一、Socket 编程

1. 套接字概念

Socket本身有“插座”的意思，在Linux环境下，用于表示进程间网络通信的特殊文件类型。本质为内核借助缓冲区形成的伪文件。

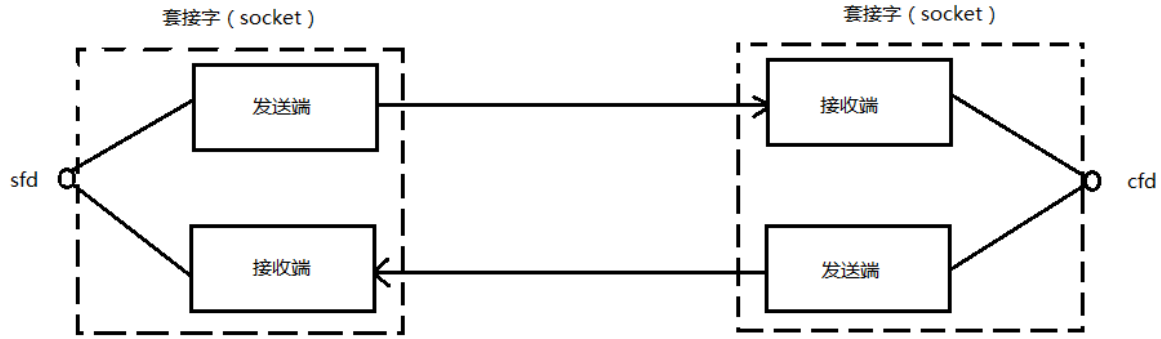
既然是文件，那么理所当然的，我们可以使用文件描述符引用套接字。与管道类似的，Linux系统将其封装成文件的目的是为了统一接口，使得读写套接字和读写文件的操作一致。区别是管道主要应用于本地进程间通信，而套接字多应用于网络进程间数据的传递。

在TCP/IP协议中，“IP地址+TCP或UDP端口号”唯一标识网络通讯中的一个进程。“IP地址+端口号”就对应一个socket。欲建立连接的两个进程各自有一个socket来标识，那么这两个socket组成的socket pair就唯一标识一个连接。因此可以用Socket来描述网络连接的一对一关系。

套接字（Socket）是计算机网络中一种常见的通信机制，它允许不同的进程在网络上进行通信。套接字是一种软件编程接口，它定义了一组用于在不同计算机之间进行通信的函数。通过使用套接字，应用程序可以发送和接收数据，以及进行其他网络通信操作。

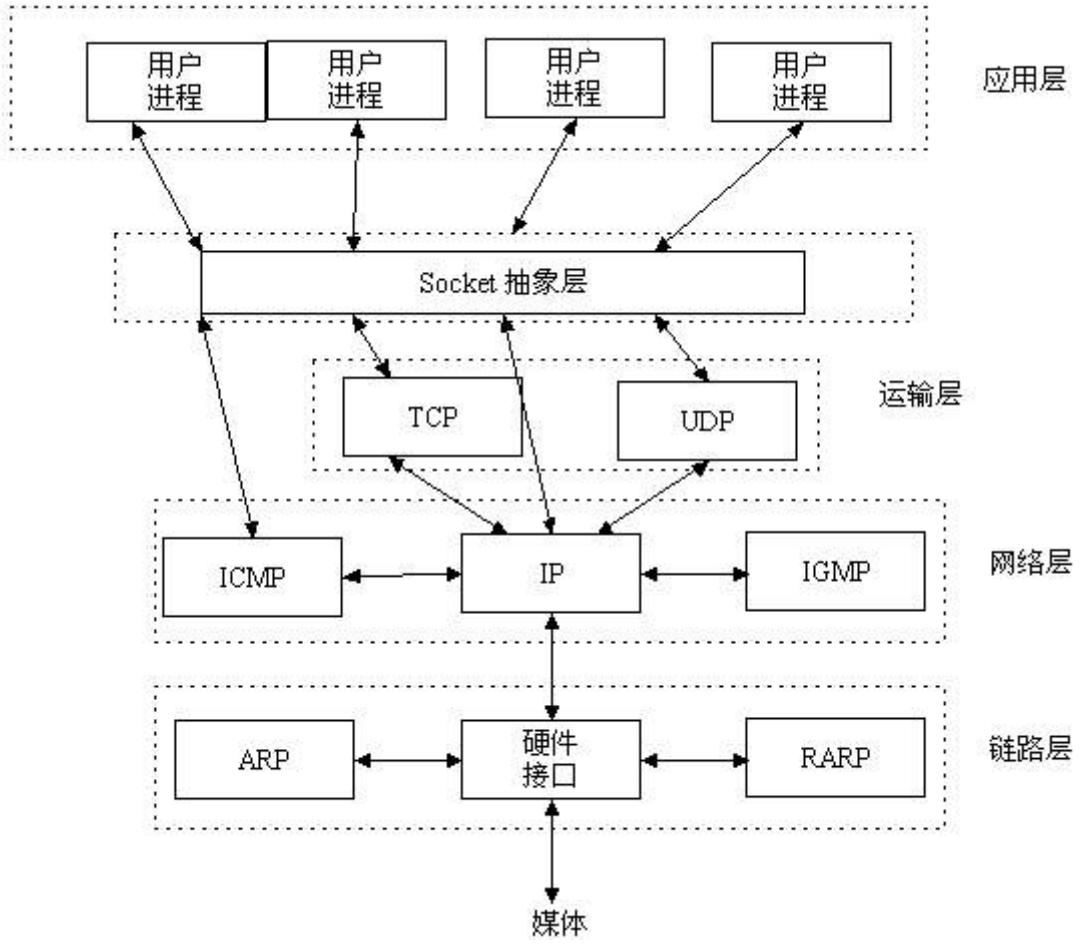
套接字通常使用TCP/IP协议栈，但也可以使用其他协议栈，例如UDP协议栈。在TCP/IP协议栈中，套接字被定义为IP地址和端口号的组合，它唯一地标识了网络上的进程。套接字可以是流套接字或数据报套接字，流套接字提供了可靠的、有序的、面向连接的数据传输服务，而数据报套接字则提供了无连接的、不可靠的数据传输服务。

套接字通信原理如下图所示：



在网络通信中，套接字一定是成对出现的。一端的发送缓冲区对应对端的接收缓冲区。我们使用同一个文件描述符索发送缓冲区和接收缓冲区。

TCP/IP协议最早在BSD UNIX上实现，为TCP/IP协议设计的应用层编程接口称为socket API。本章的主要内容是socket API，主要介绍TCP协议的函数接口，最后介绍UDP协议和UNIX Domain Socket的函数接口。



2. 网络字节序

我们已经知道，内存中的多字节数据相对于内存地址有大端和小端之分，磁盘文件中的多字节数据相对于文件中的偏移地址也有大端小端之分。网络数据流同样有大端小端之分，那么如何定义网络数据流的地址呢？发送主机通常将发送缓冲区中的数据按内存地址从低到高的顺序发出，接收主机把从网络上接到的字节依次保存在接收缓冲区中，也是按内存地址从低到高的顺序保存，因此，网络数据流的地址应这样规定：先发出的数据是低地址，后发出的数据是高地址。

TCP/IP协议规定，**网络数据流应采用大端字节序，即低地址高字节。**例如上一节的UDP段格式，地址0-1是16位的源端口号，如果这个端口号是1000（0x3e8），则地址0是0x03，地址1是0xe8，也就是先发0x03，再发0xe8，这16位在发送主机的缓冲区中也应该是低地址存0x03，高地址存0xe8。但是，如果发送主机是小端字节序的，这16位被解释成0xe803，而不是1000。因此，发送主机把1000填到发送缓冲区之前需要做字节序的转换。同样地，接收主机如果是小端字节序的，接到16位的源端口号也要做字节序的转换。如果主机是大端字节序的，发送和接收都不需要做转换。同理，32位的IP地址也要考虑网络字节序和主机字节序的问题。

大小端（Endianness）是指在计算机中存储和处理多字节数据类型（例如整数、浮点数等）时，字节的排列顺序。

在小端存储中，低位字节被存储在内存的低地址处，高位字节被存储在内存的高地址处。例如，一个16位的整数0x1234在内存中的表示方式如下：

> 地址 数据
>
> 0x1000 0x34
>
> 0x1001 0x12

在大端存储中，高位字节被存储在内存的低地址处，低位字节被存储在内存的高地址处。例如，同样是一个16位的整数0x1234在内存中的表示方式如下：

> 地址 数据
>
> 0x1000 0x12
>
> 0x1001 0x34

在网络通信中，数据传输的字节序列往往需要统一为网络字节序列（大端字节序）。因此，在进行网络通信时，需要将主机字节序列（即当前计算机的字节序列）转换为网络字节序列，或将网络字节序列转换为主机字节序列，以确保数据能够正确地被接收方解析。

通常，大部分计算机都采用小端字节序，而网络通信则通常采用大端字节序。

2.1. 在socket编程中哪些需要转，哪些不需要转

在网络通信中，为了确保不同计算机之间能够正确解析数据，需要将数据的字节序列统一为网络字节序列（大端字节序）。

> 在Socket编程中，需要将以下数据转换为大端字节序：
>
> 1. IP地址：IP地址通常采用点分十进制表示法（例如192.168.1.1），需要将其转换为32位的网络字节序列。
> 2. 端口号：端口号是16位的无符号整数，需要将其转换为16位的网络字节序列。
> 3. 整数、浮点数等多字节数据类型：这些数据类型在不同计算机之间的存储方式可能不同，因此需要将其转换为统一的字节序列。例如，一个4字节的整数，在小端字节序的计算机上存储为0x12345678，在大端字节序的计算机上存储为0x78563412，因此需要将其转换为统一的大端字节序列。

> 在Socket编程中，以下数据不需要进行字节序转换：
>
> 1. 字符串：字符串本身不涉及到多字节数据类型的存储问题，因此不需要进行字节序转换。

> 2. 字符型、布尔型、枚举型等单字节数据类型：这些数据类型只占用一个字节，不涉及到多字节数据类型的存储问题，因此也不需要进行字节序转换。

需要注意的是，对于本地计算机的Socket通信，通常也不需要进行字节序转换，因为本地计算机的字节序列（主机字节序）与网络字节序列（大端字节序）相同。只有在进行跨网络的Socket通信时，才需要进行字节序转换。

为使网络程序具有可移植性，使同样的C代码在大端和小端计算机上编译后都能正常运行，可以调用以下库函数做**网络字节序和主机字节序的转换。**

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);

```

2.2. 简单的小测试

主机向网络转：

```
#include<arpa/inet.h>
#include<stdio.h>

int main()
{
    char buf[4] = {192,168,1,2};
    int num = *(int*)buf;
    int sum = htonl(num);
    unsigned char *p = &sum;

    printf("%d.%d.%d.%d\n",*p,*(p+1),*(p+2),*(p+3));

    unsigned short a = 0x1234;
    unsigned short b = htons(a);

    printf("%x\n",b);
    return 0;
}

```

运行结果：

```
2.1.168.192
3412
```

我为大家简单的画一幅图，便于大家理解此转化过程：

网络向主机转化：

```c
int main()
{
    unsigned char buf[4] = {2,1,168,192};
    int num = *(int*)buf;
    int sum = ntohl(num);
    unsigned char * p =&sum;
    printf("%d.%d.%d.%d\n",*p,*(p+1),*(p+2),*(p+3));

    unsigned short a = 0x3412;
    unsigned short b = ntohs(a);
    printf("%x\n",b);
    return 0;
}

```

运行结果：

```
192.168.1.2
1234
```

2.3. 点分十进制转大端

```
 #include <arpa/inet.h>

      int inet_pton(int af, const char *restrict src, void
*restrict dst);
      const char *inet_ntop(int af, const void *restrict src,
                            char *restrict dst, socklen_t
size);
```

```
int main()
{
    char *src = "192.168.1.2";
    unsigned int num = 0;
    int res = inet_pton(AF_INET,src,&num);
    unsigned char * p=(unsigned char *)&num;
    printf("%d.%d.%d.%d\n",*p,*(p+1),*(p+2),*(p+3));

    char dst[16]="";
    printf("%s\n",inet_ntop(AF_INET,&num,dst,16));
    printf(dst);
    return 0;
}

```

```
192.168.1.2
192.168.1.2
192.168.1.2
```

3. sockaddr数据结构

网络通信要解决的三个问题：

1. 协议
2. ip
3. 端口

因此设计者将其封装成了一个结构体：

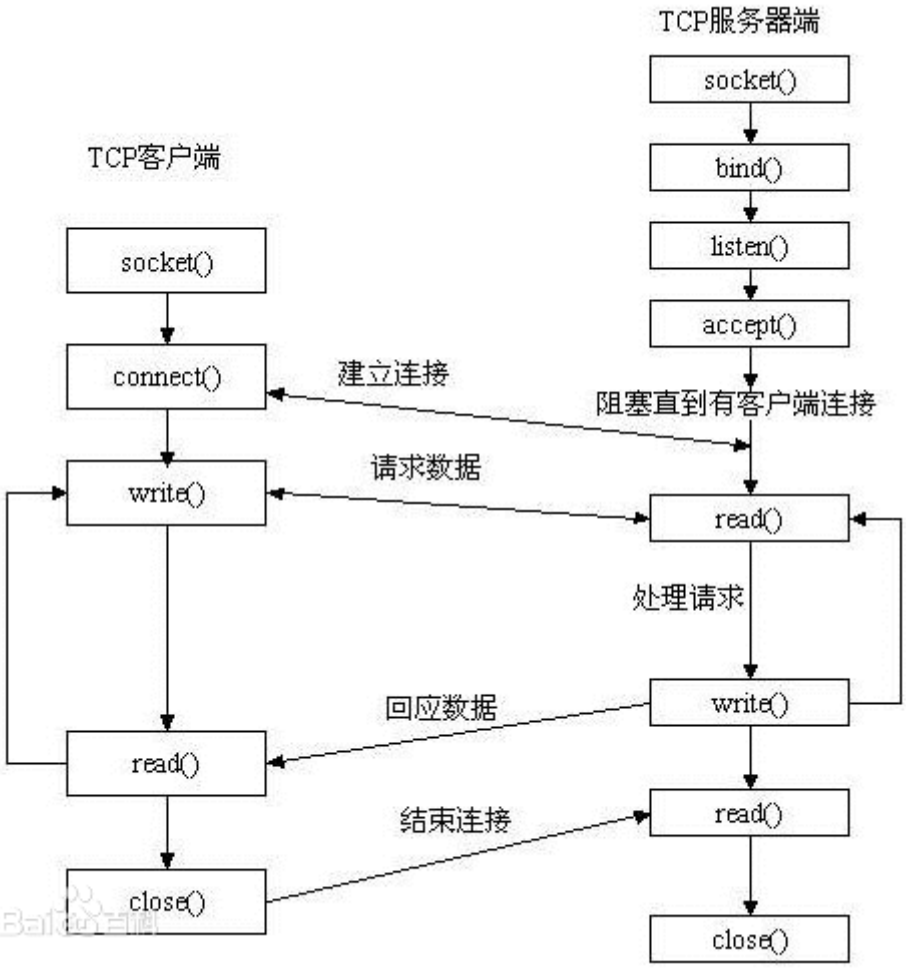
```
# ipv4套接字结构体
 struct sockaddr_in {
               sa_family_t    sin_family; /* address family: AF_INET */
               in_port_t      sin_port;   /* port in network byte order */
               struct in_addr sin_addr;   /* internet address */
           };

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


```

```
# ipv6套接字结构体（了解）
struct sockaddr_in6 {
    unsigned short int sin6_family;          /* AF_INET6 */
    __be16 sin6_port;                   /* Transport layer port # */
    __be32 sin6_flowinfo;               /* IPv6 flow information */
    struct in6_addr sin6_addr;          /* IPv6 address */
    __u32 sin6_scope_id;                /* scope id (new in RFC2553) */
};
struct in6_addr {
    union {
        __u8 u6_addr8[16];
        __be16 u6_addr16[8];
        __be32 u6_addr32[4];
    } in6_u;
    #define s6_addr         in6_u.u6_addr8
    #define s6_addr16       in6_u.u6_addr16
    #define s6_addr32       in6_u.u6_addr32
};
```

为了兼容ipv4和ipv6，设计者设计了通用套接字结构体：

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

```

4. socket 套接字通信流程



套接字的使用通常涉及以下步骤：

1. 创建套接字：通过调用系统的套接字函数创建一个新的套接字对象。
2. 绑定套接字：为套接字分配本地IP地址和端口号，使其能够在网络上被其他进程访问。
3. 监听套接字：对于流套接字，需要调用监听函数将其设置为等待连接的状态。
4. 接受连接：对于流套接字，调用接受函数来接受来自客户端的连接请求。
5. 发送和接收数据：通过调用发送和接收函数，进行数据传输。
6. 关闭套接字：当通信结束时，需要调用关闭函数来释放套接字资源。

4.1. 创建套接字

```
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
domain:
    AF_INET 这是大多数用来产生socket的协议，使用TCP或UDP来传输，用IPv4的地址
    AF_INET6 与上面类似，不过是来用IPv6的地址
    AF_UNIX 本地协议，使用在Unix和Linux系统上，一般都是当客户端和服务器在同一台及其上的时候使用
type:
    SOCK_STREAM 这个协议是按照顺序的、可靠的、数据完整的基于字节流的连接。这是一个使用最多的socket类型，这个socket是使用TCP来进行传输。
    SOCK_DGRAM 这个协议是无连接的、固定长度的传输调用。该协议是不可靠的，使用UDP来进行它的连接。
    SOCK_SEQPACKET该协议是双线路的、可靠的连接，发送固定长度的数据包进行传输。必须把这个包完整的接受才能进行读取。
    SOCK_RAW socket类型提供单一的网络访问，这个socket类型使用ICMP公共协议。（ping、traceroute使用该协议）
    SOCK_RDM 这个类型是很少使用的，在大部分的操作系统上没有实现，它是提供给数据链路层使用，不保证数据包的顺序
protocol:
传0 表示使用默认协议。
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.连接服务器

```
#include <sys/types.h>                  /* See NOTES */
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
sockdf:
    socket文件描述符
addr:
    传入参数，指定服务器端地址信息，含IP地址和端口号
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的参数形式一致，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。connect()成功返回0，出错返回-1。

客户端代码如下：

```c
#include<sys/socket.h>
#include<stdio.h>
#include<arpa/inet.h>
#include<string.h>
#include<unistd.h>
int main()
{
    int sfd = socket(AF_INET,SOCK_STREAM,0);
    if (sfd==-1){
        perror("socket:");
        return 0;
    }
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(8090);
    inet_pton(AF_INET,"10.97.23.77",&addr.sin_addr.s_addr);
    connect(sfd,(struct sockaddr *)&addr,sizeof(addr));
    char buf[1024]="";
    while(1){
        int n = read(STDIN_FILENO,buf,sizeof(buf));
        write(sfd,buf,n);
        n=read(sfd,buf,sizeof(buf));
        write(STDOUT_FILENO,buf,n);
        printf("\n");
    }
    close(sfd);
    return 0;
}
```

4.3. bind

```
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t
addrlen);
sockfd:
    socket文件描述符
addr:
    构造出IP地址加端口号
addrlen:
    sizeof(addr)长度
返回值:
    成功返回0，失败返回-1，设置errno

```

服务器程序所监听的网络地址和端口号通常是固定不变的，客户端程序得知服务器程序的地址和端口号后就可以向服务器发起连接，因此服务器需要调用bind绑定一个固定的网络地址和端口号。

bind()的作用是将参数sockfd和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。前面讲过，struct sockaddr *是一个通用指针类型，addr参数实际上可以接受多种协议的sockaddr结构体，而它们的长度各不相同，所以需要第三个参数addrlen指定结构体的长度。如：

```
struct sockaddr_in servaddr;
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(6666);
```

首先将整个结构体清零，然后设置地址类型为AF_INET，**网络地址为**INADDR_ANY，**这个宏表示本地的任意IP**地址，因为服务器可能有多个网卡，每个网卡也可能绑定多个IP地址，这样设置可以在所有的IP地址上监听，直到与某个客户端建立了连接时才确定下来到底用哪个IP地址，端口号为6666。

4.4.Listen

```
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
int listen(int sockfd, int backlog);
sockfd:
    socket文件描述符
backlog:
    排队建立3次握手队列和刚刚建立3次握手队列的链接数和

```

4.5.accept

```
#include <sys/types.h>      /* See NOTES */
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t
*addrlen);
sockdf:
    socket文件描述符
addr:
    传出参数，返回链接客户端地址信息，含IP地址和端口号
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
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

```

三方握手完成后，服务器调用accept()接受连接，如果服务器调用accept()时还没有客户端的连接请求，就阻塞等待直到有客户端连接上来。addr是一个传出参数，accept()返回时传出客户端的地址和端口号。addrlen参数是一个传入传出参数（value-result argument），传入的是调用者提供的缓冲区addr的长度以避免缓冲区溢出问题，传出的是客户端地址结构体的实际长度（有可能没有占满调用者提供的缓冲区）。如果给addr参数传NULL，表示不关心客户端的地址。

4.6.服务器代码

下面是一个简单的服务端代码书写流程：

1. 创建Socket对象：使用socket()函数创建一个Socket对象，指定其类型（TCP或UDP）和协议（IPv4或IPv6）等参数。例如：
2. ```
   perlCopy code
   int serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
   ```
3. 绑定Socket对象：使用bind()函数将Socket对象与特定的IP地址和端口号绑定。例如：
   ```

4.
```
scssCopy codestruct sockaddr_in serverAddress;
memset(&serverAddress, 0, sizeof(serverAddress));
serverAddress.sin_family = AF_INET;
serverAddress.sin_port = htons(port);
serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);

bind(serverSocket, (struct sockaddr *)&serverAddress,
sizeof(serverAddress));
```

5. 监听连接请求：使用listen()函数开始监听连接请求，并指定最大连接数。例如：

6.
```
scssCopy code
listen(serverSocket, 5);
```

7. 接受连接请求：使用accept()函数接受来自客户端的连接请求，并返回一个新的Socket对象，用于与客户端进行通信。例如：

8.
```
cCopy codestruct sockaddr_in clientAddress;
socklen_t clientAddressLen = sizeof(clientAddress);
int clientSocket = accept(serverSocket, (struct
sockaddr *)&clientAddress, &clientAddressLen);
```

9. 与客户端通信：使用send()和recv()函数向连接的对方发送数据并接收数据。例如：

10.
```
scssCopy codechar buffer[1024];
memset(buffer, 0, sizeof(buffer));
recv(clientSocket, buffer, sizeof(buffer), 0);

char message[] = "Hello, Client!";
send(clientSocket, message, sizeof(message), 0);
```

11. 关闭Socket对象：使用close()函数关闭Socket对象。例如：

12.
```
scssCopy codeclose(clientSocket);
close(serverSocket);
```

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

```c
int main(int argc,char* argv[])
{
    // 创建服务器套接字
    int server_socket = socket(AF_INET,SOCK_STREAM,0);
    if (server_socket == -1 ){
        perror("create_server_socket:");
        return 0;
    }
    // 绑定服务器
    struct sockaddr_in  serverAddress;
    memset(&serverAddress,0,sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(8080);
    char *address = "192.168.187.131";
    inet_pton(AF_INET,address,&serverAddress.sin_addr.s_addr);
    // addr.sin_addr.s_addr = INADDR_ANY //绑定的是通配地址
    if (bind(server_socket,(struct sockaddr*)&serverAddress,sizeof(serverAddress))==-1){
        perror("bind error:");
        return 0;
    }

    //监听
    listen(server_socket,128);
    struct sockaddr_in clientAddr;
    socklen_t  len=sizeof(clientAddr);
    int client_socket = accept(server_socket,(struct sockaddr*)&clientAddr,&len);
    char ip[16]="";
    printf("客户端ip:%s,端口号:%d\n",inet_ntop(AF_INET,&clientAddr.sin_addr.s_addr,ip,16),ntohs(clientAddr.sin_port));
    char buf[1024]="";
    while(1){
        //
        bzero(buf,sizeof(buf));
        char buf[1024]="";
        int n = read(STDIN_FILENO,buf,sizeof(buf));
        write(client_socket,buf,n);
        read(client_socket,buf,sizeof(buf));
        printf("%s\n",buf);
    }
    close(client_socket);
    close(server_socket);
    //ctrl+c终止后：端口过两分钟释放
```

```
    return 0;
}

```

5. 封装代码

为了方便我们书写代码，我们基本函数进行了封装：

5.1. warp.h

```h
#ifndef __WRAP_H_
#define __WRAP_H_
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <strings.h>

void perr_exit(const char *s);
int Accept(int fd, struct sockaddr *sa, socklen_t *salenptr);
int Bind(int fd, const struct sockaddr *sa, socklen_t salen);
int Connect(int fd, const struct sockaddr *sa, socklen_t
salen);
int Listen(int fd, int backlog);
int Socket(int family, int type, int protocol);
ssize_t Read(int fd, void *ptr, size_t nbytes);
ssize_t Write(int fd, const void *ptr, size_t nbytes);
int Close(int fd);
ssize_t Readn(int fd, void *vptr, size_t n);
ssize_t Writen(int fd, const void *vptr, size_t n);
ssize_t my_read(int fd, char *ptr);
ssize_t Readline(int fd, void *vptr, size_t maxlen);
int tcp4bind(short port,const char *IP);
#endif

```

5.2. warp.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
```

```c
#include <errno.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <strings.h>

void perr_exit(const char *s)
{
    perror(s);
    exit(-1);
}

int Accept(int fd, struct sockaddr *sa, socklen_t *salenptr)
{
    int n;

again:
    if ((n = accept(fd, sa, salenptr)) < 0) {
        if ((errno == ECONNABORTED) || (errno == EINTR))//如果是被信号中断和软件层次中断,不能退出
            goto again;
        else
            perr_exit("accept error");
    }
    return n;
}

int Bind(int fd, const struct sockaddr *sa, socklen_t salen)
{
    int n;

    if ((n = bind(fd, sa, salen)) < 0)
        perr_exit("bind error");

    return n;
}

int Connect(int fd, const struct sockaddr *sa, socklen_t salen)
{
    int n;

    if ((n = connect(fd, sa, salen)) < 0)
        perr_exit("connect error");

```

```
47      return n;
48  }
49
50  int Listen(int fd, int backlog)
51  {
52      int n;
53
54      if ((n = listen(fd, backlog)) < 0)
55          perr_exit("listen error");
56
57      return n;
58  }
59
60  int Socket(int family, int type, int protocol)
61  {
62      int n;
63
64      if ((n = socket(family, type, protocol)) < 0)
65          perr_exit("socket error");
66
67      return n;
68  }
69
70  ssize_t Read(int fd, void *ptr, size_t nbytes)
71  {
72      ssize_t n;
73
74  again:
75      if ( (n = read(fd, ptr, nbytes)) == -1) {
76          if (errno == EINTR)//如果是被信号中断,不应该退出
77              goto again;
78          else
79              return -1;
80      }
81      return n;
82  }
83
84  ssize_t Write(int fd, const void *ptr, size_t nbytes)
85  {
86      ssize_t n;
87
88  again:
89      if ( (n = write(fd, ptr, nbytes)) == -1) {
90          if (errno == EINTR)
91              goto again;
```

```
        else
            return -1;
    }
    return n;
}

int Close(int fd)
{
    int n;
    if ((n = close(fd)) == -1)
        perr_exit("close error");

    return n;
}

/*参三: 应该读取固定的字节数数据*/
ssize_t Readn(int fd, void *vptr, size_t n)
{
    size_t  nleft;              //usigned int 剩余未读取的字节数
    ssize_t nread;              //int 实际读到的字节数
    char   *ptr;

    ptr = vptr;
    nleft = n;

    while (nleft > 0) {
        if ((nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0;
            else
                return -1;
        } else if (nread == 0)
            break;

        nleft -= nread;
        ptr += nread;
    }
    return n - nleft;
}
/*:固定的字节数数据*/
ssize_t Writen(int fd, const void *vptr, size_t n)
{
    size_t nleft;
    ssize_t nwritten;
    const char *ptr;
```

```
137
138     ptr = vptr;
139     nleft = n;
140     while (nleft > 0) {
141         if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
142             if (nwritten < 0 && errno == EINTR)
143                 nwritten = 0;
144             else
145                 return -1;
146         }
147
148         nleft -= nwritten;
149         ptr += nwritten;
150     }
151     return n;
152 }
153
154 static ssize_t my_read(int fd, char *ptr)
155 {
156     static int read_cnt;
157     static char *read_ptr;
158     static char read_buf[100];
159
160     if (read_cnt <= 0) {
161 again:
162         if ( (read_cnt = read(fd, read_buf,
    sizeof(read_buf))) < 0) {
163             if (errno == EINTR)
164                 goto again;
165             return -1;
166         } else if (read_cnt == 0)
167             return 0;
168         read_ptr = read_buf;
169     }
170     read_cnt--;
171     *ptr = *read_ptr++;
172
173     return 1;
174 }
175
176 ssize_t Readline(int fd, void *vptr, size_t maxlen)
177 {
178     ssize_t n, rc;
179     char    c, *ptr;
180
```

```
    ptr = vptr;
    for (n = 1; n < maxlen; n++) {
        if ( (rc = my_read(fd, &c)) == 1) {
            *ptr++ = c;
            if (c  == '\n')
                break;
        } else if (rc == 0) {
            *ptr = 0;
            return n - 1;
        } else
            return -1;
    }
    *ptr  = 0;

    return n;
}

int tcp4bind(short port,const char *IP)
{
    struct sockaddr_in serv_addr;
    int lfd = Socket(AF_INET,SOCK_STREAM,0);
    bzero(&serv_addr,sizeof(serv_addr));
    if(IP == NULL){
        //如果这样使用 0.0.0.0,任意ip将可以连接
        serv_addr.sin_addr.s_addr = INADDR_ANY;
    }else{
        if(inet_pton(AF_INET,IP,&serv_addr.sin_addr.s_addr)
<= 0){
            perror(IP);//转换失败
            exit(1);
        }
    }
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port   = htons(port);
   // int opt = 1;
    //setsockopt(lfd, SOL_SOCKET, SO_REUSEADDR, &opt,
sizeof(opt));

    Bind(lfd,(struct sockaddr
*)&serv_addr,sizeof(serv_addr));
    return lfd;
}


```

6. 多进程实现并发流程

```c
#include <stdio.h>
#include <sys/socket.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include "wrap.h"
void free_process(int sig)
{
    pid_t pid;
    while(1)
    {
        pid = waitpid(-1,NULL,WNOHANG);
        if(pid <=0 )//小于0 子进程全部退出了 =0没有进程没有退出
        {
            break;
        }
        else
        {
            printf("child pid =%d\n",pid);
        }
    }



}
int main(int argc, char *argv[])
{
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set,SIGCHLD);
    sigprocmask(SIG_BLOCK,&set,NULL);
    //创建套接字,绑定
    int lfd = tcp4bind(8000,NULL);
    //监
    Listen(lfd,128);
    //提取
    //回射
    struct sockaddr_in cliaddr;
    socklen_t len = sizeof(cliaddr);
    while(1)
    {
        char ip[16]="";
```

```
43          //提取连接,
44          int cfd = Accept(lfd,(struct sockaddr
*)&cliaddr,&len);
45          printf("new client ip=%s
port=%d\n",inet_ntop(AF_INET,&cliaddr.sin_addr.s_addr,ip,16),
46                  ntohs(cliaddr.sin_port));
47          //fork创建子进程
48          pid_t pid;
49          pid = fork();
50          if(pid < 0)
51          {
52              perror("");
53              exit(0);
54          }
55          else if(pid == 0)//子进程
56          {
57              //关闭lfd
58              close(lfd);
59              while(1)
60              {
61              char buf[1024]="";
62
63              int n = read(cfd,buf,sizeof(buf));
64              if(n < 0)
65              {
66                  perror("");
67                  close(cfd);
68                  exit(0);
69              }
70              else if(n == 0)//对方关闭j
71              {
72                  printf("client close\n");
73                  close(cfd);
74                  exit(0);
75
76              }
77              else
78              {
79                  printf("%s\n",buf);
80                  write(cfd,buf,n);
81              //  exit(0);
82              }
83              }
84
85          }
```

```
        else//父进程
        {
            close(cfd);
            //回收
            //注册信号回调
            struct sigaction act;
            act.sa_flags =0;
            act.sa_handler = free_process;
            sigemptyset(&act.sa_mask);
            sigaction(SIGCHLD,&act,NULL);
            sigprocmask(SIG_UNBLOCK,&set,NULL);

        }
    }
    //关闭



    return 0;
}


```

7. 线程版本服务器

```
#include <stdio.h>
#include <pthread.h>
#include "wrap.h"
typedef struct c_info
{
    int cfd;
    struct sockaddr_in cliaddr;

}CINFO;
void* client_fun(void *arg);
int main(int argc, char *argv[])
{
    if(argc < 2)
    {
        printf("argc < 2???   \n ./a.out 8000 \n");
        return 0;
    }

```

```
19      pthread_attr_t attr;
20      pthread_attr_init(&attr);
21
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
22      short port = atoi(argv[1]);
23      int lfd = tcp4bind(port,NULL);//创建套接字 绑定
24      Listen(lfd,128);
25      struct sockaddr_in cliaddr;
26      socklen_t len = sizeof(cliaddr);
27      CINFO *info;
28      while(1)
29      {
30          int cfd = Accept(lfd,(struct sockaddr
*)&cliaddr,&len);
31          char ip[16]="";
32          pthread_t pthid;
33          info = malloc(sizeof(CINFO));//体会这块的妙处
34          info->cfd = cfd;
35          info->cliaddr= cliaddr;
36          pthread_create(&pthid,&attr,client_fun,info);
37
38
39
40      }
41
42      return 0;
43  }
44
45  void* client_fun(void *arg)
46  {
47      CINFO *info = (CINFO *)arg;
48      char ip[16]="";
49
50      printf("new client ip=%s port=%d\n",inet_ntop(AF_INET,&
(info->cliaddr.sin_addr.s_addr),ip,16),
51          ntohs(info->cliaddr.sin_port));
52      while(1)
53      {
54          char buf[1024]="";
55          int count=0;
56          count = read(info->cfd,buf,sizeof(buf));
57          if(count < 0)
58          {
59              perror("");
60              break;
```

```
61
62          }
63          else if(count == 0)
64          {
65              printf("client close\n");
66              break;
67          }
68          else
69          {
70              printf("%s\n", buf);
71              write(info->cfd,buf,count);
72
73          }
74
75
76      }
77      close(info->cfd);
78      free(info);
79  }
80
81
```

二、高并发服务器开发

高并发服务器是指能够处理大量并发连接请求的服务器，通常用于网络游戏、社交网络、在线视频等需要处理大量并发请求的应用场景。

要实现高并发服务器，通常需要考虑以下几个方面：

1. **使用多线程或多进程模型**：为了充分利用多核CPU资源，可以采用多线程或多进程模型，将每个连接请求分配给一个线程或进程进行处理。
2. **使用异步I/O模型**：异步I/O模型将网络I/O操作交给操作系统异步处理，避免了轮询等CPU占用操作，提高了服务器性能。
3. **采用高效的多路复用技术**：多路复用技术可以监控多个Socket对象的I/O状态，从而实现高效的I/O处理，提高服务器性能。
4. **采用高性能的网络框架和库**：现在有很多高性能的网络框架和库，如Boost.Asio、libevent、muduo等，可以帮助开发人员快速实现高性能服务器。
5. **避免阻塞和死锁**：在并发环境下，线程间的竞争和资源共享可能会引发阻塞和死锁问题，需要仔细设计程序逻辑，采用合适的同步机制来避免这些问题。

综上所述，实现高并发服务器需要综合考虑多个因素，采用合适的技术和框架，仔细设计程序逻辑，才能实现高性能、高可靠性的服务器

1. 名词解释

1.0. I/O模型

I/O是指输入/输出（Input/Output）操作，它是计算机系统中的一种基本操作，用于数据在内存和外部设备（如磁盘、网络等）之间的传输。在计算机程序中，I/O操作通常用来读取或写入文件、网络通信、用户输入等。

在I/O模型中，I/O操作通常是指从外部设备读取或写入数据的过程。例如，在网络通信中，当服务器接收到一个请求时，会进行读取操作，将请求数据从网络中读取到内存中；当服务器需要向客户端发送响应时，会进行写入操作，将响应数据从内存中写入到网络中。I/O模型根据不同的方式来管理I/O操作的执行和完成，可以提高应用程序的性能和可伸缩性。

I/O模型是指操作系统中用来实现I/O操作的一些标准模型或接口。常见的I/O模型有以下几种：

1. **阻塞I/O模型（Blocking I/O）**：当应用程序发起一个I/O操作时，会一直等待直到操作完成，期间线程或进程会被阻塞，不能执行其他任务。如果数据没有准备好或网络不可用，应用程序会一直阻塞在该I/O操作上，直到超时或其他错误发生。

2. **非阻塞I/O模型（Non-blocking I/O）**：当应用程序发起一个I/O操作时，会立即返回一个错误码或0，如果数据没有准备好或网络不可用，应用程序可以进行其他任务，之后再次查询I/O操作状态。这种方式需要应用程序轮询I/O操作状态，效率较低。
3. **多路复用I/O模型（Multiplexing I/O）：**通过使用select、poll、epoll等系统调用来同时监视多个socket的I/O事件，以实现高效的I/O操作和高并发性能。可以同时监视多个socket，并在有I/O事件发生时及时处理，避免了单线程阻塞等待的问题。
4. **异步I/O模型（Asynchronous I/O）**：当应用程序发起一个I/O操作时，不需要等待操作完成，而是可以继续处理其他任务。当I/O操作完成后，系统会通知应用程序，应用程序再进行相应的处理。这种方式可以提高应用程序的并发性能和响应速度，但是需要更多的代码来处理I/O事件和错误情况。

不同的I/O模型有不同的适用场景和优缺点，需要根据具体情况选择合适的模型。

1.1. 阻塞等待

阻塞等待是指程序在执行某个操作时，如果遇到需要等待某个事件发生才能继续执行的情况，会暂时停止执行，并将CPU时间片释放给其他进程或线程使用，直到等待的事件发生后再继续执行。这个过程中程序被阻塞了，等待的时间也是无法控制的。

在网络编程中，阻塞等待经常用于等待网络I/O操作完成。当程序调用了一个阻塞式的网络I/O函数时，如果当前的I/O操作无法立即完成，程序会阻塞等待，直到I/O操作完成后才会继续执行下一条语句。

阻塞等待虽然简单易用，但是也有一些缺点。首先，阻塞等待会占用大量的CPU时间，降低系统性能。其次，由于无法控制等待时间，如果等待时间过长，会造成用户体验不佳。因此，在实际开发中，为了提高系统性能和用户体验，通常会采用异步I/O或者多路复用等技术来替代阻塞等待。

1.2. 非阻塞忙轮询

非阻塞忙轮询是指程序在执行某个操作时，如果遇到需要等待某个事件发生才能继续执行的情况，程序会使用一个循环来不断地轮询等待的事件是否发生，而不会暂停执行，并且不会将CPU时间片释放给其他进程或线程使用。在每次循环中，程序会检查事件是否已经发生，如果发生了则处理事件，如果还没有发生则继续循环等待。

在网络编程中，非阻塞忙轮询经常用于实现非阻塞I/O操作。当程序调用了一个非阻塞式的网络I/O函数时，如果当前的I/O操作无法立即完成，程序会立即返回，而不是一直等待，程序可以继续执行其他任务，但是为了及时处理I/O事件，程序会不断地轮询I/O操作的状态，直到操作完成。

非阻塞忙轮询的优点是实现简单，容易理解和掌握。但是其缺点也很明显，即会浪费大量的CPU时间和资源，造成系统性能和能耗的浪费。因此，在实际开发中，通常会采用异步I/O或者多路复用等技术来替代非阻塞忙轮询，以提高系统性能和节约系统资源。

1.3.异步I/O

异步I/O是一种高效的I/O操作模型，可以在不阻塞主线程的情况下完成I/O操作。在异步I/O模型中，当应用程序发起一个I/O操作后，操作系统会立即返回，并在后台执行这个I/O操作，不会阻塞应用程序主线程。当I/O操作完成时，操作系统会通知应用程序，应用程序可以回调指定的函数来处理I/O数据。

异步I/O模型的优点在于可以实现高并发、低延迟的网络通信。由于异步I/O操作不会阻塞主线程，因此可以处理大量的并发请求，而且可以避免线程上下文切换和内核态和用户态之间的切换，提高系统性能和效率。同时，异步I/O模型也适用于高延迟网络通信，可以在I/O操作完成之前处理其他任务，提高系统的响应速度和用户体验。

在异步I/O模型中，需要使用特定的API函数来发起异步I/O操作，如Windows系统中的IOCP（I/O Completion Ports）、Linux系统中的epoll、kqueue等。异步I/O模型需要使用回调函数来处理I/O完成事件，需要编写一定的异步编程代码，但是可以提供更高的性能和更好的用户体验。

1.4. 非阻塞I/O

非阻塞I/O是一种I/O操作模型，与传统的阻塞I/O模型不同，非阻塞I/O可以在没有数据可读或可写时立即返回，并不会阻塞线程或进程的执行。非阻塞I/O常常使用非阻塞socket来实现。

在非阻塞I/O模型中，当应用程序发起一个I/O操作后，操作系统会立即返回，不会阻塞主线程。如果数据还没有准备好，应用程序会得到一个错误码，但是可以继续执行其他任务，等到数据准备好之后再进行I/O操作。如果数据已经准备好，应用程序可以读取或写入数据，并进行相应的处理。

与阻塞I/O模型相比，非阻塞I/O可以实现更高的并发性和更快的响应速度，但是也需要更多的代码来管理I/O状态和处理错误情况。通常情况下，非阻塞I/O会和其他技术一起使用，如多路复用和线程池等，来提高系统的性能和稳定性。

需要注意的是，非阻塞I/O并不是异步I/O，它们是不同的I/O操作模型。在异步I/O模型中，操作系统会在后台执行I/O操作，不会阻塞主线程，并在I/O操作完成后通知应用程序。而在非阻塞I/O模型中，应用程序需要主动轮询I/O操作的状态，并在数据准备好时进行I/O操作。

1.5. 阻塞I/O

阻塞I/O是一种I/O操作模型，常常使用阻塞socket来实现。在阻塞I/O模型中，当应用程序发起一个I/O操作时，它会一直等待直到操作完成，期间线程或进程会被阻塞，不能执行其他任务。如果数据没有准备好或网络不可用，应用程序会一直阻塞在该I/O操作上，直到超时或其他错误发生。

阻塞I/O模型的优点是编程简单易懂，适用于单线程或多线程编程环境。但是在高并发或高负载的情况下，阻塞I/O会导致性能下降，因为它不能同时处理多个I/O请求，需要等待当前I/O请求完成后才能处理下一个请求。

为了解决阻塞I/O的性能问题，人们开发了其他的I/O操作模型，如非阻塞I/O、多路复用I/O和异步I/O等。这些模型可以同时处理多个I/O请求，并且不会阻塞主线程的执行。

1.6.多路复用I/O

多路复用I/O是一种I/O操作模型，通过使用select、poll、epoll等系统调用来同时监视多个socket的I/O事件，以实现高效的I/O操作和高并发性能。

在多路复用I/O模型中，应用程序首先通过select、poll、epoll等系统调用，将需要监听的socket加入到一个监视集合中。**当一个socket有I/O事件发生时，系统调用会立即返回，并告知应用程序有哪些socket有事件发生。**应用程序可以通过事件类型来确定是读事件还是写事件，并进行相应的处理。这种方式可以同时监视多个socket，并在有I/O事件发生时及时处理，避免了单线程阻塞等待的问题。

多路复用I/O模型可以提高应用程序的性能和并发性能，但也需要更多的代码来处理I/O事件和错误情况，因此比阻塞I/O模型要复杂一些。另外，由于多路复用I/O模型使用了系统调用，因此会产生一定的开销，但这个开销通常可以通过其他优化技术来降低。

需要注意的是，多路复用I/O和非阻塞I/O、异步I/O等不同，它们是不同的I/O操作模型。在非阻塞I/O和异步I/O中，应用程序不会被阻塞，可以处理其他任务，而在多路复用I/O中，应用程序需要等待系统调用返回，并轮询事件类型来处理I/O事件。

2. select、poll、epoll

多路IO转接服务器也叫做多任务IO服务器。该类服务器实现的主旨思想是，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。select、poll和epoll是常见的多路复用I/O机制，用于同时监视多个文件描述符，等待其中任意一个或多个文件描述符上有I/O事件发生。

2.1. select

select是最早的多路复用I/O机制，适用于同时监视少量的文件描述符。它的原理是将多个文件描述符打包成一个位图，然后通过系统调用实现监视，当有I/O事件发生时，select返回并将该文件描述符的位设置为1，否则设置为0，然后应用程序可以通过读取位图的方式获取有I/O事件发生的文件描述符。

select的缺点是效率低，时间复杂度为O(n)，并且存在文件描述符数量的限制，通常在单线程服务器或客户端程序中使用。

2.2.poll

poll是select的改进版，与select类似，也是通过将多个文件描述符打包成一个数组来实现监视。不同的是，**poll不再受到文件描述符数量的限制**，并且能够更好地处理大量的文件描述符，因为poll将文件描述符数组存储在内核中，而不是用户空间中，减少了内核态和用户态之间的数据拷贝。

poll的缺点是效率依然不高，时间复杂度为O(n)，并且仍然需要轮询文件描述符数组来检查有无I/O事件发生。

2.3.epoll

epoll是最新的多路复用I/O机制，适用于同时监视大量的文件描述符。它通过一个文件描述符（epoll file descriptor）来管理所有需要监视的文件描述符，当有I/O事件发生时，epoll会立即通知应用程序，不再需要像select和poll那样轮询文件描述符数组。

epoll的优点是效率高，时间复杂度为O(1)，可以同时监视大量的文件描述符，并且能够处理边缘触发模式，支持水平触发和边缘触发两种模式。它是目前大多数高性能服务器的首选，比如Nginx、Redis等。

以上是对select、poll和epoll的简单介绍，它们是Linux系统中常用的I/O多路复用机制，各自有不同的特点和适用场景。

3. select

3.1.文件描述符表

文件描述符表是由操作系统内核维护的，其具体实现方式可能因操作系统而异，但通常会采用数组(位图)的形式来管理进程的文件和I/O设备。下面以Linux操作系统为例，简单介绍文件描述符表的实现方式。

在Linux系统中，**文件描述符表是进程控制块（Process Control Block，PCB）的一部分**，由内核维护。当进程创建时，内核会分配一个PCB，其中包括了该进程的所有信息，包括文件描述符表。文件描述符表是一个数组，定义在头文件<fcntl.h>中，它的默认大小是1024个元素。

当进程调用open()、socket()、pipe()等函数打开一个文件或I/O设备时，内核会分配一个未被占用的文件描述符，并将其作为返回值返回给进程。**文件描述符是一个整数，用于唯一标识打 开的文件或设备。**内核会在文件描述符表中为该文件或设备分配一个元素，并将文件描述符与该元素关联起来。

当进程使用文件描述符进行读写或其他操作时，内核会根据文件描述符找到对应的文件或设备，并进行相应的操作。如果进程关闭了文件或设备，内核会释放对应的文件描述符，并将文件描述符表中对应的元素设置为未被占用状态，以便后续进程再次使用。

需要注意的是，文件描述符表是进程私有的，不同进程之间的文件描述符表是相互独立的。此外，不同的操作系统可能会有不同的实现方式，但基本的概念和作用是类似的。

在Linux系统中，文件描述符表的定义通常包含在<fcntl.h>头文件中，具体定义方式如下：

```
typedef int fd_t;
typedef struct {
    fd_set fds_bits[FD_SETSIZE / NFDBITS];
} fd_set;
```

其中，fd_t是文件描述符类型的定义，通常是int类型。fd_set是文件描述符集合类型的定义，用于管理一组文件描述符。它是一个结构体类型，包含一个长度为FD_SETSIZE的位数组，每个位表示一个文件描述符的状态（被占用或空闲）。

需要注意的是，这里的FD_SETSIZE常量定义了文件描述符集合的最大大小，通常是1024或更大。在实际使用中，可以根据需要调整该值。

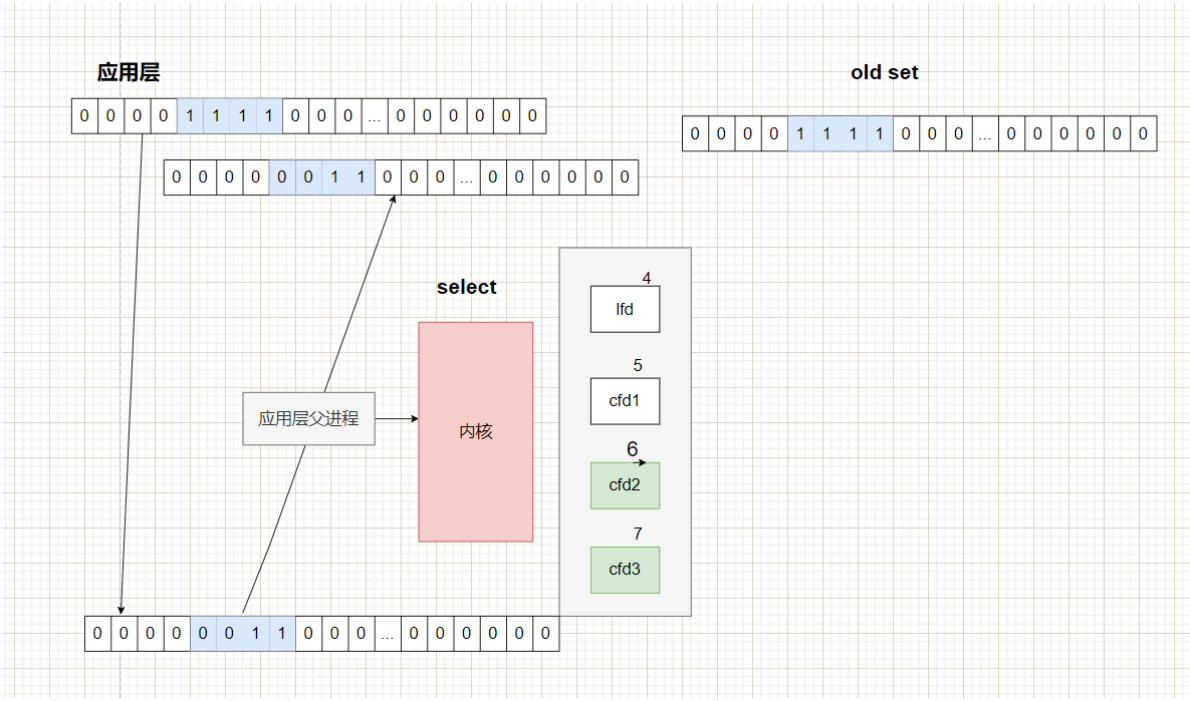
3.2. API

```
#include <sys/select.h>
/* According to earlier standards */
#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加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
    readfds:      监控有读数据到达文件描述符集合，传入传出参数
    writefds:     监控写数据到达文件描述符集合，传入传出参数
    exceptfds:    监控异常发生达文件描述符集合,如带外数据到达异常，传入传出参数
    timeout:      定时阻塞监控时间，3种情况
                  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
    struct timeval {
        long tv_sec; /* seconds */秒
        long tv_usec; /* microseconds */微秒
    };
    返回文件描述符变化的个数
    void FD_CLR(int fd, fd_set *set);   //把文件描述符集合里fd位清0
    int FD_ISSET(int fd, fd_set *set);  //测试文件描述符集合里fd是否置1
    void FD_SET(int fd, fd_set *set);   //把文件描述符集合里fd位置1
    void FD_ZERO(fd_set *set);          //把文件描述符集合里所有位清0

```

3.3.select工作流程

1. 创建一个fd_set类型的读集合和写集合，用于保存需要监听的文件描述符。
2. 将需要监听的文件描述符添加到读集合和写集合中，使用FD_SET宏实现。
3. 调用select函数，传入读集合、写集合和超时时间等参数，等待I/O事件的发生。
4. select函数返回时，通过检查读集合和写集合中的文件描述符，判断哪些文件描述符上发生了I/O事件。
5. 根据需要，对发生I/O事件的文件描述符进行相应的读写操作，处理完毕后继续等待下一次I/O事件的发生。



3.4. select 示例代码

```c
#include <stdio.h>
#include <sys/select.h>
#include <sys/types.h>
#include <unistd.h>
#include "wrap.h"
#include <sys/time.h>
#define PORT 8888
int main(int argc, char *argv[])
{
    //创建套接字,绑定
    int lfd = tcp4bind(PORT,NULL);
    //监听
    Listen(lfd,128);
    int maxfd = lfd;//最大的文件描述符
    fd_set oldset,rset;
    FD_ZERO(&oldset);
    FD_ZERO(&rset);
    //将lfd添加到oldset集合中
    FD_SET(lfd,&oldset);
    while(1)
    {
        rset = oldset;//将oldset赋值给需要监听的集合rset

        int n = select(maxfd+1,&rset,NULL,NULL,NULL);
        if(n < 0)
        {
            perror("");
```

```
28              break;
29          }
30          else if(n == 0)
31          {
32              continue;//如果没有变化,重新监听
33          }
34          else//监听到了文件描述符的变化
35          {
36              //lfd变化 代表有新的连接到来
37              if( FD_ISSET(lfd,&rset))
38              {
39                  struct sockaddr_in cliaddr;
40                  socklen_t len =sizeof(cliaddr);
41                  char ip[16]="";
42                  //提取新的连接
43                  int cfd = Accept(lfd,(struct
sockaddr*)&cliaddr,&len);
44                  printf("new client ip=%s
port=%d\n",inet_ntop(AF_INET,&cliaddr.sin_addr.s_addr,ip,16),
45                          ntohs(cliaddr.sin_port));
46                  //将cfd添加至oldset集合中,以下次监听
47                  FD_SET(cfd,&oldset);
48                  //更新maxfd
49                  if(cfd > maxfd)
50                      maxfd = cfd;
51                  //如果只有lfd变化,continue
52                  if(--n == 0)
53                      continue;
54
55              }
56
57
58              //cfd  遍历lfd之后的文件描述符是否在rset集合中,如果在则
cfd变化
59              for(int i = lfd+1;i<=maxfd;i++)
60              {
61                  //如果i文件描述符在rset集合中
62                  if(FD_ISSET(i,&rset))
63                  {
64                      char buf[1500]="";
65                      int ret = Read(i,buf,sizeof(buf));
66                      if(ret < 0)//出错,将cfd关闭,从oldset中删除
cfd
67                      {
68                          perror("");
```

```
                        close(i);
                        FD_CLR(i,&oldset);
                        continue;
                    }
                    else if(ret == 0)
                    {
                        printf("client close\n");
                        close(i);
                        FD_CLR(i,&oldset);

                    }
                    else
                    {
                        printf("%s\n",buf);
                        Write(i,buf,ret);

                    }

                }


            }


        }


    return 0;
}
```

3.5. 数组改进select

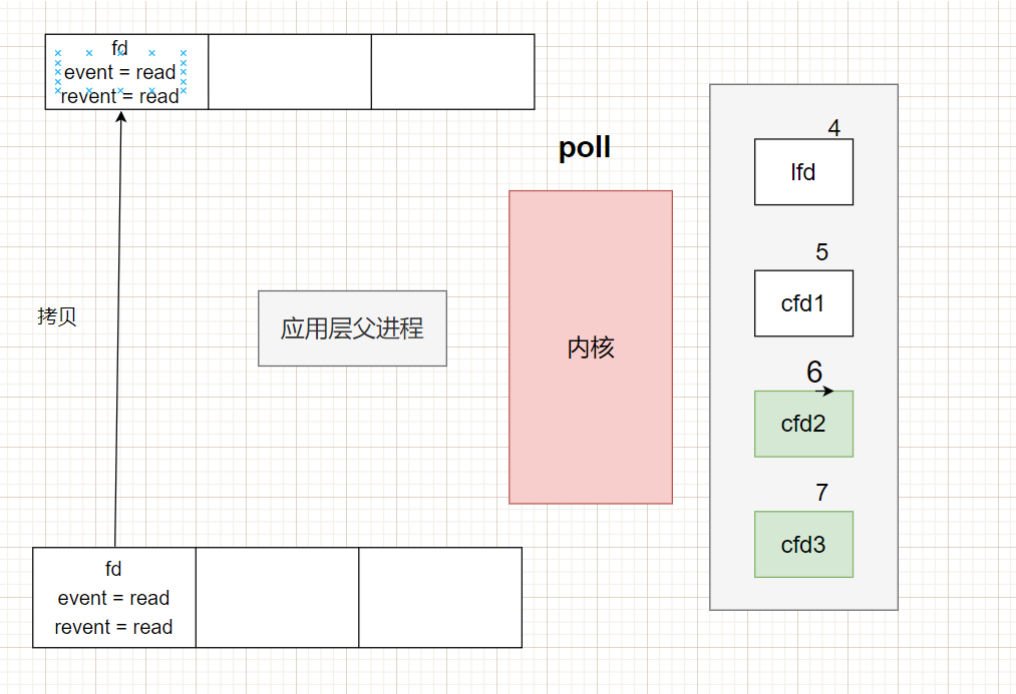
```
//进阶版select，通过数组防止遍历1024个描述符
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <ctype.h>
```

```c

#include "wrap.h"

#define SERV_PORT 8888

int main(int argc, char *argv[])
{
    int i, j, n, maxi;

    int nready, client[FD_SETSIZE];                 /* 自定义数组client, 防止遍历1024个文件描述符  FD_SETSIZE默认为1024 */
    int maxfd, listenfd, connfd, sockfd;
    char buf[BUFSIZ], str[INET_ADDRSTRLEN];         /* #define INET_ADDRSTRLEN 16 */
    struct sockaddr_in clie_addr, serv_addr;
    socklen_t clie_addr_len;
    fd_set rset, allset;                            /* rset 读事件文件描述符集合 allset用来暂存 */

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);
    //端口复用
    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_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port= htons(SERV_PORT);


    Bind(listenfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
    Listen(listenfd, 128);

    maxfd = listenfd;
  /* 起初 listenfd 即为最大文件描述符 */

    maxi = -1;
   /* 将来用作client[]的下标, 初始值指向0个元素之前下标位置 */
    for (i = 0; i < FD_SETSIZE; i++)
        client[i] = -1;
  /* 用-1初始化client[] */

```

```
45      FD_ZERO(&allset);
46      FD_SET(listenfd, &allset);
    /* 构造select监控文件描述符集 */
47
48      while (1) {
49          rset = allset;
    /* 每次循环时都从新设置select监控信号集 */
50
51          nready = select(maxfd+1, &rset, NULL, NULL, NULL);
    //2  1--lfd  1--connfd
52          if (nready < 0)
53              perr_exit("select error");
54
55          if (FD_ISSET(listenfd, &rset)) {
    /* 说明有新的客户端链接请求 */
56
57              clie_addr_len = sizeof(clie_addr);
58              connfd = Accept(listenfd, (struct sockaddr
*)&clie_addr, &clie_addr_len);       /* Accept 不会阻塞 */
59              printf("received from %s at PORT %d\n",
60                      inet_ntop(AF_INET, &clie_addr.sin_addr,
str, sizeof(str)),
61                      ntohs(clie_addr.sin_port));
62
63              for (i = 0; i < FD_SETSIZE; i++)
64                  if (client[i] < 0) {
    /* 找client[]中没有使用的位置 */
65                      client[i] = connfd;
   /* 保存accept返回的文件描述符到client[]里 */
66                      break;
67                  }
68
69              if (i == FD_SETSIZE) {
    /* 达到select能监控的文件个数上限 1024 */
70                  fputs("too many clients\n", stderr);
71                  exit(1);
72              }
73
74              FD_SET(connfd, &allset);
    /* 向监控文件描述符集合allset添加新的文件描述符connfd */
75
76              if (connfd > maxfd)
77                  maxfd = connfd;
   /* select第一个参数需要 */
78
```

```
79              if (i > maxi)
80                  maxi = i;
  /* 保证maxi存的总是client[]最后一个元素下标 */
81
82              if (--nready == 0)
83                  continue;
84          }
85
86          for (i = 0; i <= maxi; i++) {
        /* 检测哪个clients 有数据就绪 */
87
88              if ((sockfd = client[i]) < 0)
89                  continue;//数组内的文件描述符如果被释放有可能变成-1
90              if (FD_ISSET(sockfd, &rset)) {
91
92                  if ((n = Read(sockfd, buf, sizeof(buf))) ==
0) {    /* 当client关闭链接时,服务器端也关闭对应链接 */
93                      Close(sockfd);
94                      FD_CLR(sockfd, &allset);
        /* 解除select对此文件描述符的监控 */
95                      client[i] = -1;
96                  } else if (n > 0) {
97                      for (j = 0; j < n; j++)
98                          buf[j] = toupper(buf[j]);
99                      Write(sockfd, buf, n);
100                     Write(STDOUT_FILENO, buf, n);
101                 }
102                 if (--nready == 0)
103                     break;
        /* 跳出for, 但还在while中 */
104             }
105         }
106     }
107
108     Close(listenfd);
109
110     return 0;
111 }
112
113
```

4. poll



4.1. API

```
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
    struct pollfd {
        int fd; /* 文件描述符 */
        short events; /* 监控的事件 */
        short revents; /* 监控事件中满足条件返回的事件 */
    };
    POLLIN          普通或带外优先数据可读,即POLLRDNORM |
POLLRDBAND
    POLLRDNORM      数据可读
    POLLRDBAND      优先级带数据可读
    POLLPRI         高优先级可读数据
    POLLOUT         普通或带外数据可写
    POLLWRNORM      数据可写
    POLLWRBAND      优先级带数据可写
    POLLERR         发生错误
    POLLHUP         发生挂起
    POLLNVAL        描述字不是一个打开的文件

    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

```

如果不再监控某个文件描述符时，可以把pollfd中，fd设置为-1，poll不再监控此pollfd，下次返回时，把revents设置为0。

相较于select而言，poll的优势：

1. **传入、传出事件分离。无需每次调用时，重新设定监听事件。**
2. **文件描述符上限，可突破1024限制。能监控的最大上限数可使用配置文件调整。**

4.2. 代码

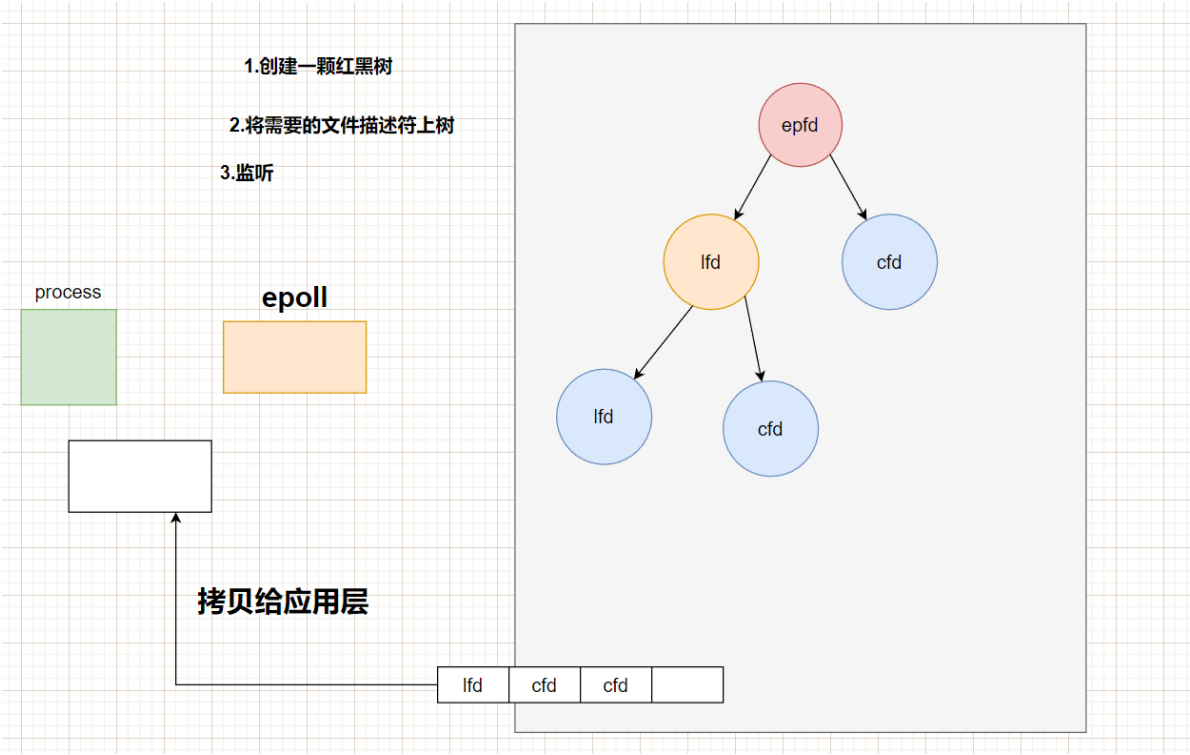
```
int main(int argc, char *argv[]) {
    int i, j, maxi, listenfd, connfd, sockfd;
    int nready;
    ssize_t n;
    char buf[MAXLINE], str[INET_ADDRSTRLEN];
    socklen_t clilen;
    struct pollfd client[OPEN_MAX];
    struct sockaddr_in cliaddr, servaddr;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

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

    Bind(listenfd, (struct sockaddr *) &servaddr,
sizeof(servaddr));

    Listen(listenfd, 20);

    client[0].fd = listenfd;
    client[0].events = POLLIN;                    /* listenfd
监听普通读事件 */
    for (i = 1; i < OPEN_MAX; i++)
        client[i].fd = -1;                          /* 用-1
初始化client[]里剩下元素 */
    maxi = 0;                                      /*
client[]数组有效元素中最大元素下标 */

    for (;;) {
        nready = poll(client, maxi + 1, -1);           /* 阻
塞 */
```

```
        if (client[0].revents & POLLIN) {        /* 有客户端链接请求 */
            clilen = sizeof(cliaddr);
            connfd = Accept(listenfd, (struct sockaddr *) &cliaddr, &clilen);
            printf("received from %s at PORT %d\n",
                   inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
                   ntohs(cliaddr.sin_port));
            for (i = 1; i < OPEN_MAX; i++) {
                if (client[i].fd < 0) {
                    client[i].fd = connfd;    /* 找到client[]中空闲的位置，存放accept返回的connfd */
                    break;
                }
            }

            if (i == OPEN_MAX)
                perr_exit("too many clients");

            client[i].events = POLLIN;        /* 设置刚刚返回的connfd，监控读事件 */
            if (i > maxi)
                maxi = i;                         /* 更新client[]中最大元素下标 */
            if (--nready <= 0)
                continue;                         /* 没有更多就绪事件时,继续回到poll阻塞 */
        }
        for (i = 1; i <= maxi; i++) {             /* 检测client[] */
            if ((sockfd = client[i].fd) < 0)
                continue;
            if (client[i].revents & POLLIN) {
                if ((n = Read(sockfd, buf, MAXLINE)) < 0) {
                    if (errno == ECONNRESET) { /* 当收到 RST标志时 */
                        /* connection reset by client */
                        printf("client[%d] aborted connection\n", i);
                        Close(sockfd);
                        client[i].fd = -1;
                    } else {
                        perr_exit("read error");
                    }
```

```
64                  } else if (n == 0) {
65                      /* connection closed by client */
66                      printf("client[%d] closed connection\n",
    i);
67                      Close(sockfd);
68                      client[i].fd = -1;
69                  } else {
70                      for (j = 0; j < n; j++)
71                          buf[j] = toupper(buf[j]);
72                      Writen(sockfd, buf, n);
73                  }
74                  if (--nready <= 0)
75                      break;                  /* no more readable
    descriptors */
76              }
77          }
78      }
79      return 0;
80  }
81
```

5.重点：epoll

5.1.epoll的工作原理



5.2. epoll的API

1.创建"红黑树"

```
#include <sys/epoll.h>
int epoll_create(int size)
    size：监听数目（内核参考值）
    返回值：成功：非负文件描述符；失败：-1，设置相应的errno
```

2.事件"上树"

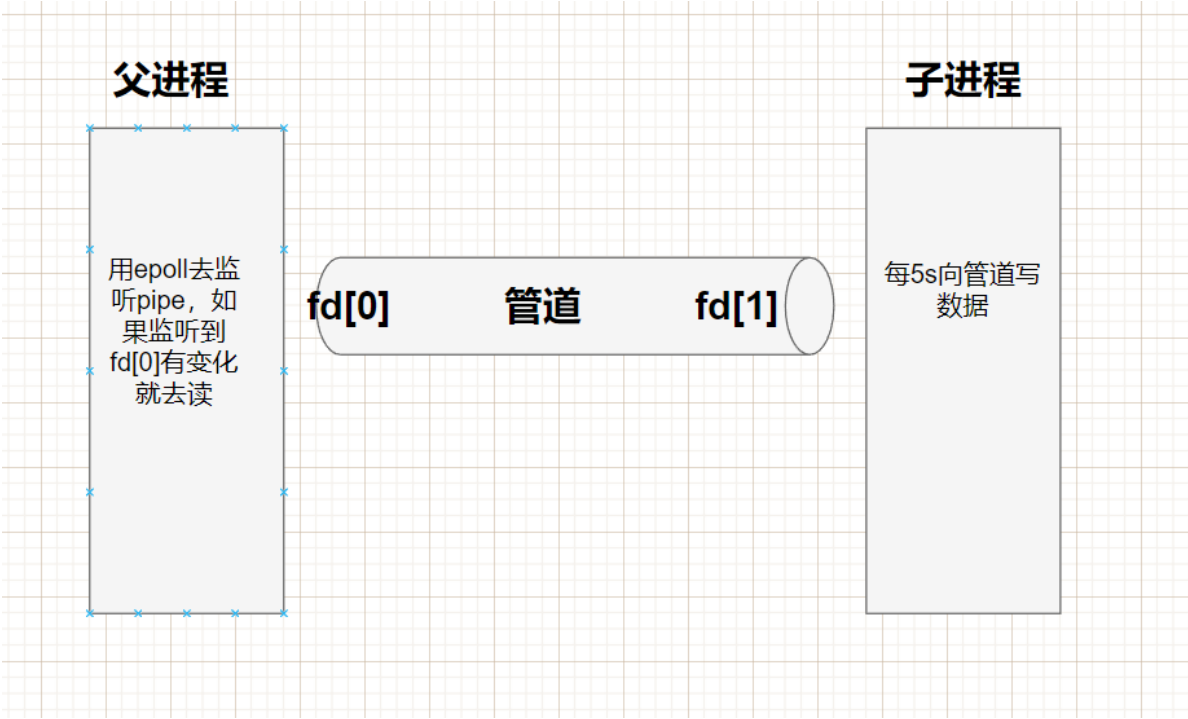
```
#include <sys/epoll.h>
 int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)
        epfd:    为epoll_creat的句柄
        op:      表示动作，用3个宏来表示：
            EPOLL_CTL_ADD (注册新的fd到epfd),
            EPOLL_CTL_MOD (修改已经注册的fd的监听事件),
            EPOLL_CTL_DEL (从epfd删除一个fd);

        event:   告诉内核需要监听的事件

        struct epoll_event {
            __uint32_t events; /* Epoll events */
            epoll_data_t data; /* User data variable */
        };
        typedef union epoll_data {
            void *ptr;
            int fd;
            uint32_t u32;
            uint64_t u64;
        } epoll_data_t;

        EPOLLIN :    表示对应的文件描述符可以读（包括对端SOCKET正常关闭）
        EPOLLOUT:    表示对应的文件描述符可以写
        EPOLLPRI:    表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）
        EPOLLERR:    表示对应的文件描述符发生错误
        EPOLLHUP:    表示对应的文件描述符被挂断;
        EPOLLET:     将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)而言的
        EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里
```

```
        返回值：成功：0；失败：-1，设置相应的errno
```

3.事件"监听"

```
#include <sys/epoll.h>
    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，可简单看作数组。
        maxevents：   告之内核这个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
```

5.3.上手epoll




```

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/epoll.h>
int main(int argc, char *argv[])
{
    int fd[2];
```

```
    pipe(fd);
    //创建子进程
    pid_t pid;
    pid = fork();
    if(pid < 0)
        perror("");
    else if(pid == 0)
    {
        close(fd[0]);
        char buf[5];
        char ch='a';
        while(1)
        {
            sleep(3);
            memset(buf,ch++,sizeof(buf));
            write(fd[1],buf,5);


        }

    }
    else
    {
        close(fd[1]);
        //创建树
        int epfd = epoll_create(1);
        struct epoll_event ev,evs[1];
        ev.data.fd = fd[0];
        ev.events = EPOLLIN;
        epoll_ctl(epfd,EPOLL_CTL_ADD,fd[0],&ev);//上树
        //监听
        while(1)
        {
            int n = epoll_wait(epfd,evs,1,-1);
            if(n == 1)
            {
                char buf[128]="";
                int ret  = read(fd[0],buf,sizeof(buf));
                if(ret <= 0)
                {
                    close(fd[0]);
                    epoll_ctl(epfd,EPOLL_CTL_DEL,fd[0],&ev);
                    break;
                }
                else
```

```
                {
                    printf("%s\n",buf);
                }

            }


        }


    }

    return 0;
}


```

epoll服务器代码

```


#include <stdio.h>
#include <fcntl.h>
#include "wrap.h"
#include <sys/epoll.h>

int main(int argc, char *argv[]) {
    //创建套接字 绑定
    int lfd = tcp4bind(8000, NULL);
    //监听
    Listen(lfd, 128);
    //创建树
    int epfd = epoll_create(1);
    //将lfd上树
    struct epoll_event ev, evs[1024];
    ev.data.fd = lfd;
    ev.events = EPOLLIN;
    epoll_ctl(epfd, EPOLL_CTL_ADD, lfd, &ev);
    //while监听
    while (1) {
        int nready = epoll_wait(epfd, evs, 1024, -1);//监听
        printf("epoll wait ____________________\n");
```

```
        if (nready < 0) {
            perror("");
            break;
        } else if (nready == 0) {
            continue;
        } else//有文件描述符变化
        {
            for (int i = 0; i < nready; i++) {
                //判断lfd变化,并且是读事件变化
                if (evs[i].data.fd == lfd && evs[i].events & EPOLLIN) {
                    struct sockaddr_in cliaddr;
                    char ip[16] = "";
                    socklen_t len = sizeof(cliaddr);
                    int cfd = Accept(lfd, (struct sockaddr *) &cliaddr, &len);//提取新的连接

                    printf("new client ip=%s port =%d\n", inet_ntop(AF_INET, &cliaddr.sin_addr.s_addr, ip, 16),
                           ntohs(cliaddr.sin_port));
                    //设置cfd为非阻塞
                    int flags = fcntl(cfd, F_GETFL);//获取的cfd的标志位
                    flags |= O_NONBLOCK;//非阻塞标志位
                    fcntl(cfd, F_SETFL, flags);
                    //将cfd上树
                    ev.data.fd = cfd;
                    ev.events = EPOLLIN | EPOLLET;//边缘触发
                    epoll_ctl(epfd, EPOLL_CTL_ADD, cfd, &ev);

                } else if (evs[i].events & EPOLLIN)//cfd变化 ,而且是读事件变化
                {
                    while (1) {
                        char buf[4] = "";
                        //如果读一个缓冲区,缓冲区没有数据,如果是带阻塞,就阻塞等待,如果
                        //是非阻塞,返回值等于-1,并且会将errno 值设置为EAGAIN
                        int n = read(evs[i].data.fd, buf, sizeof(buf));
                        if (n < 0)//出错,cfd下树
                        {
                            //如果缓冲区读干净了,这个时候应该跳出while(1)循环,继续监听
```

```
                                if (errno == EAGAIN) {
                                    break;

                                }
                                //普通错误
                                perror("");
                                close(evs[i].data.fd);//将cfd关闭
                                epoll_ctl(epfd, EPOLL_CTL_DEL, evs[i].data.fd, &evs[i]);
                                break;
                            } else if (n == 0)//客户端关闭 , 下树
                            {
                                printf("client close\n");
                                close(evs[i].data.fd);//将cfd关闭
                                epoll_ctl(epfd, EPOLL_CTL_DEL, evs[i].data.fd, &evs[i]);//下树
                                break;
                            } else {
                                //  printf("%s\n",buf);
                                write(STDOUT_FILENO, buf, 4);
                                write(evs[i].data.fd, buf, n);
                            }
                        }

                    }


                }


            }


    }

    return 0;
}
```

5.4. epoll的工作流程

epoll是Linux系统中的一种I/O多路复用机制，与其他I/O多路复用机制（如select和poll）相比，它具有更高的效率和更好的扩展性，适用于高并发的服务器程序。

epoll的工作原理如下：

1. 创建一个epoll实例，调用epoll_create函数，创建一个epoll对象，返回一个文件描述符，用于后续的操作。
2. 注册事件，调用epoll_ctl函数，将需要监听的文件描述符添加到epoll对象中，同时设置需要监听的事件类型，包括读事件、写事件和错误事件等。这个过程实际上是在建立一个事件表，用来记录需要监听的文件描述符及其对应的事件类型。
3. 等待事件，调用epoll_wait函数，等待文件描述符上的事件发生。在等待过程中，epoll_wait函数会阻塞进程，直到有文件描述符上的事件发生或者超时时间到达。
4. 处理事件，当有文件描述符上的事件发生时，epoll_wait函数会返回，返回结果中包含了所有发生事件的文件描述符及其对应的事件类型。通过遍历返回结果，可以获取到所有需要处理的文件描述符及其对应的事件类型，进而进行相应的处理操作。在处理完所有发生的事件之后，需要重新调用epoll_wait函数等待下一次事件的发生。

epoll的工作流程相较于select和poll有以下优点：

1. 高效：epoll使用红黑树来存储需要监听的文件描述符，能够支持非常大的并发连接，并且能够在其中高效地查找和处理事件。
2. 省资源：epoll在等待事件发生时，不需要像select和poll那样需要将需要监听的所有文件描述符拷贝到内核空间，而是在epoll_ctl注册时就将需要监听的文件描述符传递给内核，避免了不必要的拷贝操作，从而减少了内存开销和CPU占用率。
3. 支持边缘触发和水平触发：epoll支持两种事件触发方式，即**边缘触发**（Edge Triggered）和**水平触发**（Level Triggered），可以根据实际需要选择相应的触发方式，更加灵活。

总之，epoll是一种高效、省资源、可扩展的I/O多路复用机制，在高并发服务器中得到了广泛的应用。

5.5.水平触发和边缘触发

水平触发（Level Triggered）和边缘触发（Edge Triggered）是I/O多路复用机制中常用的两种事件触发方式。

水平触发是指，**当一个文件描述符上有事件发生时，只要该事件尚未被处理，那么下次epoll_wait返回时仍会通知该文件描述符上的事件。**也就是说，只要文件描述符上有事件发生，epoll_wait就会返回，然后应用程序需要自行处理该事件，处理完毕后再次调用epoll_wait等待下一个事件的发生。因此，在水平触发模式下，应用程序需要不断地检查该文件描述符上是否有事件发生，否则就会一直等待下去。

边缘触发是指，当一个文件描述符上有事件发生时，只会触发一次epoll_wait返回，即只有当该文件描述符上有新的事件发生时才会通知应用程序。也就是说，边缘触发模式只通知事件的变化，而不是一直通知文件描述符上的事件状态。在边缘触发模式下，如果应用程序未能及时处理该事件，那么下一次epoll_wait将不会返回该文件描述符上的事件。因此，在边缘触发模式下，应用程序需要快速处理事件，以免错过事件的发生。

需要注意的是，水平触发和边缘触发适用于不同的应用场景。**一般来说，对于非常繁忙的网络应用，边缘触发更适合，因为它可以避免频繁的epoll_wait调用和处理已经发生的事件。**而对于一般的网络应用，水平触发就足够了，因为它可以比较方便地处理文件描述符上的所有事件。

5.6.EAGAIN

在Linux环境下开发经常会碰到很多错误(设置errno)，其中`EAGAIN`是其中比较常见的一个错误(比如用在非阻塞操作中)。在man手册关于read的解释如下：

> RETURN VALUE
>
> On success, the number of bytes read is returned(zero indicates end of file), and the file position is
>
> advanced by this number. It is not an error if this number is smaller than the number of bytes requested;
>
> this may happen for example because fewer bytes are actually available right now (maybe because we
>
> were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read()
>
> was interrupted by a signal. See also NOTES.
>
> ERRORS
>
> EAGAIN The file decriptor fd refers to a file other than a socket and has been marked nonblocking
>
> (O_NONBLOCK), and the read would block. See open(2) for futher details on the O_NONBLOCK
>
> flag.
>
> EAGAIN or EOWOULDBLOCK

> The file descriptor fd refers to a socket and has been marked nonblocking (O_NONBLOCK), and the read
> would block. POSIX.1-2001 allows either error to be returned for this case, and does not require these
> constants to have the same value, so a portable application should check for both possibilities.

从字面上来看，是提示再试一次。这个错误经常出现在当应用程序进行一些非阻塞(non-blocking)操作(对文件或socket)的时候。例如，以O_NONBLOCK的标志打开file/socket/FIFO，如果你连续做read操作而没有数据可读。此时程序不会阻塞起来等待数据准备就绪返回，read函数会返回一个错误EAGAIN，提示你的应用程序现在没有数据可读请稍后再试。

又例如，当一个系统调用(比如fork)因为没有足够的资源(比如虚拟内存)而执行失败，返回EAGAIN提示其再调用一次(也许下次就能成功)。