传输层

2016-3-2

作者：朱林峰

目录

传输层

1 传输层协议概述

从通信和信息处理的角度看，传输层向它上面的应用层提供通信服务，它属于面向通信部分的最高层，同时也是用户功能中的最低层。当网络边缘部分中的两个主机使用网络的核心部分的功能进行端到端的通信时，只有主机的协议栈才有运输层，而网络核心部分中的路由器在转发分组时都只用到下三层的功能。

IP 协议提供主机到主机的通信，而传输层协议则为进程之间提供端到端的逻辑通信。

引入传输层的原因：

- 增加端口号寻址，提供进程到进程的通信
- 增加复用和分用的功能。复用和分用是分别针对发送方和接收方而言的，复用是指发送方的不同应用程序都可以使用同一个运输层协议传送数据，而分用是指接收方的运输层在剥去报文首部后能够把这些数据正确交付到目的应用进程。
- 差错检测。IP 层报文头中的校验和字段只是校验 IP 报文首部，而不对数据作校验。
- 流量控制和纠错功能
- TCP 协议还能消除网络层的不可靠性，提供连接管理
- 为高层应用屏蔽网络通信的细节，提供完善的通信能力

应用	应用层协议	运输层协议
名字转换	DNS	UDP
文件传送	TFTP	UDP
路由选择协议	RIP	UDP
IP 地址配置	DHCP	UDP
网络管理	SNMP	UDP
远程文件服务器	NFS	UDP
IP 电话	专用协议	UDP
流式多媒体通信	专用协议	UDP
电子邮件	SMTP	TCP
远程终端接入	TELNET	TCP
万维网	HTTP	TCP
文件传送	FTP	TCP

2 UDP

2.1 UDP 概述

UDP（User Datagram Protocol，用户数据报协议，协议号 17）只在 IP 的数据报服务上增加了很少一点的功能，这就是复用和分用的功能以及差错检测的功能。UDP 的主要特点：

- UDP 是无连接的，即发送数据之前不需要建立连接，因此减少了开销和发送数据之前的时延。
- UDP 尽最大努力交付，不保证可靠交付，因此主机不需要维护复杂的连接状态机
- UDP 是面向报文的。发送方的 UDP 对应应用程序交下来的报文，在添加首部后就向下交付给 IP 层。UDP 对应用层交下来的报文，既不合并，也不拆分，直接整个发送整个接收。因此，应用程序必须选择合适大小的报文，若报文太长，UDP 把它交给 IP 层后，IP 层在传送时可能要进行分片，这会降低 IP 层的效率。反之，若报文太短，会使 IP 数据报的首部相对长度太大，也降低了 IP 层的效率。

- UDP 没有拥塞控制，因此网络出现的拥塞不会使源主机的发送速率降低。这对某些实时应用（如 IP 通话、视屏会议等）是很重要的，它们要求源主机以恒定的速率发送数据，并且允许丢失一些数据。
- UDP 支持一对一、一对多、多对一和多对多的交互通信
- UDP 首部开销小，只有 8 字节，比 TCP 的 20 个字节要少。
- UDP 的简单特性也给应用层留下了二次开发的空间。

2.2 UDP 报文格式

16	32
source port	destination port
length	checksum
data	

- 源端口：在需要对方回信时选用，不需要时可用全 0
- 目的端口：
- 长度：整个 UDP 报文的长度，包括首部，其最小值为 8
- 校验和：检测整个 UDP 报文在传输中是否有差错，有就丢弃

3 TCP

3.1 TCP 概述

TCP（Transmission Control Protocol，传输控制协议，协议号 6）是 TCP/IP 体系中非常复杂的一个协议。TCP 的主要特点为：

- TCP 是面向连接的传输层协议，应用程序在使用 TCP 协议之前，必须先建立 TCP 连接，在数据传送完毕后，必须释放已经建立的 TCP 连接。
- TCP 提供一对一的通信。
- TCP 提供可靠的交付服务，通过 TCP 连接传送的数据，无差错、不丢失、不重复、并且按序到达。
- TCP 提供全双工通信。TCP 允许通信双方在任何时候都能相互发送数据，通信两端都有接受数据的缓存和发送数据的缓存。
- 面向字节流，不保证接收方收到的数据块和发送方发出的数据块具有对应的大小关系。

3.2 TCP 报文格式

16								32
源端口								目的端口
序号								
确认号								
数据偏移	保留	URG	ACR	PSH	RST	SYN	FIN	窗口
校验和								紧急指针
选项（长度可变）								填充

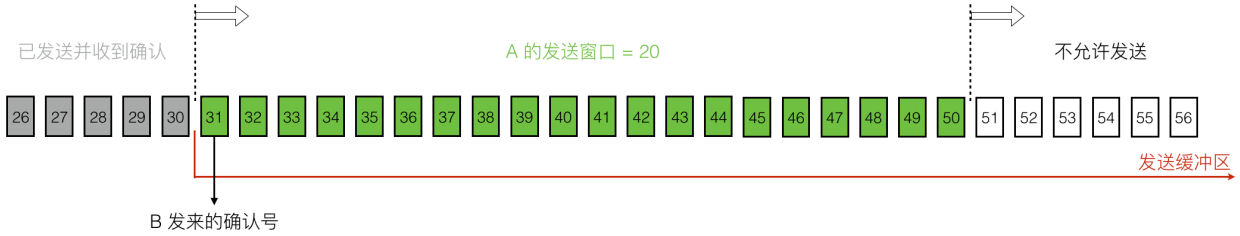
报文长度：TCP 首部前 20 个字节是固定的，后面 4N 个字节是根据需要而增加的选项，因此 TCP 首部最小长度为 20 字节。

- 源端口和目的端口：各占两个字节
- 序号：占 4 个字节，循环递增。TCP 面向字节流，为传输的每个直接都按顺序编号，此处序号字段代表此次发送数据第一个字节的序号。如果序号为 301，而数据长度为 100，则下一个数据包的序号就为 401。
- 确认号：占 4 个字节，是期望收到对方下一个报文的第一个数据字节的序号。

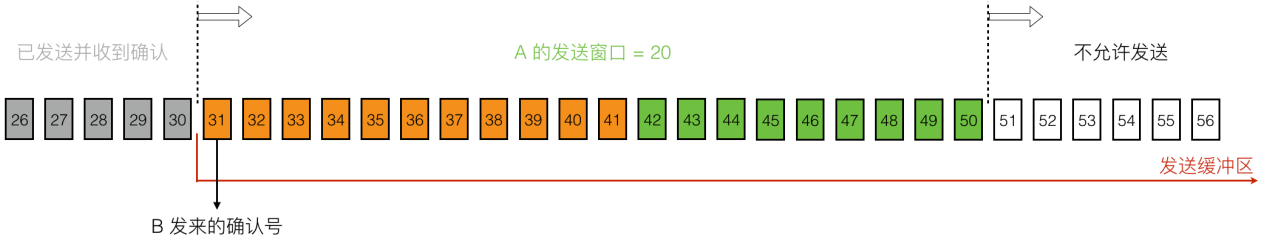
- 数据偏移：占 4 位，指出 TCP 数据距首部的偏移量，也就是 TCP 首部的实际长度。数据偏移以 4字节为单位计算，其最大值为 15，故 TCP 首部最大长度为 15 * 4 = 60 字节。
- 窗口：2 字节，因为接收方的缓存是有限的，因此接收方用窗口值告诉对方允许发送的数据量。
- 校验和：2 字节，首部和数据部分的校验和。
- URG（紧急）：当 URG = 1 时，表明紧急指针字段有效。它告诉系统此报文中有紧急数据，应尽快传送（相当于高优先级的数据），而不要按原来的排队顺序来传送。同时紧急指针指出紧急数据的字节数，紧急数据结束后就是普通数据。
- ACK（确认）：仅当 ACK = 1 时确认号才有效。TCP 规定，在连接建立后所有传送报文都必须把 ACK 置 1.
- PSH（推送）：提高报文处理的优先级，使之不用一直在缓冲区里等待。推送操作很少用。
- RST（复位）：当 RST = 1时，表明 TCP 连接中出现严重差错，必须释放连接，然后再重新建立连接。RST 置 1 还用来拒绝一个非法的报文段或拒绝打开一个连接。
- SYN（同步）：在连接建立时用来同步序号。当 SYN = 1 而 ACK = 0 时，表明这是一个连接请求报文段。对方若同意建立连接，则在响应报文中设 SYN = 1 和 ACK = 1。因此，SYN = 1 表示这是一个连接请求或连接接受报文，不能携带数据，但要消耗一个序列号。
- FIN（终止）：用来释放一个连接。当 FIN = 1 时，表明此报文的发送方数据已经发送完毕，并要求释放连接。

3.3 TCP 可靠传输的实现

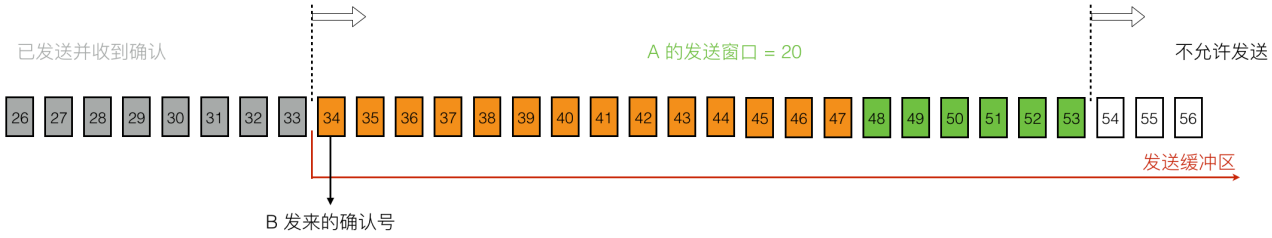
3.3.1 以字节为单位的滑动窗口



1. 灰色字节为 A 已经发送给 B 并且收到 B 的确认号为 31。此时灰色字节被移出发送缓冲区
2. 绿色字节是 A 的当前发送窗口，允许发送。发送窗口后沿由 B 的确认号（31）确定。窗口大小由 B 发来的窗口字段确定。
3. 尚未发送的字节都存储于发送缓冲区内。已发送的字节从缓冲区移出，缓冲区增大。



1. 橘色字节为 A 已经发送的数据，但尚未收到来自 B 的比 31 大的确认号



1. A 继续发送窗口内的数据，发送至序列号 47
2. 同时 A 收到来自 B 的比 31 要大的确认号 34，因此 31，32，33 都移出窗口和缓冲区

3. 因为来自 B 的窗口仍然为 20 字节，因此 A 的发送窗口要向右滑动 3 个字节，以保证窗口大小为 20 字节

对 B 的接收而言，若收到的报文中序号是 31，则发回给 A 的确认号不小于 31。有如下举例：

- 若 B 收到的字节为 32，33，34，与序号 31 不是连续的，则 B 返回的确认号依然为 31
- 若 B 收到的字节为 31，32，34，35，则 B 返回的确认号为 34

滑动窗口算法有一些优化措施：

- 接收方对收到的不连续数据不必丢弃，而是临时保存在接收缓冲区中，等到丢失的数据到齐后，再按序交付给上层应用
- TCP 要求接收方有累积确认的功能，就是接收数据达到一定程度后再一并确认，也可以在自己有数据要发送的时候一并捎上。但接收方不应过分推迟发送确认，一面发送方超时重传。

3.3.2 超时重传

TCP 发送方在规定时间内没有收到确认就要重传已经发送的报文段。重传时间为报文往返时间的平均值。

3.4 TCP 连接管理

3.4.1 概述

套接字

套接字：socket = （IP 地址： 端口号）

每一个 TCP 连接唯一的被两个套接字所确定：

TCP 连接 ::= {$socket_1$，$socket_2$} = {（IP_1：$port_1$），（IP_2：$port_2$）}

socket 其他语义：

1. 允许应用程序访问网络协议的应用编程接口 API，即传输层和应用层之间的一种接口，称为 socket API。
2. 在 socket API 总是会用的一个函数名也叫做 socket。
3. 调用 socket 函数的端点称为 socket
4. socket 函数返回值称为 socket 描述符，简称 socket。
5. 在操作系统内核中连网协议的 Berkeley 实现，称为 socket 实现。

TCP 连接的建立采用客户端服务器的方式。主动发起连接请求的进程叫做客户端，被动等待连接请求的进程叫做服务器。

TCP 连接建立过程中要解决以下三个问题：

1. 要使每一方都能够确知对方的存在
2. 要允许双方协商一些参数（如最大窗口值、是否使用窗口扩大选项和时间戳以及服务质量等）
3. 能够对运输实体资源（如缓存大小、连接表中的项目等）进行分配

3.4.2 TCP 连接的建立

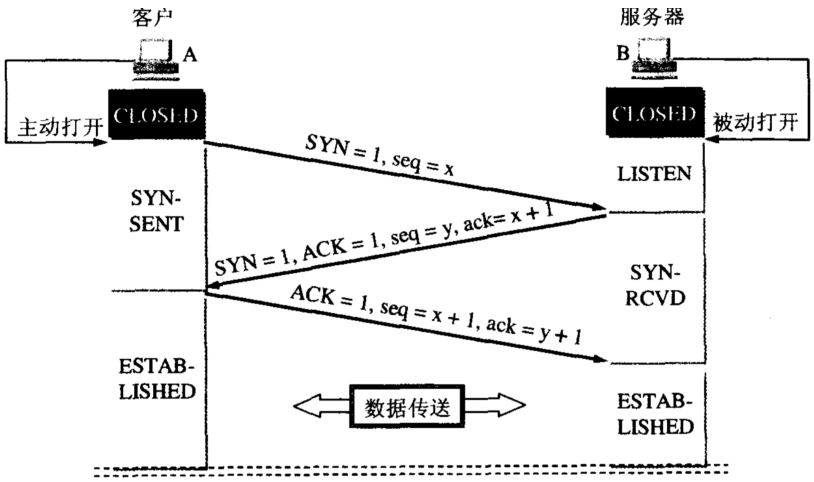


图 5-31　用三次握手建立 TCP 连接

TCB（Transmission Control Block，传输控制块）存储了每一个连接中的一些重要信息，如：TCP 连接表，到发送和接收缓存的指针，到重传队列的指针，当前的发送和接收序列号等。

1. 服务器 B 先创建 TCB，准备接受客户进程的连接请求。然后服务器就处于 LISTEN 状态，等待客户端的连接请求。如有，即作出响应。
2. 客户端 A 创建 TCB，然后向 B 发出连接请求报文。
3. A 收到 B 的同意建立连接的报文后，还要向 B 进行确认。

3.4.3 TCP 连接的释放

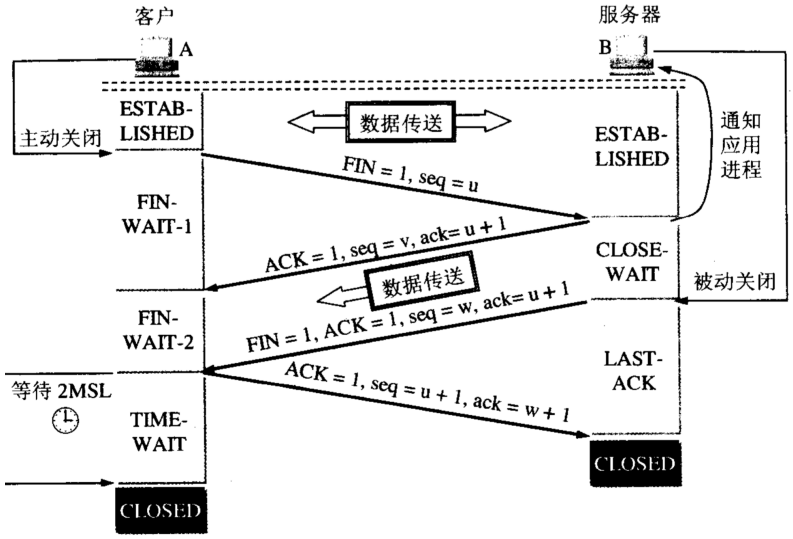


图 5-32　TCP 连接释放的过程

1. A 先发出释放连接的报文，并停止再发送数据，主动关闭 TCP 连接，等待 B 的确认
2. B 收到连接释放报文后即发出确认，这时 A 到 B 的连接就释放了，这时的 TCP 处于半关闭状态，即 A 已经没有数据要发给 B 了，但 B 若发送数据，A 仍要接收。B 到 A 的连接并未关闭。
3. A 收到 B 的确认后，开始等待 B 的连接释放报文。
4. B 给 A 发送连接释放报文。
5. A 收到报文后，还要对此发出确认。此后 A 等待 2MSL 时间后，才能进入 CLOSED 状态。A 撤销相应的 TCB，结束 TCP 连接。

除了主动释放连接，TCP 还有保活计时器。服务器每收到一次客户端的数据，就重置计时器；若计时器超时，服务器就每隔一定时间发送探测报文，若一连发送 10 个探测报文后客户端仍无响应，服务器就认为客户端出了故障，自主关闭连接。

3.4.4 TCP 的状态机

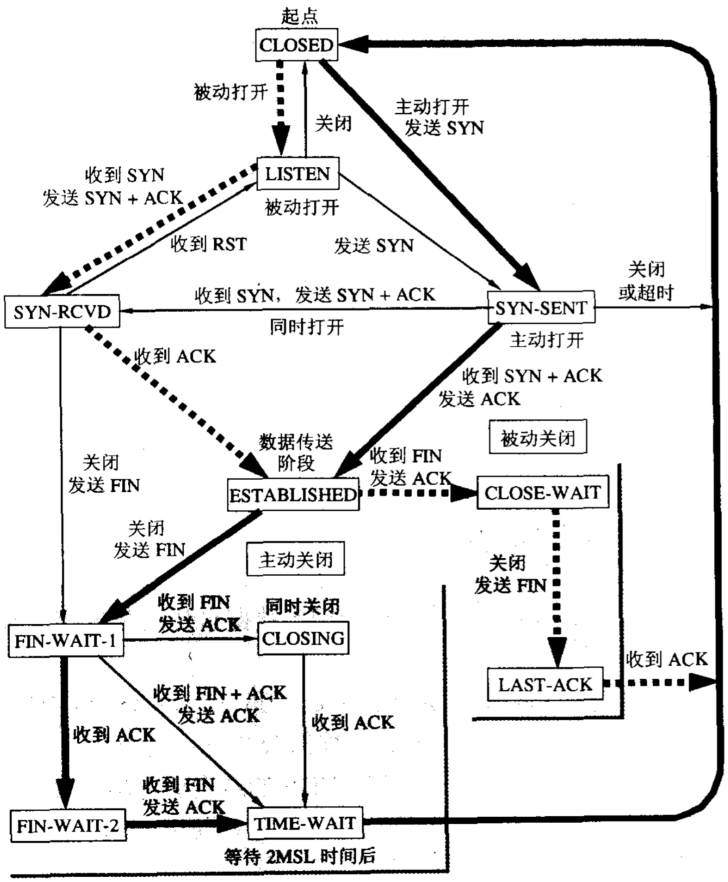


图 5-33　TCP 的有限状态机

3.5 TCP 的拥塞控制

1. 慢开始和拥塞避免

   慢开始和拥塞避免的思想都是动态改变滑动窗口的大小。

   慢开始用于刚开始发送数据时，使用很小的滑动窗口，如果网络通畅，则每次报文往返都让滑动窗口的值加倍。

   拥塞避免用于数据发送一段时间后，滑动窗口的值比较大，如果网络通畅，则每次报文往返都让滑动窗口的值线性增加。

   如果网络发生拥塞，则滑动窗口减半，拥塞窗口重置为 1，又进行慢开始算法。

2. 快重传

   接收方每收到一个失序的报文段后就立即发出重复确认，而不用等到自己发送数据时才进行捎带确认。只要接收方一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待重传计时器超时。由于发送方尽早重传未被确认的报文段，因此采用快重传后可以使整个网络的吞吐量提高约 20%。

3. 快恢复

   与快重传配合使用的是快恢复。

   首先当发送方连续收到三个重复确认时，执行滑动窗口减半的“乘法减小”算法。这是为了预防网络发生拥塞。

   然后并不执行慢开始算法，而是直接执行拥塞避免算法，时滑动窗口值线性增加。

3.6 定时器

TCP中一共有四个定时器：

- 超时定时器：超时重传是 TCP 协议保证数据可靠性的另一个重要机制，其原理是在发送某一个数据以后就开启一个计时器，在一定时间内如果没有得到发送的数据报的 ACK 报文，那么就重新发送数据，

直到发送成功为止。超时的时间设置很关键，如果太长则浪费了网络资源；如果太短，则造成过多的重传。

- 2MLS定时器：2MLS 定时器在释放连接时主动释放的一端使用。一是为了保证 A 的最后一个 ACK 报文能够到达 B，如果 B 收不到 A 的 ACK 确认报文就重传释放连接请求报文，A 再次发送 ACK 确认报文并重置 2MLS 定时器。如果 2MLS 定时器超时，则确定双方都正常关闭了连接。二是保证旧的连接时间内产生的所有报文都从网络中消失，这样下一个新的连接就不会出现旧的连接请求报文了。
- 保活定时器：在 TCP 连接已经建立，但是很长一段时间并没有报文交互，这样 TCP 都不能判断对端是否还在，连接是否还正常，所以会隔一段时间发一个定时器，但是现在一般保活定时器都由应用层来控制，TCP 自己保活定时器用的不多。
- 坚持定时器：当 A 收到 B 发过来的 window = 0 的 ACK 时，会不再往 B 端发送报文，会一直等到 B 端发过来 window > 0的 ACK 过来，再发送报文。但是现在会出现一个问题就是，如果 B 的 ACK 报文丢了，那么就会进入到 A 在等 ACK，B 也在等 A 的数据，进入了一个死锁状态。现在设置一个坚持定时器，A 等坚持定时器超时就给 B 发送数据，如果 A 回复的还是 window = 0，就再等待。

4 UDP 与 TCP 比较

- UDP 简单，报文头短，网络和系统开销小；TCP 复杂，报文头长，网络和系统开销大
- UDP 无连接，来了报文可直接发送；TCP 必须建立连接后才可发送报文。
- UDP 无连接，因此可以多对多通信；TCP 建立端到端的连接，因此只能一对一通信。
- UDP 是面向报文的传输，也就是传输以报文为单位；TCP 是面向字节流的传输，以字节为单位。
- UDP 不作可靠性保证，尽最大努力交付，也不关心报文达到的先后顺序；TCP 保证无差错、不丢失、不重复、并且按序到达的数据传输。
- UDP 没有拥塞控制；TCP 有拥塞控制。

5 网络编程

5.1 套接字接口

5.1.1 数据类型

types.h

```
typedef long          ssize_t;
typedef unsigned long size_t;
```

in.h

```
typedef __uint32_t in_addr_t; /* base type for internet address */
typedef __uint16_t  in_port_t;

struct in_addr {
    in_addr_t s_addr;
};//ipv4  地址

struct sockaddr_in {
    __uint8_t sin_len;
    sa_family_t sin_family;
    in_port_t sin_port;
    struct in_addr sin_addr;
    char  sin_zero[8];
};//ipv4 套接字
```

in6.h

```
struct in6_addr {
    union {
         __uint8_t   __u6_addr8[16];
         __uint16_t  __u6_addr16[8];
         __uint32_t  __u6_addr32[4];
```

```
    } __u6_addr;   /* 128-bit IP6 address */
};//ipv6 地址

struct sockaddr_in6 {
    __uint8_t sin6_len; /* length of this struct(sa_family_t) */
    sa_family_t sin6_family; /* AF_INET6 (sa_family_t) */
    in_port_t sin6_port; /* Transport layer port # (in_port_t) */
    __uint32_t sin6_flowinfo; /* IP6 flow information */
    struct in6_addr sin6_addr; /* IP6 address */
    __uint32_t sin6_scope_id; /* scope zone index */
};//ipv6 套接字
```

socket.h

```
typedef __uint8_t  sa_family_t;
typedef __uint32_t  __darwin_socklen_t; /* socklen_t (duh) */
typedef __darwin_socklen_t socklen_t;

struct sockaddr {
    __uint8_t sa_len;  /* total length */
    sa_family_t sa_family; /* [XSI] address family */
    char  sa_data[14]; /* [XSI] addr value (actually larger) */
};//ipv4 通用套接字

struct sockaddr_storage {
    __uint8_t ss_len;  /* address length */
    sa_family_t ss_family; /* [XSI] address family */
    char    __ss_pad1[_SS_PAD1SIZE];
    __int64_t __ss_align; /* force structure storage alignment */
    char    __ss_pad2[_SS_PAD2SIZE];
};//ipv4 & ipv6 通用套接字
```

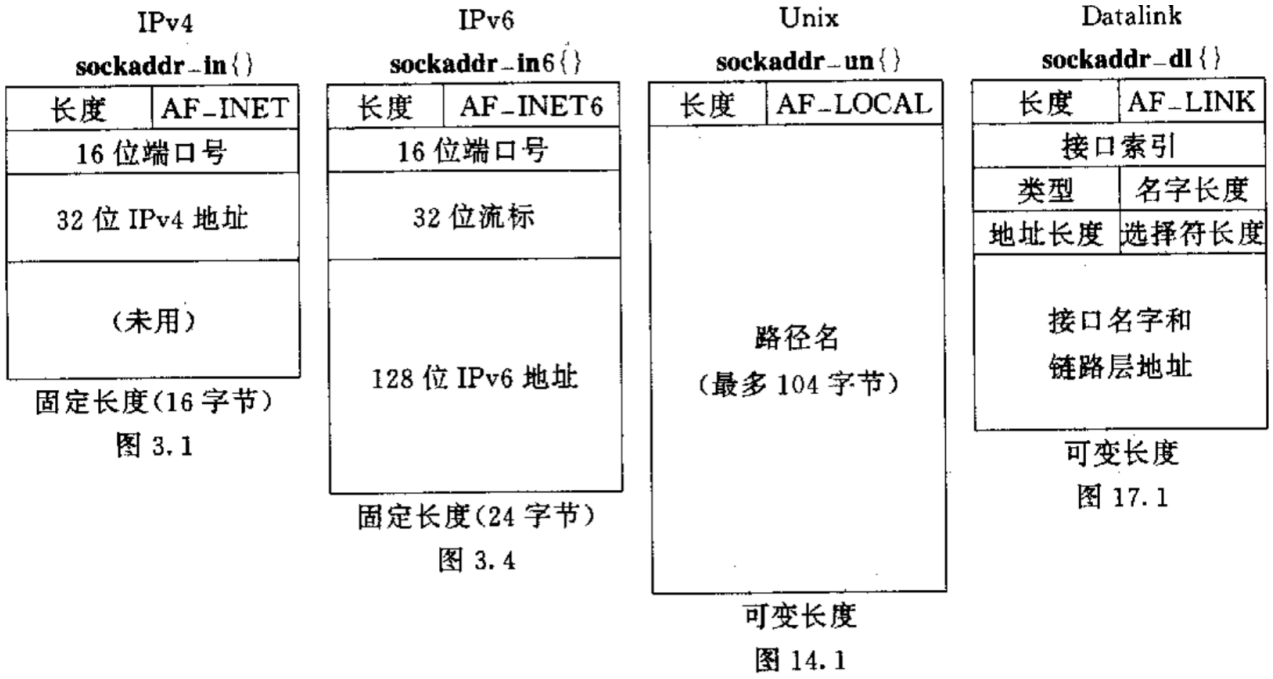


图 3.5　不同套接口地址结构的比较

5.1.2 字节操纵函数

string.h

```
extern int bcmp(const void *, const void *, size_t);
extern void bcopy(const void *src, void *dest, size_t);
extern void bzero(void *, size_t);
```

```
extern void *memcpy(void *dest, const void *src, size_t);
extern int memcmp(const void *, const void *, size_t);
extern void *memmove(void *, const void *, size_t);
extern void *memset(void *, int, size_t);
```

in.h

```
uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);
uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t net32bitvalue);
```

inet.h

```
int inet_aton(const char *strptr, struct in_addr *addrptr);
    //Returns: 1 if string was valid, 0 on error

char *inet_ntoa(struct in_addr inaddr);
    //Returns: pointer to dotted-decimal string


int inet_pton(int family, const char *strptr, void *addrptr);
    //Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error

const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t
len);
    //Returns: pointer to result if OK, NULL on error
```

5.1.3 套接字函数

socket.h

family	说明
AF_UNIX = 1	local to host(pipes)
AF_INET = 2	IPV4 协议
AF_INET6 = 30	IPV6协议
AF_ROUT = 17	路由套接字

type	说明
SOCK_STREAM	字节流套接字
SOCK_DGRAM	数据报套接字
SOCK_SEQPACKET	有序分组套接字
SOCK_RAW	原始套接字

protocol	说明
0	根据 family 和 type 选择默认的协议
IPPROTO_TCP	TCP 协议
IPPROTO_UDP	UDP 协议
IPPROTO_SCTP	SCTP 协议

```
int socket (int family, int type, int protocol);
    //Returns: non-negative descriptor if OK, -1 on error
```

```
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
    //Returns: 0 if OK, -1 on error
```

connect 失败则套接字不再可用，应立即关闭。

connect 失败原因：

- 服务器响应超时
- 收到 RST = 1 的回复报文，比如服务器没有运行、拒绝连接之类
- 网络不可达

```
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
    //Returns: 0 if OK,-1 on error
```

服务器在启动时捆绑它们众所周知的端口，如果一个 TCP 客户端或服务器未曾调用 bind() 绑定一个端口，当调用 connect 或 listen 时，内核就要为相应的套接字分配一个临时端口。让内核临时选择端口对 TCP 客户端来说是正常的，然而对 TCP 服务器来说却极为罕见，因为服务器是通过它们众所周知的端口被大家认识的。

进程可以把一个特定的 IP 地址捆绑到它的套接字上，不过这个 IP 地址必须是主机网络接口之一。对于 TCP 客户端，这就为该套接字发送的 IP 数据报指派了源 IP 地址。对于 TCP 服务器，这就限定该套接字只接受目的地址为这个 IP 地址的客户连接报文。TCP 客户通常不把 IP 地址绑定到套接字上，而是由内核根据网络接口情况分配源 IP 地址；服务器若没绑定 IP 地址，则把客户端的目的 IP 地址作为自己的源 IP 地址。

```
int listen (int sockfd, int backlog);
    //Returns: 0 if OK, -1 on error
```

listen 函数只由服务器调用，它做两件事情：

- socket 函数创建一个套接字时，它被假设为一个主动套接字，也就是说，它是一个将调用 connect 发起连接的客户端套接字。listen 函数把一个未连接的套接字转换成一个被动套接字，指示内核应接受指向该套接字的连接请求。
- backlog 参数规定了内核应该为相应套接字排队的最大连接个数。

为理解 backlog 参数，我们需要认识到内核为任何一个给定的监听套接字维护两个队列：

- 未完成连接队列：正在进行三次握手的连接，这些套接字处于 SYN_RCVD 状态。
- 已完成连接队列：已完成三次握手的连接，这些套接字处于 ESTABLISHED 状态。

如果新的客户端连接过来，添加到未完成连接队列末；如果三次握手完毕，则将连接移到已完成连接队列末。调用 accept 函数时，返回已完成连接队列的头，如果队列为空，进程投入睡眠，直到队列不为空才被唤醒。

```
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
    //Returns: non-negative descriptor if OK, -1 on error
```

accepte 函数由 TCP 服务器调用，用于从已完成连接队列头返回下一个已完成连接。如果队列为空，且套接字设置为默认的阻塞方式，则进程进入睡眠。

- sockfd：为服务器之前创建的监听套接字，服务器通常只创建一个监听套接字，在服务器的生命周期内一直存在。
- cliaddr：指向套接字结构体的指针，此套接字结构体的内存空间必须预先存在。内核为每个由服务器进程接受的客户连接创建一个已连接套接字，套接字的内容即保存在 cliaddr 所指向的结构体中。当服务器完成对某个给定客户的服务时，这个已连接套接字就被关闭。
- addrlen：是值-结果参数。调用前，addrlen 指向变量的值为 cliaddr 结构体的长度；返回时，其指向变量的值为由内核放在该套接字地址结构内的确切字节数。
- int 型返回值：如果返回值不为负，则代表连接套接字的描述符；否则表示发生错误。

如果对客户端的连接信息不感兴趣，可以将 cliaddr 和 addrlen 均设为 NULL。

```
#include <unistd.h>
int close (int sockfd);
    //Returns: 0 if OK, -1 on error
```

close 函数关闭套接字和关闭其他描述符是一样的，它只是减少描述符的引用计数，只要描述符的引用计数大于 0，TCP 连接就不会主动释放连接。

close 一个 TCP 套接字的默认行为是把该套接字标记成已关闭，然后立即返回到调用进程。该套接字不能再被该进程使用。然而 TCP 将尝试发送已排队等待发到对端的任何数据，发送完毕而且套接字的引用计数为 0，才开始释放连接。

5.1.4 套接字函数调用顺序

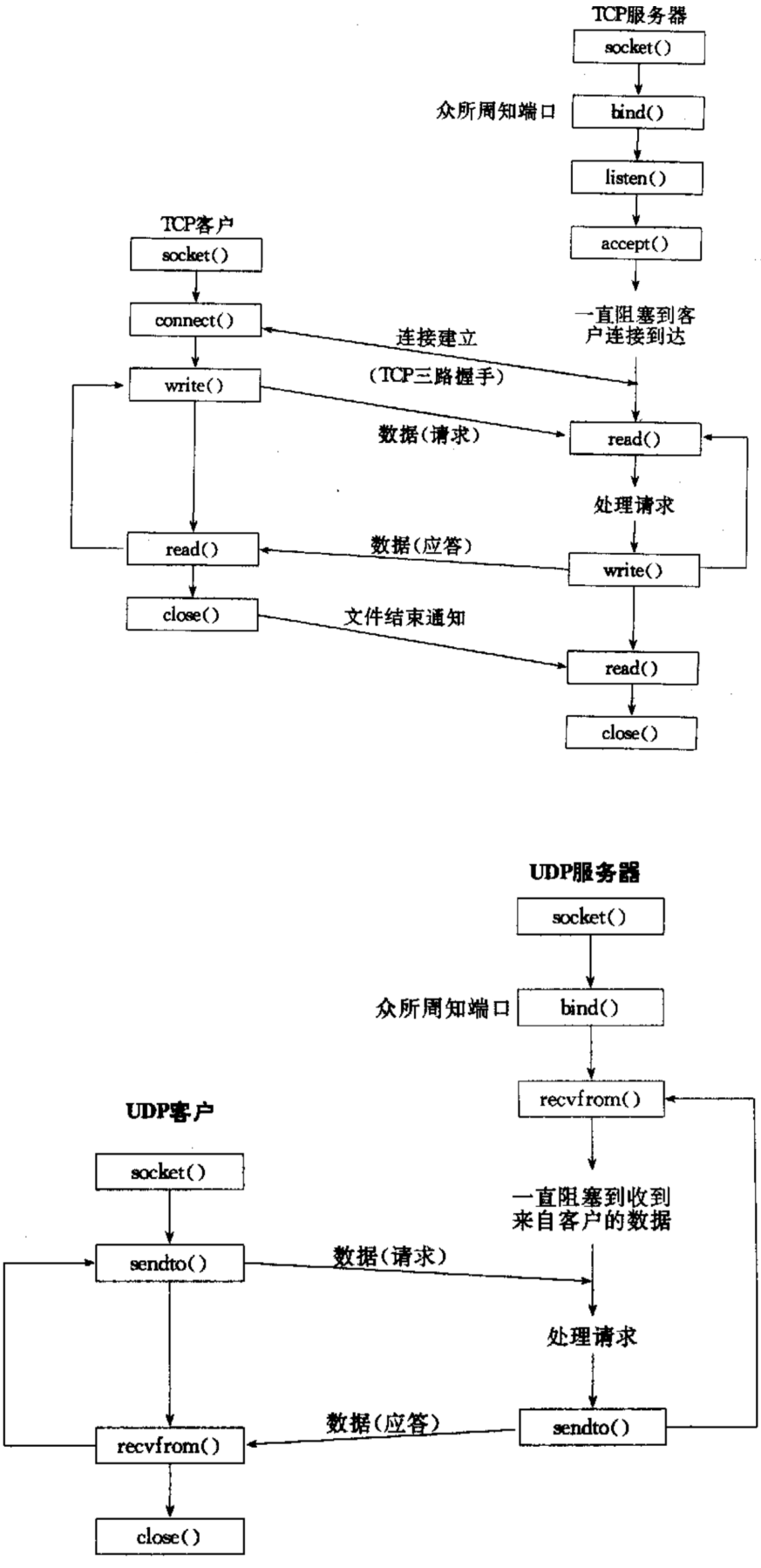


图 8.1　UDP 客户-服务器程序所用套接口函数

5.2 例程

1. TCP read 例程

```
1. ssize_t readn(int fd, void *vptr, size_t n)
2. {
3.     size_t nleft;
4.     ssize_t nread;
5.     char *ptr;
6.
7.     ptr = vptr;
8.     nleft = n;
9.     while (nleft > 0)
10.    {
11.        if ((nread = read(fd, ptr, nleft)) < 0) {
12.            if (errno == EINTR)/* 如果是因为信号中断没有读到数据
13.                nread = 0;     /* and call read() again */
14.            else
15.                return (-1);
16.        } else if (nread == 0) /* end of file */
17.        {
18.            break;
19.        }
20.        nleft -= nread;
21.        ptr += nread;
22.    }//end of while
23.    return (n - nleft);
24.}
```

2. TCP write 例程

```
1. ssize_t writen(int fd, const void *vptr, size_t n)
2. {
3.     size_t nleft;
4.     ssize_t nwritten;
5.     const char *ptr;
6.
7.     ptr = vptr;
8.     nleft = n;
9.     while (nleft > 0)
10.    {
11.        if ((nwritten = write(fd, ptr, nleft)) <= 0) {
12.            if (nwritten < 0 && errno == EINTR)
13.                nwritten = 0;
14.            else
15.                return (-1);
16.        }
17.        nleft -= nwritten;
18.        ptr += nwritten;
19.    }
       return n;
20.}
```

3. 典型并发服务的轮廓

```
pid_t pid;
int listenfd, connfd;
listenfd = socket( ... );
bind(listenfd, ... );
listen(listenfd, LISTENQ);

for ( ; ; ) {
    connfd = accept(listenfd, ... );
    if((pid = fork()) == 0)
```

```
    {
        close(listenfd);   /* child close listen socket */
        doit(connfd);      /* child process the request */
        close(connfd);     /* child close connect socket */
        exit(0);           /* child exit */
    }
    close(connfd);         /* parent closes connected socket */
}
```