计算机网络

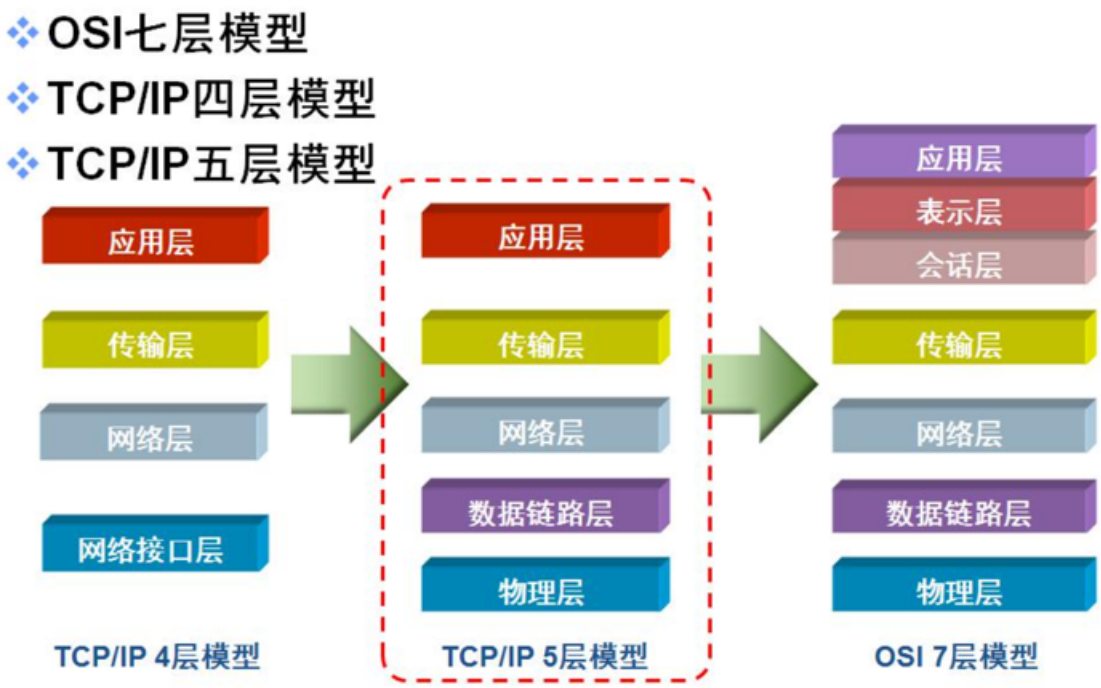
一、网络基础

1、协议概念

协议，网络协议的简称，网络协议是通信计算机双方必须共同遵从的一组约定。协议可以理解为规则，有：原始协议、标准协议。

比如A、B之间传输文件，第一次传文件名、第二次传文件大小、第三次传文件内容，按照这个步骤进行传输。例如：TCP协议、UDP协议、HTTP协议、FTP协议等

2、分层模型



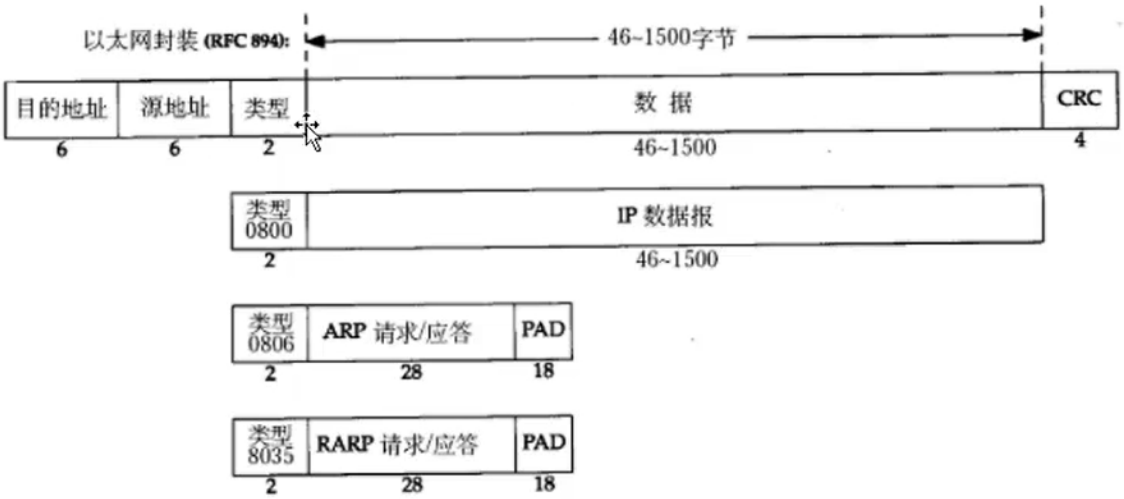
3、协议格式

链路层的以太网帧 （帧头与帧尾）、网络层的IP数据报、传输层的数据段（这三个是由操作系统进行操作的）、应用层是由用户操作的。

3.1、以太网帧格式

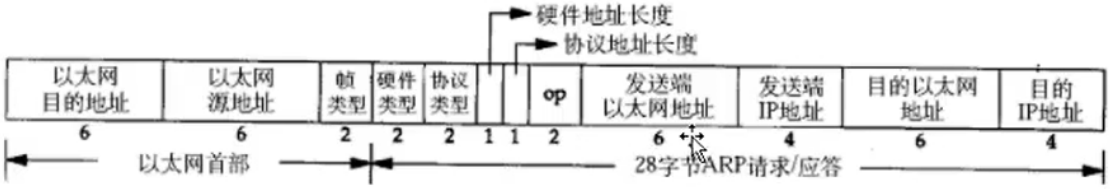
mac地址：也就是硬件地址，网卡的编号，表示网卡的唯一性。

以太网的帧格式如下所示：



以太网帧格式

ARP 数据报的格式如下所示：

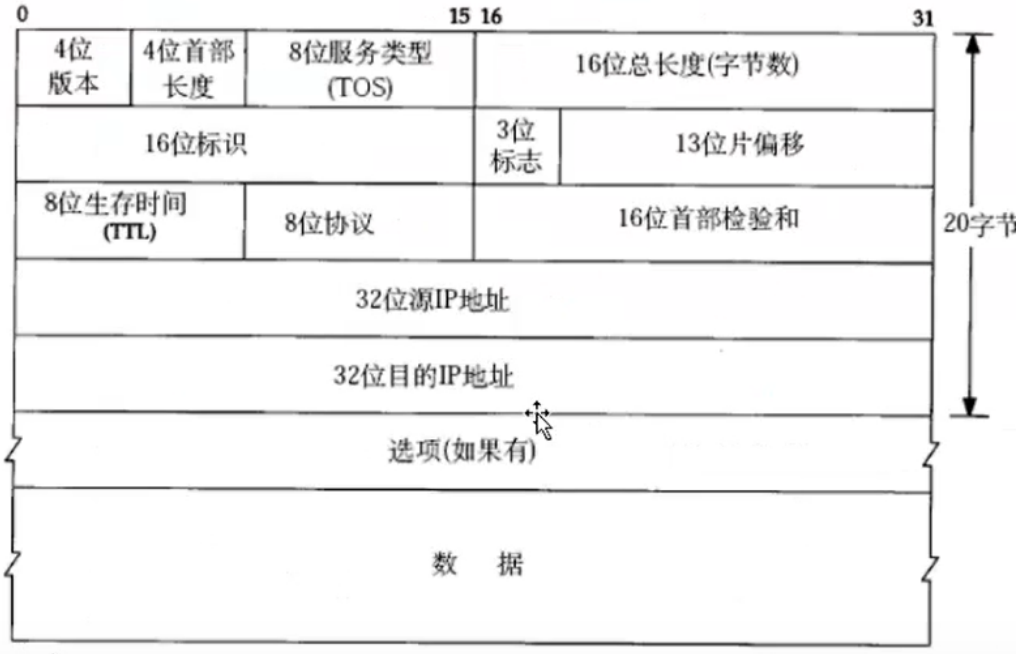


ARP 数据报格式

arp数据报就是为了获取下一跳的mac地址。

3.2、IP段格式

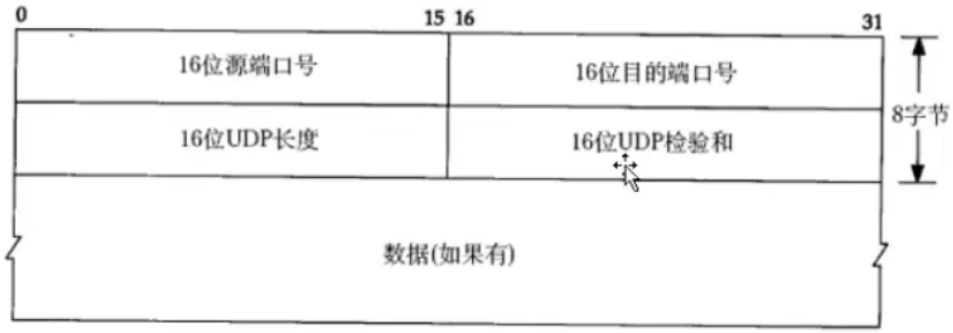
IP 段格式



IP数据报的首部长度与数据长度都是可变的，但总是4的倍数。另外，4位版本号，也就是IPv4与IPv6，4位首部长度，就是IP数据报的首部长度，最小是20字节，最大是60字节，因为4位首部长度最大表示的是2^4-1=15，4 * 15也就是60字节。8位的TTL，也就是下一跳的个数，最大也就是255，然后就是32位的源IP地址与32位的目的IP地址，也就是源IP与目的IP都占4字节。

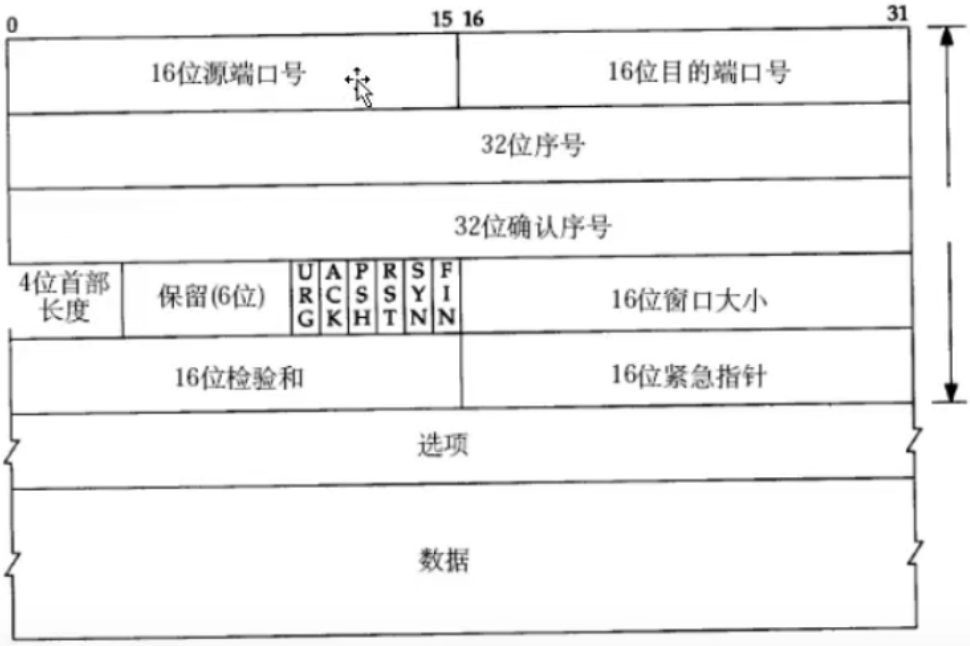
3.3、TCP/UDP数据报格式

UDP 数据报格式



UDP 数据段

TCP 数据报格式



4、TCP协议

TCP协议是一个传输层的协议；TCP是面向连接的协议；TCP是一个可靠的协议；TCP是全双工的协议；TCP是一个字节流的协议；TCP是一个可以进行流量控制的协议

三次握手

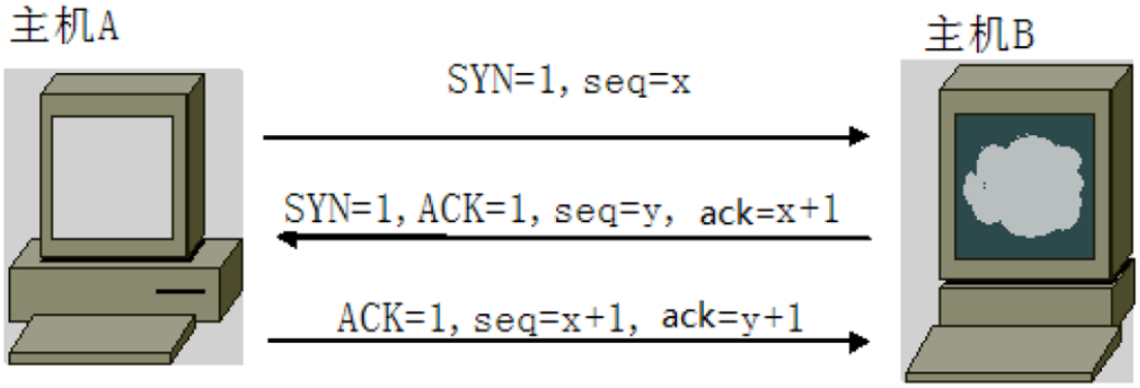
TCP建立连接的过程叫做握手，握手需要在客户端和服务器之间交换三个TCP报文段。三次握手的目的是为了建立可靠连接。

第一次握手：客户端给服务器发送一个同步报文段SYN，并指定客户端的初始序列号ISN，此时客户端处于SYN_SENT状态。首部的同步位SYN = 1(SYN只是一个比特位，0表示不是SYN，1表示是SYN)，初始序列号seq = m。SYN = 1的报文段不能携带任何的数据，但要指定序号。

第二次握手：服务器接收到来自客户端的同步报文段SYN后，会以自己的SYN报文作为应答，并且也指定了自己的初始序列号ISN。同时会把客户端的 seq+1 确认序列号ack的值，表示自己已经收到了客户端的同步报文段SYN，此时服务器处于SYN_RCVD的状态。确认报文段中SYN = 1,ACK = 1(ACK也只是一个比特位，0表示不是ACK，1表示是 ACK)，确认序列号ack = m+1，初始序列号seq = n。

第三次握手：客户端收到来自服务器的同步报文段SYN之后，会发送一个确认报文段ACK，以服务器的seq+1作为ack的值，表明自己已经收到来自服务器的同步报文段SYN。客户端进入ESTABLISHED状态，服务器确认报文段ACK之后，也会进入ESTABLISHED状态。确认报文段中，ACK = 1，确认序列号ack = n+1，序列号 seq = m+1。

双方已经建立起连接，可以正常的发送数据。



四次挥手

TCP的连接释放。由客户端到服务器需要一个FIN和ACK，再由服务器到客户端需要一个FIN和ACK，因此通常被称为四次握手。在断开连接之前客户端和服务器都处于ESTABLISHED状态，双方都可以主动断开连接，以客户端主动断开连接为优。

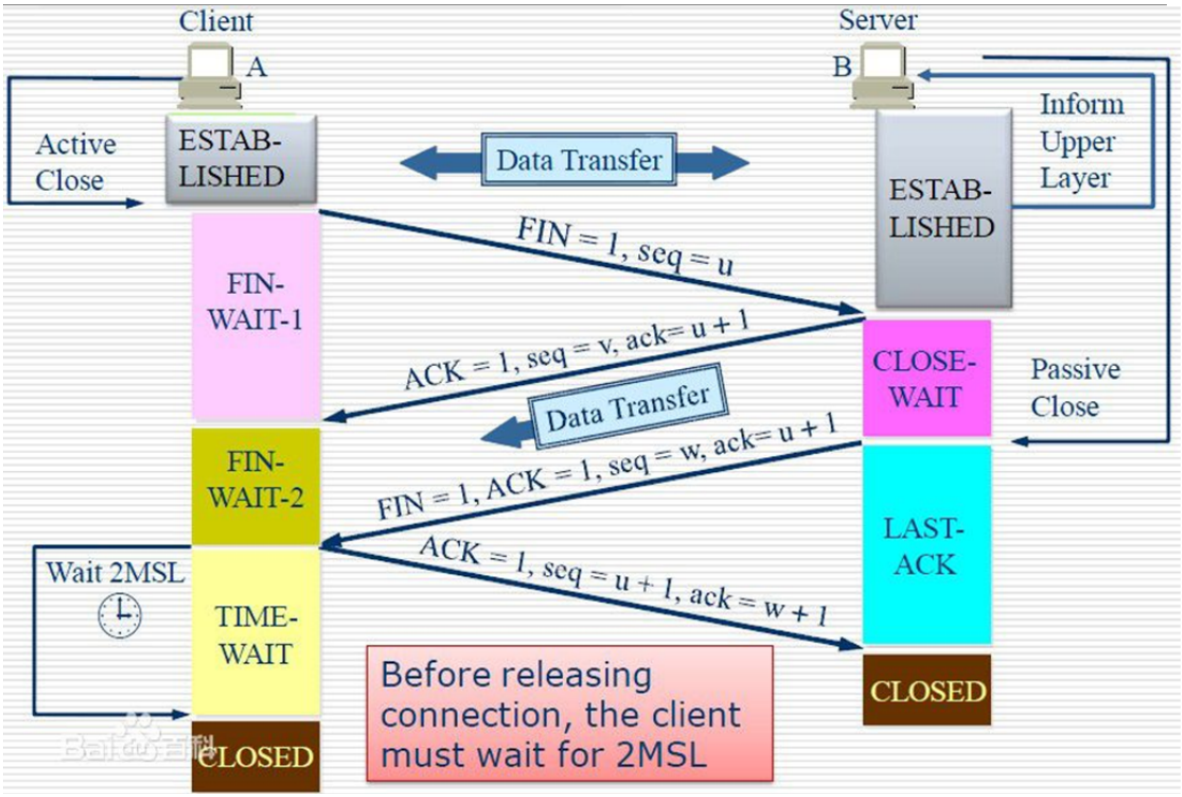
第一次挥手：客户端打算断开连接，向服务器发送FIN报文(FIN标记位被设置为1，1表示为FIN，0表示不是)，FIN报文中会指定一个序列号，之后客户端进入FIN_WAIT_1状态。也就是客户端发出连接释放报文段(FIN报文)，指定序列号seq = u，主动关闭TCP连接，等待服务器的确认。

第二次挥手：服务器收到连接释放报文段(FIN报文)后，就向客户端发送ACK应答报文，以客户端的FIN报文的序列号 seq+1 作为ACK应答报文段的确认序列号ack = seq+1 = u + 1。接着服务器进入CLOSE_WAIT(等待关闭)状态，此时的TCP处于半关闭状态(下面会说什么是半关闭状态)，客户端到服务器的连接释放。客户端收到来自服务器的ACK应答报文段后，进入FIN_WAIT_2状态。

第三次挥手：服务器也打算断开连接，向客户端发送连接释放(FIN)报文段，之后服务器进入LASK_ACK(最后确认)状态，等待客户端的确认。服务器的连接释放(FIN)报文段的FIN=1，ACK=1，序列号seq=m，确认序列号ack=u+1。

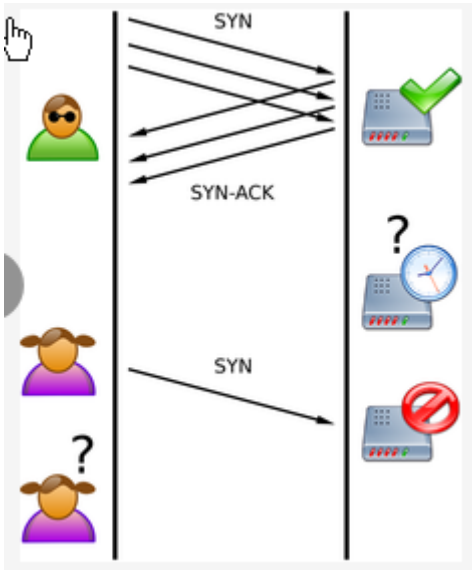
第四次挥手：客户端收到来自服务器的连接释放(FIN)报文段后，会向服务器发送一个ACK应答报文段，以连接释放(FIN)报文段的确认序号 ack 作为ACK应答报文段的序列号 seq，以连接释放(FIN)报文段的序列号 seq+1作为确认序号ack。之后客户端进入TIME_WAIT(时间等待)状态，服务器收到ACK应答报文段后，服务器就进入CLOSE(关闭)状态，到此服务器的连接已经完成关闭。

SYN请求建立连接的标志，ACK应答的标志。



SYN攻击

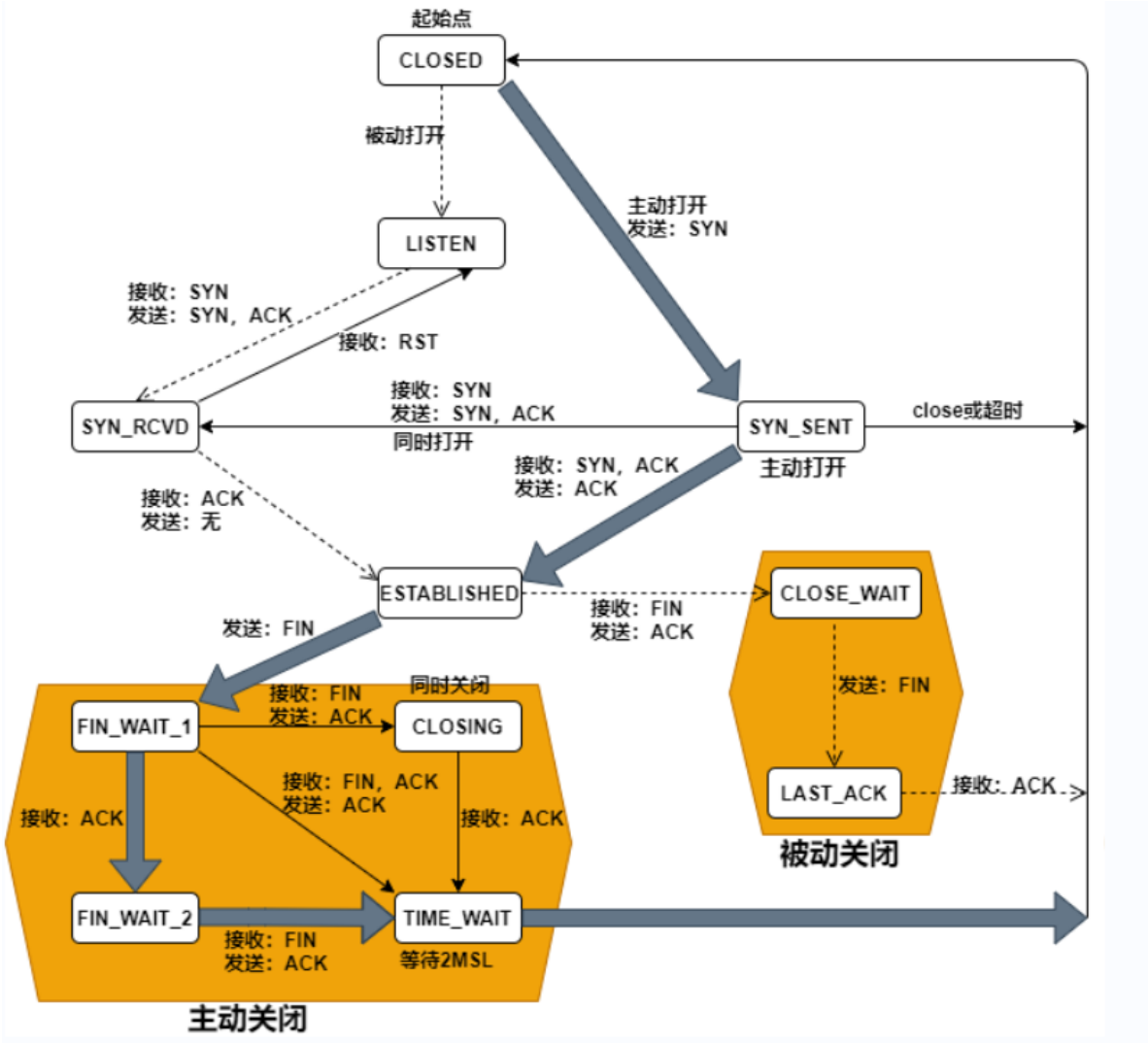
SYN攻击就是Client在短时间内伪造大量不存在的IP地址，并向Server不断地发送SYN包，Server回复确认包，并等待Client的确认，由于源地址是不存在的，因此，Server需要不断重发直至超时，这些伪造的SYN包将产时间占用未连接队列（内核会为每个这样的连接分配资源的），导致正常的SYN请求因为队列满而被丢弃，从而引起网络堵塞甚至系统瘫痪。SYN攻击是一种典型的DDOS攻击，检测SYN攻击的方式非常简单，即当Server上有大量半连接状态且源IP地址是随机的，则可以断定遭到SYN攻击了。



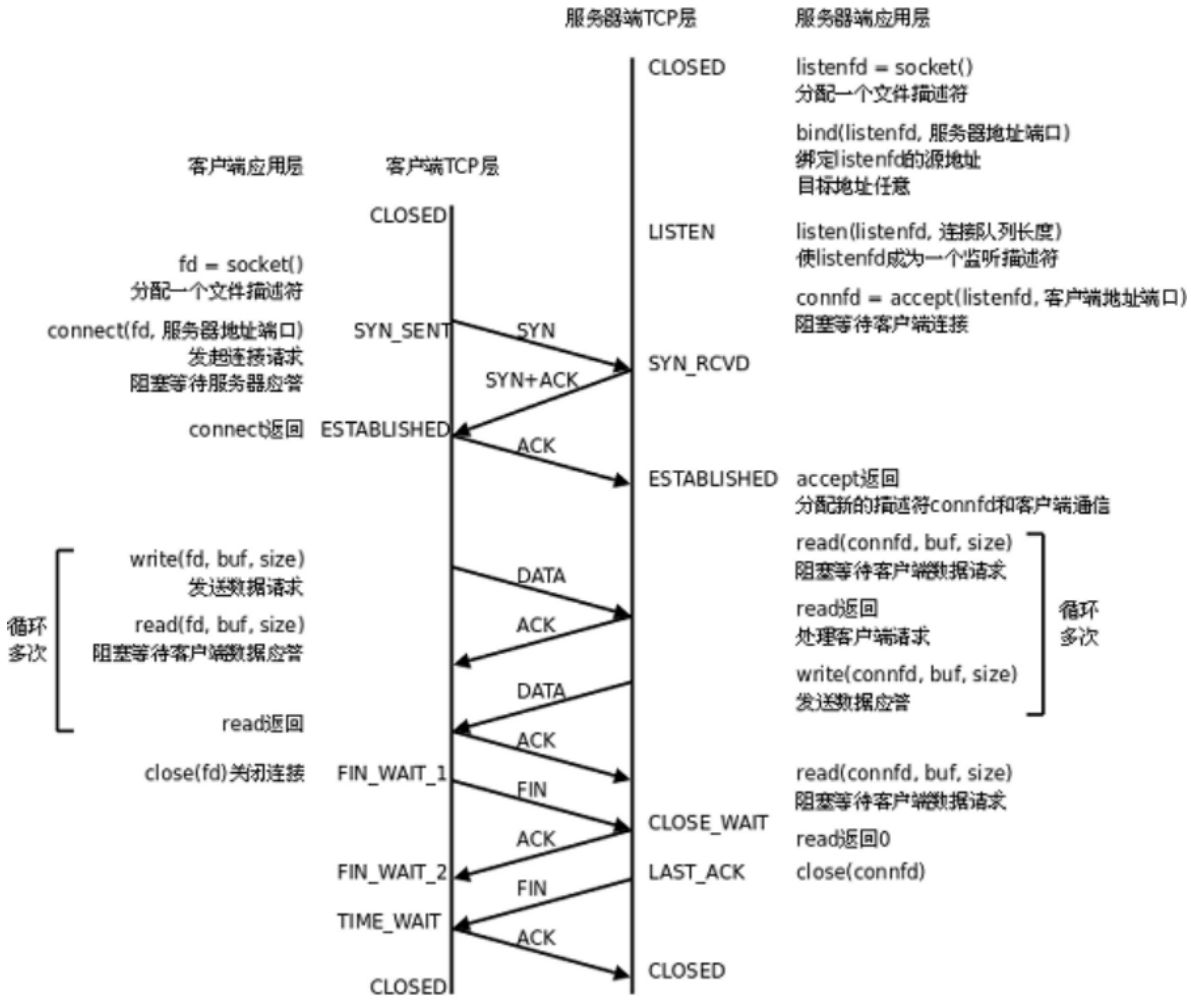
5、状态迁移图的解析

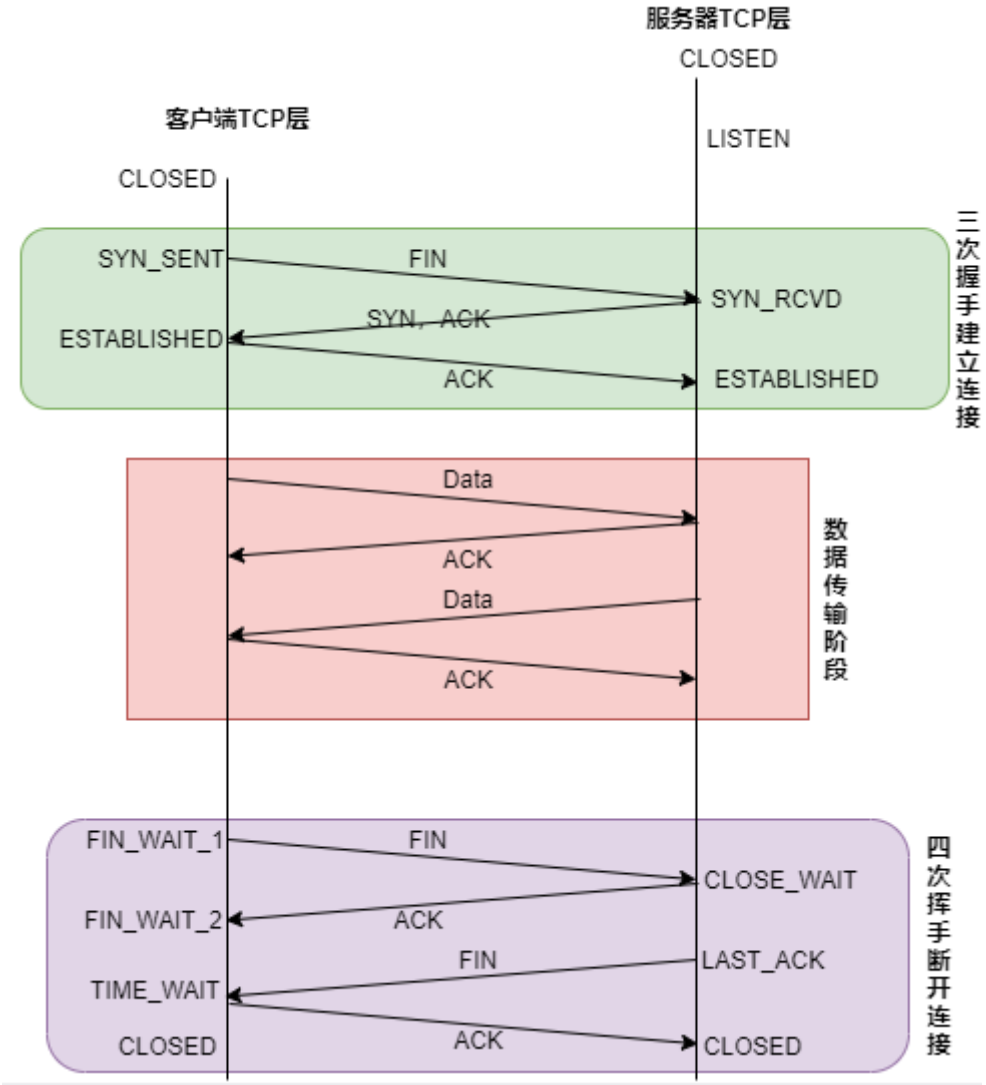
粗实线：主动发起连接与主动关闭连接；虚线：被动发起连接与被动关闭连接；细实线，两端同时操作的部分。

为何主动发起关闭的一端要设置TIME_WAIT的状态：确保最后一个ACK可以顺利到达对端。



6、TCP通信状态与程序结合分析


服务器端TCP层
服务器端应用层
CLOSED
listenfd = socket()
分配一个文件描述符
bind(listenfd, 服务器地址端口)
绑定listenfd的源地址
目标地址任意
客户端应用层
客户端TCP层
CLOSED
LISTEN
listen(listenfd, 连接队列长度)
使listenfd成为一个监听描述符
fd = socket()
分配一个文件描述符
connect(fd, 服务器地址端口)
发起连接请求
阻塞等待服务器应答
SYN_SENT
SYN
connfd = accept(listenfd, 客户端地址端口)
阻塞等待客户端连接
SYN_RCVD
SYN+ACK
connect返回
ESTABLISHED
ACK
ESTABLISHED
accept返回
分配新的描述符connfd和客户端通信
write(fd, buf, size)
发送数据请求
DATA
read(connfd, buf, size)
阻塞等待客户端数据请求
循环
多次
read(fd, buf, size)
阻塞等待客户端数据应答
ACK
read返回
处理客户端请求
循环
多次
DATA
write(connfd, buf, size)
发送数据应答
read返回
ACK
close(fd)关闭连接
FIN_WAIT_1
FIN
read(connfd, buf, size)
阻塞等待客户端数据请求
CLOSE_WAIT
ACK
read返回0
FIN_WAIT_2
LAST_ACK
close(connfd)
FIN
TIME_WAIT
ACK
CLOSED
CLOSED



二、网络编程

1、基础

IP地址可以在网络环境中唯一标识一台主机。端口号可以在主机中唯一标识一个进程。所以在网络环境中唯一标识一个进程可以使用IP地址与端口号Port。

2、字节序

TCP/IP协议规定，**网络数据流应采用大端字节序。**

大端：低地址存高位，高地址存低位；小端：低地址存低位，高地址存高位（x86采用小端存储）

网络字节序，就是在网络中进行传输的字节序列，采用的是大端法。主机字节序，就是本地计算机中存储数据采用的字节序列，采用的是小端法。

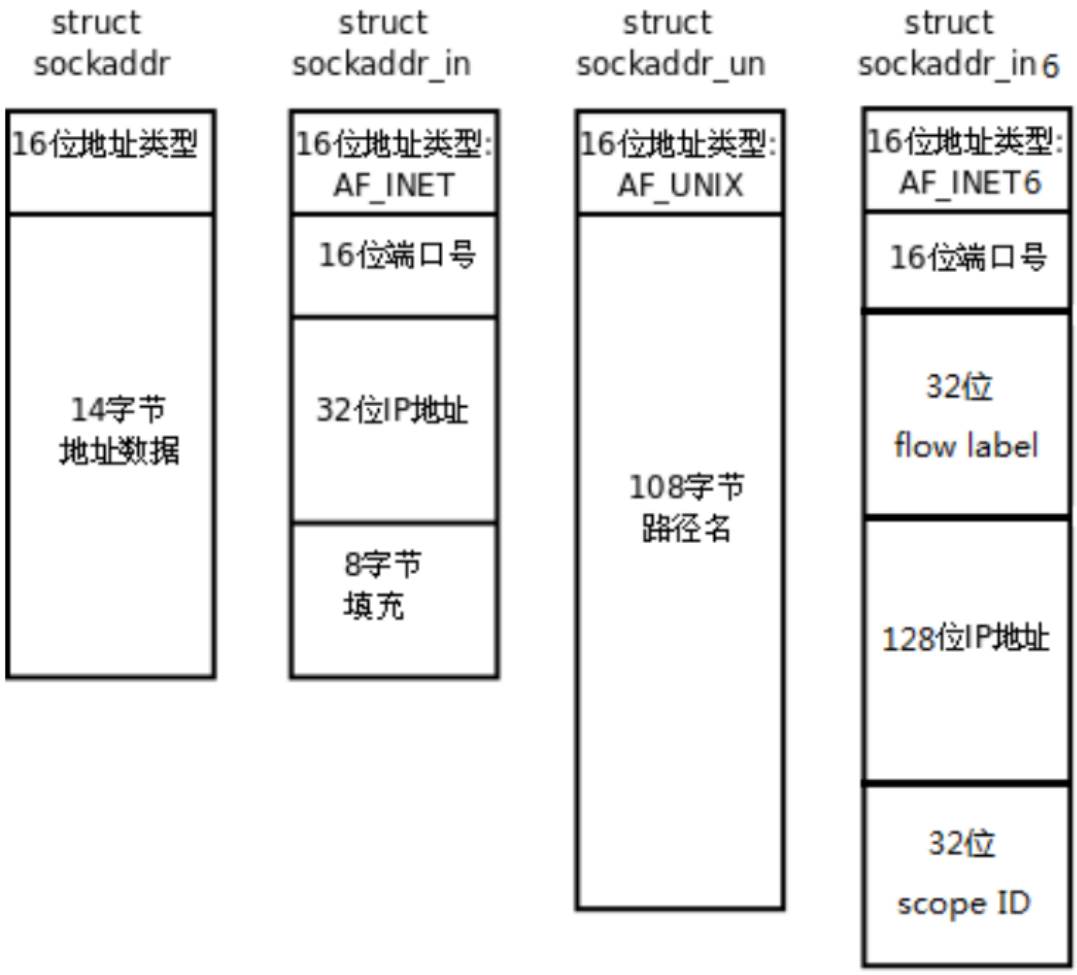
对应的函数

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);//h = host n = network l = long s = short
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

```c
#include <arpa/inet.h>
//点分十进制字符串转换为网络字节序
int inet_pton(int af, const char *src, void *dst);

//网络字节序转换为点分十进制字符串
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h> //包含了<netinet/in.h>，后者包含了<sys/socket.h>

typedef uint32_t in_addr_t;
struct in_addr
{
    in_addr_t s_addr;
};

//将cp所指C字符串转换成一个32位的网络字节序二进制值，并通过inp指针来存储，成功返回1，失败返回0
int inet_aton(const char *cp, struct in_addr *inp);

//将一个32位的网络字节序二进制IPv4地址转换成相应的点分十进制数串,由该函数的返回值所指向的字符串驻留在静态内存中,这意味着该函数是不可重入的
char *inet_ntoa(struct in_addr in);

//inet_addr函数转换网络主机地址（如192.168.1.10)为网络字节序二进制值，如果参数char *cp无效，函数返回-1(INADDR_NONE),这个函数在处理地址为255.255.255.255时也返回－1,255.255.255.255是一个有效的地址，不过inet_addr无法处理;
//返回值为32位的网络字节序二进制
in_addr_t inet_addr(const char *cp);//ok

in_addr_t inet_network(const char *cp);

struct in_addr inet_makeaddr(in_addr_t net, in_addr_t host);

in_addr_t inet_lnaof(struct in_addr in);

in_addr_t inet_netof(struct in_addr in);

#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

3、常用结构体

man 7 ip可以查看相应的结构体，也可以使用 命令sudo grep -rn "struct sockaddr_in {" /usr 进行搜索。




```c
struct sockaddr
{
    sa_family_t sa_family;      /* address family, AF_xxx */
    char sa_data[14];           /* 14 bytes of protocol address */
};

struct sockaddr_in
{
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr
{
    uint32_t       s_addr;     /* address in network byte order */
};
```

IPv4和IPv6的地址格式定义在netinet/in.h中，IPv4地址用sockaddr_in结构体表示，IPv6地址使用sockaddr_in6结构体表示。UNIX Domain Socket的地址格式定义在sys/un.h中，使用sockaddr_un结构体表示。所有的地址类型分别定义为常数AF_INET、AF_INET6、AF_UNIX。

```c
struct sockaddr_in addr;
addr.sin_family = AF_INET/AF_INET6/AF_UNIX;
addr.sin_port = htons/ntohs;
addr.sin_addr.s_addr = htonl/ntohl/inet_pton/inet_ntop
```

4、网络编程相关函数

4.1、socket函数

创建套接字

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
//创建套接字函数
int socket(int domain, int type, int protocol);
domain:AF_INET/AF_INET6/AF_UNIX
type:SOCK_STREAM/SOCK_DGRAM  前者默认是TCP，后者默认是UDP
protocol:传0表示使用默认协议

//函数返回值
成功，返回指向新创建的socket的文件描述符，失败返回-1，设置errno
```

4.2、bind函数

因为服务器程序所监听的网络地址与端口号是固定不变的，所以需要使用bind函数进行绑定。bind函数将sockfd与addr绑定在一起，使sockfd这个用于网络通讯的文件描述符监听addr所描述的地址和端口号。**绑定IP与端口号**

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
//绑定服务的端口号与IP地址
sockfd:上面socket创建的套接字
addr：所要绑定的ip地址与端口号
addrlen:前面addr结构体的长度

//函数返回值
成功，返回指向新创建的socket的文件描述符，失败返回-1，设置errno
```

4.3、listen函数

用来指定监听上限数值（允许同时多少个客户端与服务器建立连接），**指定最大同时发起连接数。**

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

int listen(int sockfd, int backlog);
sockfd:socket创建的文件描述符
backlog:排队建立3次握手队列和刚刚建立3次握手队列的连接数和。
```

可以使用命令进行查看：

```
cat /proc/sys/net/ipv4/tcp_max_syn_backlog
```

4.4、accept函数

接收连接请求的函数，**阻塞等待客户端发起连接**

如果客户端还没有来得及连接，此时accept函数会处于阻塞状态。

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
sockfd:socket创建的文件描述符
addr：传出参数，返回连接客户端地址信息，包含IP地址与端口号
addrlen:传入传出参数（值-结果），传入sizeof(addr)大小，函数返回时返回真正接收到地址结构体的大小。

//函数返回值
成功返回一个新的socket文件描述符，用于和客户端通信，失败返回-1，设置errno
```

4.5、connect函数

客户端调用该函数，连接到服务器上。**发起连接**

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
sockfd：是客户端自己使用socket得到的文件描述符。
addr:传入参数，指定服务器端地址信息，包含IP地址与端口号
addrlen:传入参数，传入sizeof(addr)大小

返回值：成功返回0，失败返回-1，设置errno
```

客户端需要调用connect连接服务器，connect和bind的采纳数形式一致，区别在于bind的参数是自己的地址，而connect的参数是对方的地址。

4.6、close函数

关闭套接字创建的文件描述符。

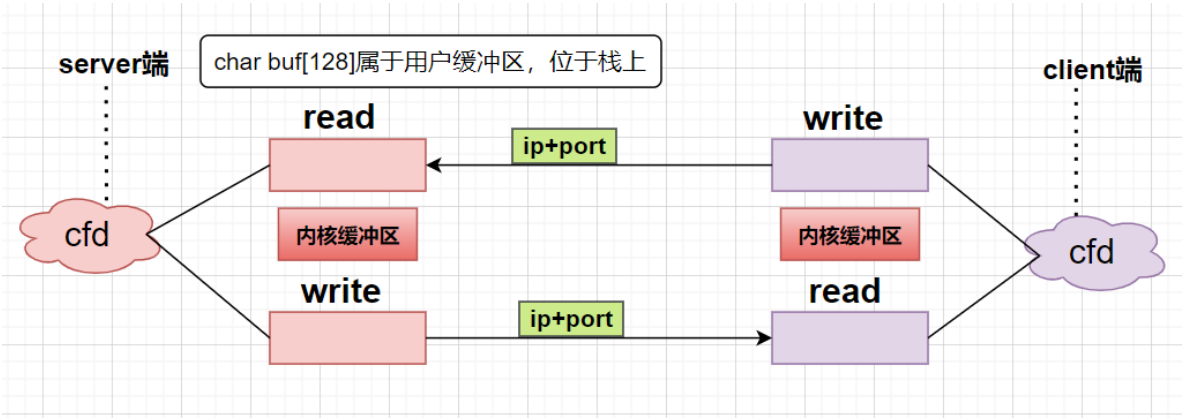
```
#include <unistd.h>
int close(int fd);
```

客户端其实也是需要bind端口号与IP地址，如果没有显示绑定的话，操作系统会自动分配一个IP地址与端口号。但是服务器是不能不使用bind函数，让操作系统随机分配IP地址与端口号，这样的话客户端就不知道服务器的IP地址与端口号，就不知道怎么连接到服务器上了，也不知道连接到那个服务器上。

本地随机的有效数字类型的IP，INADDR_ANY。

INADDR_ANY解析：转换过来就是0.0.0.0，泛指本机的意思，表示本机的所有IP，因为有些电脑不止一块网卡，如果某个应用程序只监听某个端口，那么其他端口过来的数据就接收不了。

5、网络编程代码



5.1、端口复用

让同一个端口可以进行重复使用，不至于等待2MSL的时间

```
#include <sys/types.h>
#include <sys/socket.h>
int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t
*optlen);

int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
```

```
int opt = 1;
setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

int opt = 1;
setsockopt(listenfd, SOL_SOCKET, SO_REUSEPORT, &opt, sizeof(opt));
```

5.2、服务器端源码

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <ctype.h>

#define SERV_IP "127.0.0.1"
#define SERV_PORT 6666

int main()
{
    int sfd, cfd;
    struct sockaddr_in serv_addr, clie_addr;
    socklen_t clie_addr_len;
    char buf[BUFSIZ], clie_IP[BUFSIZ];
    int nByte, idx;

    sfd = socket(AF_INET, SOCK_STREAM, 0);

    int opt = 1;
```

```
    setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));//允许端口复用

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(SERV_PORT);
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);

    bind(sfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    listen(sfd, 128);

    clie_addr_len = sizeof(clie_addr);
    cfd = accept(sfd, (struct sockaddr *)&clie_addr, &clie_addr_len);
    printf("client IP: %s, port: %d\n",
           inet_ntop(AF_INET, &clie_addr.sin_addr.s_addr, clie_IP, sizeof(clie_IP)),
           ntohs(clie_addr.sin_port));

    while(1)
    {
        nByte = read(cfd, buf, sizeof(buf));

        for(idx = 0; idx < nByte; ++idx)
        {
            buf[idx] = toupper(buf[idx]);
        }

        write(cfd, buf, nByte);
    }

    close(sfd);
    close(cfd);
    return 0;
}
```

5.3、客户端源码

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <ctype.h>
#include <string.h>

#define SERV_IP "127.0.0.1"
#define SERV_PORT 6666

int main()
{
    int cfd;
    struct sockaddr_in serv_addr;
    char buf[BUFSIZ];
    int nByte;

    cfd = socket(AF_INET, SOCK_STREAM, 0);
```

```
20
21      memset(&serv_addr, 0, sizeof(serv_addr));
22      serv_addr.sin_family = AF_INET;
23      serv_addr.sin_port = htons(SERV_PORT);
24      serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
25      /* inet_pton(cfd, SERV_IP, &serv_addr.sin_addr.s_addr); */
26
27      connect(cfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
28
29      while(1)
30      {
31          fgets(buf, sizeof(buf), stdin);//hello world  ----> hello world\n\0
32          write(cfd, buf, strlen(buf));
33
34          nByte = read(cfd, buf, sizeof(buf));
35          write(STDOUT_FILENO, buf, nByte);
36      }
37
38      close(cfd);
39      return 0;
40  }
```

5.4、read返回值

1、大于0，实际读到的字节数，并且buf=1024

- 如果read读到的数据的长度等于buf，返回的就是1024
- 如果read读到的数据长度小于buf，那就是小于1024的数值。

2、返回值为0，数据读完（读到文件、管道、socket末尾 ---对端关闭）

3、返回值为-1，表明出现异常

- errno == EINTR 说明被信号中断 所以需要重启或者退出
- errno == EAGAIN（EWOULDBLOCK）非阻塞方式读，并且没有数据
- 其他值 真的出现错误 perror打印 exit

5.5、readn/writen函数的封装

因为以太网帧一次只能传送1500字节的数据，所以使用read函数一次最多只能读到1500字节，就返回退出。

```
1   ssize_t readn(int fd, void *vptr, size_t n)
2   {
3       size_t nleft;//usigned int剩余未读取的字节数
4       size_t nread;//int 实际读到的字节数
5       char *ptr;
6
7       nleft = n;//n未读取字节数
8       ptr = vptr;
9
10      while(nleft > 0)
11      {
12          if((nread = read(fd, ptr, nleft)) < 0)
13          {
14              if(errno == EINTR)
15              {
16                  nread = 0;
```

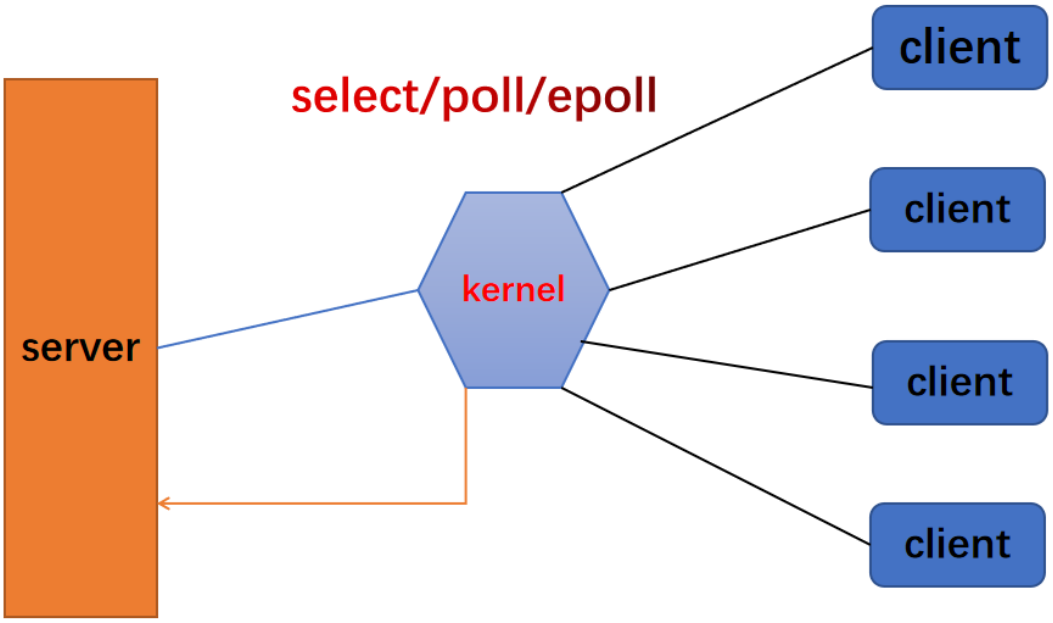
```
            }
            else
            {
                return -1;
            }
        }
        else if(0 == nread)
        {
            break;
        }

        nleft -= nread;
        ptr += nread;
    }

    return (n - nleft);
}
```

```
ssize_t writen(int fd, const void *vptr, size_t n)
{
    size_t nleft;
    size_t nwritten;
    const char *ptr;

    nleft = n;
    ptr = vptr;

    while(nleft > 0)
    {
        if((nwritten = write(fd, ptr, nleft)) <= 0)
        {
            if(nwritten < 0 && errno == EINTR)
            {
                nwritten = 0;
            }
            else
            {
                return -1;
            }
        }

        nleft -= nwritten;
        ptr += nwritten;
    }

    return n;
}
```

三、IO多路复用

1、概念与原理图

多进程与多线程并发服务器，不经常使用这种作为大型服务器开发的原因是，所有的监听与访问请求都由服务器操作。可以使用多路IO转接服务器（也叫多任务IO服务器），思想：不再由应用程序自己监视客户端连接，取而代之由内核替应用程序监视文件。



2、select

2.1、接口解析

```
#include <sys/select.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);

nfds:监控的文件描述符集里最大文件描述符+1，因为此参数会告诉内核检测前多少个文件描述符的状态。
readfs/writes/exceptfds:监控有读数据/写数据/异常发生到达文件描述符集合，三个都是传入传出参数。
timeout:定时阻塞监控时间，3中情况：
    1、NULL，永远等下去
    2、设置timeval，等待固定时间
    3、设置timeval里时间均为0，检查描述字后立即返回，轮询。

    fd_set：本质是个位图。

struct timeval
{
     long    tv_sec;         /* seconds */
     long    tv_usec;        /* microseconds */
};
返回值：
    成功：所监听的所有的监听集合中，满足条件的总数。
    失败：返回-1.
```

```
void FD_ZERO(fd_set *set);//将set清空为0
void FD_SET(int fd, fd_set *set);//将fd设置到set集合中
void FD_CLR(int fd, fd_set *set);//将fd从set中清除出去
int  FD_ISSET(int fd, fd_set *set);//判断fd是否在集合中
```

2.2、优缺点

1、文件描述符上限（1024），同时监听的文件描述符1024个，历史原因，不好修改，除非重新编译Linux内核。2、当监听的文件描述符个数比较稀疏的时候（比如3， 600， 1023），循环判断比较麻烦，所以需要自定义数据结构：数组
3、监听集合与满足监听条件的集合是同一个，需要将原有集合保存。

2.3、代码实现（C语言）

server端

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/select.h>
#include <sys/time.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>
#include <unistd.h>
#include <ctype.h>

#define SERV_PORT 8888

int main()
{
    int listenfd, connfd, sockfd;
    struct sockaddr_in serv_addr, clie_addr;
    socklen_t clie_addr_len;
    int ret, maxfd, maxi, i, j, nready, nByte;
    fd_set rset, allset;
    int client[FD_SETSIZE];
    char buf[BUFSIZ], str[BUFSIZ];

    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if(-1 == listenfd)
    {
        perror("socket error");
        exit(-1);
    }

    int opt = 1;
    setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    bzero(&serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(SERV_PORT);
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);

    ret = bind(listenfd, (struct sockaddr *)&serv_addr,
sizeof(serv_addr));
```

```
41      if(-1 == ret)
42      {
43          perror("bind error");
44          exit(-1);
45      }
46
47      ret = listen(listenfd, 128);
48      if(-1 == ret)
49      {
50          perror("listen error");
51          exit(-1);
52      }
53
54      maxfd = listenfd;
55
56      maxi = -1;
57      for(i = 0; i < FD_SETSIZE; ++i)
58      {
59          client[i] = -1;
60      }
61
62      FD_ZERO(&allset);
63      FD_SET(listenfd, &allset);
64
65      while(1)
66      {
67          rset = allset;
68          nready = select(maxfd + 1, &rset, NULL, NULL, NULL);
69          if(nready < 0)
70          {
71              perror("select error");
72              exit(-1);
73          }
74
75          if(FD_ISSET(listenfd, &rset))
76          {
77              clie_addr_len = sizeof(clie_addr);
78              connfd = accept(listenfd, (struct sockaddr *)&clie_addr,
&clie_addr_len);
79              if(-1 == connfd)
80              {
81                  perror("accept error");
82                  exit(-1);
83              }
84              printf("receive from %s from port %d\n",
85                     inet_ntop(AF_INET, &clie_addr.sin_addr, str,
sizeof(str)),
86                     ntohs(clie_addr.sin_port));
87
88              for(i = 0; i < FD_SETSIZE; ++i)
89              {
90                  if(client[i] < 0)
91                  {
92                      client[i] = connfd;
93                      break;
94                  }
95              }
96
```

```
            if(i == FD_SETSIZE)
            {
                fputs("too many clients\n", stderr);
                exit(1);
            }

            FD_SET(connfd, &allset);

            if(connfd > maxfd)
            {
                maxfd = connfd;
            }

            if(i > maxi)
            {
                maxi = i;
            }

            if(--nready == 0)
            {
                continue;
            }
        }

        for(i = 0; i <= maxi; ++i)
        {
            if((sockfd = client[i]) < 0)
            {
                continue;
            }

            if(FD_ISSET(sockfd, &rset))
            {
                if((nByte = read(sockfd, buf, sizeof(buf))) == 0)
                {
                    close(sockfd);
                    FD_CLR(sockfd, &allset);
                    client[i] = -1;
                }
                else if(nByte > 0)
                {
                    for(j = 0; j < nByte; ++j)
                    {
                        buf[j] = toupper(buf[j]);
                    }
                    write(sockfd, buf, nByte);
                    write(STDOUT_FILENO, buf, nByte);
                }

                if(--nready == 0)
                {
                    break;
                }
            }
        }
    }

    close(listenfd);
```

```
    close(connfd);
    return 0;
}
```

client端

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

#define SERV_IP "127.0.0.1"
#define SERV_PORT 8888

int main()
{
    int cfd;
    struct sockaddr_in serv_addr;
    char buf[BUFSIZ];
    int nByte;

    cfd = socket(AF_INET, SOCK_STREAM, 0);
    if(-1 == cfd)
    {
        perror("socket error");
        exit(-1);
    }

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(SERV_PORT);
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    /* inet_pton(cfd, SERV_IP, &serv_addr.sin_addr.s_addr); */

    connect(cfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    while(1)
    {
        fgets(buf, sizeof(buf), stdin);//hello world  ----> hello world\n\0
        write(cfd, buf, strlen(buf));

        nByte = read(cfd, buf, sizeof(buf));

        write(STDOUT_FILENO, buf, nByte);
    }

    close(cfd);
    return 0;
}
```

3、poll

3.1、接口解析

```
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);

struct pollfd
{
    int   fd;         /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
};

fds:文件描述符数组。
    events：POLLIN/POLLOUT/POLLERR
nfds：监控数组中有多少文件描述符需要被监控。
timeout 毫秒级等待
        -1:阻塞等，#define INFTIM -1  Linux中没有定义此宏
        0:立即返回，不阻塞进程
        >0:等待指定毫秒数，如当前系统时间精度不够毫秒，向上取值。
函数返回值：满足监听条件的文件描述符的数目。
```

3.2、优缺点

优点：
1、突破文件描述符1024的上限
2、监听与返回的集合分离
3、搜索范围变小（已经知道是那几个数组）

缺点：

监听1000个文件描述符，但是只有3个满足条件，这样也需要全部遍历，效率依旧低。

cat /proc/sys/fs/file-max 查看一个进程可以打开的文件描述符的上限数。

sudo vi /etc/security/limits.conf。在文件尾部写入以下配置,soft软限制，hard硬限制。

```
soft nofile 65536
hard nofile 100000
```

3.3、代码实现（C语言）

server端

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <poll.h>
#include <errno.h>

#define SERV_PORT 8888
#define OPEN_MAX 1024
```

```
14
15  int main()
16  {
17      int i, j, n, maxi;
18      int nready, ret;
19      int listenfd, connfd, sockfd;
20      char buf[BUFSIZ], str[INET_ADDRSTRLEN];
21      struct sockaddr_in serv_addr, clie_addr;
22      socklen_t clie_addr_len;
23      struct pollfd client[OPEN_MAX];
24
25      listenfd = socket(AF_INET, SOCK_STREAM, 0);
26      if(-1 == listenfd)
27      {
28          perror("socket error");
29          exit(-1);
30      }
31
32      int opt = 1;
33      setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
34
35      memset(&serv_addr, 0, sizeof(serv_addr));
36      serv_addr.sin_family = AF_INET;
37      serv_addr.sin_port = htons(SERV_PORT);//本地字节序port与ip都要转换为网络字
    节序
38      serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);//因为要在网络上传输
39
40      ret = bind(listenfd, (struct sockaddr *)&serv_addr,
    sizeof(serv_addr));
41      if(-1 == ret)
42      {
43          perror("bind error");
44          exit(-1);
45      }
46
47      ret = listen(listenfd, 128);
48      if(-1 == ret)
49      {
50          perror("listen error");
51          exit(-1);
52      }
53
54      client[0].fd = listenfd;
55      client[0].events = POLLIN;
56
57      for(i = 1; i < OPEN_MAX; ++i)
58      {
59          client[i].fd = -1;//将数组初始化为-1
60      }
61
62      maxi = 0;
63
64      while(1)
65      {
66          nready = poll(client, maxi + 1, -1);
67          if(nready < 0)
68          {
69              perror("poll error");
```

```
70              exit(-1);
71          }
72
73          if(client[0].revents & POLLIN)
74          {
75              clie_addr_len = sizeof(clie_addr);
76              connfd = accept(listenfd, (struct sockaddr *)&clie_addr,
    &clie_addr_len);//立即连接，此时不会阻塞等
77              if(-1 == connfd)
78              {
79                  perror("accept error");
80                  exit(-1);
81              }
82
83              printf("received from %s at port %d\n",
84                     inet_ntop(AF_INET, &clie_addr.sin_addr.s_addr, str,
    sizeof(str)),
85                     ntohs(clie_addr.sin_port));
86
87              for(i = 1; i < OPEN_MAX; ++i)
88              {
89                  if(client[i].fd < 0)//因为初始化为-1,所以在此作为判断条件
90                  {
91                      client[i].fd = connfd;
92                      break;//直接跳出，免得继续判断,浪费时间
93                  }
94              }
95
96              if(i == OPEN_MAX)//select监听的文件描述符有上限，最大只能监听1024个
97              {
98                  fputs("too many clients\n", stderr);
99                  exit(1);
100             }
101
102             client[i].events = POLLIN;
103
104             if(i > maxi)
105             {
106                 maxi = i;//因为文件描述符有新增，导致自定义数组有变化，所以需要重新
    修改maxi的值
107             }
108
109             if(--nready == 0)//意思不明确
110             {
111                 continue;
112             }
113         }
114
115         for(i = 1; i <= maxi; ++i)
116         {
117             if((sockfd = client[i].fd) < 0)
118             {
119                 continue;
120             }
121
122             if(client[i].revents & POLLIN)
123             {
124                 if((n = read(sockfd, buf, sizeof(buf))) < 0)
```

```
                {
                    if(errno == ECONNRESET)
                    {
                        printf("client[%d] abort connect\n", i);
                        close(sockfd);
                        client[i].fd = -1;
                    }
                    else
                    {
                        perror("read n = 0 error");
                    }
                }
                else if(n > 0)
                {
                    for(j = 0; j < n; ++j)
                    {
                        buf[j] = toupper(buf[j]);
                    }
                    write(sockfd, buf, n);
                    write(STDOUT_FILENO, buf, n);
                }
                else
                {
                    close(sockfd);
                    client[i].fd = -1;
                }

                if(--nready == 0)
                {
                    break;
                }
            }
        }
    }
    close(listenfd);
    close(connfd);

    return 0;
}
```

client端

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

#define SERV_IP "127.0.0.1"
#define SERV_PORT 8888

int main()
{
    int cfd;
```

```c
    struct sockaddr_in serv_addr;
    char buf[BUFSIZ];
    int nByte;

    cfd = socket(AF_INET, SOCK_STREAM, 0);
    if(-1 == cfd)
    {
        perror("socket error");
        exit(-1);
    }

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(SERV_PORT);
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    /* inet_pton(cfd, SERV_IP, &serv_addr.sin_addr.s_addr); */

    connect(cfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    while(1)
    {
        fgets(buf, sizeof(buf), stdin);//hello world  ----> hello world\n\0
        write(cfd, buf, strlen(buf));

        nByte = read(cfd, buf, sizeof(buf));

        write(STDOUT_FILENO, buf, nByte);
    }

    close(cfd);
    return 0;
}
```

4、epoll

4.1、接口解析

是Linux下IO多路复用接口select/poll的增强版本，能显著提高程序在大量并发连接中只有**少量活跃**的情况下的系统CPU利用率，因为它会复用文件描述符集合来传递结果而不是迫使开发者每次等待事件之前都必须重新准备要侦听的文件描述符集合，另一个原因是获取事件的时候，**它无须遍历整个被侦听的描述符集，只要遍历哪些被内核IO事件唤醒而加入Ready队列的描述符集合**就行了。

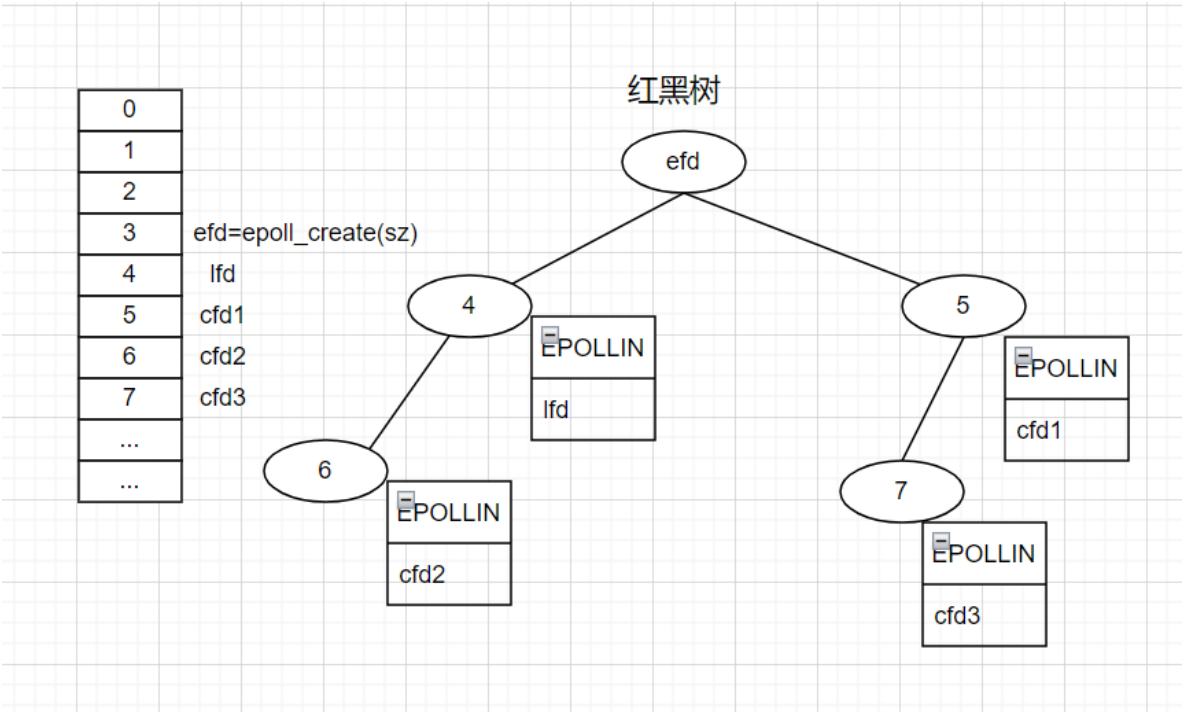
```c
#include <sys/epoll.h>
int epoll_create(int size);
size：参数size用来告知内核监听的文件描述符的个数，与内存大小有关。


//控制某个epoll监控的文件描述符上的事件：注册、修改、删除
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
epfd：epoll_create函数返回的值
op：EPOLL_CTL_ADD/EPOLL_CTL_MOD/EPOLL_CTL_DEL
fd：将哪个文件描述符以op的方式加在以epfd建立的树上
event：告诉内核需要监听的事情。
struct epoll_event
{
```

```
    uint32_t    events;
    epoll_data_t data;
};
typedef union epoll_data
{
    void  *ptr;
    int   fd;
    uint32_t  u32;
    uint64_t  u64;
} epoll_data_t;

//等待所监控文件描述符上有事件的产生
int epoll_wait(int epfd, struct epoll_event *events,int maxevents, int
timeout);
events：用来存内核得到事件的集合（这里是个传出参数）
maxevents：告知内核这个events有多大，这个maxevents的值不能大于创建epoll_create时的
size
timeout：是超时时间
          -1：阻塞
          =0：立即返回，非阻塞
          >0：指定毫秒
返回值：成功返回有多少文件描述符就绪，时间到时返回0，出错返回-1
```

4.2、优缺点

1、文件描述符数目没有上限：通过epoll_ctl()来注册一个文件描述符，内核中使用红黑树的数据结构来管理所有需要监控的文件描述符。
2、基于事件就绪通知方式：一旦被监听的某个文件描述符就绪，内核会采用类似于callback的回调机制，迅速激活这个文件描述符，这样随着文件描述符数量的增加，也不会影响判定就绪的性能。
3、维护就绪队列：当文件描述符就绪，就会被放到内核中的一个就绪队列中，这样调用epoll_weit获取就绪文件描述符的时候，只要取队列中的元素即可，操作的时间复杂度恒为O(1)。

4.3、图解

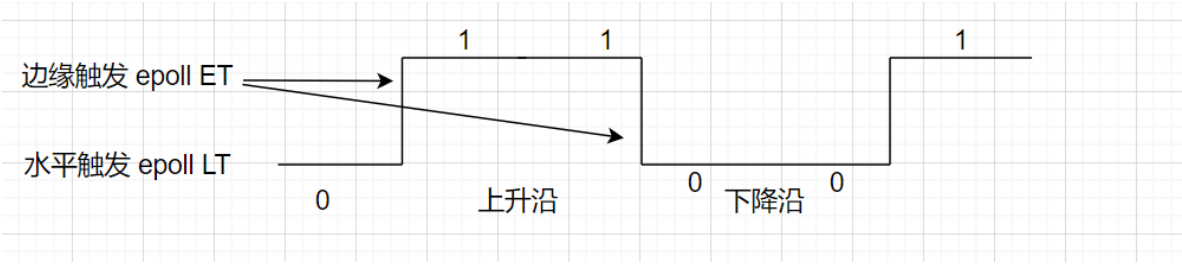


4.4、类型区别

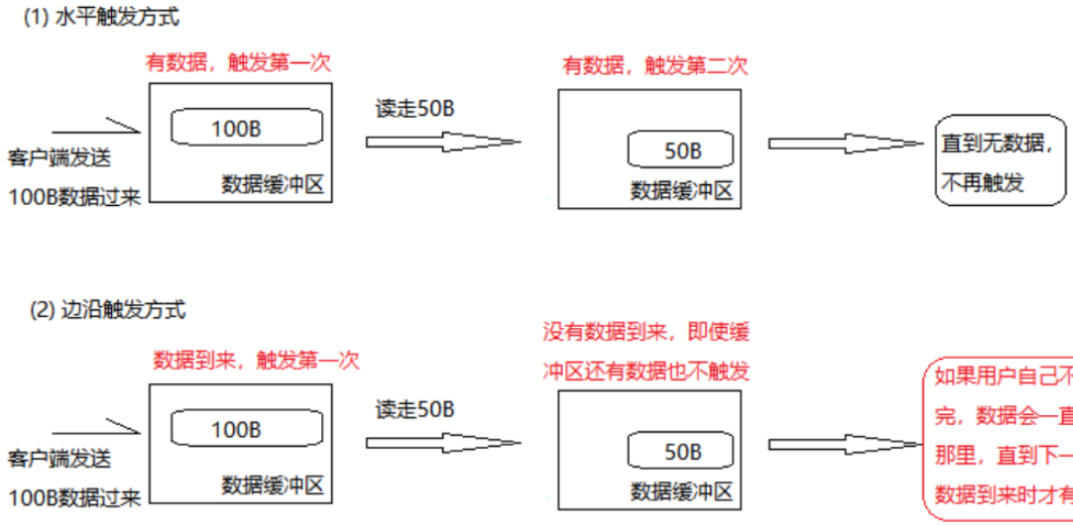
水平触发(level-trggered)
只要文件描述符关联的**读内核缓冲区**非空，有数据可以读取，就一直发出可读信号进行通知；当文件描述符关联的**内核写缓冲区**不满，有空间可以写入，就一直发出可写信号进行通知LT模式支持阻塞和非阻塞两种方式。epoll默认的模式是LT。

边缘触发(edge-triggered)
当文件描述符关联的**读内核缓冲区由空转化为非空**的时候，则发出可读信号进行通知；当文件描述符关联的**内核写缓冲区由满转化为不满的时候**，则发出可写信号进行通知。

两者的区别在哪里呢？水平触发是只要读缓冲区有数据，就会一直触发可读信号，而边缘触发仅仅在空变为非空的时候通知一次。LT(level triggered)是缺省的工作方式，并且同时支持block和no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。传统的select/poll都是这种模型的代表.



当设置了边缘触发以后，以可读事件为例，对"有数据到来"这事件为触发。



select/poll/epoll除了应用于fd外，像管道、文件也是可以的。

4.5、代码实现（C语言）

server端

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/select.h>
#include <sys/time.h>
#include <sys/epoll.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
```

```c
#include <ctype.h>

#define SERV_PORT 8888
#define OPEN_MAX 5000

int main()
{
    int listenfd, connfd, sockfd, epfd;
    struct sockaddr_in serv_addr, clie_addr;
    socklen_t clie_addr_len;
    int ret, i, j, nready, nByte;
    char buf[BUFSIZ], str[BUFSIZ];
    struct epoll_event evt, ep[OPEN_MAX];


    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if(-1 == listenfd)
    {
        perror("socket error");
        exit(-1);
    }

    int opt = 1;
    setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    bzero(&serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(SERV_PORT);
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);

    ret = bind(listenfd, (struct sockaddr *)&serv_addr,
sizeof(serv_addr));
    if(-1 == ret)
    {
        perror("bind error");
        exit(-1);
    }

    ret = listen(listenfd, 128);
    if(-1 == ret)
    {
        perror("listen error");
        exit(-1);
    }

    epfd = epoll_create(OPEN_MAX);
    if(-1 == epfd)
    {
        perror("epoll_create error");
        exit(-1);
    }

    evt.events = EPOLLIN;
    evt.data.fd = listenfd;

    ret = epoll_ctl(epfd, EPOLL_CTL_ADD, listenfd, &evt);
    if(-1 == ret)
    {
```

```
68          perror("epoll_ctl error");
69          exit(-1);
70      }
71
72      while(1)
73      {
74          nready = epoll_wait(epfd, ep, OPEN_MAX, -1);
75          if(nready < 0)
76          {
77              perror("select error");
78              exit(-1);
79          }
80
81          for(i = 0; i < nready; ++i)
82          {
83              if(!(ep[i].events & EPOLLIN))
84              {
85                  continue;
86              }
87
88              if(ep[i].data.fd == listenfd)//如果是连接事件
89              {
90                  clie_addr_len = sizeof(clie_addr);
91                  connfd = accept(listenfd, (struct sockaddr *)&clie_addr,
&clie_addr_len);
92                  if(-1 == connfd)
93                  {
94                      perror("accept error");
95                      exit(-1);
96                  }
97                  printf("receive from %s from port %d\n",
98                         inet_ntop(AF_INET, &clie_addr.sin_addr, str,
sizeof(str)),
99                         ntohs(clie_addr.sin_port));
100
101                 evt.events = EPOLLIN;
102                 evt.data.fd = connfd;
103                 epoll_ctl(epfd, EPOLL_CTL_ADD, connfd, &evt);
104             }
105             else //不是连接建立事件，而是读写事件(信息传递事件)
106             {
107                 sockfd = ep[i].data.fd;
108                 nByte = read(sockfd, buf, sizeof(buf));
109                 if(nByte == 0)
110                 {
111                     ret = epoll_ctl(epfd, EPOLL_CTL_DEL, sockfd, NULL);
112                     if(-1 == ret)
113                     {
114                         perror("epoll_ctl error");
115                     }
116                     close(sockfd);
117                     printf("client[%d] closed connection\n", sockfd);
118                 }
119                 else if(nByte < 0)
120                 {
121                     perror("epoll_ctl error");
122                     ret = epoll_ctl(epfd, EPOLL_CTL_DEL, sockfd, NULL);
123                     if(-1 == ret)
```

```
                    {
                        perror("epoll_ctl error");
                    }
                    close(sockfd);

                }
                else
                {
                    for(j = 0; j < nByte; ++j)
                    {
                        buf[j] = toupper(buf[j]);
                    }
                    write(sockfd, buf, nByte);
                    write(STDOUT_FILENO, buf, nByte);
                }
            }
        }
    }

    close(listenfd);
    close(connfd);

    return 0;
}
```

client端

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

#define SERV_IP "127.0.0.1"
#define SERV_PORT 8888

int main()
{
    int cfd;
    struct sockaddr_in serv_addr;
    char buf[BUFSIZ];
    int nByte;

    cfd = socket(AF_INET, SOCK_STREAM, 0);
    if(-1 == cfd)
    {
        perror("socket error");
        exit(-1);
    }

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(SERV_PORT);
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
    /* inet_pton(cfd, SERV_IP, &serv_addr.sin_addr.s_addr); */

    connect(cfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    while(1)
    {
        fgets(buf, sizeof(buf), stdin);//hello world  ----> hello world\n\0
        write(cfd, buf, strlen(buf));

        nByte = read(cfd, buf, sizeof(buf));

        write(STDOUT_FILENO, buf, nByte);
    }

    close(cfd);
    return 0;
}
```