.2 封装

上层协议是如何使用下层协议提供的服务的呢？其实这是通过封装（encapsulation）实现的。应用程序数据在发送到物理网络上之前，将沿着协议栈从上往下依次传递。每层协议都将在上层数据的基础上加上自己的头部信息（有时还包括尾部信息），以实现该层的功能，这个过程就称为封装，如图 1-4 所示。

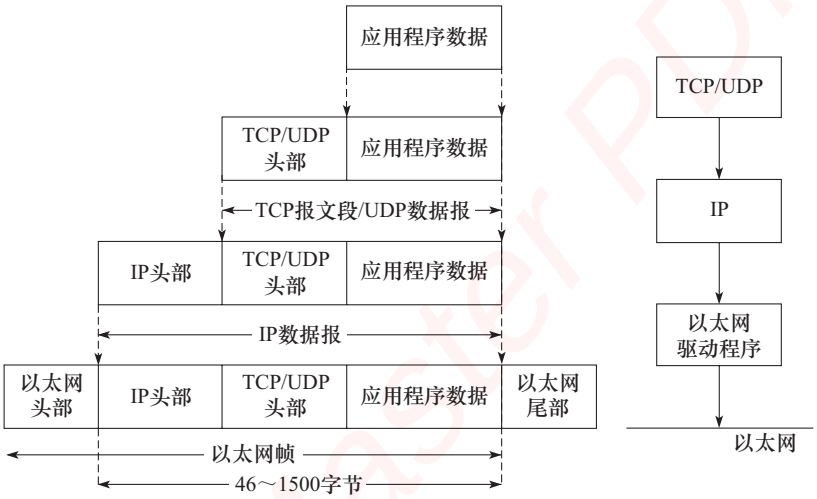

图 1-4 封装

经过 TCP 封装后的数据称为 TCP 报文段（TCP message segment），或者简称 TCP 段。前文提到，TCP 协议为通信双方维持一个连接，并且在内核中存储相关数据。这部分数据中的 TCP 头部信息和 TCP 内核缓冲区（发送缓冲区或接收缓冲区）数据一起构成了 TCP 报文段，如图 1-5 中的虚线框所示。

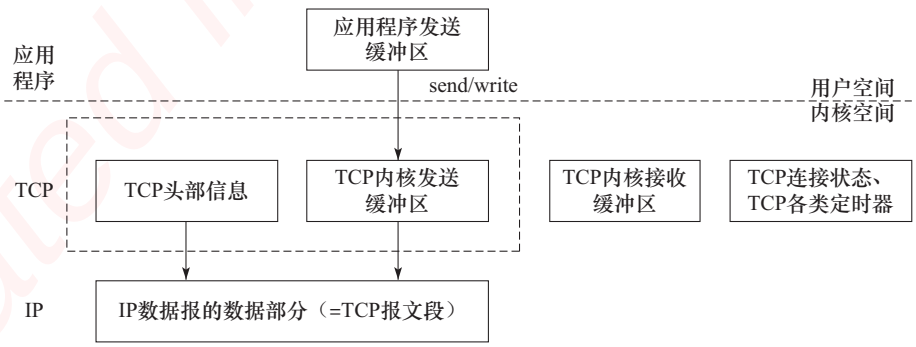

图 1-5 TCP 报文段封装过程

当发送端应用程序使用 send（或者 write）函数向一个 TCP 连接写入数据时，内核中的 TCP 模块首先把这些数据复制到与该连接对应的 TCP 内核发送缓冲区中，然后 TCP 模块调用 IP 模块提供的服务，传递的参数包括 TCP 头部信息和 TCP 发送缓冲区中的数据，即 TCP 报文段。关于 TCP 报文段头部的细节，我们将在第 3 章讨论。

经过 UDP 封装后的数据称为 UDP 数据报（UDP datagram）。UDP 对应用程序数据的封装与 TCP 类似。不同的是，UDP 无须为应用层数据保存副本，因为它提供的服务是不可靠的。当一个 UDP 数据报被成功发送之后，UDP 内核缓冲区中的该数据报就被丢弃了。如果应用程序检测到该数据报未能被接收端正确接收，并打算重发这个数据报，则应用程序需要重新从用户空间将该数据报拷贝到 UDP 内核发送缓冲区中。

经过 IP 封装后的数据称为 IP 数据报（IP datagram）。IP 数据报也包括头部信息和数据部分，其中数据部分就是一个 TCP 报文段、UDP 数据报或者 ICMP 报文。我们将在第 2 章详细讨论 IP 数据报的头部信息。

经过数据链路层封装的数据称为帧（frame）。传输媒介不同，帧的类型也不同。比如，以太网上传输的是以太网帧（ethernet frame），而令牌环网络上传输的则是令牌环帧（token ring frame）。以以太网帧为例，其封装格式如图 1-6 所示。

以太网帧使用 6 字节的目的物理地址和 6 字节的源物理地址来表示通信的双方。关于类型（type）字段，我们将在后面讨论。4 字节 CRC 字段对帧的其他部分提供循环冗余校验。

帧的最大传输单元（Max Transmit Unit，MTU），即帧最多能携带多少上层协议数据（比如 IP 数据报），通常受到网络类型的限制。图 1-6 所示的以太网帧的 MTU 是 1500 字节。正因为如此，过长的 IP 数据报可能需要被分片（fragment）传输。

目的物理地址	源物理地址	类型	数据	CRC
6字节	6字节	2字节	46～1500字节	4字节

图 1-6　以太网帧封装

帧才是最终在物理网络上传送的字节序列。至此，封装过程完成。

1.3　分用

当帧到达目的主机时，将沿着协议栈自底向上依次传递。各层协议依次处理帧中本层负责的头部数据，以获取所需的信息，并最终将处理后的帧交给目标应用程序。这个过程称为分用（demultiplexing）。分用是依靠头部信息中的类型字段实现的。标准文档 RFC 1700 定义了所有标识上层协议的类型字段以及每个上层协议对应的数值。图 1-7 显示了以太网帧的分用过程。

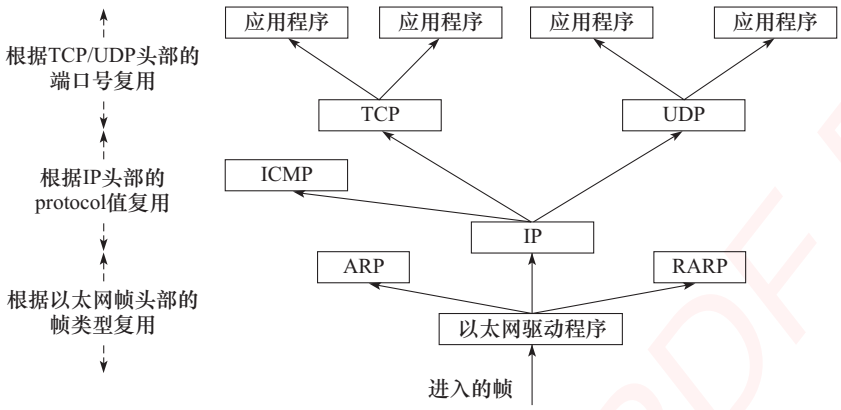

图 1-7　以太网帧的分用过程

因为 IP 协议、ARP 协议和 RARP 协议都使用帧传输数据，所以帧的头部需要提供某个字段（具体情况取决于帧的类型）来区分它们。以以太网帧为例，它使用 2 字节的类型字段来标识上层协议（见图 1-6）。如果主机接收到的以太网帧类型字段的值为 0x800，则帧的数据部分为 IP 数据报（见图 1-4），以太网驱动程序就将帧交付给 IP 模块；若类型字段的值为 0x806，则帧的数据部分为 ARP 请求或应答报文，以太网驱动程序就将帧交付给 ARP 模块；若类型字段的值为 0x835，则帧的数据部分为 RARP 请求或应答报文，以太网驱动程序就将帧交付给 RARP 模块。

同样，因为 ICMP 协议、TCP 协议和 UDP 协议都使用 IP 协议，所以 IP 数据报的头部采用 16 位的协议（protocol）字段来区分它们。

TCP 报文段和 UDP 数据报则通过其头部中的 16 位的端口号（port number）字段来区分上层应用程序。比如 DNS 协议对应的端口号是 53，HTTP 协议（Hyper-Text Transfer Protocol，超文本传送协议）对应的端口号是 80。所有知名应用层协议使用的端口号都可在 /etc/services 文件中找到。

帧通过上述分用步骤后，最终将封装前的原始数据送至目标服务（图 1-7 中的 ARP 服务、RARP 服务、ICMP 服务或者应用程序）。这样，在顶层目标服务看来，封装和分用似乎没有发生过。

1.4　测试网络

为了深入理解网络通信和网络编程，我们准备了图 1-8 所示的测试网络，其中包括两台主机 A 和 B，以及一个连接到因特网的路由器。后文如没有特别声明，所有测试硬件指的都是该网络。我们将使用机器名来标识测试机器。

该测试网络主要用于分析 ARP 协议、IP 协议、ICMP 协议、TCP 协议和 DNS 协议。我们通过抓取该网络上的以太网帧，查看其中的以太网帧头部、IP 数据报头部、TCP 报文段头部信息，以获取网络通信的细节。这样，以理论结合实践，我们就清楚 TCP/IP 通信具体是

如何进行的了。作者编写的多个客户端、服务器程序都是使用该网络来调试和测试的。

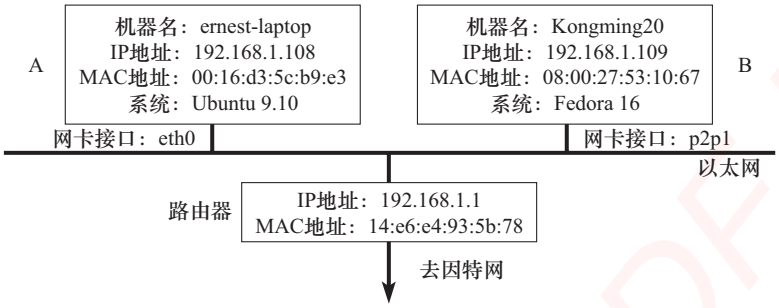

图 1-8　测试网络

对于路由器，我们仅列出了其 LAN 网络 IP 地址（192.168.1.1），而忽略了 ISP（Internet Service Provider，因特网服务提供商）给它分配的 WAN 网络 IP 地址，因为全书的讨论都不涉及它。

1.5　ARP 协议工作原理

ARP 协议能实现任意网络层地址到任意物理地址的转换，不过本书仅讨论从 IP 地址到以太网地址（MAC 地址）的转换。其工作原理是：主机向自己所在的网络广播一个 ARP 请求，该请求包含目标机器的网络地址。此网络上的其他机器都将收到这个请求，但只有被请求的目标机器会回应一个 ARP 应答，其中包含自己的物理地址。

1.5.1　以太网 ARP 请求 / 应答报文详解

以太网 ARP 请求 / 应答报文的格式如图 1-9 所示。

硬件类型	协议类型	硬件地址长度	协议地址长度	操作	发送端以太网地址	发送端IP地址	目的端以太网地址	目的端IP地址
2字节	2字节	1字节	1字节	2字节	6字节	4字节	6字节	4字节

图 1-9　以太网 ARP 请求 / 应答报文

图 1-9 所示以太网 ARP 请求 / 应答报文各字段具体介绍如下。

- 硬件类型字段定义物理地址的类型，它的值为 1 表示 MAC 地址。
- 协议类型字段表示要映射的协议地址类型，它的值为 0x800，表示 IP 地址。
- 硬件地址长度字段和协议地址长度字段，顾名思义，其单位是字节。对 MAC 地址来说，其长度为 6；对 IP（v4）地址来说，其长度为 4。
- 操作字段指出 4 种操作类型：ARP 请求（值为 1）、ARP 应答（值为 2）、RARP 请求（值为 3）和 RARP 应答（值为 4）。
- 最后 4 个字段指定通信双方的以太网地址和 IP 地址。发送端填充除目的端以太网地址外的其他 3 个字段，以构建 ARP 请求并发送之。接收端发现该请求的目的端 IP 地

址是自己，就把自己的以太网地址填进去，然后交换两个目的端地址和两个发送端地址，以构建 ARP 应答并返回之（当然，如前所述，操作字段需要设置为 2）。

由图 1-9 可知，ARP 请求 / 应答报文的长度为 28 字节。如果再加上以太网帧头部和尾部的 18 字节（见图 1-6），则一个携带 ARP 请求 / 应答报文的以太网帧长度为 46 字节。不过有的实现要求以太网帧数据部分长度至少为 46 字节（见图 1-4），此时 ARP 请求 / 应答报文将增加一些填充字节，以满足这个要求。在这种情况下，一个携带 ARP 请求 / 应答报文的以太网帧长度为 64 字节。

1.5.2 ARP 高速缓存的查看和修改

通常，ARP 维护一个高速缓存，其中包含经常访问（比如网关地址）或最近访问的机器的 IP 地址到物理地址的映射。这样就避免了重复的 ARP 请求，提高了发送数据包的速度。

Linux 下可以使用 arp 命令来查看和修改 ARP 高速缓存。比如，ernest-laptop 在某一时刻（注意，ARP 高速缓存是动态变化的）的 ARP 缓存内容如下（使用 arp-a 命令）：

```
Kongming20 (192.168.1.109) at 08:00:27:53:10:67 [ether] on eth0
?(192.168.1.1) at 14:e6:e4:93:5b:78 [ether] on eth0
```

其中，第一项描述的是另一台测试机器 Kongming20（注意，其 IP 地址、MAC 地址都与图 1-8 描述的一致），第二项描述的是路由器。下面两条命令则分别删除和添加一个 ARP 缓存项：

```
$ sudo arp -d 192.168.1.109                    # 删除 Kongming20 对应的 ARP 缓存项
$ sudo arp -s 192.168.1.109 08:00:27:53:10:67  # 添加 Kongming20 对应的 ARP 缓存项
```

1.5.3 使用 tcpdump 观察 ARP 通信过程

为了清楚地了解 ARP 的运作过程，我们从 ernest-laptop 上执行 telnet 命令登录 Kongming20 的 echo 服务（已经开启），并用 tcpdump（详见第 17 章）抓取这个过程中两台测试机器之间交换的以太网帧。具体的操作过程如下：

```
$ sudo arp -d 192.168.1.109                    # 清除 ARP 缓存中 Kongming20 对应的项
$ sudo tcpdump -i eth0 -ent '(dst 192.168.1.109 and src 192.168.1.108)or
(dst 192.168.1.108 and src 192.168.1.109)'     # 如无特殊声明，抓包都在机器 ernest-
                                                 laptop 上执行
$ telnet 192.168.1.109 echo                    # 开启另一个终端执行 telnet 命令
Trying 192.168.1.109...
Connected to 192.168.1.109.
Escape character is '^]'.
^]（回车）                                      # 输入 Ctrl+] 并回车

telnet> quit（回车）
Connection closed.
```

在执行 telnet 命令之前，应先清除 ARP 缓存中与 Kongming20 对应的项，否则 ARP 通信不被执行，我们也就无法抓取到期望的以太网帧。当执行 telnet 命令并在两台通信主机之间建立

TCP 连接后（telnet 输出“Connected to 192.168.1.109”），输入 Ctrl+］以调出 telnet 程序的命令提示符，然后在 telnet 命令提示符后输入 quit，退出 telnet 客户端程序（因为 ARP 通信在 TCP 连接建立之前就已经完成，故我们不关心后续内容）。tcpdump 抓取到的众多数据包中，只有最靠前的两个和 ARP 通信有关系，现在将它们列出（数据包前面的编号是笔者加入的，后同）：

```
    1. 00:16:d3:5c:b9:e3 > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806), length 42:
Request who-has 192.168.1.109 tell 192.168.1.108, length 28
    2. 08:00:27:53:10:67 > 00:16:d3:5c:b9:e3, ethertype ARP (0x0806), length 60:
Reply 192.168.1.109 is-at 08:00:27:53:10:67, length 46
```

由 tcpdump 抓取的数据包本质上是以太网帧，我们通过该命令的众多选项来控制帧的过滤（比如用 dst 和 src 指定通信的目的端 IP 地址和源端 IP 地址）和显示（比如用 -e 选项开启以太网帧头部信息的显示）。

第一个数据包中，ARP 通信的源端的物理地址是 00:16:d3:5c:b9:e3（ernest-laptop），目的端的物理地址是 ff:ff:ff:ff:ff:ff，这是以太网的广播地址，用以表示整个 LAN。该 LAN 上的所有机器都会收到并处理这样的帧。数值 0x806 是以太网帧头部的类型字段的值，它表示分用的目标是 ARP 模块。该以太网帧的长度为 42 字节（实际上是 46 字节，tcpdump 未统计以太网帧尾部 4 字节的 CRC 字段），其中数据部分长度为 28 字节。“Request”表示这是一个 ARP 请求，“who-has 192.168.1.109 tell 192.168.1.108”则表示是 ernest-laptop 要查询 Kongming20 的 IP 地址。

第二个数据包中，ARP 通信的源端的物理地址是 08:00:27:53:10:67（Kongming20），目的端的物理地址是 00:16:d3:5c:b9:e3（ernest-laptop）。“Reply”表示这是一个 ARP 应答，“192.168.1.109 is-at 08:00:27:53:10:67”则表示目标机器 Kongming20 报告其物理地址。该以太网帧的长度为 60 字节（实际上是 64 字节），可见它使用了填充字节来满足最小帧长度。

为了便于理解，我们将上述讨论用图 1-10 来详细说明。

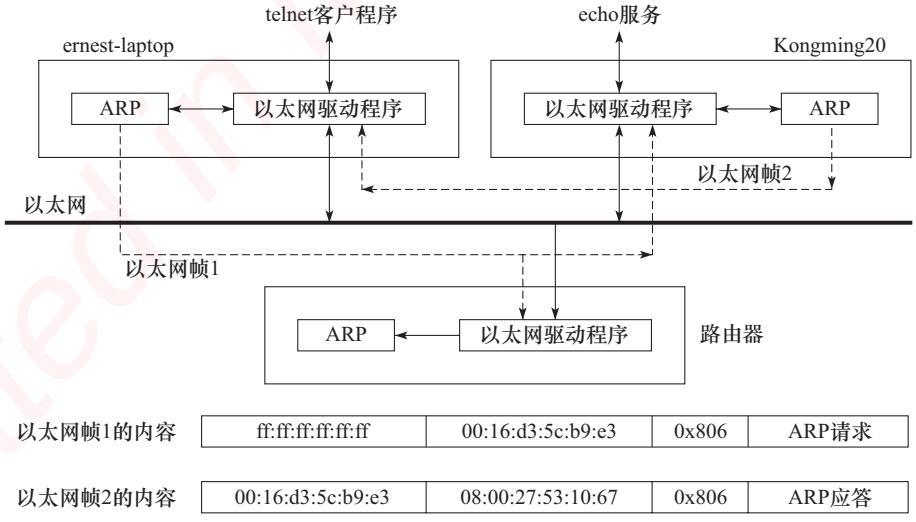

图 1-10　ARP 通信过程

关于该图，需要说明三点：

第一，我们将两次传输的以太网帧按照图 1-6 所描述的以太网帧封装格式绘制在图的下半部分。

第二，ARP 请求和应答是从以太网驱动程序发出的，而并非像图中描述的那样从 ARP 模块直接发送到以太网上，所以我们将它们用虚线表示，这主要是为了体现携带 ARP 数据的以太网帧和其他以太网帧（比如携带 IP 数据报的以太网帧）的区别。

第三，路由器也将接收到以太网帧 1，因为该帧是一个广播帧。不过很显然，路由器并没有回应其中的 ARP 请求，正如前文讨论的那样。

1.6 DNS 工作原理

我们通常使用机器的域名来访问这台机器，而不直接使用其 IP 地址，比如访问因特网上的各种网站。那么如何将机器的域名转换成 IP 地址呢？这就需要使用域名查询服务。域名查询服务有很多种实现方式，比如 NIS（Network Information Service，网络信息服务）、DNS 和本地静态文件等。本节主要讨论 DNS。

1.6.1 DNS 查询和应答报文详解

DNS 是一套分布式的域名服务系统。每个 DNS 服务器上都存放着大量的机器名和 IP 地址的映射，并且是动态更新的。众多网络客户端程序都使用 DNS 协议来向 DNS 服务器查询目标主机的 IP 地址。DNS 查询和应答报文的格式如图 1-11 所示。

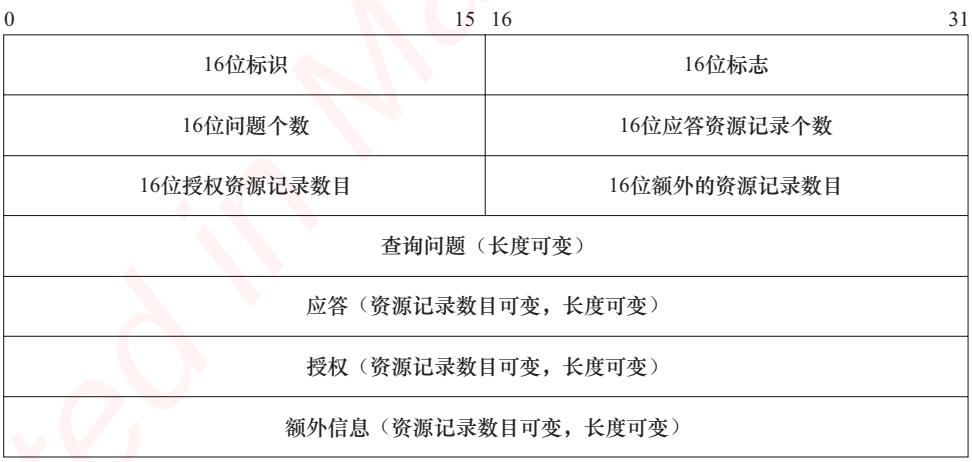

图 1-11　DNS 查询和应答报文

16 位标识[⊖]字段用于标记一对 DNS 查询和应答，以此区分一个 DNS 应答是哪个 DNS 查

⊖ “标识”和“标志”在《现代汉语词典（第5版）》中表示同一含义，但是在本书中（计算机业界也是如此），它们为两个概念，代表不同的含义，读者在阅读时应严格区分。

询的回应。

16 位标志字段用于协商具体的通信方式和反馈通信状态。DNS 报文头部的 16 位标志字段的细节如图 1-12 所示。

QR	opcode	AA	TC	RD	RA	zero	rcode
1位	4位	1位	1位	1位	1位	3位	4位

图 1-12　DNS 报文头部的标志字段

图 1-12 中各标志的含义分别是：

- QR，查询 / 应答标志。0 表示这是一个查询报文，1 表示这是一个应答报文。
- opcode，定义查询和应答的类型。0 表示标准查询，1 表示反向查询（由 IP 地址获得主机域名），2 表示请求服务器状态。
- AA，授权应答标志，仅由应答报文使用。1 表示域名服务器是授权服务器。
- TC，截断标志，仅当 DNS 报文使用 UDP 服务时使用。因为 UDP 数据报有长度限制，所以过长的 DNS 报文将被截断。1 表示 DNS 报文超过 512 字节，并被截断。
- RD，递归查询标志。1 表示执行递归查询，即如果目标 DNS 服务器无法解析某个主机名，则它将向其他 DNS 服务器继续查询，如此递归，直到获得结果并把该结果返回给客户端。0 表示执行迭代查询，即如果目标 DNS 服务器无法解析某个主机名，则它将自己知道的其他 DNS 服务器的 IP 地址返回给客户端，以供客户端参考。
- RA，允许递归标志。仅由应答报文使用，1 表示 DNS 服务器支持递归查询。
- zero，这 3 位未用，必须都设置为 0。
- rcode，4 位返回码，表示应答的状态。常用值有 0（无错误）和 3（域名不存在）。

接下来的 4 个字段则分别指出 DNS 报文的最后 4 个字段的资源记录数目。对查询报文而言，它一般包含 1 个查询问题，而应答资源记录数、授权资源记录数和额外资源记录数则为 0。应答报文的应答资源记录数则至少为 1，而授权资源记录数和额外资源记录数可为 0 或非 0。

查询问题的格式如图 1-13 所示。

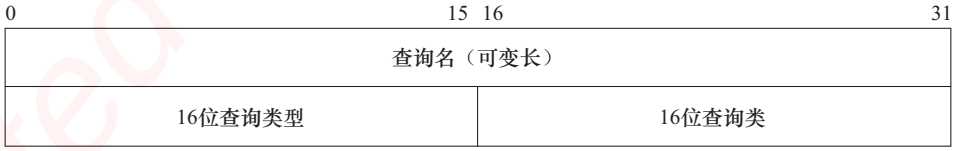

图 1-13　DNS 查询问题的格式

图 1-13 中，查询名以一定的格式封装了要查询的主机域名。16 位查询类型表示如何执行查询操作，常见的类型有如下几种：

- 类型 A，值是 1，表示获取目标主机的 IP 地址。

❑ 类型 CNAME，值是 5，表示获得目标主机的别名。

❑ 类型 PTR，值是 12，表示反向查询。

16 位查询类通常为 1，表示获取因特网地址（IP 地址）。

应答字段、授权字段和额外信息字段都使用资源记录（Resource Record，RR）格式。资源记录格式如图 1-14 所示。

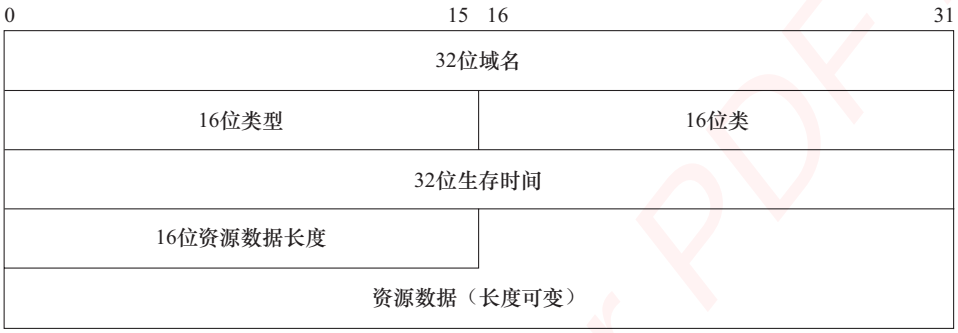

图 1-14　资源记录格式

图 1-14 中，32 位域名是该记录中与资源对应的名字，其格式和查询问题中的查询名字段相同。16 位类型和 16 位类字段的含义也与 DNS 查询问题的对应字段相同。

32 位生存时间表示该查询记录结果可被本地客户端程序缓存多长时间，单位是秒。

16 位资源数据长度字段和资源数据字段的内容取决于类型字段。对类型 A 而言，资源数据是 32 位的 IPv4 地址，而资源数据长度则为 4（以字节为单位）。

至此，我们简要地介绍了 DNS 协议。我们将在后面给出一个 DNS 通信的具体例子。DNS 协议的更多细节请参考其 RFC 文档（DNS 协议存在诸多 RFC 文档，每个 RFC 文档介绍其一个侧面，比如 RFC 1035 介绍的是域名的实现和规范，RFC 1886 则描述 DNS 协议对 IPv6 的扩展支持）。

1.6.2　Linux 下访问 DNS 服务

我们要访问 DNS 服务，就必须先知道 DNS 服务器的 IP 地址。Linux 使用 /etc/resolv.conf 文件来存放 DNS 服务器的 IP 地址。机器 ernest-laptop 上，该文件的内容如下：

```
# Generated by Network Manager
nameserver 219.239.26.42
nameserver 124.207.160.106
```

其中的两个 IP 地址分别是首选 DNS 服务器地址和备选 DNS 服务器地址。文件中的注释语句“Generated by Network Manager”告诉我们，这两个 DNS 服务器地址是由网络管理程序写入的。

Linux 下一个常用的访问 DNS 服务器的客户端程序是 host，比如下面的命令是向首选 DNS 服务器 219.239.26.42 查询机器 www.baidu.com 的 IP 地址：

```
$ host -t A www.baidu.com
www.baidu.com is an alias for www.a.shifen.com.
www.a.shifen.com has address 119.75.217.56
www.a.shifen.com has address 119.75.218.77
```

host 命令的输出告诉我们，机器名 www.baidu.com 是 www.a.shifen.com. 的别名，并且该机器名对应两个 IP 地址。host 命令使用 DNS 协议和 DNS 服务器通信，其 -t 选项告诉 DNS 协议使用哪种查询类型。我们这里使用的是 A 类型，即通过机器的域名获得其 IP 地址（但实际上返回的资源记录中还包含机器的别名）。关于 host 命令的详细使用方法，请参考其 man 手册。

1.6.3 使用 tcpdump 观察 DNS 通信过程

为了看清楚 DNS 通信的过程，下面我们将从 ernest-laptop 上运行 host 命令以查询主机 www.baidu.com 对应的 IP 地址，并使用 tcpdump 抓取这一过程中 LAN 上传输的以太网帧。具体的操作过程如下：

```
$ sudo tcpdump -i eth0 -nt -s 500 port domain
$ host -t A www.baidu.com
```

这一次执行 tcpdump 抓包时，我们使用“port domain”来过滤数据包，表示只抓取使用 domain（域名）服务的数据包，即 DNS 查询和应答报文。tcpdump 的输出如下：

```
1. IP 192.168.1.108.34319 > 219.239.26.42.53: 57428+ A? www.baidu.com. (31)
2. IP 219.239.26.42.53 > 192.168.1.108.34319: 57428 3/4/4 CNAME www.a.shifen.com.,
A 119.75.218.77, A 119.75.217.56 (226)
```

这两个数据包开始的“IP”指出，它们后面的内容描述的是 IP 数据报。tcpdump 以“IP 地址 . 端口号”的形式来描述通信的某一端；以“>”表示数据传输的方向，“>”前面是源端，后面是目的端。可见，第一个数据包是测试机器 ernest-laptop（IP 地址是 192.168.1.108）向其首选 DNS 服务器（IP 地址是 219.239.26.42）发送的 DNS 查询报文（目标端口 53 是 DNS 服务使用的端口，这一点我们在前面介绍过），第二个数据包是服务器反馈的 DNS 应答报文。

第一个数据包中，数值 57428 是 DNS 查询报文的标识值，因此该值也出现在 DNS 应答报文中。“+”表示启用递归查询标志。“A?”表示使用 A 类型的查询方式。“www.baidu.com”则是 DNS 查询问题中的查询名。括号中的数值 31 是 DNS 查询报文的长度（以字节为单位）。

第二个数据包中，“3/4/4”表示该报文中包含 3 个应答资源记录、4 个授权资源记录和 4 个额外信息记录。“CNAME www.a.shifen.com.，A 119.75.218.77，A 119.75.217.56”则表示 3 个应答资源记录的内容。其中 CNAME 表示紧随其后的记录是机器的别名，A 表示紧随其后的记录是 IP 地址。该应答报文的长度为 226 字节。

注意 我们抓包的时候没有开启 tcpdump 的 -X 选项（或者 -x 选项）。如果使用 -X 选项，我们将能看到 DNS 报文的每一个字节，也就能明白上面 31 字节的查询报文和 226 字节的应答报文的具体含义。限于篇幅，这里不再讨论，读者不妨自己分析。

1.7 socket 和 TCP/IP 协议族的关系

前文提到，数据链路层、网络层、传输层协议是在内核中实现的。因此操作系统需要实现一组系统调用，使得应用程序能够访问这些协议提供的服务。实现这组系统调用的 API（Application Programming Interface，应用程序编程接口）主要有两套：socket 和 XTI。XTI 现在基本不再使用，本书仅讨论 socket。图 1-1 显示了 socket 与 TCP/IP 协议族的关系。

由 socket 定义的这一组 API 提供如下两点功能：一是将应用程序数据从用户缓冲区中复制到 TCP/UDP 内核发送缓冲区，以交付内核来发送数据（比如图 1-5 所示的 send 函数），或者是从内核 TCP/UDP 接收缓冲区中复制数据到用户缓冲区，以读取数据；二是应用程序可以通过它们来修改内核中各层协议的某些头部信息或其他数据结构，从而精细地控制底层通信的行为。比如可以通过 setsockopt 函数来设置 IP 数据报在网络上的存活时间。我们将在第 5 章详细讨论这一组 API。

值得一提的是，socket 是一套通用网络编程接口，它不但可以访问内核中 TCP/IP 协议栈，而且可以访问其他网络协议栈（比如 X.25 协议栈、UNIX 本地域协议栈等）。

第 2 章　IP 协议详解

IP 协议是 TCP/IP 协议族的核心协议，也是 socket 网络编程的基础之一。本章从两个方面较为深入地探讨 IP 协议：

- IP 头部信息。IP 头部信息出现在每个 IP 数据报中，用于指定 IP 通信的源端 IP 地址、目的端 IP 地址，指导 IP 分片和重组，以及指定部分通信行为。
- IP 数据报的路由和转发。IP 数据报的路由和转发发生在除目标机器之外的所有主机和路由器上。它们决定数据报是否应该转发以及如何转发。

由于 32 位表示的 IP 地址即将全部使用完，因此人们开发出了新版本的 IP 协议，称为 IPv6 协议，而原来的版本则称为 IPv4 协议。本章前面部分的讨论都是基于 IPv4 协议的，只在最后一节简要讨论 IPv6 协议。

在开始讨论前，我们先简单介绍一下 IP 服务。

2.1　IP 服务的特点

IP 协议是 TCP/IP 协议族的动力，它为上层协议提供无状态、无连接、不可靠的服务。

无状态（stateless）是指 IP 通信双方不同步传输数据的状态信息，因此所有 IP 数据报的发送、传输和接收都是相互独立、没有上下文关系的。这种服务最大的缺点是无法处理乱序和重复的 IP 数据报。比如发送端发送出的第 *N* 个 IP 数据报可能比第 *N*+1 个 IP 数据报后到达接收端，而同一个 IP 数据报也可能经过不同的路径多次到达接收端。在这两种情况下，接收端的 IP 模块无法检测到乱序和重复，因为这些 IP 数据报之间没有任何上下文关系。接收端的 IP 模块只要收到了完整的 IP 数据报（如果是 IP 分片的话，IP 模块将先执行重组），就将其数据部分（TCP 报文段、UDP 数据报或者 ICMP 报文）上交给上层协议。那么从上层协议来看，这些数据就可能是乱序的、重复的。面向连接的协议，比如 TCP 协议，则能够自己处理乱序的、重复的报文段，它递交给上层协议的内容绝对是有序的、正确的。

虽然 IP 数据报头部提供了一个标识字段（见后文）用以唯一标识一个 IP 数据报，但它是被用来处理 IP 分片和重组的，而不是用来指示接收顺序的。

无状态服务的优点也很明显：简单、高效。我们无须为保持通信的状态而分配一些内核资源，也无须每次传输数据时都携带状态信息。在网络协议中，无状态是很常见的，比如 UDP 协议和 HTTP 协议都是无状态协议。以 HTTP 协议为例，一个浏览器的连续两次网页请求之间没有任何关联，它们将被 Web 服务器独立地处理。

无连接（connectionless）是指 IP 通信双方都不长久地维持对方的任何信息。这样，上层协议每次发送数据的时候，都必须明确指定对方的 IP 地址。

不可靠是指 IP 协议不能保证 IP 数据报准确地到达接收端，它只是承诺尽最大努力（best effort）。很多种情况都能导致 IP 数据报发送失败。比如，某个中转路由器发现 IP 数据报在网络上存活的时间太长（根据 IP 数据报头部字段 TTL 判断，见后文），那么它将丢弃之，并返回一个 ICMP 错误消息（超时错误）给发送端。又比如，接收端发现收到的 IP 数据报不正确（通过校验机制），它也将丢弃之，并返回一个 ICMP 错误消息（IP 头部参数错误）给发送端。无论哪种情况，发送端的 IP 模块一旦检测到 IP 数据报发送失败，就通知上层协议发送失败，而不会试图重传。因此，使用 IP 服务的上层协议（比如 TCP 协议）需要自己实现数据确认、超时重传等机制以达到可靠传输的目的。

2.2 IPv4 头部结构

2.2.1 IPv4 头部结构

IPv4 的头部结构如图 2-1 所示。其长度通常为 20 字节，除非含有可变长的选项部分。

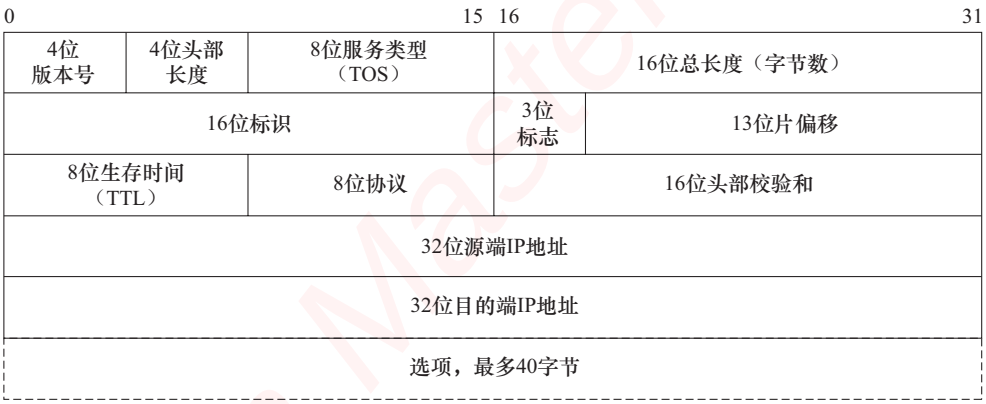

图 2-1　IPv4 头部结构

4 位版本号（version）指定 IP 协议的版本。对 IPv4 来说，其值是 4。其他 IPv4 协议的扩展版本（如 SIP 协议和 PIP 协议），则具有不同的版本号（它们的头部结构也和图 2-1 不同）。

4 位头部长度（header length）标识该 IP 头部有多少个 32 bit 字（4 字节）。因为 4 位最大能表示 15，所以 IP 头部最长是 60 字节。

8 位服务类型（Type Of Service，TOS）包括一个 3 位的优先权字段（现在已经被忽略），4 位的 TOS 字段和 1 位保留字段（必须置 0）。4 位的 TOS 字段分别表示：最小延时，最大吞吐量，最高可靠性和最小费用。其中最多有一个能置为 1，应用程序应该根据实际需要来设置它。比如像 ssh 和 telnet 这样的登录程序需要的是最小延时的服务，而文件传输程序 ftp 则需要最大吞吐量的服务。

16 位总长度（total length）是指整个 IP 数据报的长度，以字节为单位，因此 IP 数据报的最大长度为 65 535（2^{16}–1）字节。但由于 MTU 的限制，长度超过 MTU 的数据报都将被分片传输，所以实际传输的 IP 数据报（或分片）的长度都远远没有达到最大值。接下来的 3 个字段则描述了如何实现分片。

16 位标识（identification）唯一地标识主机发送的每一个数据报。其初始值由系统随机生成；每发送一个数据报，其值就加 1。该值在数据报分片时被复制到每个分片中，因此同一个数据报的所有分片都具有相同的标识值。

3 位标志字段的第一位保留。第二位（Don’t Fragment，DF）表示“禁止分片”。如果设置了这个位，IP 模块将不对数据报进行分片。在这种情况下，如果 IP 数据报长度超过 MTU 的话，IP 模块将丢弃该数据报并返回一个 ICMP 差错报文。第三位（More Fragment，MF）表示“更多分片”。除了数据报的最后一个分片外，其他分片都要把它置 1。

13 位分片偏移（fragmentation offset）是分片相对原始 IP 数据报开始处（仅指数据部分）的偏移。实际的偏移值是该值左移 3 位（乘 8）后得到的。由于这个原因，除了最后一个 IP 分片外，每个 IP 分片的数据部分的长度必须是 8 的整数倍（这样才能保证后面的 IP 分片拥有一个合适的偏移值）。

8 位生存时间（Time To Live，TTL）是数据报到达目的地之前允许经过的路由器跳数。TTL 值被发送端设置（常见的值是 64）。数据报在转发过程中每经过一个路由，该值就被路由器减 1。当 TTL 值减为 0 时，路由器将丢弃数据报，并向源端发送一个 ICMP 差错报文。TTL 值可以防止数据报陷入路由循环。

8 位协议（protocol）用来区分上层协议，我们在第 1 章讨论过。/etc/protocols 文件定义了所有上层协议对应的 protocol 字段的数值。其中，ICMP 是 1，TCP 是 6，UDP 是 17。/etc/protocols 文件是 RFC 1700 的一个子集。

16 位头部校验和（header checksum）由发送端填充，接收端对其使用 CRC 算法以检验 IP 数据报头部（注意，仅检验头部）在传输过程中是否损坏。

32 位的源端 IP 地址和目的端 IP 地址用来标识数据报的发送端和接收端。一般情况下，这两个地址在整个数据报的传递过程中保持不变，而不论它中间经过多少个中转路由器。关于这一点，我们将在第 4 章进一步讨论。

IPv4 最后一个选项字段（option）是可变长的可选信息。这部分最多包含 40 字节，因为 IP 头部最长是 60 字节（其中还包含前面讨论的 20 字节的固定部分）。可用的 IP 选项包括：

- 记录路由（record route），告诉数据报途经的所有路由器都将自己的 IP 地址填入 IP 头部的选项部分，这样我们就可以跟踪数据报的传递路径。
- 时间戳（timestamp），告诉每个路由器都将数据报被转发的时间（或时间与 IP 地址对）填入 IP 头部的选项部分，这样就可以测量途经路由之间数据报传输的时间。
- 松散源路由选择（loose source routing），指定一个路由器 IP 地址列表，数据报发送过

程中必须经过其中所有的路由器。

- 严格源路由选择（strict source routing），和松散源路由选择类似，不过数据报只能经过被指定的路由器。

关于 IP 头部选项字段更详细的信息，请参考 IP 协议的标准文档 RFC 791。不过这些选项字段很少被使用，使用松散源路由选择和严格源路由选择选项的例子大概仅有 traceroute 程序。此外，作为记录路由 IP 选项的替代品，traceroute 程序使用 UDP 报文和 ICMP 报文实现了更可靠的记录路由功能，详情请参考文档 RFC 1393。

2.2.2 使用 tcpdump 观察 IPv4 头部结构

为了深入理解 IPv4 头部中每个字段的含义，我们从测试机器 ernest-laptop 上执行 telnet 命令登录本机，并用 tcpdump 抓取这个过程中 telnet 客户端程序和 telnet 服务器程序之间交换的数据包。具体的操作过程如下：

```
$ sudo tcpdump -ntx -i lo                # 抓取本地回路上的数据包
$ telnet 127.0.0.1                       # 开启另一个终端执行 telnet 命令登录本机
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Ubuntu 9.10
ernest-laptop login: ernest              # 输入用户名并回车
Password:                                # 输入密码并回车
```

此时观察 tcpdump 输出的第一个数据包，其内容如代码清单 2-1 所示。

代码清单 2-1　用 tcpdump 抓取数据包

```
IP 127.0.0.1.41621 > 127.0.0.1.23: Flags [S], seq 3499745539, win 32792,
options [mss 16396,sackOK,TS val 40781017 ecr 0,nop,wscale 6], length 0
    0x0000:  4510 003c a5da 4000 4006 96cf 7f00 0001
    0x0010:  7f00 0001 a295 0017 d099 e103 0000 0000
    0x0020:  a002 8018 fe30 0000 0204 400c 0402 080a
    0x0030:  026e 44d9 0000 0000 0103 0306
```

该数据包描述的是一个 IP 数据报。由于我们是使用 telnet 登录本机的，所以 IP 数据报的源端 IP 地址和目的端 IP 地址都是“127.0.0.1”。telnet 服务器程序使用的端口号是 23（参见 /etc/services 文件），而 telnet 客户端程序使用临时端口号 41621 与服务器通信。关于临时端口号，我们将在第 3 章讨论。“Flags”、“seq”、“win”和“options”描述的都是 TCP 头部信息，这也将在第 3 章讨论。“length”指出该 IP 数据报所携带的应用程序数据的长度。

这次抓包我们开启了 tcpdump 的 -x 选项，使之输出数据包的二进制码。此数据包共包含 60 字节，其中前 20 字节是 IP 头部，后 40 字节是 TCP 头部，不包含应用程序数据（length 值为 0）。现在我们分析 IP 头部的每个字节，如表 2-1 所示。

表 2-1　IPv4 头部各个字段详解

十六进制数	十进制表示 ㊀	IP 头部信息
0x4	4	IP 版本号
0x5	5	头部长度为 5 个 32 位（20 字节）
0x10		TOS 选项中最小延时服务被开启
0x003c	60	数据报总长度，60 字节
0xa5da		数据报标识
0x4		设置了禁止分片标志
0x000	0	分片偏移
0x40	64	TTL 被设为 64
0x06	6	协议字段为 6，表示上层协议是 TCP 协议
0x96cf		IP 头部校验和
0x7f000001		32 位源端 IP 地址 127.0.0.1
0x7f000001		32 位目的端 IP 地址 127.0.0.1

由表 2-1 可见，telnet 服务选择使用具有最小延时的服务，并且默认使用的传输层协议是 TCP 协议（回顾第 1 章讨论的分用）。这些都符合我们通常的理解。这个 IP 数据报没有被分片，因为它没有携带任何应用程序数据。接下来我们将抓取并讨论被分片的 IP 数据报。

2.3　IP 分片

前文曾提到，当 IP 数据报的长度超过帧的 MTU 时，它将被分片传输。分片可能发生在发送端，也可能发生在中转路由器上，而且可能在传输过程中被多次分片，但只有在最终的目标机器上，这些分片才会被内核中的 IP 模块重新组装。

IP 头部中的如下三个字段给 IP 的分片和重组提供了足够的信息：数据报标识、标志和片偏移。一个 IP 数据报的每个分片都具有自己的 IP 头部，它们具有相同的标识值，但具有不同的片偏移。并且除了最后一个分片外，其他分片都将设置 MF 标志。此外，每个分片的 IP 头部的总长度字段将被设置为该分片的长度。

以太网帧的 MTU 是 1500 字节（可以通过 ifconfig 命令或者 netstat 命令查看），因此它携带的 IP 数据报的数据部分最多是 1480 字节（IP 头部占用 20 字节）。考虑用 IP 数据报封装一个长度为 1481 字节的 ICMP 报文（包括 8 字节的 ICMP 头部，所以其数据部分长度为 1473 字节），则该数据报在使用以太网帧传输时必须被分片，如图 2-2 所示。

图 2-2 中，长度为 1501 字节的 IP 数据报被拆分成两个 IP 分片，第一个 IP 分片长度为 1500 字节，第二个 IP 分片的长度为 21 字节。每个 IP 分片都包含自己的 IP 头部（20 字节），且第一个 IP 分片的 IP 头部设置了 MF 标志，而第二个 IP 分片的 IP 头部则没有设置该标志，因为它已经是最后一个分片了。原始 IP 数据报中的 ICMP 头部内容被完整地复制到了第一

㊀ 此列中的空格表示我们并不关心相应字段的十进制值。

个 IP 分片中。第二个 IP 分片不包含 ICMP 头部信息，因为 IP 模块重组该 ICMP 报文的时候只需要一份 ICMP 头部信息，重复传送这个信息没有任何益处。1473 字节的 ICMP 报文数据的前 1472 字节被 IP 模块复制到第一个 IP 分片中，使其总长度为 1500 字节，从而满足 MTU 的要求；而多出的最后 1 字节则被复制到第二个 IP 分片中。

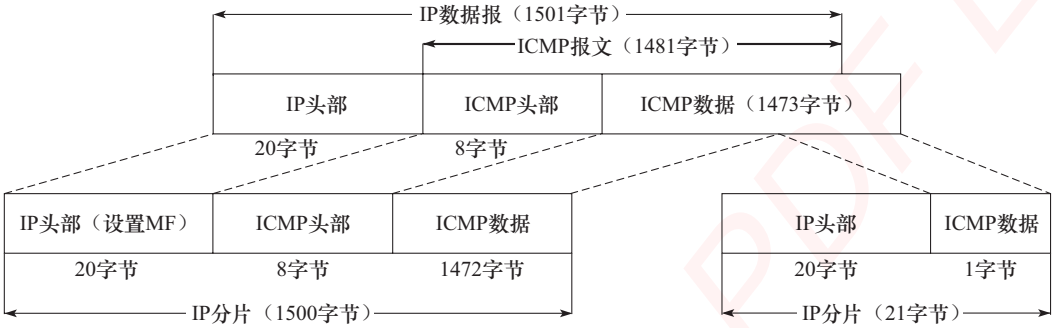

图 2-2　携带 ICMP 报文的 IP 数据报被分片

需要指出的是，ICMP 报文的头部长度取决于报文的类型，其变化范围很大。图 2-2 以 8 字节为例，因为后面的例子用到了 ping 程序，而 ping 程序使用的 ICMP 回显和应答报文的头部长度是 8 字节。

为了看清楚 IP 分片的具体过程，考虑从 ernest-laptop 来 ping 机器 Kongming20，每次传送 1473 字节的数据（这是 ICMP 报文的数据部分）以强制引起 IP 分片，并用 tcpdump 抓取这一过程中双方交换的数据包。具体操作过程如下：

```
$ sudo tcpdump -ntv -i eth0 icmp          # 只抓取 ICMP 报文
$ ping Kongming20 -s 1473                 # 用 -s 选项指定每次发送 1473 字节的数据
```

下面我们考察 tcpdump 输出的一个 IP 数据报的两个分片，其内容如下：

```
1. IP(tos 0x0, ttl 64, id 61197, offset 0, flags [+], proto ICMP (1), length 1500)
     192.168.1.108 > 192.168.1.110: ICMP echo request, id 41737, seq 1, length 1480
2. IP(tos 0x0, ttl 64, id 61197, offset 1480, flags [none], proto ICMP (1), length 21)
     192.168.1.108 > 192.168.1.110: icmp
```

这两个 IP 分片的标识值都是 61197，说明它们是同一个 IP 数据报的分片。第一个分片的片偏移值为 0，而第二个则是 1480。很显然，第二个分片的片偏移值实际上也是第一个分片的 ICMP 报文的长度。第一个分片设置了 MF 标志以表示还有后续分片，所以 tcpdump 输出“flags［+］”。而第二个分片则没有设置任何标志，所以 tcpdump 输出“flags［none］”。这个两个分片的长度分别为 1500 字节和 21 字节，这与图 2-2 描述的一致。

最后，IP 层传递给数据链路层的数据可能是一个完整的 IP 数据报，也可能是一个 IP 分片，它们统称为 IP 分组（packet）。本书如无特殊声明，不区分 IP 数据报和 IP 分组。

2.4　IP 路由

IP 协议的一个核心任务是数据报的路由，即决定发送数据报到目标机器的路径。为了理

解 IP 路由过程，我们先简要分析 IP 模块的基本工作流程。

2.4.1 IP 模块工作流程

IP 模块基本工作流程如图 2-3 所示。

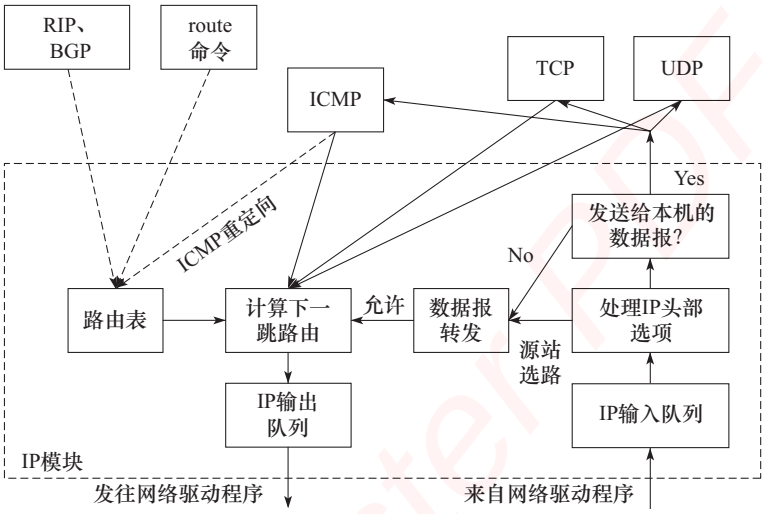

图 2-3 IP 模块基本工作流程

我们从右往左来分析图 2-3。当 IP 模块接收到来自数据链路层的 IP 数据报时，它首先对该数据报的头部做 CRC 校验，确认无误之后就分析其头部的具体信息。

如果该 IP 数据报的头部设置了源站选路选项（松散源路由选择或严格源路由选择），则 IP 模块调用数据报转发子模块来处理该数据报。如果该 IP 数据报的头部中目标 IP 地址是本机的某个 IP 地址，或者是广播地址，即该数据报是发送给本机的，则 IP 模块就根据数据报头部中的协议字段来决定将它派发给哪个上层应用（分用）。如果 IP 模块发现这个数据报不是发送给本机的，则也调用数据报转发子模块来处理该数据报。

数据报转发子模块将首先检测系统是否允许转发，如果不允许，IP 模块就将数据报丢弃。如果允许，数据报转发子模块将对该数据报执行一些操作，然后将它交给 IP 数据报输出子模块。我们将在后面讨论数据报转发的具体过程。

IP 数据报应该发送至哪个下一跳路由（或者目标机器），以及经过哪个网卡来发送，就是 IP 路由过程，即图 2-3 中“计算下一跳路由”子模块。IP 模块实现数据报路由的核心数据结构是路由表。这个表按照数据报的目标 IP 地址分类，同一类型的 IP 数据报将被发往相同的下一跳路由器（或者目标机器）。我们将在后面讨论 IP 路由过程。

IP 输出队列中存放的是所有等待发送的 IP 数据报，其中除了需要转发的 IP 数据报外，还包括封装了本机上层数据（ICMP 报文、TCP 报文段和 UDP 数据报）的 IP 数据报。

图 2-3 中的虚线箭头显示了路由表更新的过程。这一过程是指通过路由协议或者 route 命令调整路由表，使之更适应最新的网络拓扑结构，称为 IP 路由策略。我们将在后面简单讨

论它。

2.4.2 路由机制

要研究 IP 路由机制，需要先了解路由表的内容。我们可以使用 route 命令或 netstat 命令查看路由表。在测试机器 ernest-laptop 上执行 route 命令，输出内容如代码清单 2-2 所示。

代码清单 2-2 路由表实例

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags   Metric   Ref    Use Iface
default         192.168.1.1     0.0.0.0         UG      0        0        0 eth0
192.168.1.0     *               255.255.255.0   U       1        0        0 eth0
```

该路由表包含两项，每项都包含 8 个字段，如表 2-2 所示。

表 2-2 路由表内容

字　段	含　义
Destination	目标网络或主机
Gateway	网关地址，* 表示目标和本机在同一个网络，不需要路由
Genmask	网络掩码
Flags	路由项标志，常见标志有如下 5 种（更多标志见 route 命令的 man 手册）： ❑ U，该路由项是活动的； ❑ H，该路由项的目标是一台主机； ❑ G，该路由项的目标是网关； ❑ D，该路由项是由重定向生成的； ❑ M，该路由项被重定向修改过
Metric	路由距离，即到达指定网络所需的中转数
Ref	路由项被引用的次数（Linux 未使用）
Use	该路由项被使用的次数
Iface	该路由项对应的输出网卡接口

代码清单 2-2 所示的路由表中，第一项的目标地址是 default，即所谓的默认路由项。该项包含一个“G”标志，说明路由的下一跳目标是网关，其地址是 192.168.1.1（这是测试网络中路由器的本地 IP 地址）。另外一个路由项的目标地址是 192.168.1.0，它指的是本地局域网。该路由项的网关地址为 *，说明数据报不需要路由中转，可以直接发送到目标机器。

那么路由表是如何按照 IP 地址分类的呢？或者说给定数据报的目标 IP 地址，它将匹配路由表中的哪一项呢？这就是 IP 的路由机制，分为 3 个步骤：

1）查找路由表中和数据报的目标 IP 地址完全匹配的主机 IP 地址。如果找到，就使用该路由项，没找到则转步骤 2。

2）查找路由表中和数据报的目标 IP 地址具有相同网路 ID 的网络 IP 地址（比如代码清单 2-2 所示的路由表中的第二项）。如果找到，就使用该路由项；没找到则转步骤 3。

3）选择默认路由项，这通常意味着数据报的下一跳路由是网关。

因此，对于测试机器 ernest-laptop 而言，所有发送到 IP 地址为 192.168.1.* 的机器的 IP 数据报都可以直接发送到目标机器（匹配路由表第二项），而所有访问因特网的请求都将通过网关来转发（匹配默认路由项）。

2.4.3 路由表更新

路由表必须能够更新，以反映网络连接的变化，这样 IP 模块才能准确、高效地转发数据报。route 命令可以修改路由表。我们看如下几个例子（在机器 ernest-laptop 上执行）：

```
$ sudo route add -host 192.168.1.109 dev eth0
$ sudo route del -net 192.168.1.0 netmask 255.255.255.0
$ sudo route del default
$ sudo route add default gw 192.168.1.109 dev eth0
```

第 1 行表示添加主机 192.168.1.109（机器 Kongming20）对应的路由项。这样设置之后，所有从 ernest-laptop 发送到 Kongming20 的 IP 数据报将通过网卡 eth0 直接发送至目标机器的接收网卡。第 2 行表示删除网络 192.168.1.0 对应的路由项。这样，除了机器 Kongming20 外，测试机器 ernest-laptop 将无法访问该局域网上的任何其他机器（能访问到 Kongming20 是由于执行了上一条命令）。第 3 行表示删除默认路由项，这样做的后果是无法访问因特网。第 4 行表示重新设置默认路由项，不过这次其网关是机器 Kongming20（而不是能直接访问因特网的路由器）！经过上述修改后的路由表如下：

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags   Metric   Ref   Use Iface
Kongming20      *               255.255.255.255 UH      0        0     0   eth0
default         Kongming20      0.0.0.0         UG      0        0     0   eth0
```

这个新的路由表中，第一个路由项是主机路由项，所以它被设置了“H”标志。我们设计这样一个路由表的目的是为后文讨论 ICMP 重定向提供环境。

通过 route 命令或其他工具手工修改路由表，是静态的路由更新方式。对于大型的路由器，它们通常通过 BGP（Border Gateway Protocol，边际网关协议）、RIP（Routing Information Protocol，路由信息协议）、OSPF 等协议来发现路径，并更新自己的路由表。这种更新方式是动态的、自动的。这部分内容超出了本书的讨论范围，感兴趣的读者可阅读参考资料 1。

2.5 IP 转发

前文提到，不是发送给本机的 IP 数据报将由数据报转发子模块来处理。路由器都能执行数据报的转发操作，而主机一般只发送和接收数据报，这是因为主机上 /proc/sys/net/ipv4/ip_forward 内核参数默认被设置为 0。我们可以通过修改它来使能主机的数据报转发功能（在测试机器 Kongming20 上以 root 身份执行）：

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

对于允许 IP 数据报转发的系统（主机或路由器），数据报转发子模块将对期望转发的数据报执行如下操作：

1）检查数据报头部的 TTL 值。如果 TTL 值已经是 0，则丢弃该数据报。

2）查看数据报头部的严格源路由选择选项。如果该选项被设置，则检测数据报的目标 IP 地址是否是本机的某个 IP 地址。如果不是，则发送一个 ICMP 源站选路失败报文给发送端。

3）如果有必要，则给源端发送一个 ICMP 重定向报文，以告诉它一个更合理的下一跳路由器。

4）将 TTL 值减 1。

5）处理 IP 头部选项。

6）如果有必要，则执行 IP 分片操作。

2.6 重定向

图 2-3 显示了 ICMP 重定向报文也能用于更新路由表，因此本节我们简要讨论 ICMP 重定向。

2.6.1 ICMP 重定向报文

ICMP 重定向报文格式如图 2-4 所示。

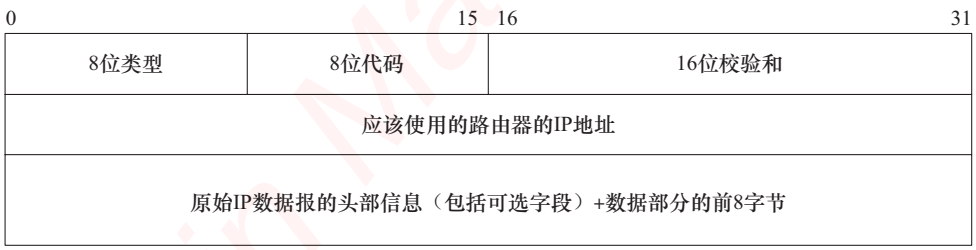

图 2-4 ICMP 重定向报文格式

我们在 1.1 节讨论过 ICMP 报文头部的 3 个固定字段：8 位类型、8 位代码和 16 位校验和。ICMP 重定向报文的类型值是 5，代码字段有 4 个可选值，用来区分不同的重定向类型。本书仅讨论主机重定向，其代码值为 1。

ICMP 重定向报文的数据部分含义很明确，它给接收方提供了如下两个信息：

- 引起重定向的 IP 数据报（即图 2-4 中的原始 IP 数据报）的源端 IP 地址。
- 应该使用的路由器的 IP 地址。

接收主机根据这两个信息就可以断定引起重定向的 IP 数据报应该使用哪个路由器来转发，并且以此来更新路由表（通常是更新路由表缓冲，而不是直接更改路由表）。

/proc/sys/net/ipv4/conf/all/send_redirects 内核参数指定是否允许发送 ICMP 重定向报文，

而 /proc/sys/net/ipv4/conf/all/accept_redirects 内核参数则指定是否允许接收 ICMP 重定向报文。一般来说，主机只能接收 ICMP 重定向报文，而路由器只能发送 ICMP 重定向报文。

2.6.2 主机重定向实例

2.4.3 节中，我们把机器 ernest-laptop 的网关设置成了机器 Kongming20，2.5 节中我们又使能了 Kongming20 的数据报转发功能，因此机器 ernest-laptop 将通过 Kongming20 来访问因特网，比如在 ernest-laptop 上执行如下 ping 命令：

```
$ ping www.baidu.com
PING www.a.shifen.com (119.75.217.56) 56(84) bytes of data.
From Kongming20 (192.168.1.109): icmp_seq=1 Redirect Host(New nexthop: 192.168.1.1)
64 bytes from 119.75.217.56: icmp_seq=1 ttl=54 time=6.78 ms

--- www.a.shifen.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 6.789/6.789/6.789/0.000 ms
```

从 ping 命令的输出来看，Kongming20 给 ernest-laptop 发送了一个 ICMP 重定向报文，告诉它请通过 192.168.1.1 来访问目标机器，因为这对 ernest-laptop 来说是更合理的路由方式。当主机 ernest-laptop 收到这样的 ICMP 重定向报文后，它将更新其路由表缓冲（使用命令 route -Cn 查看），并使用新的路由方式来发送后续数据报。上面讨论的重定向过程可用图 2-5 来总结。

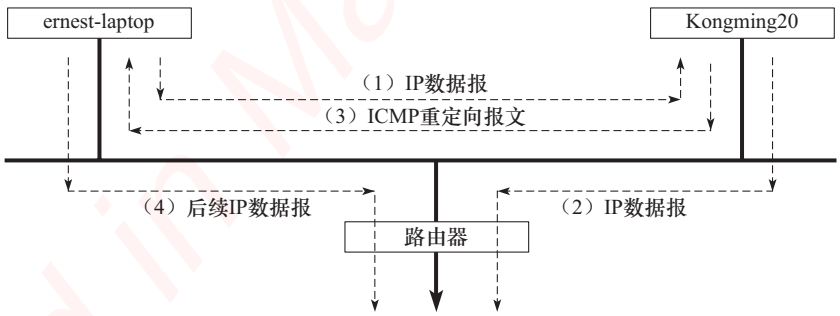

图 2-5 主机重定向过程

2.7 IPv6 头部结构

IPv6 协议是网络层技术发展的必然趋势。它不仅解决了 IPv4 地址不够用的问题，还做了很大的改进。比如，增加了多播和流的功能，为网络上多媒体内容的质量提供精细的控制；引入自动配置功能，使得局域网管理更方便；增加了专门的网络安全功能等。本节简要地讨论 IPv6 头部结构，它的更多细节请参考其标准文档 RFC 2460。

2.7.1 IPv6 固定头部结构

IPv6 头部由 40 字节的固定头部和可变长的扩展头部组成。图 2-6 所示是 IPv6 的固定头部结构。

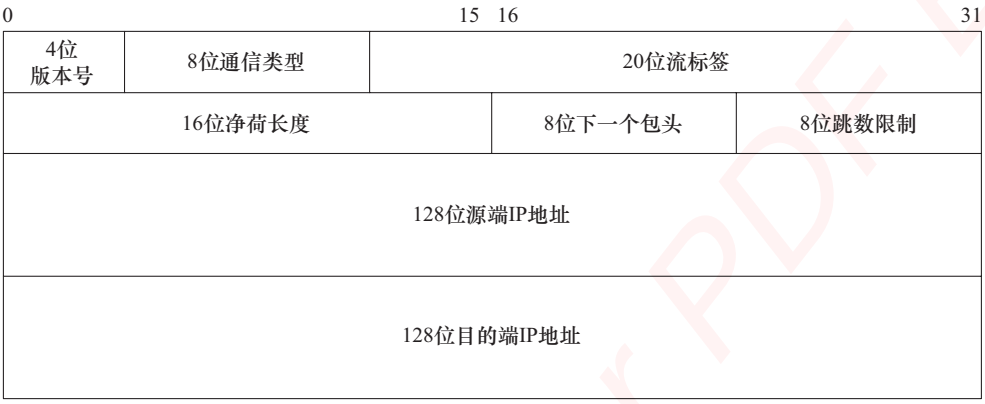

图 2-6 IPv6 固定头部结构

4 位版本号（version）指定 IP 协议的版本。对 IPv6 来说，其值是 6。

8 位通信类型（traffic class）指示数据流通信类型或优先级，和 IPv4 中的 TOS 类似。

20 位流标签（flow label）是 IPv6 新增加的字段，用于某些对连接的服务质量有特殊要求的通信，比如音频或视频等实时数据传输。

16 位净荷长度（payload length）指的是 IPv6 扩展头部和应用程序数据长度之和，不包括固定头部长度。

8 位下一个包头（next header）指出紧跟 IPv6 固定头部后的包头类型，如扩展头（如果有的话）或某个上层协议头（比如 TCP，UDP 或 ICMP）。它类似于 IPv4 头部中的协议字段，且相同的取值有相同的含义。

8 位跳数限制（hop limit）和 IPv4 中的 TTL 含义相同。

IPv6 用 128 位（16 字节）来表示 IP 地址，使得 IP 地址的总量达到了 2^{128} 个。所以有人说，“IPv6 使得地球上的每粒沙子都有一个 IP 地址”。

32 位表示的 IPv4 地址一般用点分十进制来表示，而 IPv6 地址则用十六进制字符串表示，比如“FE80:0000:0000:0000:1234:5678:0000:0012”。可见，IPv6 地址用“:”分割成 8 组，每组包含 2 字节。但这种表示方法过于麻烦，通常可以使用所谓的零压缩法来将其简写，也就是省略连续的、全零的组。比如，上面的例子使用零压缩法可表示为“FE80::1234:5678:0000:0012”。不过零压缩法对一个 IPv6 地址只能使用一次，比如上面的例子中，字节组“5678”后面的全零组就不能再省略，否则我们就无法计算每个“::”之间省略了多少个全零组。

2.7.2 IPv6 扩展头部

可变长的扩展头部使得 IPv6 能支持更多的选项，并且很便于将来的扩展需要。它的长度可以是 0，表示数据报没使用任何扩展头部。一个数据报可以包含多个扩展头部，每个扩展头部的类型由前一个头部（固定头部或扩展头部）中的下一个报头字段指定。目前可以使用的扩展头部如表 2-3 所示。

表 2-3 IPv6 扩展头部

扩展头部	含 义
Hop-by-Hop	逐跳选项头部，它包含每个路由器都必须检查和处理的特殊参数选项
Destination options	目的选项头部，指定由最终目的节点处理的选项
Routing	路由头部，指定数据报要经过哪些中转路由器，功能类似于 IPv4 的松散源路由选择选项和记录路由选项
Fragment	分片头部，处理分片和重组的细节
Authentication	认证头部，提供数据源认证、数据完整性检查和反重播保护
Encapsulating Security Payload	加密头部，提供加密服务
No next header	没有后续扩展头部

注意 IPv6 协议并不是 IPv4 协议的简单扩展，而是完全独立的协议。用以太网帧封装的 IPv6 数据报和 IPv4 数据报具有不同的类型值。第 1 章提到，IPv4 数据报的以太网帧封装类型值是 0x800，而 IPv6 数据报的以太网帧封装类型值是 0x86dd（见 RFC 2464）。

第 3 章　TCP 协议详解

TCP 协议是 TCP/IP 协议族中另一个重要的协议。和 IP 协议相比，TCP 协议更靠近应用层，因此在应用程序中具有更强的可操作性。一些重要的 socket 选项都和 TCP 协议相关。

本章从如下四方面来讨论 TCP 协议：

- TCP 头部信息。TCP 头部信息出现在每个 TCP 报文段中，用于指定通信的源端端口号、目的端端口号，管理 TCP 连接，控制两个方向的数据流。
- TCP 状态转移过程。TCP 连接的任意一端都是一个状态机。在 TCP 连接从建立到断开的整个过程中，连接两端的状态机将经历不同的状态变迁。理解 TCP 状态转移对于调试网络应用程序将有很大的帮助。
- TCP 数据流。通过分析 TCP 数据流，我们就可以从网络应用程序外部来了解应用层协议和通信双方交换的应用程序数据。这一部分将讨论两种类型的 TCP 数据流：交互数据流和成块数据流。TCP 数据流中有一种特殊的数据，称为紧急数据，我们也将简单讨论之。
- TCP 数据流的控制。为了保证可靠传输和提高网络通信质量，内核需要对 TCP 数据流进行控制。这一部分讨论 TCP 数据流控制的两个方面：超时重传和拥塞控制。

不过在详细讨论 TCP 协议之前，我们先简单介绍一下 TCP 服务的特点，以及它和 UDP 服务的区别。

3.1　TCP 服务的特点

传输层协议主要有两个：TCP 协议和 UDP 协议。TCP 协议相对于 UDP 协议的特点是：面向连接、字节流和可靠传输。

使用 TCP 协议通信的双方必须先建立连接，然后才能开始数据的读写。双方都必须为该连接分配必要的内核资源，以管理连接的状态和连接上数据的传输。TCP 连接是全双工的，即双方的数据读写可以通过一个连接进行。完成数据交换之后，通信双方都必须断开连接以释放系统资源。

TCP 协议的这种连接是一对一的，所以基于广播和多播（目标是多个主机地址）的应用程序不能使用 TCP 服务。而无连接协议 UDP 则非常适合于广播和多播。

我们在 1.1 节中简单介绍过字节流服务和数据报服务的区别。这种区别对应到实际编程中，则体现为通信双方是否必须执行相同次数的读、写操作（当然，这只是表现形式）。当发送端应用程序连续执行多次写操作时，TCP 模块先将这些数据放入 TCP 发送缓冲区中。当 TCP 模块真正开始发送数据时，发送缓冲区中这些等待发送的数据可能被封装成一个或多个

TCP 报文段发出。因此，TCP 模块发送出的 TCP 报文段的个数和应用程序执行的写操作次数之间没有固定的数量关系。

当接收端收到一个或多个 TCP 报文段后，TCP 模块将它们携带的应用程序数据按照 TCP 报文段的序号（见后文）依次放入 TCP 接收缓冲区中，并通知应用程序读取数据。接收端应用程序可以一次性将 TCP 接收缓冲区中的数据全部读出，也可以分多次读取，这取决于用户指定的应用程序读缓冲区的大小。因此，应用程序执行的读操作次数和 TCP 模块接收到的 TCP 报文段个数之间也没有固定的数量关系。

综上所述，发送端执行的写操作次数和接收端执行的读操作次数之间没有任何数量关系，这就是字节流的概念：应用程序对数据的发送和接收是没有边界限制的。UDP 则不然。发送端应用程序每执行一次写操作，UDP 模块就将其封装成一个 UDP 数据报并发送之。接收端必须及时针对每一个 UDP 数据报执行读操作（通过 recvfrom 系统调用），否则就会丢包（这经常发生在较慢的服务器上）。并且，如果用户没有指定足够的应用程序缓冲区来读取 UDP 数据，则 UDP 数据将被截断。

图 3-1 和图 3-2 显示了 TCP 字节流服务和 UDP 数据报服务的上述区别。两图中省略了传输层以下的通信细节。

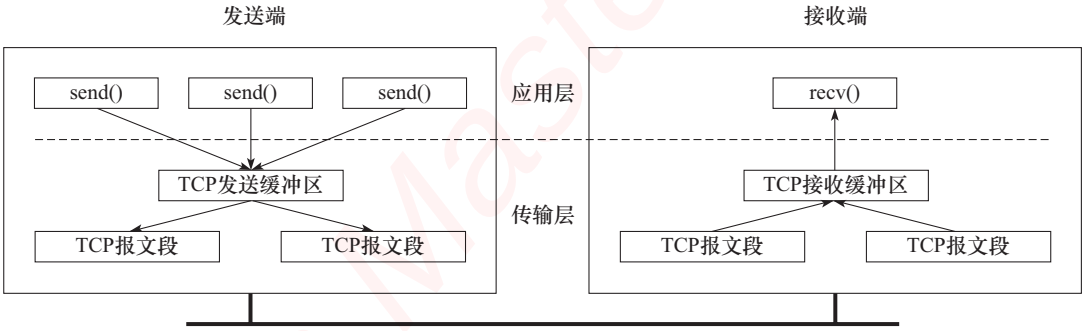

图 3-1　TCP 字节流服务

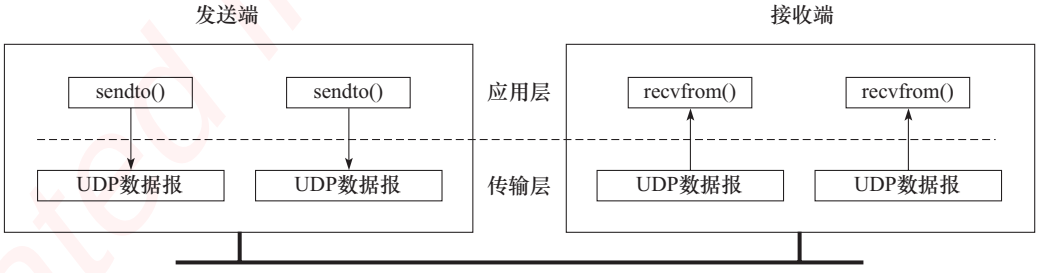

图 3-2　UDP 数据报服务

TCP 传输是可靠的。首先，TCP 协议采用发送应答机制，即发送端发送的每个 TCP 报文段都必须得到接收方的应答，才认为这个 TCP 报文段传输成功。其次，TCP 协议采用超

时重传机制，发送端在发送出一个 TCP 报文段之后启动定时器，如果在定时时间内未收到应答，它将重发该报文段。最后，因为 TCP 报文段最终是以 IP 数据报发送的，而 IP 数据报到达接收端可能乱序、重复，所以 TCP 协议还会对接收到的 TCP 报文段重排、整理，再交付给应用层。

UDP 协议则和 IP 协议一样，提供不可靠服务。它们都需要上层协议来处理数据确认和超时重传。

3.2 TCP 头部结构

TCP 头部信息出现在每个 TCP 报文段中，用于指定通信的源端端口，目的端端口，管理 TCP 连接等，本节详细介绍 TCP 的头部结构，包括固定头部结构和头部选项。

3.2.1 TCP 固定头部结构

TCP 头部结构如图 3-3 所示，其中的诸多字段为管理 TCP 连接和控制数据流提供了足够的信息。

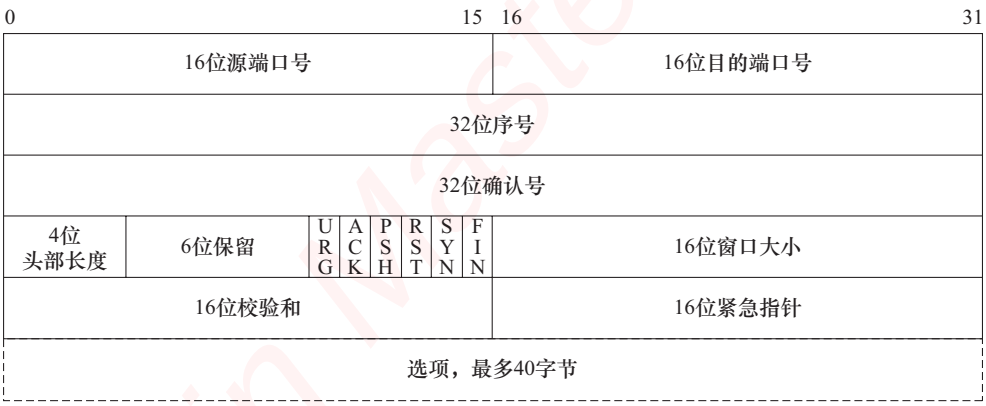

图 3-3　TCP 头部结构

16 位端口号（port number）：告知主机该报文段是来自哪里（源端口）以及传给哪个上层协议或应用程序（目的端口）的。进行 TCP 通信时，客户端通常使用系统自动选择的临时端口号，而服务器则使用知名服务端口号。1.3 节中提到过，所有知名服务使用的端口号都定义在 /etc/services 文件中。

32 位序号（sequence number）：一次 TCP 通信（从 TCP 连接建立到断开）过程中某一个传输方向上的字节流的每个字节的编号。假设主机 A 和主机 B 进行 TCP 通信，A 发送给 B 的第一个 TCP 报文段中，序号值被系统初始化为某个随机值 ISN（Initial Sequence Number，初始序号值）。那么在该传输方向上（从 A 到 B），后续的 TCP 报文段中序号值将被系统设置成 ISN 加上该报文段所携带数据的第一个字节在整个字节流中的偏移。例

如，某个 TCP 报文段传送的数据是字节流中的第 1025 ～ 2048 字节，那么该报文段的序号值就是 ISN+1025。另外一个传输方向（从 B 到 A）的 TCP 报文段的序号值也具有相同的含义。

32 位确认号（acknowledgement number）：用作对另一方发送来的 TCP 报文段的响应。其值是收到的 TCP 报文段的序号值加 1。假设主机 A 和主机 B 进行 TCP 通信，那么 A 发送出的 TCP 报文段不仅携带自己的序号，而且包含对 B 发送来的 TCP 报文段的确认号。反之，B 发送出的 TCP 报文段也同时携带自己的序号和对 A 发送来的报文段的确认号。

4 位头部长度（header length）：标识该 TCP 头部有多少个 32bit 字（4 字节）。因为 4 位最大能表示 15，所以 TCP 头部最长是 60 字节。

6 位标志位包含如下几项：

- URG 标志，表示紧急指针（urgent pointer）是否有效。
- ACK 标志，表示确认号是否有效。我们称携带 ACK 标志的 TCP 报文段为确认报文段。
- PSH 标志，提示接收端应用程序应该立即从 TCP 接收缓冲区中读走数据，为接收后续数据腾出空间（如果应用程序不将接收到的数据读走，它们就会一直停留在 TCP 接收缓冲区中）。
- RST 标志，表示要求对方重新建立连接。我们称携带 RST 标志的 TCP 报文段为复位报文段。
- SYN 标志，表示请求建立一个连接。我们称携带 SYN 标志的 TCP 报文段为同步报文段。
- FIN 标志，表示通知对方本端要关闭连接了。我们称携带 FIN 标志的 TCP 报文段为结束报文段。

16 位窗口大小（window size）：是 TCP 流量控制的一个手段。这里说的窗口，指的是接收通告窗口（Receiver Window，RWND）。它告诉对方本端的 TCP 接收缓冲区还能容纳多少字节的数据，这样对方就可以控制发送数据的速度。

16 位校验和（TCP checksum）：由发送端填充，接收端对 TCP 报文段执行 CRC 算法以检验 TCP 报文段在传输过程中是否损坏。注意，这个校验不仅包括 TCP 头部，也包括数据部分。这也是 TCP 可靠传输的一个重要保障。

16 位紧急指针（urgent pointer）：是一个正的偏移量。它和序号字段的值相加表示最后一个紧急数据的下一字节的序号。因此，确切地说，这个字段是紧急指针相对当前序号的偏移，不妨称之为紧急偏移。TCP 的紧急指针是发送端向接收端发送紧急数据的方法。我们将在后面讨论 TCP 紧急数据。

3.2.2 TCP 头部选项

TCP 头部的最后一个选项字段（options）是可变长的可选信息。这部分最多包含 40 字节，因为 TCP 头部最长是 60 字节（其中还包含前面讨论的 20 字节的固定部分）。典型的 TCP 头部选项结构如图 3-4 所示。

kind（1字节）	length（1字节）	info（*n*字节）

图 3-4　TCP 头部选项的一般结构

选项的第一个字段 kind 说明选项的类型。有的 TCP 选项没有后面两个字段，仅包含 1 字节的 kind 字段。第二个字段 length（如果有的话）指定该选项的总长度，该长度包括 kind 字段和 length 字段占据的 2 字节。第三个字段 info（如果有的话）是选项的具体信息。常见的 TCP 选项有 7 种，如图 3-5 所示。

kind=0						
kind=1						
kind=2	length=4	最大segment长度（2字节）				
kind=3	length=3	移位数（1字节）				
kind=4	length=2					
kind=5	length =*N**8+2	第1块 左边沿	第1块 右边沿	…	第*N*块 左边沿	第*N*块 右边沿
kind=8	length=10	时间戳值（4字节）			时间戳回显应答（4字节）	

图 3-5　7 种 TCP 选项

kind=0 是选项表结束选项。

kind=1 是空操作（nop）选项，没有特殊含义，一般用于将 TCP 选项的总长度填充为 4 字节的整数倍。

kind=2 是最大报文段长度选项。TCP 连接初始化时，通信双方使用该选项来协商最大报文段长度（Max Segment Size，MSS）。TCP 模块通常将 MSS 设置为（MTU–40）字节（减掉的这 40 字节包括 20 字节的 TCP 头部和 20 字节的 IP 头部）。这样携带 TCP 报文段的 IP 数据报的长度就不会超过 MTU（假设 TCP 头部和 IP 头部都不包含选项字段，并且这也是一般情况），从而避免本机发生 IP 分片。对以太网而言，MSS 值是 1460（1500–40）字节。

kind=3 是窗口扩大因子选项。TCP 连接初始化时，通信双方使用该选项来协商接收通告窗口的扩大因子。在 TCP 的头部中，接收通告窗口大小是用 16 位表示的，故最大为 65 535 字节，但实际上 TCP 模块允许的接收通告窗口大小远不止这个数（为了提高 TCP 通信的吞吐量）。窗口扩大因子解决了这个问题。假设 TCP 头部中的接收通告窗口大小是 N，窗口扩大因子（移位数）是 M，那么 TCP 报文段的实际接收通告窗口大小是 N 乘 2^M，或者说 N 左移 M 位。注意，M 的取值范围是 0 ～ 14。我们可以通过修改 /proc/sys/net/ipv4/tcp_window_scaling 内核变量来启用或关闭窗口扩大因子选项。

和 MSS 选项一样，窗口扩大因子选项只能出现在同步报文段中，否则将被忽略。但同步报文段本身不执行窗口扩大操作，即同步报文段头部的接收通告窗口大小就是该 TCP 报文段的实际接收通告窗口大小。当连接建立好之后，每个数据传输方向的窗口扩大因子就固定不变了。关于窗口扩大因子选项的细节，可参考标准文档 RFC 1323。

kind=4 是选择性确认（Selective Acknowledgment，SACK）选项。TCP 通信时，如果某个 TCP 报文段丢失，则 TCP 模块会重传最后被确认的 TCP 报文段后续的所有报文段，这样原先已经正确传输的 TCP 报文段也可能重复发送，从而降低了 TCP 性能。SACK 技术正是为改善这种情况而产生的，它使 TCP 模块只重新发送丢失的 TCP 报文段，不用发送所有未被确认的 TCP 报文段。选择性确认选项用在连接初始化时，表示是否支持 SACK 技术。我们可以通过修改 /proc/sys/net/ipv4/tcp_sack 内核变量来启用或关闭选择性确认选项。

kind=5 是 SACK 实际工作的选项。该选项的参数告诉发送方本端已经收到并缓存的不连续的数据块，从而让发送端可以据此检查并重发丢失的数据块。每个块边沿（edge of block）参数包含一个 4 字节的序号。其中块左边沿表示不连续块的第一个数据的序号，而块右边沿则表示不连续块的最后一个数据的序号的下一个序号。这样一对参数（块左边沿和块右边沿）之间的数据是没有收到的。因为一个块信息占用 8 字节，所以 TCP 头部选项中实际上最多可以包含 4 个这样的不连续数据块（考虑选项类型和长度占用的 2 字节）。

kind=8 是时间戳选项。该选项提供了较为准确的计算通信双方之间的回路时间（Round Trip Time，RTT）的方法，从而为 TCP 流量控制提供重要信息。我们可以通过修改 /proc/sys/net/ipv4/tcp_timestamps 内核变量来启用或关闭时间戳选项。

3.2.3 使用 tcpdump 观察 TCP 头部信息

在 2.3 节中，我们利用 tcpdump 抓取了一个数据包并分析了其中的 IP 头部信息，本节分析其中与 TCP 协议相关的部分（后面的分析中，我们将所有 tcpdump 抓取到的数据包都称为 TCP 报文段，因为 TCP 报文段既是数据包的主要内容，也是我们主要讨论的对象）。为了方便阅读，先将该 TCP 报文段的内容复制于代码清单 3-1 中。

代码清单 3-1　用 tcpdump 抓取数据包

```
    IP 127.0.0.1.41621 > 127.0.0.1.23: Flags [S], seq 3499745539, win 32792,
options [mss 16396,sackOK,TS val 40781017 ecr 0,nop,wscale 6], length 0
        0x0000:  4510 003c a5da 4000 4006 96cf 7f00 0001
        0x0010:  7f00 0001 a295 0017 d099 e103 0000 0000
        0x0020:  a002 8018 fe30 0000 0204 400c 0402 080a
        0x0030:  026e 44d9 0000 0000 0103 0306
```

tcpdump 输出 Flags[S]，表示该 TCP 报文段包含 SYN 标志，因此它是一个同步报文段。如果 TCP 报文段包含其他标志，则 tcpdump 也会将该标志的首字母显示在“Flags”后的方括号中。

seq 是序号值。因为该同步报文段是从 127.0.0.1.41621（客户端 IP 地址和端口号）到

127.0.0.1.23（服务器 IP 地址和端口号）这个传输方向上的第一个 TCP 报文段，所以这个序号值也就是此次通信过程中该传输方向的 ISN 值。并且，因为这是整个通信过程中的第一个 TCP 报文段，所以它没有针对对方发送来的 TCP 报文段的确认值（尚未收到任何对方发送来的 TCP 报文段）。

win 是接收通告窗口的大小。因为这是一个同步报文段，所以 win 值反映的是实际的接收通告窗口大小。

options 是 TCP 选项，其具体内容列在方括号中。mss 是发送端（客户端）通告的最大报文段长度。通过 ifconfig 命令查看回路接口的 MTU 为 16436 字节，因此可以预想到 TCP 报文段的 MSS 为 16396（16436–40）字节。sackOK 表示发送端支持并同意使用 SACK 选项。TS val 是发送端的时间戳。ecr 是时间戳回显应答。因为这是一次 TCP 通信的第一个 TCP 报文段，所以它针对对方的时间戳的应答为 0（尚未收到对方的时间戳）。紧接着的 nop 是一个空操作选项。wscale 指出发送端使用的窗口扩大因子为 6。

接下来我们分析 tcpdump 输出的字节码中 TCP 头部对应的信息，它从第 21 字节开始，如表 3-1 所示。

表 3-1　TCP 头部

十六进制数	十进制表示[⊖]	TCP 头部信息
0xa295	41621	源端口号
0x0017	23	目的端口号
0xd099e103	3499745539	序号
0x00000000	0	确认号
0xa	10	TCP 头部长度为 10 个 32 位（40 字节）
0x002		设置了 SYN 标志
0x8018	32792	接收通告窗口大小
0xfe30		头部校验和
0x0000		没设置 URG 标志，所以紧急指针值无意义
0x0204		最大报文段长度选项的 kind 值和 length 值
0x400c	16396	最大报文段长度
0x0402		允许 SACK 选项
0x080a		时间戳选项的 kind 值和 length 值
0x026e44d9	40781017	时间戳
0x00000000	0	回显应答时间戳
0x01		空操作选项
0x0303		窗口扩大因子选项的 kind 值和 length 值
0x06	6	窗口扩大因子为 6

⊖ 此列中的空格表示我们并不关心相应字段的十进制值。

从表 3-1 中可见，TCP 报文段头部的二进制码和 tcpdump 输出的 TCP 报文段描述信息完全对应。在后面的 tcpdump 输出中，我们将省略大部分 TCP 头部信息，仅显示序号、确认号、窗口大小以及标志位等与主题相关的字段。

3.3 TCP 连接的建立和关闭

本节我们讨论建立和关闭 TCP 连接的过程。

3.3.1 使用 tcpdump 观察 TCP 连接的建立和关闭

首先从 ernest-laptop 上执行 telnet 命令登录 Kongming20 的 80 端口，然后抓取这一过程中客户端和服务器交换的 TCP 报文段。具体操作过程如下：

```
    $ sudo tcpdump -i eth0 -nt '(src 192.168.1.109 and dst 192.168.1.108) or (src
192.168.1.108 and dst 192.168.1.109)'
    $ telnet 192.168.1.109 80
    Trying 192.168.1.109...
    Connected to 192.168.1.109.
    Escape character is '^]'.
    ^]（回车） # 输入 ctrl+] 并回车

    telnet> quit（回车）
    Connection closed.
```

当执行 telnet 命令并在两台通信主机之间建立 TCP 连接后（telnet 输出“Connected to 192.168.1.109”），输入 Ctrl+］以调出 telnet 程序的命令提示符，然后在 telnet 命令提示符后输入 quit 以退出 telnet 客户端程序，从而结束 TCP 连接。整个过程中（从连接建立到结束），tcpdump 输出的内容如代码清单 3-2 所示。

代码清单 3-2　建立和关闭 TCP 连接的过程

```
1. IP 192.168.1.108.60871 > 192.168.1.109.80: Flags [S], seq 535734930, win
   5840, length 0
2. IP 192.168.1.109.80 > 192.168.1.108.60871: Flags [S.], seq 2159701207, ack
   535734931, win 5792, length 0
3. IP 192.168.1.108.60871 > 192.168.1.109.80: Flags [.], ack 1, win 92, length 0
4. IP 192.168.1.108.60871 > 192.168.1.109.80: Flags [F.], seq 1, ack 1, win 92,
   length 0
5. IP 192.168.1.109.80 > 192.168.1.108.60871: Flags [.], ack 2, win 91, length 0
6. IP 192.168.1.109.80 > 192.168.1.108.60871: Flags [F.], seq 1, ack 2, win 91,
   length 0
7. IP 192.168.1.108.60871 > 192.168.1.109.80: Flags [.], ack 2, win 92, length 0
```

因为整个过程并没有发生应用层数据的交换，所以 TCP 报文段的数据部分的长度（length）总是 0。为了更清楚地表示建立和关闭 TCP 连接的整个过程，我们将 tcpdump 输出的内容绘制成图 3-6 所示的时序图。

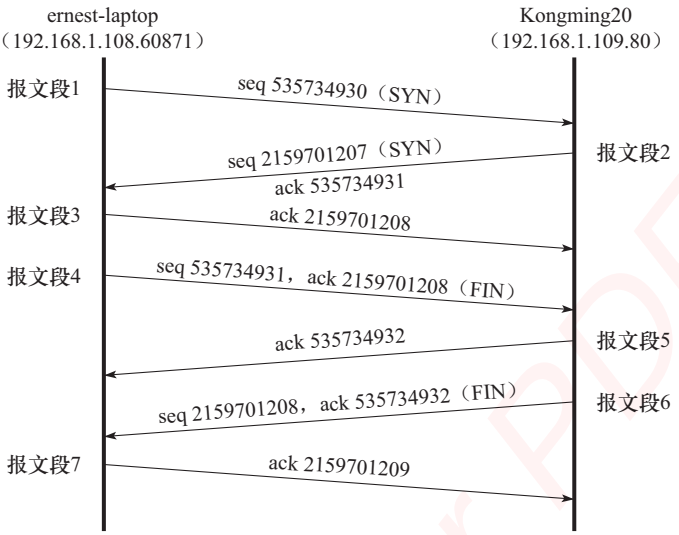

图 3-6　TCP 连接的建立和关闭时序图

第 1 个 TCP 报文段包含 SYN 标志，因此它是一个同步报文段，即 ernest-laptop（客户端）向 Kongming20（服务器）发起连接请求。同时，该同步报文段包含一个 ISN 值为 535734930 的序号。第 2 个 TCP 报文段也是同步报文段，表示 Kongming20 同意与 ernest-laptop 建立连接。同时它发送自己的 ISN 值为 2159701207 的序号，并对第 1 个同步报文段进行确认。确认值是 535734931，即第 1 个同步报文段的序号值加 1。前文说过，序号值是用来标识 TCP 数据流中的每一字节的。但同步报文段比较特殊，即使它并没有携带任何应用程序数据，它也要占用一个序号值。第 3 个 TCP 报文段是 ernest-laptop 对第 2 个同步报文段的确认。至此，TCP 连接就建立起来了。建立 TCP 连接的这 3 个步骤被称为 TCP 三次握手。

从第 3 个 TCP 报文段开始，tcpdump 输出的序号值和确认值都是相对初始 ISN 值的偏移。当然，我们可以开启 tcpdump 的 -S 选项来选择打印序号的绝对值。

后面 4 个 TCP 报文段是关闭连接的过程。第 4 个 TCP 报文段包含 FIN 标志，因此它是一个结束报文段，即 ernest-laptop 要求关闭连接。结束报文段和同步报文段一样，也要占用一个序号值。Kongming20 用 TCP 报文段 5 来确认该结束报文段。紧接着 Kongming20 发送自己的结束报文段 6，ernest-laptop 则用 TCP 报文段 7 给予确认。实际上，仅用于确认目的的确认报文段 5 是可以省略的，因为结束报文段 6 也携带了该确认信息。确认报文段 5 是否出现在连接断开的过程中，取决于 TCP 的延迟确认特性。延迟确认将在后面讨论。

在连接的关闭过程中，因为 ernest-laptop 先发送结束报文段（telnet 客户端程序主动退出），故称 ernest-laptop 执行主动关闭，而称 Kongming20 执行被动关闭。

一般而言，TCP 连接是由客户端发起，并通过三次握手建立（特殊情况是所谓同时打开[1]）的。TCP 连接的关闭过程相对复杂一些。可能是客户端执行主动关闭，比如前面的例子；也可能是服务器执行主动关闭，比如服务器程序被中断而强制关闭连接；还可能是同时关闭

（和同时打开一样，非常少见）。

3.3.2 半关闭状态

TCP 连接是全双工的，所以它允许两个方向的数据传输被独立关闭。换言之，通信的一端可以发送结束报文段给对方，告诉它本端已经完成了数据的发送，但允许继续接收来自对方的数据，直到对方也发送结束报文段以关闭连接。TCP 连接的这种状态称为半关闭（half close）状态，如图 3-7 所示。

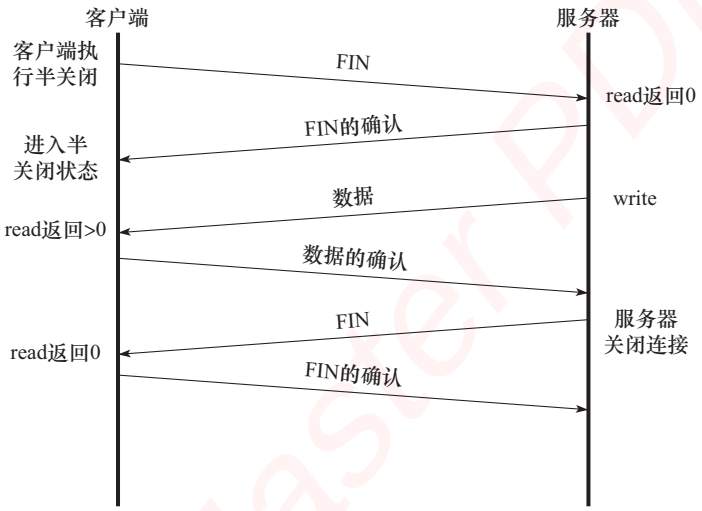

图 3-7 半关闭状态

请注意，在图 3-7 中，服务器和客户端应用程序判断对方是否已经关闭连接的方法是：read 系统调用返回 0（收到结束报文段）。当然，Linux 还提供其他检测连接是否被对方关闭的方法，这将在后续章节讨论。

socket 网络编程接口通过 shutdown 函数提供了对半关闭的支持，我们将在后续章节讨论它。这里强调一下，虽然我们介绍了半关闭状态，但是使用半关闭的应用程序很少见。

3.3.3 连接超时

前面我们讨论的是很快建立连接的情况。如果客户端访问一个距离它很远的服务器，或者由于网络繁忙，导致服务器对于客户端发送出的同步报文段没有应答，此时客户端程序将产生什么样的行为呢？显然，对于提供可靠服务的 TCP 来说，它必然是先进行重连（可能执行多次），如果重连仍然无效，则通知应用程序连接超时。

为了观察连接超时，我们模拟一个繁忙的服务器环境，在 ernest-laptop 上执行下面的操作：

```
$ sudo iptables -F
$ sudo iptables -I INPUT -p tcp --syn -i eth0 -j DROP
```

iptable 命令用于过滤数据包，这里我们利用它来丢弃所有接收到的连接请求（丢弃所有同步报文段，这样客户端就无法得到任何确认报文段）。

接下来从 Kongming20 上执行 telnet 命令登录到 ernest-laptop，并用 tcpdump 抓取这个过程中双方交换的 TCP 报文段。具体操作如下：

```
$ sudo tcpdump -n -i eth0 port 23        # 仅抓取 telnet 客户端和服务器交换的数据包
$ date; telnet 192.168.1.108; date       # 在 telnet 命令前后都执行 date 命令，以计算超时时间
Mon Jun 11 21:23:35 CST 2012
Trying 192.168.1.108...
telnet: connect to address 192.168.1.108: Connection timed out
Mon Jun 11 21:24:38 CST 2012
```

从两次 date 命令的输出来看，Kongming20 建立 TCP 连接的超时时间是 63 s。本次 tcpdump 的输出如代码清单 3-3 所示。

代码清单 3-3　TCP 超时重连

```
1. 21:23:35.612136 IP 192.168.1.109.39385 > 192.168.1.108.telnet: Flags [S],
   seq 1355982096, length 0
2. 21:23:36.613146 IP 192.168.1.109.39385 > 192.168.1.108.telnet: Flags [S],
   seq 1355982096, length 0
3. 21:23:38.617279 IP 192.168.1.109.39385 > 192.168.1.108.telnet: Flags [S],
   seq 1355982096, length 0
4. 21:23:42.625140 IP 192.168.1.109.39385 > 192.168.1.108.telnet: Flags [S],
   seq 1355982096, length 0
5. 21:23:50.641344 IP 192.168.1.109.39385 > 192.168.1.108.telnet: Flags [S],
   seq 1355982096, length 0
6. 21:24:06.673331 IP 192.168.1.109.39385 > 192.168.1.108.telnet: Flags [S],
   seq 1355982096, length 0
```

这次抓包我们保留了 tcpdump 输出的时间戳（不使用其 -t 选项），以便推理 Linux 的超时重连策略。

我们一共抓取到 6 个 TCP 报文段，它们都是同步报文段，并且具有相同的序号值，这说明后面 5 个同步报文段都是超时重连报文段。观察这些 TCP 报文段被发送的时间间隔，它们分别为 1 s、2 s、4 s、8 s 和 16 s（由于定时器精度的问题，这些时间间隔都有一定偏差），可以推断最后一个 TCP 报文段的超时时间是 32 s（63 s–16 s–8 s–4 s–2 s–1 s）。因此，TCP 模块一共执行了 5 次重连操作，这是由 /proc/sys/net/ipv4/tcp_syn_retries 内核变量所定义的。每次重连的超时时间都增加一倍。在 5 次重连均失败的情况下，TCP 模块放弃连接并通知应用程序。

在应用程序中，我们可以修改连接超时时间，具体方法将在本书后续章节中进行介绍。

3.4　TCP 状态转移

TCP 连接的任意一端在任一时刻都处于某种状态，当前状态可以通过 netstat 命令（见第 17 章）查看。本节我们要讨论的是 TCP 连接从建立到关闭的整个过程中通信两端状态的变

化。图 3-8 是完整的状态转移图，它描绘了所有的 TCP 状态以及可能的状态转换。

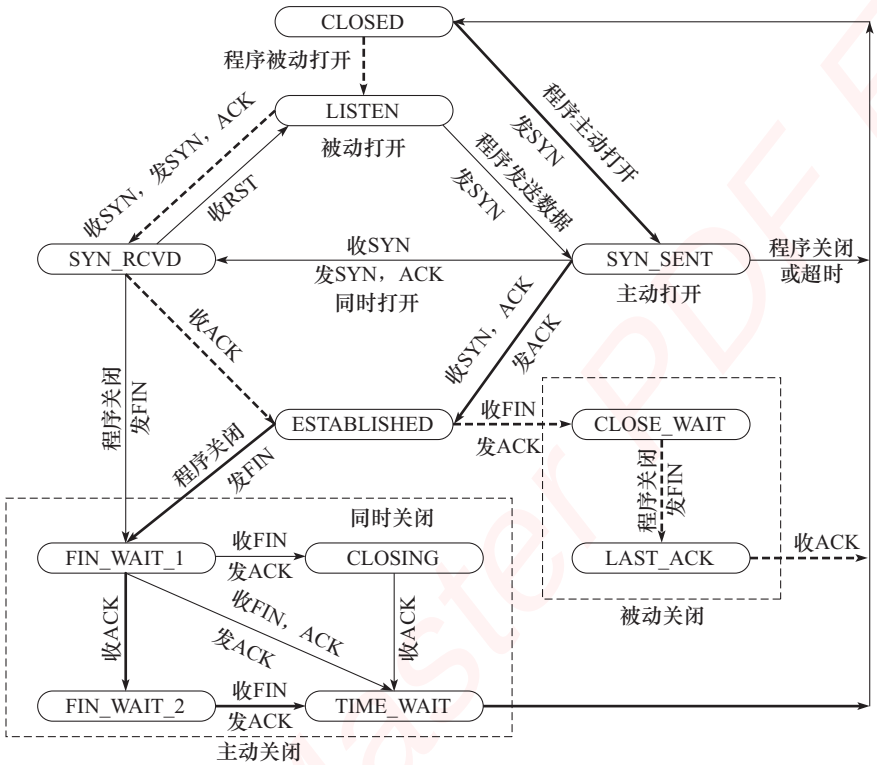

图 3-8　TCP 状态转移过程

图 3-8 中的粗虚线表示典型的服务器端连接的状态转移；粗实线表示典型的客户端连接的状态转移。CLOSED 是一个假想的起始点，并不是一个实际的状态。

3.4.1　TCP 状态转移总图

我们先讨论服务器的典型状态转移过程，此时我们说的连接状态都是指该连接的服务器端的状态。

服务器通过 listen 系统调用（见第 5 章）进入 LISTEN 状态，被动等待客户端连接，因此执行的是所谓的被动打开。服务器一旦监听到某个连接请求（收到同步报文段），就将该连接放入内核等待队列中，并向客户端发送带 SYN 标志的确认报文段。此时该连接处于 SYN_RCVD 状态。如果服务器成功地接收到客户端发送回的确认报文段，则该连接转移到 ESTABLISHED 状态。ESTABLISHED 状态是连接双方能够进行双向数据传输的状态。

当客户端主动关闭连接时（通过 close 或 shutdown 系统调用向服务器发送结束报文段），服务器通过返回确认报文段使连接进入 CLOSE_WAIT 状态。这个状态的含义很明确：等待

服务器应用程序关闭连接。通常，服务器检测到客户端关闭连接后，也会立即给客户端发送一个结束报文段来关闭连接。这将使连接转移到 LAST_ACK 状态，以等待客户端对结束报文段的最后一次确认。一旦确认完成，连接就彻底关闭了。

下面讨论客户端的典型状态转移过程，此时我们说的连接状态都是指该连接的客户端的状态。

客户端通过 connect 系统调用（见第 5 章）主动与服务器建立连接。connect 系统调用首先给服务器发送一个同步报文段，使连接转移到 SYN_SENT 状态。此后，connect 系统调用可能因为如下两个原因失败返回：

- 如果 connect 连接的目标端口不存在（未被任何进程监听），或者该端口仍被处于 TIME_WAIT 状态的连接所占用（见后文），则服务器将给客户端发送一个复位报文段，connect 调用失败。
- 如果目标端口存在，但 connect 在超时时间内未收到服务器的确认报文段，则 connect 调用失败。

connect 调用失败将使连接立即返回到初始的 CLOSED 状态。如果客户端成功收到服务器的同步报文段和确认，则 connect 调用成功返回，连接转移至 ESTABLISHED 状态。

当客户端执行主动关闭时，它将向服务器发送一个结束报文段，同时连接进入 FIN_WAIT_1 状态。若此时客户端收到服务器专门用于确认目的的确认报文段（比如图 3-6 中的 TCP 报文段 5），则连接转移至 FIN_WAIT_2 状态。当客户端处于 FIN_WAIT_2 状态时，服务器处于 CLOSE_WAIT 状态，这一对状态是可能发生半关闭的状态。此时如果服务器也关闭连接（发送结束报文段），则客户端将给予确认并进入 TIME_WAIT 状态。关于 TIME_WAIT 状态的含义，我们将在下一节讨论。

图 3-8 还给出了客户端从 FIN_WAIT_1 状态直接进入 TIME_WAIT 状态的一条线路（不经过 FIN_WAIT_2 状态），前提是处于 FIN_WAIT_1 状态的服务器直接收到带确认信息的结束报文段（而不是先收到确认报文段，再收到结束报文段）。这种情况对应于图 3-6 中的服务器不发送 TCP 报文段 5。

前面说过，处于 FIN_WAIT_2 状态的客户端需要等待服务器发送结束报文段，才能转移至 TIME_WAIT 状态，否则它将一直停留在这个状态。如果不是为了在半关闭状态下继续接收数据，连接长时间地停留在 FIN_WAIT_2 状态并无益处。连接停留在 FIN_WAIT_2 状态的情况可能发生在：客户端执行半关闭后，未等服务器关闭连接就强行退出了。此时客户端连接由内核来接管，可称之为孤儿连接（和孤儿进程类似）。Linux 为了防止孤儿连接长时间存留在内核中，定义了两个内核变量：/proc/sys/net/ipv4/tcp_max_orphans 和 /proc/sys/net/ipv4/tcp_fin_timeout。前者指定内核能接管的孤儿连接数目，后者指定孤儿连接在内核中生存的时间。

至此，我们简单地讨论了服务器和客户端程序的典型 TCP 状态转移路线。对应于图 3-6 所示的 TCP 连接的建立与断开过程，客户端和服务器的状态转移如图 3-9 所示。

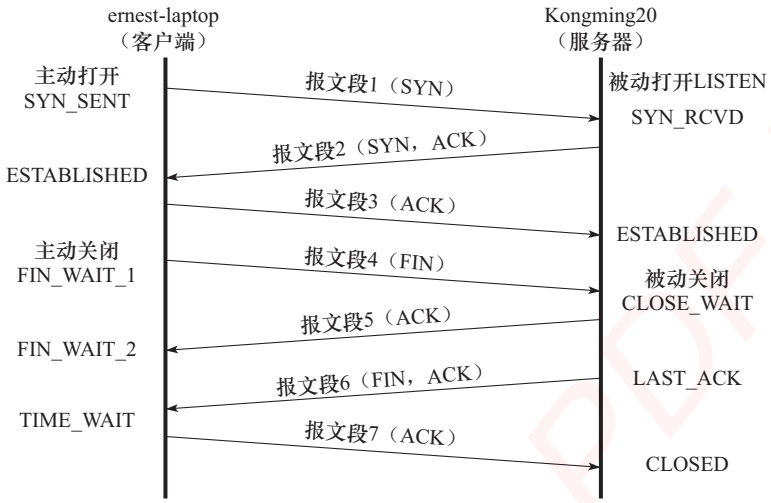

图 3-9　TCP 连接的建立和断开过程中客户端和服务器的状态变化

图 3-8 还描绘了其他非典型的 TCP 状态转移路线，比如同时关闭与同时打开，本书不予讨论。

3.4.2　TIME_WAIT 状态

从图 3-9 来看，客户端连接在收到服务器的结束报文段（TCP 报文段 6）之后，并没有直接进入 CLOSED 状态[㊀]，而是转移到 TIME_WAIT 状态。在这个状态，客户端连接要等待一段长为 2MSL（Maximum Segment Life，报文段最大生存时间）的时间，才能完全关闭。MSL 是 TCP 报文段在网络中的最大生存时间，标准文档 RFC 1122 的建议值是 2 min。

TIME_WAIT 状态存在的原因有两点：

- 可靠地终止 TCP 连接。
- 保证让迟来的 TCP 报文段有足够的时间被识别并丢弃。

第一个原因很好理解。假设图 3-9 中用于确认服务器结束报文段 6 的 TCP 报文段 7 丢失，那么服务器将重发结束报文段。因此客户端需要停留在某个状态以处理重复收到的结束报文段（即向服务器发送确认报文段）。否则，客户端将以复位报文段来回应服务器，服务器则认为这是一个错误，因为它期望的是一个像 TCP 报文段 7 那样的确认报文段。

在 Linux 系统上，一个 TCP 端口不能被同时打开多次（两次及以上）。当一个 TCP 连接处于 TIME_WAIT 状态时，我们将无法立即使用该连接占用着的端口来建立一个新连接。反过来思考，如果不存在 TIME_WAIT 状态，则应用程序能够立即建立一个和刚关闭的连接相似的连接（这里说的相似，是指它们具有相同的 IP 地址和端口号）。这个新的、和原来相似的连接被称为原来的连接的化身（incarnation）。新的化身可能接收到属于原来的连接的、携

㊀ 请读者根据语境判断连接的状态是指客户端状态还是服务器状态，后同。

带应用程序数据的 TCP 报文段（迟到的报文段），这显然是不应该发生的。这就是 TIME_WAIT 状态存在的第二个原因。

另外，因为 TCP 报文段的最大生存时间是 MSL，所以坚持 2MSL 时间的 TIME_WAIT 状态能够确保网络上两个传输方向上尚未被接收到的、迟到的 TCP 报文段都已经消失（被中转路由器丢弃）。因此，一个连接的新的化身可以在 2MSL 时间之后安全地建立，而绝对不会接收到属于原来连接的应用程序数据，这就是 TIME_WAIT 状态要持续 2MSL 时间的原因。

有时候我们希望避免 TIME_WAIT 状态，因为当程序退出后，我们希望能够立即重启它。但由于处在 TIME_WAIT 状态的连接还占用着端口，程序将无法启动（直到 2MSL 超时时间结束）。考虑一个例子：在测试机器 ernest-laptop 上以客户端方式运行 nc（用于创建网络连接的工具，见第 17 章）命令，登录本机的 Web 服务，且明确指定客户端使用 12345 端口与服务器通信。然后从终端输入 Ctrl+C 终止客户端程序，接着又立即重启 nc 程序，以完全相同的方式再次连接本机的 Web 服务。具体操作如下：

```
$ nc -p 12345 192.168.1.108 80
ctrl+C                                  # 中断客户端程序
$ nc -p 12345 192.168.1.108 80          # 重启客户端程序，重新建立连接
nc: bind failed: Address already in use # 输出显示连接失败，因为 12345 端口仍被占用
$ netstat -nat                          # 用 netstat 命令查看连接状态
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 192.168.1.108:12345    192.168.1.108:80      TIME_WAIT
```

这里我们使用 netstat 命令查看连接的状态。其输出显示，客户端程序被中断后，连接进入 TIME_WAIT 状态，12345 端口仍被占用，所以客户端重启失败。

对客户端程序来说，我们通常不用担心上面描述的重启问题。因为客户端一般使用系统自动分配的临时端口号来建立连接，而由于随机性，临时端口号一般和程序上一次使用的端口号（还处于 TIME_WAIT 状态的那个连接使用的端口号）不同，所以客户端程序一般可以立即重启。上面的例子仅仅是为了说明问题，我们强制客户端使用 12345 端口，这才导致立即重启客户端程序失败。

但如果是服务器主动关闭连接后异常终止，则因为它总是使用同一个知名服务端口号，所以连接的 TIME_WAIT 状态将导致它不能立即重启。不过，我们可以通过 socket 选项 SO_REUSEADDR 来强制进程立即使用处于 TIME_WAIT 状态的连接占用的端口，这将在第 5 章讨论。

3.5 复位报文段

在某些特殊条件下，TCP 连接的一端会向另一端发送携带 RST 标志的报文段，即复位报文段，以通知对方关闭连接或重新建立连接。本节讨论产生复位报文段的 3 种情况。

3.5.1 访问不存在的端口

3.4.1 小节提到，当客户端程序访问一个不存在的端口时，目标主机将给它发送一个复位

报文段。考虑从 Kongming20 上执行 telnet 命令登录 ernest-laptop 上一个不存在的 54321 端口，并用 tcpdump 抓取该过程中两台主机交换的 TCP 报文段。具体操作过程如下：

```
$ sudo tcpdump -nt -i eth0 port 54321   # 仅抓取发送至和来自 54321 端口的 TCP 报文段
$ telnet 192.168.1.108 54321
Trying 192.168.1.108...
telnet: connect to address 192.168.1.108: Connection refused
```

telnet 程序的输出显示连接被拒绝了，因为这个端口不存在。tcpdump 抓取到的 TCP 报文段内容如下：

```
1. IP 192.168.1.109.42001 > 192.168.1.108.54321: Flags [S], seq 21621375, win
   14600, length 0
2. IP 192.168.1.108.54321 > 192.168.1.109.42001: Flags [R.], seq 0, ack
   21621376, win 0, length 0
```

由此可见，ernest-laptop 针对 Kongming20 的连接请求（同步报文段）回应了一个复位报文段（tcpdump 输出 R 标志）。因为复位报文段的接收通告窗口大小为 0，所以可以预见：收到复位报文段的一端应该关闭连接或者重新连接，而不能回应这个复位报文段。

实际上，当客户端程序向服务器的某个端口发起连接，而该端口仍被处于 TIME_WAIT 状态的连接所占用时，客户端程序也将收到复位报文段。

3.5.2 异常终止连接

前面讨论的连接终止方式都是正常的终止方式：数据交换完成之后，一方给另一方发送结束报文段。TCP 提供了异常终止一个连接的方法，即给对方发送一个复位报文段。一旦发送了复位报文段，发送端所有排队等待发送的数据都将被丢弃。

应用程序可以使用 socket 选项 SO_LINGER 来发送复位报文段，以异常终止一个连接。我们将在第 5 章讨论 SO_LINGER 选项。

3.5.3 处理半打开连接

考虑下面的情况：服务器（或客户端）关闭或者异常终止了连接，而对方没有接收到结束报文段（比如发生了网络故障），此时，客户端（或服务器）还维持着原来的连接，而服务器（或客户端）即使重启，也已经没有该连接的任何信息了。我们将这种状态称为半打开状态，处于这种状态的连接称为半打开连接。如果客户端（或服务器）往处于半打开状态的连接写入数据，则对方将回应一个复位报文段。

举例来说，我们在 Kongming20 上使用 nc 命令模拟一个服务器程序，使之监听 12345 端口，然后从 ernest-laptop 运行 telnet 命令登录到该端口上，接着拔掉 ernest-laptop 的网线，并在 Kongming20 上中断服务器程序。显然，此时 ernest-laptop 上运行的 telnet 客户端程序维持着一个半打开连接。然后接上 ernest-laptop 的网线，并从客户端程序往半打开连接写入 1 字节的数据“a”。同时，运行 tcpdump 程序抓取整个过程中 telnet 客户端和 nc 服务器交换的 TCP 报文段。具体操作过程如下：

```
$ nc -l 12345                                  # 在 Kongming20 上运行服务器程序
$ sudo tcpdump -nt -i eth0 port 12345
$ telnet 192.168.1.109 12345                   # 在 ernest-laptop 上运行客户端程序
Trying 192.168.1.109...
Connected to 192.168.1.109.
Escape character is '^]'.                      # 此时断开 ernest-laptop 的网线，并重启服务器
a（回车）                                      # 向半打开连接输入字符 a
Connection closed by foreign host.
```

telnet 的输出显示，连接被服务器关闭了。tcpdump 抓取到的 TCP 报文段内容如下：

```
1. IP 192.168.1.108.55100 > 192.168.1.109.12345: Flags [S], seq 3093809365,
   length 0
2. IP 192.168.1.109.12345 > 192.168.1.108.55100: Flags [S.], seq 1495337791,
   ack 3093809366, length 0
3. IP 192.168.1.108.55100 > 192.168.1.109.12345: Flags [.], ack 1, length 0
4. IP 192.168.1.108.55100 > 192.168.1.109.12345: Flags [P.], seq 1:4, ack 1,
   length 3
5. IP 192.168.1.109.12345 > 192.168.1.108.55100: Flags [R], seq 1495337792,
   length 0
```

该输出内容中，前 3 个 TCP 报文段是正常建立 TCP 连接的 3 次握手的过程。第 4 个 TCP 报文段由客户端发送给服务器，它携带了 3 字节的应用程序数据，这 3 字节依次是：字母“a”、回车符“\r”和换行符“\n”。不过因为服务器程序已经被中断，所以 Kongming20 对客户端发送的数据回应了一个复位报文段 5。

3.6 TCP 交互数据流

前面讨论了 TCP 连接及其状态，从本节开始我们讨论通过 TCP 连接交换的应用程序数据。TCP 报文段所携带的应用程序数据按照长度分为两种：交互数据和成块数据。交互数据仅包含很少的字节。使用交互数据的应用程序（或协议）对实时性要求高，比如 telnet、ssh 等。成块数据的长度则通常为 TCP 报文段允许的最大数据长度。使用成块数据的应用程序（或协议）对传输效率要求高，比如 ftp。本节我们讨论交互数据流。

考虑如下情况：在 ernest-laptop 上执行 telnet 命令登录到本机，然后在 shell 命令提示符后执行 ls 命令，同时用 tcpdump 抓取这一过程中 telnet 客户端和 telnet 服务器交换的 TCP 报文段。具体操作过程如下：

```
$ tcpdump -nt -i lo port 23
$ telnet 127.0.0.1
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Ubuntu 9.10
ernest-laptop login: ernest（回车）            # 输入用户名并回车
Password:（回车）                              # 输入密码并回车


ernest@ernest-laptop:~$ ls（回车）
```

上述过程将引起客户端和服务器交换很多 TCP 报文段。下面我们仅列出我们感兴趣的、执行 ls 命令产生的 tcpdump 输出，如代码清单 3-4 所示。

代码清单 3-4　TCP 交互数据流

```
1. IP 127.0.0.1.58130 > 127.0.0.1.23: Flags [P.], seq 1408334812:1408334813,
   ack 1415955507, win 613, length 1
2. IP 127.0.0.1.23 > 127.0.0.1.58130: Flags [P.], seq 1:2, ack 1, win 512,
   length 1
3. IP 127.0.0.1.58130 > 127.0.0.1.23: Flags [.], ack 2, win 613, length 0
4. IP 127.0.0.1.58130 > 127.0.0.1.23: Flags [P.], seq 1:2, ack 2, win 613,
   length 1
5. IP 127.0.0.1.23 > 127.0.0.1.58130: Flags [P.], seq 2:3, ack 2, win 512,
   length 1
6. IP 127.0.0.1.58130 > 127.0.0.1.23: Flags [.], ack 3, win 613, length 0
7. IP 127.0.0.1.58130 > 127.0.0.1.23: Flags [P.], seq 2:4, ack 3, win 613,
   length 2
8. IP 127.0.0.1.23 > 127.0.0.1.58130: Flags [P.], seq 3:176, ack 4, win 512,
   length 173
9. IP 127.0.0.1.58130 > 127.0.0.1.23: Flags [.], ack 176, win 630, length 0
10. IP 127.0.0.1.23 > 127.0.0.1.58130: Flags [P.], seq 176:228, ack 4, win 512,
    length 52
11. IP 127.0.0.1.58130 > 127.0.0.1.23: Flags [.], ack 228, win 630, length 0
```

TCP 报文段 1 由客户端发送给服务器，它携带 1 个字节的应用程序数据，即字母“1”。TCP 报文段 2 是服务器对 TCP 报文段 1 的确认，同时回显字母“1”。TCP 报文段 3 是客户端对 TCP 报文段 2 的确认。第 4 ～ 6 个 TCP 报文段是针对字母“s”的上述过程。TCP 报文段 7 传送的 2 字节数据分别是：客户端键入的回车符和流结束符（EOF，本例中是 0x00）。TCP 报文段 8 携带服务器返回的客户查询的目录的内容（ls 命令的输出），包括该目录下文件的文件名及其显示控制参数。TCP 报文段 9 是客户端对 TCP 报文段 8 的确认。TCP 报文段 10 携带的也是服务器返回给客户端的数据，包括一个回车符、一个换行符、客户端登录用户的 PS1 环境变量（第一级命令提示符）。TCP 报文段 11 是客户端对 TCP 报文段 10 的确认。

在上述过程中，客户端针对服务器返回的数据所发送的确认报文段（TCP 报文段 6、9 和 11）都不携带任何应用程序数据（长度为 0），而服务器每次发送的确认报文段（TCP 报文段 2、5、8 和 10）都包含它需要发送的应用程序数据。服务器的这种处理方式称为延迟确认，即它不马上确认上次收到的数据，而是在一段延迟时间后查看本端是否有数据需要发送，如果有，则和确认信息一起发出。因为服务器对客户请求处理得很快，所以它发送确认报文段的时候总是有数据一起发送。延迟确认可以减少发送 TCP 报文段的数量。而由于用户的输入速度明显慢于客户端程序的处理速度，所以客户端的确认报文段总是不携带任何应用程序数据。前文曾提到，在 TCP 连接的建立和断开过程中，也可能发生延迟确认。

上例是在本地回路运行的结果，在局域网中也能得到基本相同的结果，但在广域网就未必如此了。广域网上的交互数据流可能经受很大的延迟，并且，携带交互数据的微小 TCP 报文段数量一般很多（一个按键输入就导致一个 TCP 报文段），这些因素都可能导致拥塞发生。

解决该问题的一个简单有效的方法是使用 Nagle 算法。

Nagle 算法要求一个 TCP 连接的通信双方在任意时刻都最多只能发送一个未被确认的 TCP 报文段，在该 TCP 报文段的确认到达之前不能发送其他 TCP 报文段。另一方面，发送方在等待确认的同时收集本端需要发送的微量数据，并在确认到来时以一个 TCP 报文段将它们全部发出。这样就极大地减少了网络上的微小 TCP 报文段的数量。该算法的另一个优点在于其自适应性：确认到达得越快，数据也就发送得越快。

3.7 TCP 成块数据流

下面考虑用 FTP 协议传输一个大文件。在 ernest-laptop 上启动一个 vsftpd 服务器程序（升级的、安全版的 ftp 服务器程序），并执行 ftp 命令登录该服务器上，然后在 ftp 命令提示符后输入 get 命令，从服务器下载一个几百兆的大文件。同时用 tcpdump 抓取这一个过程中 ftp 客户端和 vsftpd 服务器交换的 TCP 报文段。具体操作过程如下：

```
$ sudo tcpdump -nt -i eth0 port 20              # vsftpd 服务器程序使用端口号 20
$ ftp 127.0.0.1
Connected to 127.0.0.1.
220 (vsFTPd 2.3.0)
Name (127.0.0.1:ernest): ernest（回车）          # 输入用户名并回车
331 Please specify the password.
Password:（回车）                                # 输入密码并回车
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> get bigfile（回车）                         # 获取大文件 bigfile
```

代码清单 3-5 是该过程的部分 tcpdump 输出。

代码清单 3-5　TCP 成块数据流

```
1. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [.], seq 205783041:205799425, ack 1,
   win 513, length 16384
2. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [.], seq 205799425:205815809, ack 1,
   win 513, length 16384
3. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [.], seq 205815809:205832193, ack 1,
   win 513, length 16384
4. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [P.], seq 205832193:205848577, ack
   1, win 513, length 16384
5. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [.], seq 205848577:205864961, ack 1,
   win 513, length 16384
6. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [.], seq 205864961:205881345, ack 1,
   win 513, length 16384
7. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [.], seq 205881345:205897729, ack 1,
   win 513, length 16384
8. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [P.], seq 205897729:205914113, ack
   1, win 513, length 16384
9. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [.], seq 205914113:205930497, ack 1,
   win 513, length 16384
```

```
10. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [.], seq 205930497:205946881, ack
    1, win 513, length 16384
11. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [.], seq 205946881:205963265, ack
    1, win 513, length 16384
12. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [P.], seq 205963265:205979649, ack
    1, win 513, length 16384
13. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [.], seq 205979649:205996033, ack
    1, win 513, length 16384
14. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [.], seq 205996033:206012417, ack
    1, win 513, length 16384
15. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [.], seq 206012417:206028801, ack
    1, win 513, length 16384
16. IP 127.0.0.1.20 > 127.0.0.1.39651: Flags [P.], seq 206028801:206045185, ack
    1, win 513, length 16384
17. IP 127.0.0.1.39651 > 127.0.0.1.20: Flags [.], ack 205815809, win 30084,
    length 0
18. IP 127.0.0.1.39651 > 127.0.0.1.20: Flags [.], ack 206045185, win 27317,
    length 0
```

注意，客户端发送的最后两个 TCP 报文段 17 和 18，它们分别是对 TCP 报文段 2 和 16 的确认（从序号值和确认值来判断）。由此可见，当传输大量大块数据的时候，发送方会连续发送多个 TCP 报文段，接收方可以一次确认所有这些报文段。那么发送方在收到上一次确认后，能连续发送多少个 TCP 报文段呢？这是由接收通告窗口（还需要考虑拥塞窗口，见后文）的大小决定的。TCP 报文段 17 说明客户端还能接收 30 084×64 字节（本例中窗口扩大因子为 6），即 1 925 376 字节的数据。而在 TCP 报文段 18 中，接收通告窗口大小为 1 748 288 字节，即客户端能接收的数据量变小了。这表明客户端的 TCP 接收缓冲区有更多的数据未被应用程序读取而停留在其中，这些数据都来自 TCP 报文段 3 ～ 16 中的一部分。服务器收到 TCP 报文段 18 后，它至少（因为接收通告窗口可能扩大）还能连续发送的未被确认的报文段数量是 1 748 288/16 384 个，即 106 个（但一般不会连续发送这么多）。其中，16 384 是成块数据的长度（见 TCP 报文段 1 ～ 16 的 length 值），很显然它小于但接近 MSS 规定的 16 396 字节。

另外一个值得注意的地方是，服务器每发送 4 个 TCP 报文段就传送一个 PSH 标志（tcpdump 输出标志 P）给客户端，以通知客户端的应用程序尽快读取数据。不过这对服务器来说显然不是必需的，因为它知道客户端的 TCP 接收缓冲区中还有空闲空间（接收通告窗口大小不为 0）。

下面我们修改系统的 TCP 接收缓冲区和 TCP 发送缓冲区的大小（如何修改将在第 16 章介绍），使之都为 4096 字节，然后重启 vsftpd 服务器，并再次执行上述操作。此次 tcpdump 的部分输出如代码清单 3-6 所示。

代码清单 3-6　修改 TCP 接收和发送缓冲区大小后的 TCP 成块数据流

```
1. IP 127.0.0.1.20 > 127.0.0.1.45227: Flags [.], seq 5195777:5197313, ack 1,
   win 3072, length 1536
2. IP 127.0.0.1.20 > 127.0.0.1.45227: Flags [.], seq 5197313:5198849, ack 1,
```

```
    win 3072, length 1536
3.  IP 127.0.0.1.45227 > 127.0.0.1.20: Flags [.], ack 5198849, win 3072, length 0
4.  IP 127.0.0.1.20 > 127.0.0.1.45227: Flags [P.], seq 5198849:5200385, ack 1,
    win 3072, length 1536
5.  IP 127.0.0.1.45227 > 127.0.0.1.20: Flags [.], ack 5200385, win 3072, length 0
```

从同步报文段（未在代码清单 3-6 中列出）得知在这次通信过程中，客户端和服务器的窗口扩大因子都为 0，因而客户端和服务器每次通告的窗口大小都是 3072 字节（没超过 4096 字节，预料之中）。因为每个成块数据的长度为 1536 字节，所以服务器在收到上一个 TCP 报文段的确认之前最多还能再发送 1 个 TCP 报文段，这正是 TCP 报文段 1 ～ 3 描述的情形。

3.8 带外数据

有些传输层协议具有带外（Out Of Band，OOB）数据的概念，用于迅速通告对方本端发生的重要事件。因此，带外数据比普通数据（也称为带内数据）有更高的优先级，它应该总是立即被发送，而不论发送缓冲区中是否有排队等待发送的普通数据。带外数据的传输可以使用一条独立的传输层连接，也可以映射到传输普通数据的连接中。实际应用中，带外数据的使用很少见，已知的仅有 telnet、ftp 等远程非活跃程序。

UDP 没有实现带外数据传输，TCP 也没有真正的带外数据。不过 TCP 利用其头部中的紧急指针标志和紧急指针两个字段，给应用程序提供了一种紧急方式。TCP 的紧急方式利用传输普通数据的连接来传输紧急数据。这种紧急数据的含义和带外数据类似，因此后文也将 TCP 紧急数据称为带外数据。

我们先来介绍 TCP 发送带外数据的过程。假设一个进程已经往某个 TCP 连接的发送缓冲区中写入了 *N* 字节的普通数据，并等待其发送。在数据被发送前，该进程又向这个连接写入了 3 字节的带外数据“abc”。此时，待发送的 TCP 报文段的头部将被设置 URG 标志，并且紧急指针被设置为指向最后一个带外数据的下一字节（进一步减去当前 TCP 报文段的序号值得到其头部中的紧急偏移值），如图 3-10 所示。

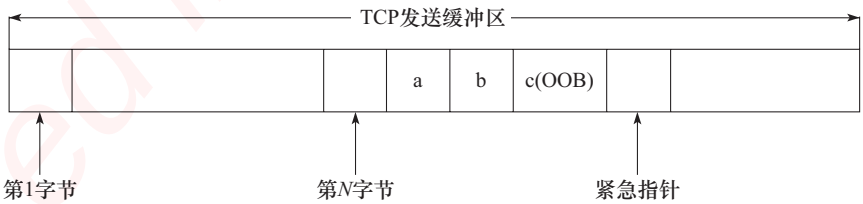

图 3-10　TCP 发送缓冲区中的紧急数据

由图 3-10 可见，发送端一次发送的多字节的带外数据中只有最后一字节被当作带外数据（字母 c），而其他数据（字母 a 和 b）被当成了普通数据。如果 TCP 模块以多个 TCP 报文段来发送图 3-10 所示 TCP 发送缓冲区中的内容，则每个 TCP 报文段都将设置 URG 标志，

并且它们的紧急指针指向同一个位置（数据流中带外数据的下一个位置），但只有一个 TCP 报文段真正携带带外数据。

现在考虑 TCP 接收带外数据的过程。TCP 接收端只有在接收到紧急指针标志时才检查紧急指针，然后根据紧急指针所指的位置确定带外数据的位置，并将它读入一个特殊的缓存中。这个缓存只有 1 字节，称为带外缓存。如果上层应用程序没有及时将带外数据从带外缓存中读出，则后续的带外数据（如果有的话）将覆盖它。

前面讨论的带外数据的接收过程是 TCP 模块接收带外数据的默认方式。如果我们给 TCP 连接设置了 SO_OOBINLINE 选项，则带外数据将和普通数据一样被 TCP 模块存放在 TCP 接收缓冲区中。此时应用程序需要像读取普通数据一样来读取带外数据。那么这种情况下如何区分带外数据和普通数据呢？显然，紧急指针可以用来指出带外数据的位置，socket 编程接口也提供了系统调用来识别带外数据（见第 5 章）。

至此，我们讨论了 TCP 模块发送和接收带外数据的过程。至于内核如何通知应用程序带外数据的到来，以及应用程序如何发送和接收带外数据，将在后续章节讨论。

3.9 TCP 超时重传

在 3.6 节～ 3.8 节中，我们讲述了 TCP 在正常网络情况下的数据流。从本节开始，我们讨论异常网络状况下（开始出现超时或丢包），TCP 如何控制数据传输以保证其承诺的可靠服务。

TCP 服务必须能够重传超时时间内未收到确认的 TCP 报文段。为此，TCP 模块为每个 TCP 报文段都维护一个重传定时器，该定时器在 TCP 报文段第一次被发送时启动。如果超时时间内未收到接收方的应答，TCP 模块将重传 TCP 报文段并重置定时器。至于下次重传的超时时间如何选择，以及最多执行多少次重传，就是 TCP 的重传策略。我们通过实例来研究 Linux 下 TCP 的超时重传策略。

在 ernest-laptop 上启动 iperf 服务器程序，然后从 Kongming20 上执行 telnet 命令登录该服务器程序。接下来，从 telnet 客户端发送一些数据（此处是“1234”）给服务器，然后断开服务器的网线并再次从客户端发送一些数据给服务器（此处是“12”）。同时，用 tcpdump 抓取这一过程中客户端和服务器交换的 TCP 报文段。具体操作过程如下：

```
$ sudo tcpdump -n -i eth0 port 5001
$ iperf -s                                  # 在 ernest-laptop 上执行
$ telnet 192.168.1.108 5001                 # 在 Kongming20 上执行
Trying 192.168.1.108...
Connected to 192.168.1.108.
Escape character is '^]'.
1234                                        # 发送完之后断开服务器网线
12
Connection closed by foreign host
```

iperf 是一个测量网络状况的工具，-s 选项表示将其作为服务器运行。iperf 默认监听 5001 端口，并丢弃该端口上接收到的所有数据，相当于一个 discard 服务器。上述操作过程

的部分 tcpdump 输出如代码清单 3-7 所示。

代码清单 3-7　TCP 超时重传

```
1. 18:44:57.580341 IP 192.168.1.109.38234 > 192.168.1.108.5001: Flags [S], seq
   2381272950, length 0
2. 18:44:57.580477 IP 192.168.1.108.5001 > 192.168.1.109.38234: Flags [S.], seq
   466032301, ack 2381272951, length 0
3. 18:44:57.580498 IP 192.168.1.109.38234 > 192.168.1.108.5001: Flags [.], ack
   1, length 0
4. 18:44:59.866019 IP 192.168.1.109.38234 > 192.168.1.108.5001: Flags [P.], seq
   1:7, ack 1, length 6
5. 18:44:59.866165 IP 192.168.1.108.5001 > 192.168.1.109.38234: Flags [.], ack
   7, length 0
6. 18:45:25.028933 IP 192.168.1.109.38234 > 192.168.1.108.5001: Flags [P.], seq
   7:11, ack 1, length 4
7. 18:45:25.230034 IP 192.168.1.109.38234 > 192.168.1.108.5001: Flags [P.], seq
   7:11, ack 1, length 4
8. 18:45:25.639407 IP 192.168.1.109.38234 > 192.168.1.108.5001: Flags [P.], seq
   7:11, ack 1, length 4
9. 18:45:26.455942 IP 192.168.1.109.38234 > 192.168.1.108.5001: Flags [P.], seq
   7:11, ack 1, length 4
10. 18:45:28.092425 IP 192.168.1.109.38234 > 192.168.1.108.5001: Flags [P.],
    seq 7:11, ack 1, length 4
11. 18:45:31.362473 IP 192.168.1.109.38234 > 192.168.1.108.5001: Flags [P.],
    seq 7:11, ack 1, length 4
12. 18:45:33.100888 ARP, Request who-has 192.168.1.108 tell 192.168.1.109,
    length 28
13. 18:45:34.098156 ARP, Request who-has 192.168.1.108 tell 192.168.1.109,
    length 28
14. 18:45:35.100887 ARP, Request who-has 192.168.1.108 tell 192.168.1.109,
    length 28
15. 18:45:37.902034 ARP, Request who-has 192.168.1.108 tell 192.168.1.109,
    length 28
16. 18:45:38.903126 ARP, Request who-has 192.168.1.108 tell 192.168.1.109,
    length 28
17. 18:45:39.901421 ARP, Request who-has 192.168.1.108 tell 192.168.1.109,
    length 28
18. 18:45:44.440049 ARP, Request who-has 192.168.1.108 tell 192.168.1.109,
    length 28
19. 18:45:45.438840 ARP, Request who-has 192.168.1.108 tell 192.168.1.109,
    length 28
20. 18:45:46.439932 ARP, Request who-has 192.168.1.108 tell 192.168.1.109,
    length 28
21. 18:45:50.976710 ARP, Request who-has 192.168.1.108 tell 192.168.1.109,
    length 28
22. 18:45:51.974134 ARP, Request who-has 192.168.1.108 tell 192.168.1.109,
    length 28
23. 18:45:52.973939 ARP, Request who-has 192.168.1.108 tell 192.168.1.109,
    length 28
```

TCP 报文段 1 ～ 3 是三次握手建立连接的过程，TCP 报文段 4 ～ 5 是客户端发送数据

“1234”（应用程序数据长度为 6，包括回车、换行两个字符，后同）及服务器确认的过程。TCP 报文段 6 是客户端第一次发送数据“12”的过程。因为服务器的网线被断开，所以客户端无法收到 TCP 报文段 6 的确认报文段。此后，客户端对 TCP 报文段 6 执行了 5 次重传，它们是 TCP 报文段 7 ～ 11，这可以从每个 TCP 报文段的序号得知。此后，数据包 12 ～ 23 都是 ARP 模块的输出内容，即 Kongming20 查询 ernest-laptop 的 MAC 地址。

我们保留了 tcpdump 输出的时间戳，以便推理 TCP 的超时重传策略。观察 TCP 报文段 6 ～ 11 被发送的时间间隔，它们分别为 0.2 s、0.4 s、0.8 s、1.6 s 和 3.2 s。由此可见，TCP 一共执行 5 次重传，每次重传超时时间都增加一倍（因此，和 TCP 超时重连的策略相似）。在 5 次重传均失败的情况下，底层的 IP 和 ARP 开始接管，直到 telnet 客户端放弃连接为止。

Linux 有两个重要的内核参数与 TCP 超时重传相关：/proc/sys/net/ipv4/tcp_retries1 和 /proc/sys/net/ipv4/tcp_retries2。前者指定在底层 IP 接管之前 TCP 最少执行的重传次数，默认值是 3。后者指定连接放弃前 TCP 最多可以执行的重传次数，默认值是 15（一般对应 13 ～ 30 min）。在我们的实例中，TCP 超时重传发生了 5 次，连接坚持的时间是 15 min（可以用 date 命令来测量）。

虽然超时会导致 TCP 报文段重传，但 TCP 报文段的重传可以发生在超时之前，即快速重传，这将在下一节中讨论。

3.10 拥塞控制

3.10.1 拥塞控制概述

TCP 模块还有一个重要的任务，就是提高网络利用率，降低丢包率，并保证网络资源对每条数据流的公平性。这就是所谓的拥塞控制。

TCP 拥塞控制的标准文档是 RFC 5681，其中详细介绍了拥塞控制的四个部分：慢启动（slow start）、拥塞避免（congestion avoidance）、快速重传（fast retransmit）和快速恢复（fast recovery）。拥塞控制算法在 Linux 下有多种实现，比如 reno 算法、vegas 算法和 cubic 算法等。它们或者部分或者全部实现了上述四个部分。/proc/sys/net/ipv4/tcp_congestion_control 文件指示机器当前所使用的拥塞控制算法。

拥塞控制的最终受控变量是发送端向网络一次连续写入（收到其中第一个数据的确认之前）的数据量，我们称为 SWND（Send Window，发送窗口㊀）。不过，发送端最终以 TCP 报文段来发送数据，所以 SWND 限定了发送端能连续发送的 TCP 报文段数量。这些 TCP 报文段的最大长度（仅指数据部分）称为 SMSS（Sender Maximum Segment Size，发送者最大段大小），其值一般等于 MSS。

发送端需要合理地选择 SWND 的大小。如果 SWND 太小，会引起明显的网络延迟；反

㊀ 这里所说的窗口实际上是指窗口的大小，这里只是保留了行业的习惯说法。

之，如果 SWND 太大，则容易导致网络拥塞。前文提到，接收方可通过其接收通告窗口（RWND）来控制发送端的 SWND。但这显然不够，所以发送端引入了一个称为拥塞窗口（Congestion Window，CWND）的状态变量。实际的 SWND 值是 RWND 和 CWND 中的较小者。图 3-11 显示了拥塞控制的输入和输出（可见，它是一个闭环反馈控制）。

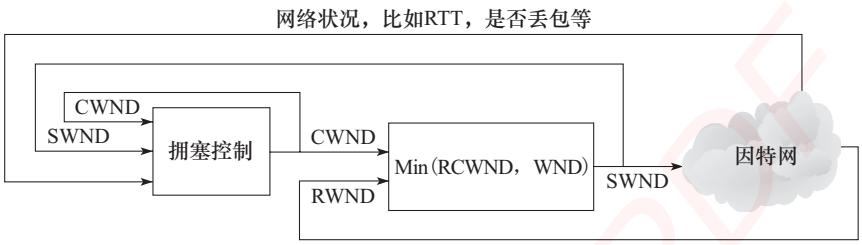

图 3-11　拥塞控制的输入和输出

3.10.2　慢启动和拥塞避免

TCP 连接建立好之后，CWND 将被设置成初始值 IW（Initial Window），其大小为 2 ～ 4 个 SMSS。但新的 Linux 内核提高了该初始值，以减小传输滞后。此时发送端最多能发送 IW 字节的数据。此后发送端每收到接收端的一个确认，其 CWND 就按照式（3-1）增加：

$$CWND+=\min(N, SMSS) \tag{3-1}$$

其中 N 是此次确认中包含的之前未被确认的字节数。这样一来，CWND 将按照指数形式扩大，这就是所谓的慢启动。慢启动算法的理由是，TCP 模块刚开始发送数据时并不知道网络的实际情况，需要用一种试探的方式平滑地增加 CWND 的大小。

但是如果不施加其他手段，慢启动必然使得 CWND 很快膨胀（可见慢启动其实不慢）并最终导致网络拥塞。因此 TCP 拥塞控制中定义了另一个重要的状态变量：慢启动门限（slow start threshold size，ssthresh）。当 CWND 的大小超过该值时，TCP 拥塞控制将进入拥塞避免阶段。

拥塞避免算法使得 CWND 按照线性方式增加，从而减缓其扩大。RFC 5681 中提到了如下两种实现方式：

- 每个 RTT 时间内按照式（3-1）计算新的 CWND，而不论该 RTT 时间内发送端收到多少个确认。
- 每收到一个对新数据的确认报文段，就按照式（3-2）来更新 CWND。

$$CWND+=SMSS*SMSS/CWND \tag{3-2}$$

图 3-12 粗略地描述了慢启动和拥塞避免发生的时机和区别。该图中，我们以 SMSS 为单位来显示 CWND（实际上它是以字节为单位的），以次数为单位来显示 RTT，这只是为了方便讨论问题。此外，我们假设当前的 ssthresh 是 16SMSS 大小（当然，实际的 ssthresh 显然远不止这么大）。

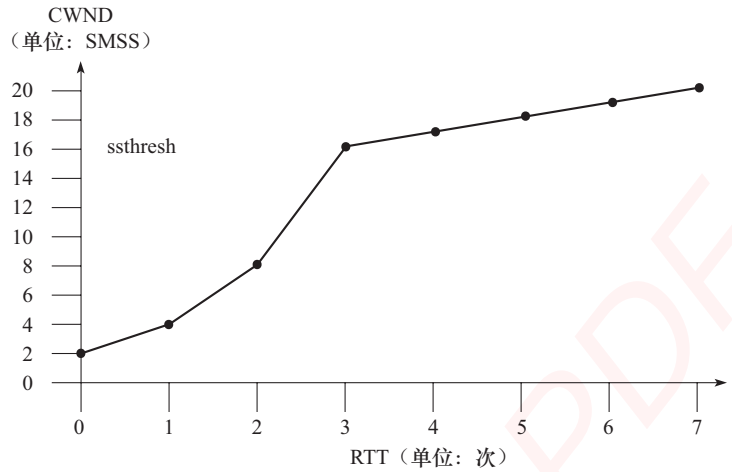

图 3-12　慢启动和拥塞避免

以上我们讨论了发送端在未检测到拥塞时所采用的积极避免拥塞的方法。接下来介绍拥塞发生时（可能发生在慢启动阶段或者拥塞避免阶段）拥塞控制的行为。不过我们先要搞清楚发送端是如何判断拥塞已经发生的。发送端判断拥塞发生的依据有如下两个：

- 传输超时，或者说 TCP 重传定时器溢出。
- 接收到重复的确认报文段。

拥塞控制对这两种情况有不同的处理方式。对第一种情况仍然使用慢启动和拥塞避免。对第二种情况则使用快速重传和快速恢复（如果是真的发生拥塞的话），这种情况将在后面讨论。注意，第二种情况如果发生在重传定时器溢出之后，则也被拥塞控制当成第一种情况来对待。

如果发送端检测到拥塞发生是由于传输超时，即上述第一种情况，那么它将执行重传并做如下调整：

$$ssthresh=max(FlightSize/2，2*SMSS) \quad (3\text{-}3)$$

$$CWMD<=SMSS$$

其中 FlightSize 是已经发送但未收到确认的字节数。这样调整之后，CWMD 将小于 SMSS，那么也必然小于新的慢启动门限值 ssthresh（因为根据式（3-3），它一定不小于 SMSS 的 2 倍），故而拥塞控制再次进入慢启动阶段。

3.10.3　快速重传和快速恢复

在很多情况下，发送端都可能接收到重复的确认报文段，比如 TCP 报文段丢失，或者接收端收到乱序 TCP 报文段并重排之等。拥塞控制算法需要判断当收到重复的确认报文段时，网络是否真的发生了拥塞，或者说 TCP 报文段是否真的丢失了。具体做法是：发送端如果连续收到 3 个重复的确认报文段，就认为是拥塞发生了。然后它启用快速重传和快速恢复

算法来处理拥塞，过程如下：

1）当收到第 3 个重复的确认报文段时，按照式（3-3）计算 ssthresh，然后立即重传丢失的报文段，并按照式（3-4）设置 CWND。

$$CWND=ssthresh+3*SMSS \quad (3\text{-}4)$$

2）每次收到 1 个重复的确认时，设置 CWND=CWND+SMSS。此时发送端可以发送新的 TCP 报文段（如果新的 CWND 允许的话）。

3）当收到新数据的确认时，设置 CWND=ssthresh（ssthresh 是新的慢启动门限值，由第一步计算得到）。

快速重传和快速恢复完成之后，拥塞控制将恢复到拥塞避免阶段，这一点由第 3 步操作可得知。

嵌入式资源免费下载

总线协议：

1. 基于 PCIe 驱动程序的数据传输卡 DMA 传输
2. 基于 PCIe 总线协议的设备驱动开发
3. CANopen 协议介绍
4. 基于 PXI 总线 RS422 数据通信卡 WDM 驱动程序设计
5. FPGA 实现 PCIe 总线 DMA 设计
6. PCI Express 协议实现与验证
7. VPX 总线技术及其实现
8. 基于 Xilinx FPGA 的 PCIE 接口实现
9. 基于 PCI 总线的 GPS 授时卡设计
10. 基于 CPCI 标准的 6U 信号处理平台的设计
11. USB30 电路保护
12. USB30 协议分析与框架设计
13. USB 30 中的 CRC 校验原理及实现
14. 基于 CPLD 的 UART 设计
15. IPMI 在 VPX 系统中的应用与设计
16. 基于 CPCI 总线的 PMC 载板设计
17. 基于 VPX 总线的工件台运动控制系统研究与开发
18. PCI Express 流控机制的研究与实现
19. UART16C554 的设计
20. 基于 VPX 的高性能计算机设计
21. 基于 CAN 总线技术的嵌入式网关设计
22. Visual C 串行通讯控件使用方法与技巧的研究
23. IEEE1588 精密时钟同步关键技术研究
24. GPS 信号发生器射频模块的一种实现方案
25. 基于 CPCI 接口的视频采集卡的设计
26. 基于 VPX 的 3U 信号处理平台的设计
27. 基于 PCI Express 总线 1394b 网络传输系统 WDM 驱动设计

VxWorks：

1. 基于 VxWorks 的多任务程序设计

2. 基于 VxWorks 的数据采集存储装置设计
3. Flash 文件系统分析及其在 VxWorks 中的实现
4. VxWorks 多任务编程中的异常研究
5. VxWorks 应用技巧两例
6. 一种基于 VxWorks 的飞行仿真实时管理系统
7. 在 VxWorks 系统中使用 TrueType 字库
8. 基于 FreeType 的 VxWorks 中文显示方案
9. 基于 Tilcon 的 VxWorks 简单动画开发
10. 基于 Tilcon 的某武器显控系统界面设计
11. 基于 Tilcon 的综合导航信息处理装置界面设计
12. VxWorks 的内存配置和管理
13. 基于 VxWorks 系统的 PCI 配置与应用
14. 基于 MPC8270 的 VxWorks BSP 的移植
15. Bootrom 功能改进经验谈
16. 基于 VxWorks 嵌入式系统的中文平台研究与实现
17. VxBus 的 A429 接口驱动
18. 基于 VxBus 和 MPC8569E 千兆网驱动开发和实现
19. 一种基于 vxBus 的 PPC 与 FPGA 高速互联的驱动设计方法
20. 基于 VxBus 的设备驱动开发
21. 基于 VxBus 的驱动程序架构分析

Linux：

1. Linux 程序设计第三版及源代码
2. NAND FLASH 文件系统的设计与实现
3. 多通道串行通信设备的 Linux 驱动程序实现
4. Zsh 开发指南-数组
5. 常用 GDB 命令中文速览
6. 嵌入式 C 进阶之道
7. Linux 串口编程实例
8. 基于 Yocto Project 的嵌入式应用设计
9. Android 应用的反编译
10. 基于 Android 行为的加密应用系统研究
11. 嵌入式 Linux 系统移植步步通
12. 嵌入式 CC++语言精华文章集锦
13. 基于 Linux 的高性能服务器端的设计与研究
14. S3C6410 移植 Android 内核
15. Android 开发指南中文版
16. 图解 Linux 操作系统架构设计与实现原理（第二版）

17. 如何在 Ubuntu 和 Linux Mint 下轻松升级 Linux 内核
18. Android 简单 mp3 播放器源码
19. 嵌入式 Linux 系统实时性的研究
20. Android 嵌入式系统架构及内核浅析
21. 基于嵌入式 Linux 操作系统内核实时性的改进方法研究

Windows CE：

1. Windows CE.NET 下 YAFFS 文件系统 NAND Flash 驱动程序设计
2. Windows CE 的 CAN 总线驱动程序设计
3. 基于 Windows CE.NET 的 ADC 驱动程序实现与应用的研究
4. 基于 Windows CE.NET 平台的串行通信实现
5. 基于 Windows CE.NET 下的 GPRS 模块的研究与开发
6. win2k 下 NTFS 分区用 ntldr 加载进 dos 源代码
7. Windows 下的 USB 设备驱动程序开发
8. WinCE 的大容量程控数据传输解决方案设计
9. WinCE6.0 安装开发详解
10. DOS 下仿 Windows 的自带计算器程序 C 源码
11. G726 局域网语音通话程序和源代码
12. WinCE 主板加载第三方驱动程序的方法
13. WinCE 下的注册表编辑程序和源代码
14. WinCE 串口通信源代码
15. WINCE 的 SD 卡程序[可实现读写的源码]
16. 基于 WinCE 的 BootLoader 研究

PowerPC：

1. Freescale MPC8536 开发板原理图
2. 基于 MPC8548E 的固件设计
3. 基于 MPC8548E 的嵌入式数据处理系统设计
4. 基于 PowerPC 嵌入式网络通信平台的实现
5. PowerPC 在车辆显控系统中的应用
6. 基于 PowerPC 的单板计算机的设计
7. 用 PowerPC860 实现 FPGA 配置
8. 基于 MPC8247 嵌入式电力交换系统的设计与实现
9. 基于设备树的 MPC8247 嵌入式 Linux 系统开发

10. 基于 MPC8313E 嵌入式系统 UBoot 的移植

ARM：

1. 基于 DiskOnChip 2000 的驱动程序设计及应用
2. 基于 ARM 体系的 PC-104 总线设计
3. 基于 ARM 的嵌入式系统中断处理机制研究
4. 设计 ARM 的中断处理
5. 基于 ARM 的数据采集系统并行总线的驱动设计
6. S3C2410 下的 TFT LCD 驱动源码
7. STM32 SD 卡移植 FATFS 文件系统源码
8. STM32 ADC 多通道源码
9. ARM Linux 在 EP7312 上的移植
10. ARM 经典 300 问
11. 基于 S5PV210 的频谱监测设备嵌入式系统设计与实现
12. Uboot 中 start.S 源码的指令级的详尽解析
13. 基于 ARM9 的嵌入式 Zigbee 网关设计与实现
14. 基于 S3C6410 处理器的嵌入式 Linux 系统移植

Hardware：

1. DSP 电源的典型设计
2. 高频脉冲电源设计
3. 电源的综合保护设计
4. 任意波形电源的设计
5. 高速 PCB 信号完整性分析及应用
6. DM642 高速图像采集系统的电磁干扰设计
7. 使用 COMExpress Nano 工控板实现 IP 调度设备
8. 基于 COM Express 架构的数据记录仪的设计与实现
9. 基于 COM Express 的信号系统逻辑运算单元设计
10. 基于 COM Express 的回波预处理模块设计