12 进程池和线程池

12.1 进程池和线程池的设计思路

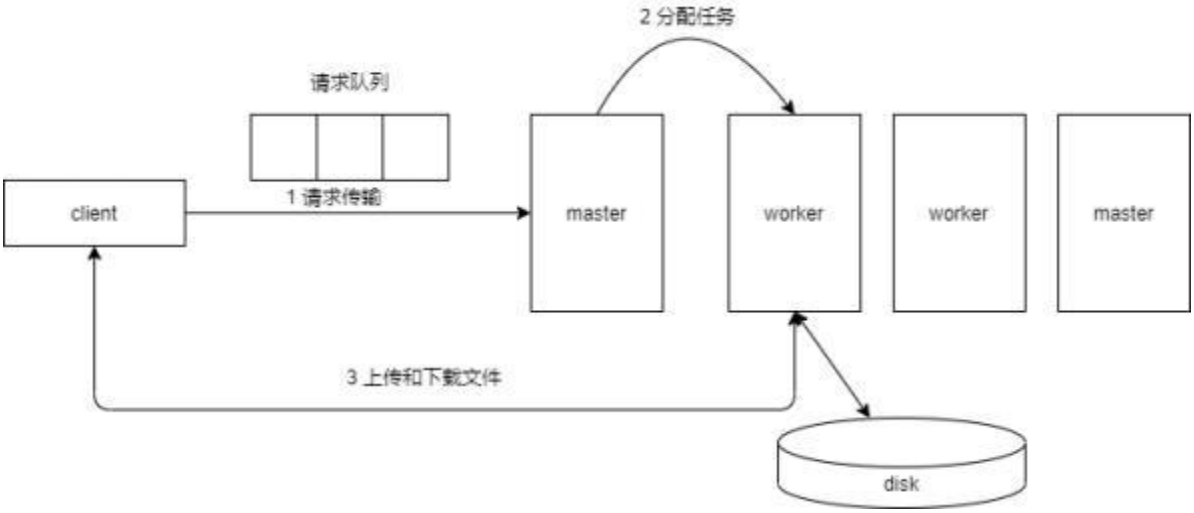
12.1.1 设计的背景

如果要实现一个网络文件传输存储（即网盘项目）应用程序，需要涉及的知识点包括文件、进程、线程和网络等等。但是之前所学习的内容都是零散的知识点，我们需要综合这些所有的知识点，然后从一个更高的层面，也就是从整体设计的角度来思考应用程序的架构，这样才能构建一个高质量的网盘项目。

一个良好的架构需要满足许多侧面的要求，其中最基本的要求是**可维护性**和**性能**：

- 可维护性是指应用程序对开发者应该足够友好，开发和维护的程序员可以很快速的就能理解程序架构并进行后续开发。为了提供可维护性，网盘的各个部分的功能应当彼此分离。在这里，我们将网盘设计成一个多进程的应用：存在一个主进程和多个子进程，主进程负责处理启动程序、管理子进程和处理客户端请求，子进程也叫工作进程，负责真正的传输文件。
- 性能是指应用程序应当充分利用操作系统资源，并且减少不要资源消耗。在多进程程序中， 一种非常浪费资源的操作就是创建进程，在网盘项目中，如果每有一个传输文件的请求，程序就要去创建一个进程，那么大量的时间都消耗在进程的创建和销毁当中。 一种优化的思路就是**“池化”**，在应用程序启动的时候就创建多个子进程，这些子进程**不会在执行过程中动态的创建和回收，它们一直存在直到应用程序终止**；如果没有任务时，子进程睡眠；而每当有传输文件的请求到来时，会唤醒某个子进程来完成任务；文件传输完成之后，子进程会重新陷入睡眠。

12.1.2 进程池的整体结构



12.2 进程池的实现

我们以一个文件下载的应用为例子来介绍进程池结构：客户端可以向服务端建立连接，随后将服务端中存储的文件通过网络传输发送到客户端，其中一个服务端可以同时处理多个客户端连接的，彼此之间互不干扰。

12.2.1 父子进程创建

首先，我们先实现最基本的功能，使用一个父进程创建若干个子进程。目前的话，父进程只负责创建子进程，创建完成之后直接陷入死循环，子进程被创建之后也直接陷入死循环。父进程需要知道每个子进程的pid和空闲情况。

下面是代码实现（注意只摘要了重要部分）：

```
//head.h
enum workerStatus {
    FREE,
    BUSY
};
typedef struct{
    pid_t pid; //工作进程的pid
    int status;//工作进程的状态
} processData_t;
//main.c
int main(int argc, char *argv[]){
    //./main 192.168.30.129 8080 3
    ARGS_CHECK(argc,4);
    int workerNum = atoi(argv[3]);
    processData_t *workerList = (processData_t
*)calloc(sizeof(processData_t),workerNum);
    //workerList记录了所有工作进程的状态
    makeChild(workerList,workerNum);
    while(1);
}
//child.c
int makeChild(processData_t *pProcssData, int processNum){
    pid_t pid;
    for(int i = 0;i < processNum; ++i){
        pid = fork();
        if(pid == 0){
            handleEvent();
        }
        pProcssData[i].pid = pid;
        pProcssData[i].status = FREE;
    }
    return 0;
}
void handleEvent(){//工作进程目前的工作是死循环
    while(1);
}
```

12.2.2 父进程处理网络连接

在创建完所有的子进程之后，父进程的下一个工作目标是准备接受客户端的TCP连接，这个工作和之前网络编程时的工作内容差不多，按照 `socket` 、 `bind`和 `listen` 的顺序执行系统调用即可。

下面是相关的代码：

```
int tcpInit(char *ip, char *port,int *pSockFd){
    *pSockFd = socket(AF_INET,SOCK_STREAM,0);
    struct sockaddr_in addr;
    bzero(&addr,sizeof(struct sockaddr_in));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr(ip);
    addr.sin_port = htons(atoi(port));
    int reuse = 1;
    int ret;
    ret = setsockopt(*pSockFd,SOL_SOCKET,SO_REUSEADDR,&reuse,sizeof(reuse));
    ERROR_CHECK(ret,-1,"setsockopt");
    ret = bind(*pSockFd,(struct sockaddr*)&addr,sizeof(struct sockaddr_in));
```

```
    ERROR_CHECK(ret,-1,"bind");
    listen(*pSockFd,10);
    return 0;
}
```

12.2.3 本地套接字

父进程和子进程的地址空间是隔离的，如果两个进程之间需要进行通信，那就要选择一种合适的进程间通信的手段，在本项目中，比较合适的方法是管道。

除了之前所使用的 `pipe`系统调用可以在父子进程间创建管道以外，还有一种方法是本地套接字。使用系统调用 `socketpair`可以在父子进程间利用 `socket`创建一个**全双工**的管道。除此以外，本地套接字可以在同一个操作系统的两个进程之间传递文件描述符。

创建套接字的方式是和管道的用法十分相似：

```
int socketpair(int domain, int type, int protocol, int sv[2]);
```

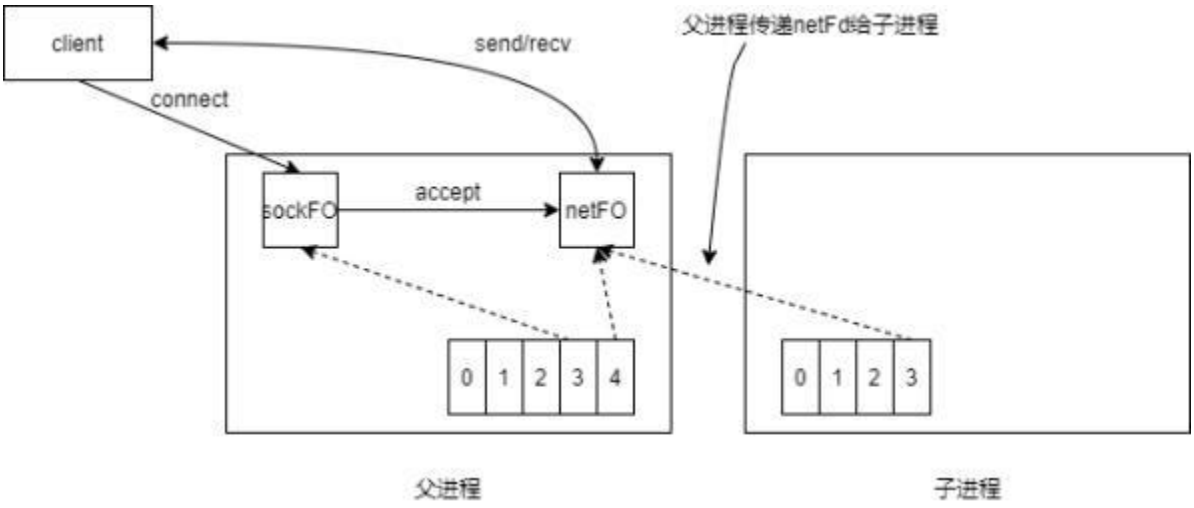
在这里 `domain`必须填写AF_LOCAL， `type`可以选择流式数据还是消息数据， `protocol`一般填0表示不需要任何额外的协议， `sv`这个参数和 `pipe` 的参数一样，是一个长度为2的整型数据，用来存储管道两端的文件描述符（值得注意的是， `sv[0]`和 `sv[1]`没有任何的区别）。 一般 `socketpair`之后会配合 `fork` 函数一起使用，从而实现父子进程之间的通信。从数据传递使用上面来看，本地套接字和网络套接字是完全一致的，但是本地套接字的效率更高，因为它在拷贝数据的时候不需要处理协议相关内容。

12.2.4 父子进程共享文件描述符

那么父进程向子进程到底需要传递哪些信息呢？除了传递一般的控制信息和文本信息（比如上传）以外，需要特别注意的是**需要传递已连接套接字的文件描述符。**

父进程会监听特定某个 `IP:PORT` ，如果有某个客户端连接之后，子进程需要能够连上 `accept`得到的已连接套接字的文件描述符，这样子进程才能和客户端进行通信。这种文件描述符的传递不是简单地传输一个整型数字就行了，而是需要让父子进程共享一个套接字文件对象。

但是这里会遇到麻烦，因为 `accept`调用是在 `fork`之后的，所以父子进程之间并不是天然地共享文件对象。倘若想要在父子进程之间共享 `acccept`调用返回的已连接套接字，需要采用一些特别的手段： 一方面，父子进程之间需要使用本地套接字来通信数据。另一方面需要使用 `sendmsg`和 `recvmsg` 函数来传递数据。



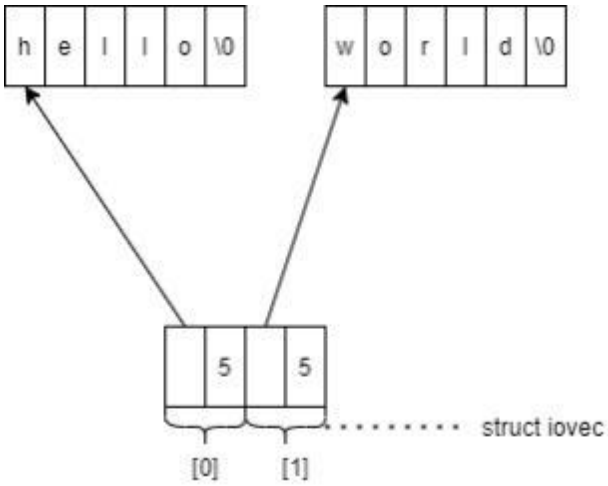

```
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
struct iovec
```

```
{                    /* Scatter/gather array items */
    void *iov_base; /* Starting address */
    size_t iov_len; /* Number of bytes to transfer */
};

struct msghdr
{
    void *msg_name;         /* optional address */
    socklen_t msg_namelen; /* size of address */
    struct iovec *msg_iov; /* scatter/gather array */
    size_t msg_iovlen;     /* # elements in msg_iov */
    void *msg_control;     /* ancillary data, see below */
    size_t msg_controllen; /* ancillary data buffer len */
    int msg_flags;         /* flags on received message */
};
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

使用 sendmsg和 recvmsg 的时候附加一个消息头部，即一个 struct msghdr类型的结构体。

首先，需要将要传递的内容存储入 msg_iov 当中，在这里需要注意的是，元素类型为 struct iovec 的数组可以存储一组离散的消息，只需要将每个消息的起始地址和本消息的长度存入数组元素中即可。（使用 writev和 readv可以直接读写一组离散的消息。）



接下来，需要将文件描述符的信息存入控制字段 msg_control 中，这个我们需要存储一个地址值，该地址指向了一个 struct cmsghdr类型的控制信息。如果存在多个控制信息，会构成一个控制信息序列，规范要求使用者绝不能直接操作控制信息序列，而是需要用一系列的 cmsg宏来间接操作。

CMSG_FIRSTHDR 用来获取序列中的第一个控制信息（ CMSG_NXTHDR 获取下一个）， CMSG_DATA 宏用来设置控制信息的具体数据的地址； CMSG_LEN 宏用来设置具体数据占据内存空间的大小。

```
//man cmsg
struct cmsghdr{
    size_t cmsg_len; /* Data byte count, including header
                        (type is socklen_t in POSIX) */
    int cmsg_level;  /* Originating protocol */
    int cmsg_type;   /* Protocol-specific type */
    /* followed by
        unsigned char cmsg_data[]; */
};
struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *msgh);
struct cmsghdr *CMSG_NXTHDR(struct msghdr *msgh, struct cmsghdr *cmsg);
size_t CMSG_ALIGN(size_t length);
```

```
size_t CMSG_SPACE(size_t length);
size_t CMSG_LEN(size_t length);
unsigned char *CMSG_DATA(struct cmsghdr *cmsg);
```

为了传递文件描述符，需要将结构体中的`cmsg_level`字段设置为`SOL_SOCKET`，而`cmsg_type`字段需要设置为`SCM_RIGHTS`，再将数据部分设置为文件描述符。这样，该文件描述符所指的文件对象就可以传递到另一个进程了。

下面是代码示例：

```
int sendFd(int pipeFd, int fdToSend){
    struct msghdr hdr;
    bzero(&hdr,sizeof(struct msghdr));
    struct iovec iov[1];
    char buf[] = "Hello";
    iov[0].iov_base = buf;
    iov[0].iov_len = 5;
    hdr.msg_iov = iov;
    hdr.msg_iovlen = 1;
    struct cmsghdr *pcmsghdr = (struct cmsghdr
*)calloc(1,sizeof(CMSG_LEN(sizeof(int))));
    pcmsghdr->cmsg_len = CMSG_LEN(sizeof(int));
    //控制信息的数据部分只有int类型的文件描述符
    pcmsghdr->cmsg_level = SOL_SOCKET;
    pcmsghdr->cmsg_type = SCM_RIGHTS;//SCM->socket-level control message
    //表示在socket层传递的是访问权力，这样接受进程就可以访问对应文件对象了
    *(int *)CMSG_DATA(pcmsghdr) = fdToSend;
    //数据部分是文件描述符
    hdr.msg_control = pcmsghdr;
    hdr.msg_controllen = CMSG_LEN(sizeof(int));
    int ret = sendmsg(pipeFd,&hdr,0);
    ERROR_CHECK(ret,-1,"sendmsg");
}
int recvFd(int pipeFd, int *pFd){
    struct msghdr hdr;
    bzero(&hdr,sizeof(struct msghdr));
    struct iovec iov[1];
    char buf[6] = {0};//除了数据内容以外，其他和sendmsg是一致的
    iov[0].iov_base = buf;
    iov[0].iov_len = 5;//这里一定不能填0
    hdr.msg_iov = iov;
    hdr.msg_iovlen = 1;
    struct cmsghdr *pcmsghdr = (struct cmsghdr
*)calloc(1,sizeof(CMSG_LEN(sizeof(int))));
    pcmsghdr->cmsg_len = CMSG_LEN(sizeof(int));
    pcmsghdr->cmsg_level = SOL_SOCKET;
    pcmsghdr->cmsg_type = SCM_RIGHTS;//SCM->socket-level control message
    hdr.msg_control = pcmsghdr;
    hdr.msg_controllen = CMSG_LEN(sizeof(int));
    int ret = recvmsg(pipeFd,&hdr,0);
    ERROR_CHECK(ret,-1,"recvmsg");
    *pFd = *(int *)CMSG_DATA(pcmsghdr);
    return 0;
}
```

要特别注意的是，传递的文件描述符在数值上完全可能是不相等的，但是它们对应的文件对象确实是同一个，自然文件读写偏移量也是共享的，和之前使用 dup或者是先打开文件再 fork 的情况是一致的，下面是例子：

```
int main(){
    int pipeFd[2];
    socketpair(AF_LOCAL,SOCK_STREAM,0,pipeFd);
    if(fork() == 0){
        int fd1 = open("file1", O_RDWR);
        printf("child fd1 = %d\n", fd1);
        write(fd1,"hello",5);
        sendFd(pipeFd[0],fd1);
    }
    else{
        int fd2 = open("file2", O_RDWR);
        int fd3;
        recvFd(pipeFd[1],&fd3);
        printf("parent fd3 = %d\n", fd3);
        write(fd3,"world", 5);
        wait(NULL);
    }
    exit(0);
}
```

至此，我们就可以实现一个进程池的服务端了：

- 启动父进程
- makeChild :父进程在创建每个子进程时，先调用 socketpair
- handleEvent :子进程被创建之后，执行进程工作函数
- recvFd :子进程等待一个文件描述符，在父进程未发送的时候，子进程处于阻塞状态
- tcpInit :父进程初始化一个网络socket
- epollFunc :父进程使用 epoll等IO多路复用机制监听网络socket和每个子进程的本地socket的一端。
- 如果有客户端通过网络连接父进程，那么父进程会 accept得到一个已连接socket。
- sendFd :选择一个空闲的子进程，将已连接socket发送给子进程，之后父进程就不再和客户端直接网络通信，而是由子进程和客户端通信。
- 当某个子进程完成了任务之后，子进程可以通过本地socket通知父进程，并且重新将自己设为空闲。

下面是样例代码：

```
//客户端
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc, 3);
    int sockFd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr;
    bzero(&addr, sizeof(struct sockaddr_in));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr(argv[1]);
    addr.sin_port = htons(atoi(argv[2]));
    int ret = connect(sockFd, (struct sockaddr*)&addr, sizeof(struct
sockaddr_in));
    ERROR_CHECK(ret, -1, "connect");
    char buf[1024] = {0};
```

```
    read(STDIN_FILENO,buf,sizeof(buf));
    send(sockFd,buf,strlen(buf)-1,0);
    bzero(buf,sizeof(buf));
    recv(sockFd,buf,sizeof(buf),0);
    puts(buf);
    close(sockFd);
    return 0;
}
//服务端主进程
int main(int argc, char *argv[]){
    //./main 192.168.135.132 5678 10
    ARGS_CHECK(argc,4);
    int workerNum = atoi(argv[3]);
    processData_t *workerList = (processData_t
*)calloc(sizeof(processData_t),workerNum);
    makeChild(workerList,workerNum);
    int sockFd;
    tcpInit(argv[1],argv[2],&sockFd);
    int epfd = epollCtor();
    epollAdd(sockFd,epfd);
    for(int i = 0;i < workerNum; ++i){
        epollAdd(workerList[i].pipeFd,epfd);
    }
    int listenSize = workerNum+1;//socket+每个进程pipe的读端
    struct epoll_event * readylist = (struct epoll_event
*)calloc(listenSize,sizeof(struct epoll_event));
    while(1){
        int readynum = epoll_wait(epfd,readylist,listenSize,-1);
        for(int i = 0;i < readynum; ++i){
            if(readylist[i].data.fd == sockFd){
                puts("accept ready");
                int netFd = accept(sockFd,NULL,NULL);
                for(int j = 0;j < workerNum; ++j){
                    if(workerList[j].status == FREE){
                        printf("No. %d worker gets his job, pid = %d\n", j,
workerList[j].pid);
                        sendFd(workerList[j].pipeFd, netFd);
                        workerList[j].status = BUSY;
                        break;
                    }
                }
                close(netFd);//父进程交给子进程一定要关闭
            }
            else{
                puts("One worker finish his task!");
                int j;
                for(j = 0;j < workerNum;++j){
                    if(workerList[j].pipeFd == readylist[i].data.fd){
                        pid_t pid;
                        int ret =
recv(workerList[j].pipeFd,&pid,sizeof(pid_t),0);
                        printf("No. %d worker finish, pid = %d\n",j,pid);
                        workerList[j].status = FREE;
                        break;
                    }
                }
            }
        }
```

```
    }
}
//服务端子进程
int makeChild(processData_t *pProcssData, int processNum)
{
    pid_t pid;
    for (int i = 0; i < processNum; ++i)
    {
        int pipeFd[2];
        socketpair(AF_LOCAL,SOCK_STREAM,0,pipeFd);
        pid = fork();
        if (pid == 0)
        {
            close(pipeFd[0]);
            handleEvent(pipeFd[1]);
        }
        close(pipeFd[1]);
        printf("pid = %d, pipefd[0] = %d\n", pid, pipeFd[0]);
        pProcssData[i].pid = pid;
        pProcssData[i].status = FREE;
        pProcssData[i].pipeFd = pipeFd[0];
    }
    return 0;
}
void handleEvent(int pipeFd)
{
    int netFd;
    while(1){
        recvFd(pipeFd,&netFd);
        char buf[1024] = {0};
        recv(netFd,buf,sizeof(buf),0);
        puts(buf);
        send(netFd,"Echo",4,0);
        close(netFd);
        pid_t pid = getpid();
        send(pipeFd,&pid,sizeof(pid_t),0);
    }
}
//TCP初始化相关代码
int tcpInit(char *ip, char *port, int *pSockFd)
{
    *pSockFd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr;
    bzero(&addr, sizeof(struct sockaddr_in));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr(ip);
    addr.sin_port = htons(atoi(port));
    int reuse = 1;
    int ret;
    ret = setsockopt(*pSockFd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse));
    ERROR_CHECK(ret, -1, "setsockopt");
    ret = bind(*pSockFd, (struct sockaddr *)&addr, sizeof(struct sockaddr_in));
    ERROR_CHECK(ret, -1, "bind");
    listen(*pSockFd, 10);
    return 0;
}
//epoll相关代码
int epollCtor(){
```

```
    int epfd = epoll_create(1);
    ERROR_CHECK(epfd,-1,"epoll_create");
    return epfd;
}
int epollAdd(int fd,int epfd){
    struct epoll_event event;
    event.events = EPOLLIN;
    event.data.fd = fd;
    int ret = epoll_ctl(epfd,EPOLL_CTL_ADD,fd,&event);
    ERROR_CHECK(ret,-1,"epoll_ctl add");
    return 0;
}
int epollDel(int fd, int epfd){
    int ret = epoll_ctl(epfd,EPOLL_CTL_DEL,fd,NULL);
    ERROR_CHECK(ret,-1,"epoll_ctl del");
    return 0;
}
```

在上述例子中，我们实现了一个最基本的进程池：客户端读取标准输入，发送给服务端，服务端回复一个相同的内容。

12.3 文件的传输

文件传输的本质实现上和 cp命令的原理是一样：应用程序需要打开源文件并且进行读取，然后将读取得到的内容写入到目标文件当中。如果是远程上传/下载文件，则需要将前述流程分解成两个应用程序，应用程序之间使用网络传输数据。

12.3.1 小文件传输和小火车

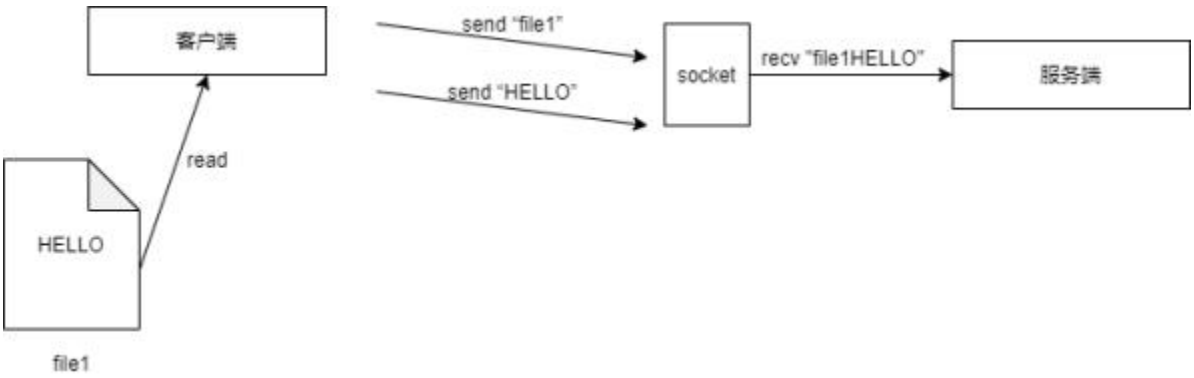
我们首先来看小文件的传输，所谓的小文件，就是指单次 send和 recv就能发送/接收完成的文件。如果一端要把文件发送给另一端，要发送两个部分的数据：其一是文件名，用于对端创建文件；另一个部分是文件内容。

假设是客户端将文件上传到服务端， 一种简单的实现方法是这样的：

```
//客户端
//...
    send(sockFd,filename,strlen(filename),0);
    ret = read(fd,buf,sizeof(buf))
    send(sockFd,buf,ret,0);
//...

//服务端
//...
    recv(netFd,filename,sizeof(filename),0);
    int fd = open(filename,O_RDONLY|O_CREAT,0666);
    ret = recv(netFd,buf,sizeof(buf),0);
    write(fd,buf,ret);
//...
```

但是这种写法会引入一个非常严重的问题，服务端在接收文件名，实际上并不知道有多长，所以它会试图把网络缓冲区的所有内容都读取出来，但是 send底层基于的协议是TCP协议——这是一种流式协议。这样的情况下，服务端没办法区分到底是哪些部分是文件名而哪些部分是文件内容。完全可能会出现服务端把文件名和文件内容混杂在一起的情况，这种就是江湖中所谓的"粘包"问题。



所以接下要我们要做的事情是在应用层上构建一个私有协议，这个协议的目的是规定TCP发送和接收的实际长度从而确定单个消息的边界。目前这个协议非常简单，可以把它看成是一个小火车，包括一个火车头和一个火车车厢。火车头里面存储一个整型数字，描述了火车车厢的长度，而火车车厢才是真正承载数据的部分。

```
typedef struct train_s{
    int size;
    char buf[1000];
} train_t;
```

下面就是使用小火车传递文件名，然后再传递文件内容的例子：

```
//接收文件
int recvFile(int netFd)
{
    train_t t;
    //先接收文件名长度
    recv(netFd, &t.dataLength, sizeof(int), 0);
    //再接收文件名
    recv(netFd, t.buf, t.dataLength, 0);
    //接收方创建一个同名文件
    int fd = open(t.buf, O_WRONLY | O_CREAT, 0666);
    ERROR_CHECK(fd, -1, "open");
    //读取网络并写入到同名文件中 -- 小文件
    int ret = recv(netFd,&t.dataLength,sizeof(int),0);
    ERROR_CHECK(ret,-1,"recv");
    ret = recv(netFd,t.buf,t.dataLength,0);
    ERROR_CHECK(ret,-1,"recv");
    write(fd, t.buf, t.dataLength);
    close(fd);
}
//发送文件
int transFile(int netFd){
    train_t t = {5,"file1"};
    //先发送文件名长度
    send(netFd,&t.dataLength,sizeof(int), 0);
    //再发送文件名
    send(netFd,t.buf,t.dataLength,0);
    int fd = open(t.buf,O_RDONLY);
    ERROR_CHECK(fd,-1,"open");
    bzero(&t,sizeof(t));
    int ret = read(fd,t.buf,sizeof(t.buf));
    t.dataLength = ret;
    send(netFd,&t.dataLength,sizeof(int),0);
    send(netFd,t.buf,t.dataLength,0);
```

```
    return 0;
}
```

12.3.2 大文件传输

当文件的内容大小少于小火车车厢的时候，上述代码的表现是非常完美的。但是如果一旦文件长度大于火车车厢大小，那么上述代码就无能为力了。

最自然的思路解决大文件问题就是使用循环机制：发送方使用一个循环来读取文件内容，每当读取一定字节的数据之后，将这些数据的大小和内容填充进小火车当中；接收方就不断的使用 `recv`接收小火车的火车头和车厢，先读取4个字节的火车头，再根据车厢长度接收后续内容。

```
//!!!!!!!!!!!注意!!!!!!!!!!!
//这是有问题的代码
int transFile(int netFd){
    train_t t = {5,"file2"};
    send(netFd,&t,4+5,0);
    int fd = open(t.buf,O_RDONLY);
    ERROR_CHECK(fd,-1,"open");
    bzero(&t,sizeof(t));
    while(1){
        t.dataLength = read(fd,t.buf,sizeof(t.buf));
        ERROR_CHECK(t.dataLength,-1,"read");
        if(t.dataLength != sizeof(t.buf)){
            printf("t.dataLength = %d\n",t.dataLength);
        }
        if(t.dataLength == 0){
            break;
        }
        send(netFd,&t,sizeof(int)+t.dataLength,0);
    }
    //结束的时候发送一个车厢为0的小火车
    t.dataLength = 0;
    send(netFd,&t,4,MSG_NOSIGNAL);
    close(fd);
    return 0;
}
int recvFile(int netFd)
{
    train_t t;
    bzero(&t,sizeof(t));
    //先接收文件名长度
    recv(netFd, &t.dataLength, sizeof(int), 0);
    //再接收文件名
    recv(netFd, t.buf, t.dataLength, 0);
    //接收方创建一个同名文件
    int fd = open(t.buf, O_WRONLY | O_CREAT, 0666);
    ERROR_CHECK(fd, -1, "open");
    int ret;
    while (1)
    {
        bzero(&t,sizeof(t));
        recv(netFd,&t.dataLength,sizeof(int),0);
        if(t.dataLength != sizeof(t.buf)){
            printf("dataLength = %d\n", t.dataLength);
        }
        if(t.dataLength <= 0){
```

```
                break;
            }
            recv(netFd,t.buf,t.dataLength,0);
            write(fd,t.buf,t.dataLength);
        }
        close(fd);
    }
```

在文件大小比较小的时候，上述的代码表现良好，但是一旦文件增大，上面的代码表现就会出现问题了，而且还是不稳定出现问题：

- 最基本的问题发送的文件和接收到的文件大小不一致。
- 另外一个问题就是服务端可能会出现死循环。

12.3.3 忽略SIGPIPE信号

首先我们来解决死循环的问题，这个死循环的表现是服务端的 `epoll_wait`总是有文件描述符处于就绪状态，这个就绪的文件描述符就是工作进程的管道。为什么这个会管道总是就绪呢？通过 `ps -elf`命令，我们可以查看所有进程的状态，会发现该工作进程处于“僵尸”状态，“僵尸”状态出现的原因是该工作进程终止但是父进程并未回收资源，所以接下来的问题就是什么原因导致工作进程终止了？

进入探查后，我们发现进程终止的原因是收到了信号 `SIGPIPE` ，产生这个信号的原因是服务端往已经关闭的网络socket中写入数据了。解决这个问题有两种方案， 一种是使用 `signal`或者 `sigaction`忽略这个信号；另一种是给 `send` 的最后一个参数加上 `MSG_NOSIGNAL` 选项，这样进程也可以忽略信号。

12.3.4 让recv取出所有数据

解决了 `SIGPIPE` 信号的问题之后，工作进程不会异常终止了，但是还会存在另一个问题，就是传递的数据和实际的数据不一致的问题。这个问题的根源其实是 `recv` ——调用 `recv` 的时候，需要传入一个整型的长度参数，但是遗憾的是，这个长度参数是描述的是最大的长度，而实际 `recv` 的长度可能并没有达到最大的长度——因为TCP是一种流式协议，它只能负责每个报文可靠有序地发送和接收，但是并不能保证传输到网络缓冲区当中的就是完整的一个小火车。

这样就有可能会到导致数据读取问题，下面就举一个例子：假设发送方需要传输两个小火车，其中每个车厢都是1000个字节，那么自然火车头都是4个字节，里面各自存储了1000 （当然是二进制形式），当两个小火车发送到socket的时候，由于TCP是流式协议，所以小火车与小火车之间边界就不见了，到了接收方这边， `recv`可能会先收到4个字节确定第一个小火车的车厢长度，再收到800字节，此时继续再`recv`就会从第一个火车车厢中继续取出4个字节，那这4个字节显然就不是第二个小火车的车厢长度了。

为了解决这个问题，一种解决方案就是给 `recv` 函数设置 `MSG_WAITALL` 属性，这样的话， `recv`在不遇到EOF或者异常关闭的情况就能一定把最大长度数据读取出来。

12.3.5 检查文件的正确性

现在我们已经能够成功地将一个文件从一方传输到另一方了，而两方看到的文件大小是完全一致的。但是，如何判断这两个文件的内容是否是一致的呢？如果都是文本文件，那么可以使用`vimdiff`比较，但是也只能局限在一台主机上面。最好的解决方式是使用md5码来解决问题。

`md5`是一种摘要散列算法。它的行为类似于之前的哈希函数，传入任何一个文件都能得到一个128bit的数据，可以用16个十六进制数来描述，这个数据称为md5码。如果两个文件的文件内容是一致的话，那么生成的md5码必然是一致的；如果两个文件的内容不同，只有极小的可以忽略不计的概率两个文件的md5码一致。如果需要生成一个文件的md5码，需要使用命令`md5sum`。

```
# client
$md5sum file2
# 计算md5码需要等待一段时间
8e9d11a16f03372c82c5134278a0bd7d  file2

# server
$md5sum file2
8e9d11a16f03372c82c5134278a0bd7d  file2
```

通过上述例子，我们就知道两个文件的内容是完全一致的了。

> 使用truncate命令可以生成指定大小的文件
>
> `truncate -s 600M file2`生成

12.3.6 封装recv MSG_WAITALL

为了简化recv的使用，可以考虑将之前带有`MSG_WAITALL`参数的`recv`封装成`recvn`，这样可以方便进一步使用。

```
int recvn(int netFd,void* pstart,int len)
{
    int total=0;
    int ret;
    char *p=(char*)pstart;
    while(total<len)
    {
        ret=recv(netFd,p+total,len-total,0);
        total+=ret;//每次接收到的字节数加到total上
    }
    return 0;
}
```

12.4 进程池1.0

下面是一个能够正确下载任意大小文件的进程的代码实现：

12.4.1 客户端

```
//client.c
typedef struct train_s
{
    int dataLength;
    char buf[1000];
```

```c
} train_t;
int recvFile(int netFd);
int recvn(int netFd,void* pstart,int len);
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc, 3);
    int sockFd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr;
    bzero(&addr, sizeof(struct sockaddr_in));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr(argv[1]);
    addr.sin_port = htons(atoi(argv[2]));
    int ret = connect(sockFd, (struct sockaddr *)&addr,
                      sizeof(struct sockaddr_in));
    ERROR_CHECK(ret, -1, "connect");
    recvFile(sockFd);
    close(sockFd);
    return 0;
}
int recvFile(int netFd)
{
    train_t t;
    bzero(&t,sizeof(t));
    //先接收文件名长度
    recvn(netFd, &t.dataLength, sizeof(int));
    //再接收文件名
    recvn(netFd, t.buf, t.dataLength);
    //接收方创建一个同名文件
    int fd = open(t.buf, O_WRONLY | O_CREAT, 0666);
    ERROR_CHECK(fd, -1, "open");
    while (1)
    {
        bzero(&t,sizeof(t));
        recvn(netFd,&t.dataLength,sizeof(int));
        if(0 == t.dataLength){
            break;
        }
        recvn(netFd,t.buf,t.dataLength);
        write(fd,t.buf,t.dataLength);
    }
    close(fd);
}
int recvn(int netFd,void* pstart,int len)
{
    int total=0;
    int ret;
    char *p=(char*)pstart;
    while(total<len)
    {
        ret=recv(netFd,p+total,len-total,0);
        total+=ret;//每次接收到的字节数加到total上
    }
    return 0;
}
```

12.4.2 服务端

```
//main.c
int main(int argc, char *argv[]){
    //./server 192.168.30.129 8080 3
    ARGS_CHECK(argc,4);
    int workerNum = atoi(argv[3]);
    processData_t *workerList = (processData_t
*)calloc(sizeof(processData_t),workerNum);
    makeChild(workerList,workerNum);
    int sockFd;
    tcpInit(argv[1],argv[2],&sockFd);
    int epfd = epollCtor();
    epollAdd(sockFd,epfd);
    for(int i = 0;i < workerNum; ++i){
        epollAdd(workerList[i].pipeFd,epfd);
    }
    int listenSize = workerNum+1;//socket+每个进程pipe的读端
    struct epoll_event * readylist = (struct epoll_event
*)calloc(listenSize,sizeof(struct epoll_event));
    while(1){
        int readynum = epoll_wait(epfd,readylist,listenSize,-1);
        for(int i = 0;i < readynum; ++i){
            if(readylist[i].data.fd == sockFd){
                puts("accept ready");
                int netFd = accept(sockFd,NULL,NULL);
                for(int j = 0;j < workerNum; ++j){
                    if(workerList[j].status == FREE){
                        printf("No. %d worker gets his job, pid = %d\n", j,
workerList[j].pid);
                        sendFd(workerList[j].pipeFd, netFd);
                        workerList[j].status = BUSY;
                        break;
                    }
                }
                close(netFd);
            }
            else{
                puts("One worker finish his task!");
                int j;
                for(j = 0;j < workerNum;++j){
                    if(workerList[j].pipeFd == readylist[i].data.fd){
                        pid_t pid;
                        int ret =
recv(workerList[j].pipeFd,&pid,sizeof(pid_t),0);
                        printf("No. %d worker finish, pid = %d\n",j,pid);
                        workerList[j].status = FREE;
                        break;
                    }
                }
            }
        }
    }
}
//worker.c
int makeChild(processData_t *pProcssData, int processNum)
{
```

```
    pid_t pid;
    for (int i = 0; i < processNum; ++i)
    {
        int pipeFd[2];
        socketpair(AF_LOCAL,SOCK_STREAM,0,pipeFd);
        pid = fork();
        if (pid == 0)
        {
            close(pipeFd[0]);
            handleEvent(pipeFd[1]);
        }
        close(pipeFd[1]);
        printf("pid = %d, pipefd[0] = %d\n", pid, pipeFd[0]);
        pProcssData[i].pid = pid;
        pProcssData[i].status = FREE;
        pProcssData[i].pipeFd = pipeFd[0];
    }
    return 0;
}
void handleEvent(int pipeFd)
{
    int netFd;
    while(1){
        recvFd(pipeFd,&netFd);
        transFile(netFd);
        close(netFd);
        pid_t pid = getpid();
        send(pipeFd,&pid,sizeof(pid_t),0);
    }
}
//tcpInit.c
int tcpInit(char *ip, char *port, int *pSockFd)
{
    *pSockFd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr;
    bzero(&addr, sizeof(struct sockaddr_in));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr(ip);
    addr.sin_port = htons(atoi(port));
    int reuse = 1;
    int ret;
    ret = setsockopt(*pSockFd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse));
    ERROR_CHECK(ret, -1, "setsockopt");
    ret = bind(*pSockFd, (struct sockaddr *)&addr, sizeof(struct sockaddr_in));
    ERROR_CHECK(ret, -1, "bind");
    listen(*pSockFd, 10);
    return 0;
}
//epollFunc.c
int epollCtor(){
    int epfd = epoll_create(1);
    ERROR_CHECK(epfd,-1,"epoll_create");
    return epfd;
}
int epollAdd(int fd,int epfd){
    struct epoll_event event;
    event.events = EPOLLIN;
    event.data.fd = fd;
```

```
    int ret = epoll_ctl(epfd,EPOLL_CTL_ADD,fd,&event);
    ERROR_CHECK(ret,-1,"epoll_ctl add");
    return 0;
}
int epollDel(int fd, int epfd){
    int ret = epoll_ctl(epfd,EPOLL_CTL_DEL,fd,NULL);
    ERROR_CHECK(ret,-1,"epoll_ctl del");
    return 0;
}
//sendFd.c
int sendFd(int pipeFd, int fdToSend){
    struct msghdr hdr;
    bzero(&hdr,sizeof(struct msghdr));
    struct iovec iov[1];
    char buf[] = "Hello";
    iov[0].iov_base = buf;
    iov[0].iov_len = 5;
    hdr.msg_iov = iov;
    hdr.msg_iovlen = 1;
    struct cmsghdr *pcmsghdr = (struct cmsghdr
*)calloc(1,sizeof(CMSG_LEN(sizeof(int))));
    pcmsghdr->cmsg_len = CMSG_LEN(sizeof(int));
    //控制信息的数据部分只有int类型的文件描述符
    pcmsghdr->cmsg_level = SOL_SOCKET;
    pcmsghdr->cmsg_type = SCM_RIGHTS;//SCM->socket-level control message
    //表示在socket层传递的是访问权力，这样接受进程就可以访问对应文件对象了
    *(int *)CMSG_DATA(pcmsghdr) = fdToSend;
    //数据部分是文件描述符
    hdr.msg_control = pcmsghdr;
    hdr.msg_controllen = CMSG_LEN(sizeof(int));
    int ret = sendmsg(pipeFd,&hdr,0);
    ERROR_CHECK(ret,-1,"sendmsg");
}
int recvFd(int pipeFd, int *pFd){
    struct msghdr hdr;
    bzero(&hdr,sizeof(struct msghdr));
    struct iovec iov[1];
    char buf[6] = {0};
    iov[0].iov_base = buf;
    iov[0].iov_len = 5;
    hdr.msg_iov = iov;
    hdr.msg_iovlen = 1;
    struct cmsghdr *pcmsghdr = (struct cmsghdr *)calloc(1,sizeof(struct
cmsghdr));
    pcmsghdr->cmsg_len = CMSG_LEN(sizeof(int));
    //控制信息的数据部分只有int类型的文件描述符
    pcmsghdr->cmsg_level = SOL_SOCKET;
    pcmsghdr->cmsg_type = SCM_RIGHTS;//SCM->socket-level control message
    hdr.msg_control = pcmsghdr;
    hdr.msg_controllen = CMSG_LEN(sizeof(int));
    int ret = recvmsg(pipeFd,&hdr,0);
    ERROR_CHECK(ret,-1,"recvmsg");
    *pFd = *(int *)CMSG_DATA(pcmsghdr);
    return 0;
}
//transFile.c
int transFile(int netFd){
    train_t t = {5,"file2"};
```

```
    send(netFd,&t,4+5,MSG_NOSIGNAL);
    int fd = open(t.buf,O_RDONLY);
    ERROR_CHECK(fd,-1,"open");
    bzero(&t,sizeof(t));
    while(1){
        t.dataLength = read(fd,t.buf,sizeof(t.buf));
        ERROR_CHECK(t.dataLength,-1,"read");
        if(t.dataLength != sizeof(t.buf)){
            printf("t.dataLength = %d\n",t.dataLength);
        }
        if(t.dataLength == 0){
            bzero(&t,sizeof(t));
            send(netFd,&t,4,MSG_NOSIGNAL);
            break;
        }
        int ret = send(netFd,&t,sizeof(int)+t.dataLength,MSG_NOSIGNAL);
        if(ret == -1){
            perror("send");
            break;
        }
    }
    close(fd);
    return 0;
}
```

12.5 进程池的其他功能

12.5.1 进度条显示

首先服务端需要传输一个文件的大小给客户端，以便客户端计算百分比。客户端也需要先接收一个长度的小火车，再读取文件内容，在显示的时候需要控制换行的显示，可以使用 `fflush`清空缓冲区。

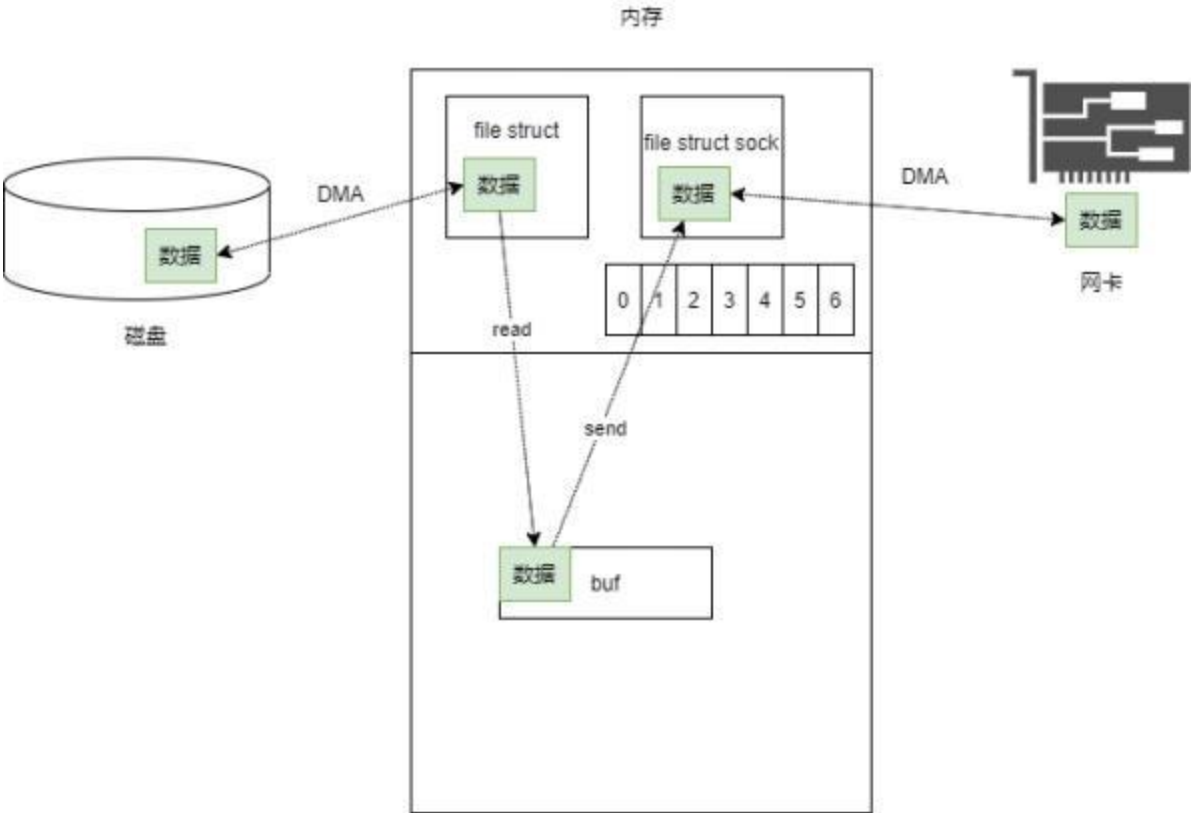
```
//服务端略
//下面是客户端
    //...
    off_t fileSize;
    bzero(&t,sizeof(t));
    recvn(netFd,&t.dataLength, sizeof(int));
    recvn(netFd,&fileSize,t.dataLength);
    printf("fileSize = %ld\n", fileSize);
    off_t doneSize = 0;
    off_t lastSize = 0;
    off_t slice = fileSize/100;
    int percentage = 0;
    while (1)
    {
        bzero(&t,sizeof(t));
        recvn(netFd,&t.dataLength,sizeof(int));
        if(0 == t.dataLength){
            break;
        }
        doneSize += t.dataLength;
        if(doneSize-lastSize >= slice){
            printf("%5.2lf%%\r", 100.0*doneSize/fileSize);
            fflush(stdout);
            lastSize = doneSize;
```

```
        }
        recvn(netFd,t.buf,t.dataLength);
        write(fd,t.buf,t.dataLength);
    }
    printf("100.00%%\n");
    //...
```

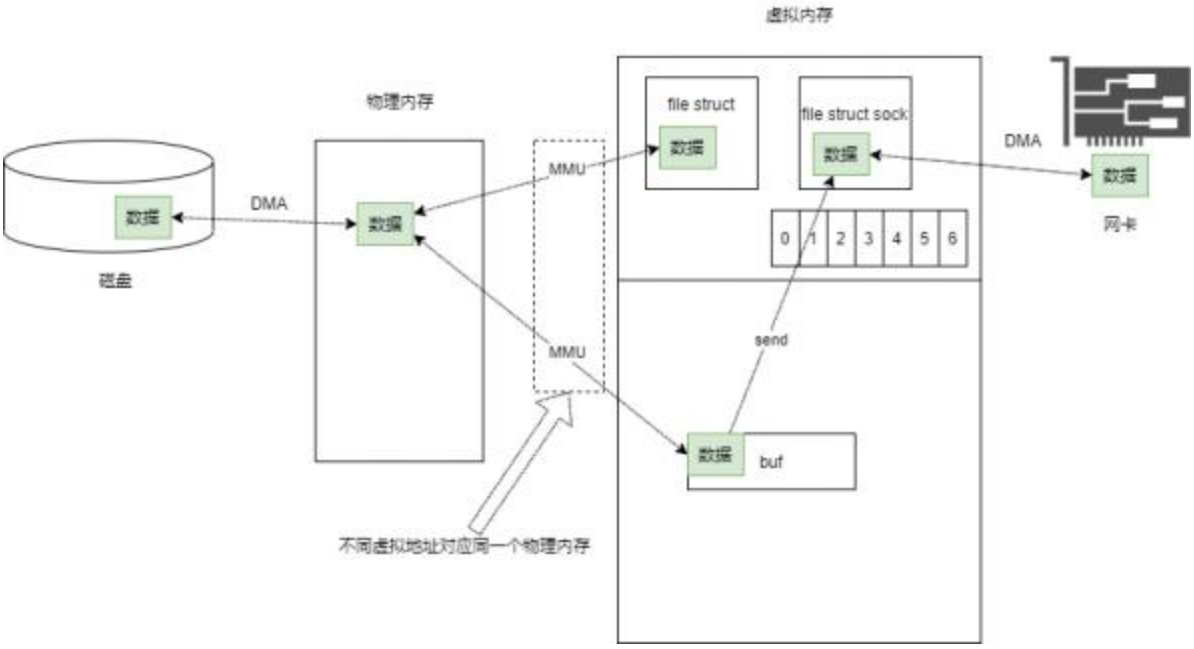
12.5.2 零拷贝、sendfile和splice

目前我们传输文件的时候是采用 `read`和 `send`来组合完成，这种当中的数据流向是怎么样的呢？首先打开一个普通文件，数据会从磁盘通过DMA设备传输到内存，即文件对象当中的内核缓冲区部分，然后调用 `read`数据会从内核缓冲区拷贝到一个用户态的buf上面（buf是 `read` 函数的参数），接下来调用 `send` ，就将数据拷贝到了网络发送缓存区，最终实现了文件传输。

但是实际上这里涉及了大量的不必要的拷贝操作，比如下图中 `read`和 `send` 的过程：



如何减少从内核文件缓冲区到用户态空间的拷贝呢？解决方案就是使用 `mmap`系统调用直接建立文件和用户态空间buf的映射。这样的话数据就减少了一次拷贝。在非常多的场景下都会使用 `mmap`来减少拷贝次数，典型的就是使用图形的应用去操作显卡设备的显存。除此以外，这种传输方式也可以减少由于系统调用导致的CPU用户态和内核态的切换次数。



下面是实现的代码：

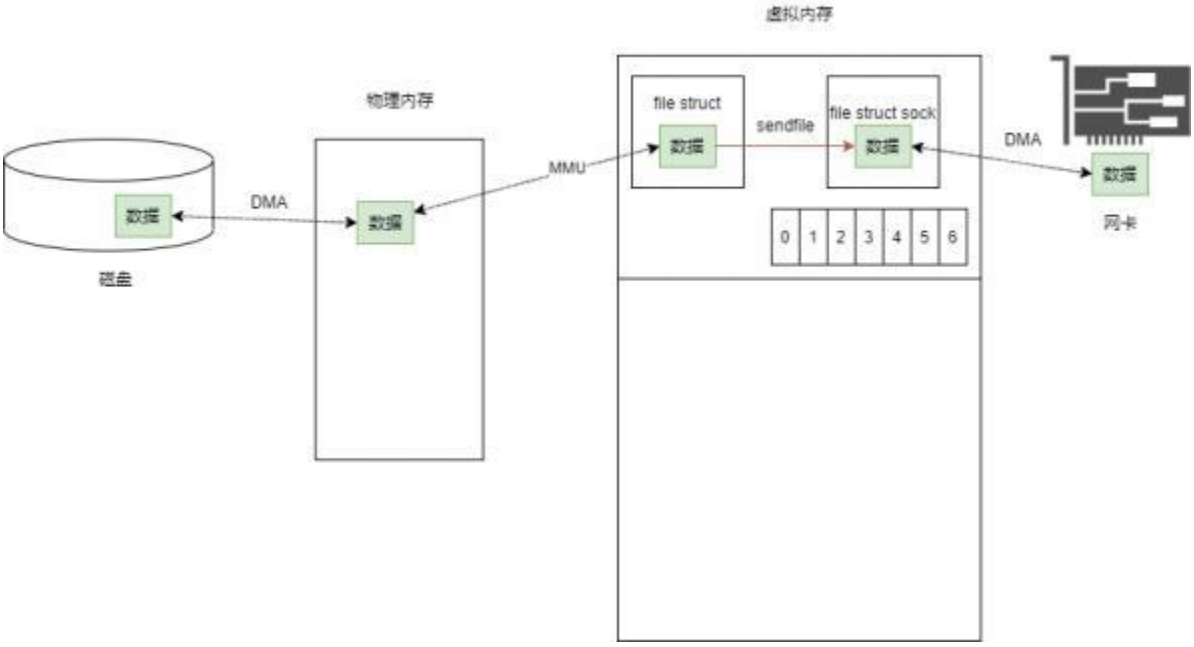
```
//客户端
int recvFile(int netFd)
{
    train_t t;
    bzero(&t,sizeof(t));
    //先接收文件名长度
    recvn(netFd, &t.dataLength, sizeof(int));
    //再接收文件名
    recvn(netFd, t.buf, t.dataLength);
    //接收方创建一个同名文件
    int fd = open(t.buf, O_RDWR| O_CREAT, 0666);
    ERROR_CHECK(fd, -1, "open");
    off_t fileSize;
    bzero(&t,sizeof(t));
    recvn(netFd,&t.dataLength, sizeof(int));
    recvn(netFd,&fileSize,t.dataLength);
    printf("fileSize = %ld\n", fileSize);
    /* case 1 分批接收
    off_t doneSize = 0;
    off_t lastSize = 0;
    off_t slice = fileSize/100;
    int percentage = 0;
    while (1)
    {
        bzero(&t,sizeof(t));
        recvn(netFd,&t.dataLength,sizeof(int));
        if(0 == t.dataLength){
            break;
        }
        doneSize += t.dataLength;
        if(doneSize-lastSize >= slice){
            printf("%5.2lf%%\r", 100.0*doneSize/fileSize);
            fflush(stdout);
            lastSize = doneSize;
        }
        recvn(netFd,t.buf,t.dataLength);
       write(fd,t.buf,t.dataLength);
```

```
    }
    */
    //case 1一次性接收完 注意此时客户端需要修改
    ftruncate(fd,fileSize);
    //前面open的权限需要改成O_RDWR
    char *p = (char *)mmap(NULL,fileSize,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
    ERROR_CHECK(p,MAP_FAILED,"mmap");
    recvn(netFd,p,fileSize);
    printf("100.00%%\n");
    munmap(p,fileSize);
    close(fd);
}
//服务端 分批发送
int transFile(int netFd){//mmap_multi
    train_t t = {5,"file2"};
    send(netFd,&t,4+5,MSG_NOSIGNAL);
    int fd = open(t.buf,O_RDONLY);
    ERROR_CHECK(fd,-1,"open");
    struct stat statbuf;
    int ret = fstat(fd,&statbuf);
    bzero(&t,sizeof(t));
    t.dataLength = sizeof(statbuf.st_size);
    memcpy(t.buf,&statbuf.st_size,t.dataLength);
    send(netFd,&t,sizeof(off_t)+4, MSG_NOSIGNAL);
    char *p = (char *)mmap(NULL,statbuf.st_size,PROT_READ,MAP_SHARED,fd,0);
    ERROR_CHECK(p,(void *)-1, "mmap");
    off_t total = 0;
    while(total < statbuf.st_size){
        if(statbuf.st_size - total > sizeof(t.buf)){
            t.dataLength = sizeof(t.buf);
        }
        else{
            t.dataLength = statbuf.st_size - total;
        }
        memcpy(t.buf,p+total,t.dataLength);
        total += t.dataLength;
        int ret = send(netFd,&t,sizeof(int)+t.dataLength,MSG_NOSIGNAL);
        if(ret == -1){
            perror("send");
            break;
        }
    }
    //发送结束标志
    t.dataLength = 0;
    send(netFd,&t,4,MSG_NOSIGNAL);
    munmap(p,statbuf.st_size);
    close(fd);
    return 0;
}
//服务端 一次性发送
int transFile(int netFd){// mmap_once
    train_t t = {5,"file2"};
    send(netFd,&t,4+5,MSG_NOSIGNAL);
    int fd = open(t.buf,O_RDONLY);
    ERROR_CHECK(fd,-1,"open");
    struct stat statbuf;
    int ret = fstat(fd,&statbuf);
    bzero(&t,sizeof(t));
```

```
    t.dataLength = sizeof(statbuf.st_size);
    memcpy(t.buf,&statbuf.st_size,t.dataLength);
    send(netFd,&t,sizeof(off_t)+4, MSG_NOSIGNAL);
    char *p = (char *)mmap(NULL,statbuf.st_size,PROT_READ,MAP_SHARED,fd,0);
    ERROR_CHECK(p,(void *)-1, "mmap");
    send(netFd,p,statbuf.st_size,MSG_NOSIGNAL);
    //发送结束标志
    t.dataLength = 0;
    send(netFd,&t,4,MSG_NOSIGNAL);
    munmap(p,statbuf.st_size);
    close(fd);
    return 0;
}
```

> 有兴趣的同学可以统计下使用read&write、分多次mmap和单次mmap的时间消耗情况。

使用 `mmap`系统调用只能减少数据从磁盘文件的文件对象到用户态空间的拷贝，但是依然无法避免从用户态到内核已连接套接字的拷贝（因为网络设备文件对象不支持 `mmap`）。 `sendfile`系统调用可以解决这个问题，它可以使数据直接在内核中传递而不需要经过用户态空间，调用 `sendfile`系统调用可以直接将磁盘文件的文件对象的数据直接传递给已连接套接字文件对象，从而直接发送到网卡设备之上（在内核的底层实现中，实际上是让内核磁盘文件缓冲区和网络缓冲区对应同一片物理内存）。




```
#include <sys/sendfile.h>
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

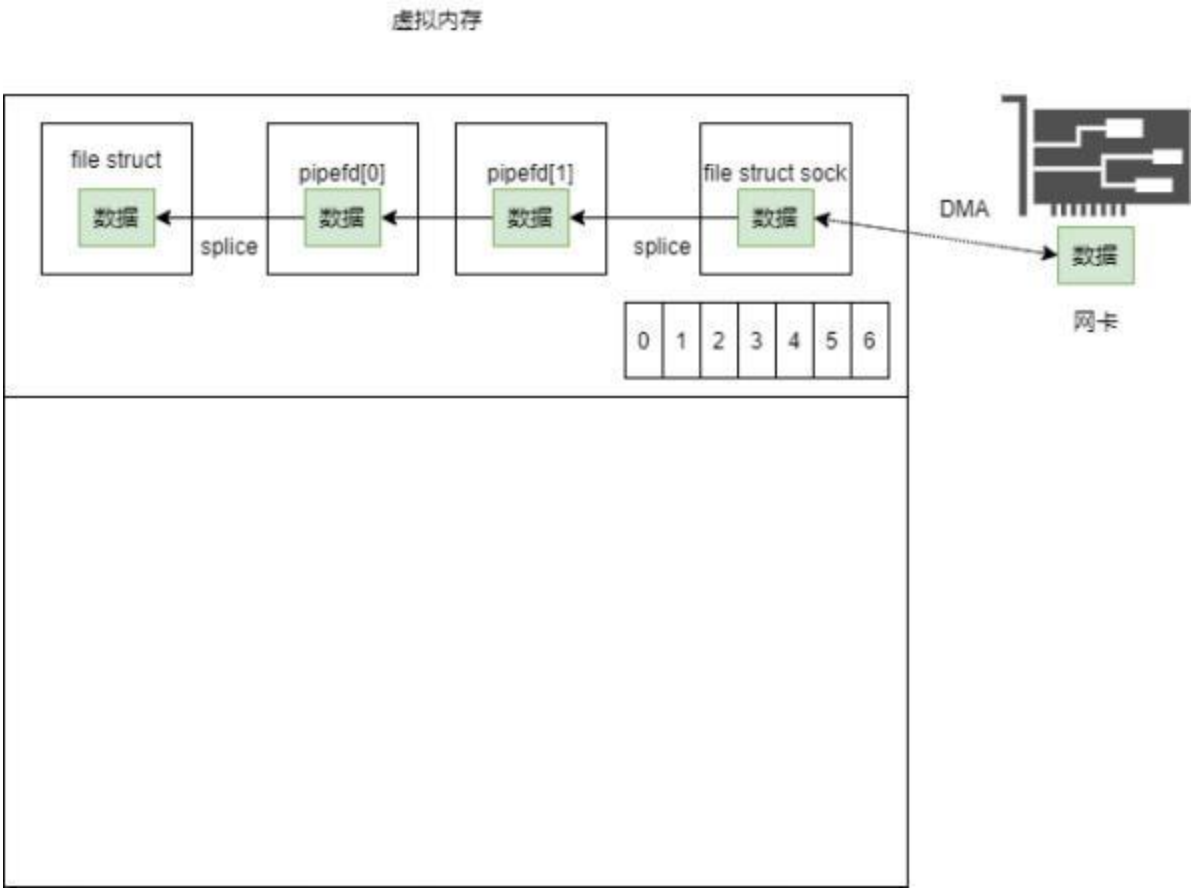
使用 `sendfile` 的时候要特别注意， `out_fd`一般只能填写网络套接字的描述符，表示写入的文件描述符， `in_fd`一般是一个磁盘文件，表示读取的文件描述符。从上述的需求可以得知， `sendfile`只能用于发送文件方的零拷贝实现，无法用于接收方，并且发送文件的大小上限是2GB。下面是使用 `sendfile` 的发送方的例子。

```
int transFile(int netFd){
    train_t t = {5,"file2"};
    send(netFd,&t,4+5,MSG_NOSIGNAL);
    int fd = open(t.buf,O_RDONLY);
    ERROR_CHECK(fd,-1,"open");
    struct stat statbuf;
    int ret = fstat(fd,&statbuf);
```

```
    bzero(&t,sizeof(t));
    t.dataLength = sizeof(statbuf.st_size);
    memcpy(t.buf,&statbuf.st_size,t.dataLength);
    send(netFd,&t,sizeof(off_t)+4, MSG_NOSIGNAL);
    //发送结束标志
    sendfile(netFd,fd,NULL,statbuf.st_size);
    t.dataLength = 0;
    send(netFd,&t,4,MSG_NOSIGNAL);
    close(fd);
    return 0;
}
```

考虑到 `sendfile`只能将数据从磁盘文件发送到网络设备中，那么接收方如何在避免使用 `mmap` 的情况下使用零拷贝技术呢？ 一种方式就是采用管道配合 `splice` 的做法。 `splice`系统调用可以直接将数据从内核管道文件缓冲区发送到另一个内核文件缓冲区，也可以反之，将一个内核文件缓冲区的数据直接发送到内核管道缓冲区中。所以只需要在内核创建一个匿名管道，这个管道用于本进程中，在磁盘文件和网络文件之间无拷贝地传递数据。

```
ssize_t splice(int fd_in, loff_t *off_in, int fd_out,
                      loff_t *off_out, size_t len, unsigned int flags);
```



```
//...
    recvn(netFd,&t.dataLength, sizeof(int));
    recvn(netFd,&fileSize,t.dataLength);
    printf("fileSize = %ld\n", fileSize);
    int pipefds[2];
    pipe(pipefds);
    int total = 0;
    while(total < fileSize){
        int ret = splice(netFd,NULL,pipefds[1],NULL,4096,SPLICE_F_MORE);
        total += ret;
        splice(pipefds[0],NULL,fd,NULL,ret,SPLICE_F_MORE);
    }
//...
```

12.5.3 进程池的退出

进程池的简单退出

进程池的简单退出要实现功能很简单，就是让父进程收到信号之后，再给每个子进程发送信号使其终止，这种实现方案只需要让父进程在一个目标信号（通常是10信号 `SIGUSR1` ）的过程给目标子进程发送信号即可。

在实现的过程需要注意的是 `signal` 函数和 `fork` 函数之间调用顺序，因为父进程会修改默认递送行为，而子进程会执行默认行为，所以 `fork`应该要在 `signal` 的后面调用。

```
processData_t *workerList;//需要改成全局变量
int workerNum;
void sigFunc(int signum){
    printf("signum = %d\n", signum);
    for(int i = 0; i < workerNum; ++i){
        kill(workerList[i].pid,SIGUSR1);
    }
    for(int i = 0; i < workerNum; ++i){
        wait(NULL);
    }
    puts("process pool is over!");
    exit(0);
}
int main(){
    //..
    makeChild(workerList,workerNum);
    signal(SIGUSR1,sigFunc);
    //注意fork和signal的顺序
}
```

使用管道通知工作进程终止

采用信号就不可避免要使用全局变量，因为信号处理函数当中只能存储有限的信息，有没有办法避免全局的进程数量和进程数组呢？ 一种解决方案就是采取“异步拉起同步”的策略：虽然还是需要创建一个管道全局变量，但是该管道只用于处理进程池退出，不涉及其他的进程属性。这个管道的读端需要使用IO多路复用机制管理起来，而当信号产生之后，主进程递送信号的时候会往管道中写入数据，此时可以依靠 `epoll` 的就绪事件，在事件处理中来完成退出的逻辑。

```
int pipeFd[2];
void sigFunc(int signum){
```

```
    printf("signum = %d\n",signum);
    write(pipeFd[1],"1",1);
}
int main(){
    //...
    pipe(pipeFd);
    epollAdd(pipeFd[0],epfd);
    //...
    //...epoll就绪事件处理
    else if(readylist[i].data.fd == pipeFd[0]){
                for(int j = 0; j < workerNum; ++j){
                    kill(workerList[j].pid,SIGINT);
                    puts("send signal to worker!");
                }
                for(int j = 0; j < workerNum; ++j){
                    wait(NULL);
                }
                printf("Parent process exit!\n");
                exit(0);
            }
    //...
}
```

进程池的优雅退出

上述的退出机制存在一个问题，就是即使工作进程正在传输文件中，父进程也会通过信号将其终止。如何实现进程池在退出的时候，子进程要完成传输文件的工作之后才能退出呢?

一种典型的方案是使用 `sigprocmask`在文件传输的过程中设置信号屏蔽字，这样可以实现上述的机制。

另一种方案就是调整 `sendFd` 的设计，每个工作进程在传输完文件之后总是循环地继续下一个事件，而在每个事件处理的开始，工作进程总是会调用 `recvFd`来使自己处于阻塞状态直到有事件到达。我们可以对进程池的终止作一些调整：用户发送信号给父进程表明将要退出进程池；随后父进程通过 `sendFd` 给所有的工作进程发送终止的信息，工作进程在完成了一次工作任务了之后就会 `recvFd`收到进程池终止的信息，然后工作进程就可以主动退出；随着所有的工作进程终止，父进程亦随后终止，整个进程池就终止了。

```
int sendFd(int pipeFd, int fdToSend, int exitFlag){
    struct msghdr hdr;
    bzero(&hdr,sizeof(struct msghdr));
    struct iovec iov[1];
    iov[0].iov_base = &exitFlag;
    iov[0].iov_len = sizeof(int);
    hdr.msg_iov = iov;
    hdr.msg_iovlen = 1;
//...
}
int recvFd(int pipeFd, int *pFd, int *exitFlag){
    struct msghdr hdr;
    bzero(&hdr,sizeof(struct msghdr));
    struct iovec iov[1];
    iov[0].iov_base = exitFlag;
    iov[0].iov_len = sizeof(int);
    hdr.msg_iov = iov;
    hdr.msg_iovlen = 1;
//.....
}
```

```
void handleEvent(int pipeFd)
{
    int netFd;
    while(1){
        int exitFlag;
        recvFd(pipeFd,&netFd,&exitFlag);
        if(exitFlag == 1){
            puts("I am closing!");
            exit(0);
        }
        //...
    }
}
//... main函数中的epoll时间循环
        for(int i = 0;i < readynum; ++i){
            if(readylist[i].data.fd == sockFd){
                puts("accept ready");
                int netFd = accept(sockFd,NULL,NULL);
                for(int j = 0;j < workerNum; ++j){
                    if(workerList[j].status == FREE){
                        printf("No. %d worker gets his job, pid = %d\n",
                            j, workerList[j].pid);
                        sendFd(workerList[j].pipeFd, netFd, 0);
                        workerList[j].status = BUSY;
                        break;
                    }
                }
                close(netFd);
            }
            else if(readylist[i].data.fd == exitpipeFd[0]){
                for(int j = 0; j < workerNum; ++j){
                    puts("set exitFlag to worker!");
                    sendFd(workerList[j].pipeFd,0,1);
                }
                for(int j = 0; j < workerNum; ++j){
                    wait(NULL);
                }
                printf("Parent process exit!\n");
                exit(0);
            }
//....
```

12.6 线程池的实现

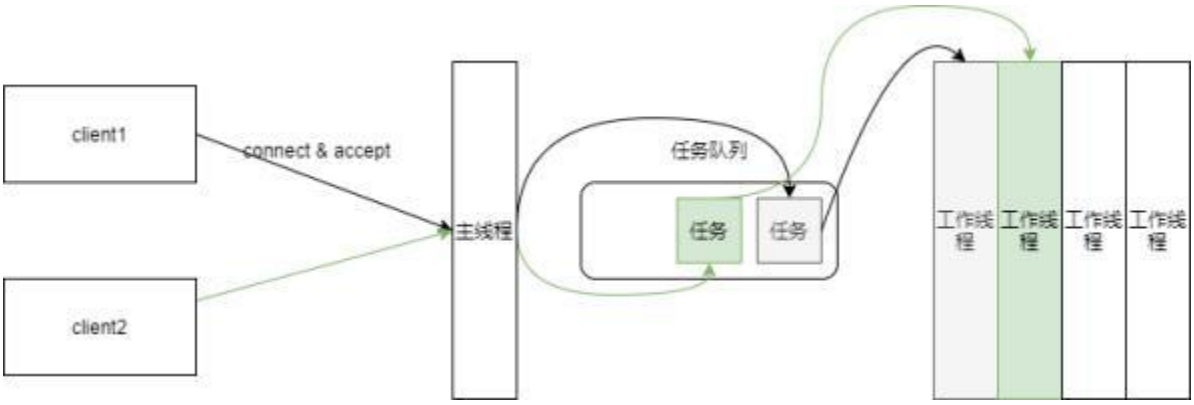
12.6.1 从进程池到线程池

使用进程池的思路来解决并发连接是一种经典的基于事件驱动模型的解决方案，但是由于进程天生具有隔离性，导致进程之间通信十分困难， 一种优化的思路就是用线程来取代进程，即所谓的线程池。

由于多线程是共享地址空间的，所以主线程和工作线程天然地通过共享文件描述符数值的形式共享网络文件对象，但是这种共享也会带来麻烦：每当有客户端发起请求时，主线程会分配一个空闲的工作线程完成任务，而任务正是在多个线程之间共享的资源，所以需要采用一定的互斥和同步的机制来避免竞争。

我们可以将任务放在一个队列中，该队列就称为任务队列，那任务队列就成为多个线程同时访问的共享资源，此时问题就转化成了 一个典型的生产者-消费者问题：任务队列中的任务就是商品，主线程是生产者，每当有连接到来的时候，就将一个任务放入任务队列，即生产商品，而各个工作线程就是消费者，每当队列中任务到来的时 候，就负责取出任务并执行。

下面是线程池的基本设计方案：



首先，我们先设计数据结构：

```
//任务节点
typedef struct task_s{
    int peerfd;
    struct task_s * pNext;
}task_t;

//阻塞式任务队列
typedef struct task_queue_s
{
    task_t * pFront;  //队列的头节点
    task_t * pRear;   //队列的尾节点
    int queSize;      //记录当前任务的数量
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int flag;//0 表示要退出，1 表示不退出
}task_queue_t;

//线程池
typedef struct threadpool_s {
    pthread_t * pthreads;  //线程id数组的首地址
    int pthreadNum;        //线程数量
    task_queue_t que;      //任务队列
}threadpool_t;

//任务队列的相应操作
int queueInit(task_queue_t * que);
int queueDestroy(task_queue_t * que);
int queueIsEmpty(task_queue_t * que);
int taskSize(task_queue_t * que);
int taskEnque(task_queue_t * que, int peerfd);
int taskDeque(task_queue_t * que);
int broadcastALL(task_queue_t* que);

//线程池的相应操作
int threadpoolInit(threadpool_t *, int num);
int threadpoolDestroy(threadpool_t *);
int threadpoolStart(threadpool_t *);
int threadpoolStop(threadpool_t *);
```

```
//实现任务队列的相关操作
int queueInit(task_queue_t * que) { //队列的初始化
    if(que) {
        que->pFront = NULL;
        que->pRear = NULL;
        que->queSize = 0;
```

```
        que->flag = 1;
        int ret = pthread_mutex_init(&que->mutex, NULL);
        THREAD_ERROR_CHECK(ret, "pthread_mutex_init");

        ret = pthread_cond_init(&que->cond, NULL);
        THREAD_ERROR_CHECK(ret, "pthread_cond_init");
    }
    return 0;
}

//队列的销毁
int queueDestroy(task_queue_t * que)
{
    if(que) {
        int ret = pthread_mutex_destroy(&que->mutex);
        THREAD_ERROR_CHECK(ret, "pthread_mutex_destroy");

        ret = pthread_cond_destroy(&que->cond);
        THREAD_ERROR_CHECK(ret, "pthread_cond_destroy");
    }
    return 0;
}

//判断队列是否为空
int queueIsEmpty(task_queue_t * que)
{
    return que->queSize == 0;
}

int taskSize(task_queue_t * que)
{
    return que->queSize;
}

//添加一个任务
int taskEnque(task_queue_t * que, int peerfd)
{
    task_t * pNode = (task_t *)calloc(1, sizeof(task_t));
    pNode->peerfd = peerfd;
    pNode->pNext = NULL;
    int ret = pthread_mutex_lock(&que->mutex);
    THREAD_ERROR_CHECK(ret, "pthread_mutex_lock");
    if(queueIsEmpty(que)) {
        que->pFront = que->pRear = pNode;
    } else {//不为空
        que->pRear->pNext = pNode;
        que->pRear = pNode;
    }
    que->queSize++;
    ret = pthread_mutex_unlock(&que->mutex);
    THREAD_ERROR_CHECK(ret, "pthread_mutex_unlock");
    //通知消费者取任务
    ret = pthread_cond_signal(&que->cond);
    THREAD_ERROR_CHECK(ret, "pthread_cond_signal");
    return 0;
}

//从任务队列中获取一个任务
int taskDeque(task_queue_t * que)
{
    int ret = pthread_mutex_lock(&que->mutex);
    THREAD_ERROR_CHECK(ret, "pthread_mutex_lock");
    int peerfd = -1;
    while(que->flag && queueIsEmpty(que)) {//虚假唤醒
        pthread_cond_wait(&que->cond, &que->mutex);
    }
    if(que->flag) {
        peerfd = que->pFront->peerfd;
```

```
        //元素出队操作
        task_t * pDelete = que->pFront;
        if(taskSize(que) == 1) {
            que->pFront = que->pRear = NULL;
        } else {
            que->pFront = que->pFront->pNext;
        }
        free(pDelete);
        que->queSize--;
    } else {
        peerfd = -1;
    }
    ret = pthread_mutex_unlock(&que->mutex);
    THREAD_ERROR_CHECK(ret, "pthread_mutex_unlock");
    return peerfd;
}
```

随后，设计与线程池相关的函数：

```
//线程池初始化
int threadpoolInit(threadpool_t * pthreadpool, int num)
{
    pthreadpool->pthreadNum = num;
    pthreadpool->pthreads = calloc(num, sizeof(pthread_t));
    queueInit(&pthreadpool->que);

    return 0;
}
//线程池销毁
int threadpoolDestroy(threadpool_t * pthreadpool)
{
    free(pthreadpool->pthreads);
    queueDestroy(&pthreadpool->que);
    return 0;
}
//启动线程池
int threadpoolStart(threadpool_t * pthreadpool){
    if(pthreadpool) {
        for(int i = 0; i < pthreadpool->pthreadNum; ++i) {
            int ret = pthread_create(&pthreadpool->pthreads[i],
                            NULL,threadFunc, pthreadpool);
            THREAD_ERROR_CHECK(ret, "pthread_create");
        }
    }
    return 0;
}
```

接下来，就是main函数中的实现了。

```
#define EPOLL_ARR_SIZE 100
int main(int argc, char ** argv)
{   //ip,port,threadNum
    ARGS_CHECK(argc, 4);

    threadpool_t threadpool;
    memset(&threadpool, 0, sizeof(threadpool));

    //初始化线程池
    threadpoolInit(&threadpool, atoi(argv[3]));

    //启动线程池
    threadpoolStart(&threadpool);

}
```

当调用了threadpoolStart函数以后，每一个子线程都运行起来了。初始情况下，因为队列中还没有任务，每一个子线程都阻塞在taskDeque函数上。

```
void * threadFunc(void* arg) {
    //不断地从任务队列中获取任务，并执行
    threadpool_t * pThreadpool = (threadpool_t*)arg;
    while(1) {
        int peerfd = taskDeque(&pThreadpool->que);
        if(peerfd > 0) {
            //发送文件的任务
            transferFile(peerfd);
        } else {//peerfd为-1的情况
            break;
        }
    }
    printf("sub thread %ld is exiting.\n", pthread_self());
    return NULL;
}
```

作为服务器，主线程内还是要启动监听套接字listenfd，同时通过epoll来监听新连接的到来。主线程需要`accept`客户端的连接，并且需要将任务加入到任务队列。当任务队列中有任务之后，某个子线程就会被唤醒，从而去执行文件传输任务。

```
    //创建监听套接字
    int listenfd = tcpInit(argv[1], argv[2]);

    //创建epoll实例
    int epfd = epoll_create1(0);
    ERROR_CHECK(epfd, -1, "epoll_create1");
    //对listenfd进行监听
    addEpollReadfd(epfd, listenfd);
    struct epoll_event * pEventArr = (struct epoll_event*)
        calloc(EPOLL_ARR_SIZE, sizeof(struct epoll_event));
    while(1) {
        int nready = epoll_wait(epfd, pEventArr, EPOLL_ARR_SIZE, -1);
        if(nready == -1 && errno == EINTR) {
            continue;
        } else if(nready == -1) {
            ERROR_CHECK(nready, -1, "epoll_wait");
        } else {
            //大于0
            for(int i = 0; i < nready; ++i) {
                int fd = pEventArr[i].data.fd;
                if(fd == listenfd) {//对新连接进行处理
                    int peerfd = accept(listenfd, NULL, NULL);
                    //需要将peerfd加入到任务队列中
                    taskEnque(&threadpool.que, peerfd);
                }
             }
        }
    }
```

12.6.2 线程池的退出

编程规范要求，信号机制不能应用在多线程应用中——其主要的原因时当多线程进程执行过程中， 一旦产生了信号，则处理信号的线程是未知的，有可能是主线程也有可能是子线程。这样的话，线程终止的处理就非常繁琐，也不够清晰明了。

解决这个问题方式是在原来设计的基础上引入多进程机制：将线程池改造成一个父进程和一个子进程组成的应用程序。其中父进程负责递送信号，而子进程负责创建和运行线程池（也就是对应之前的已完成代码），父子进程之间通过管道通信。当信号的产生的时候，父进程递送该信号，并且在信号处理函数的执行过程中，写入一个消息给管道，此外，子进程会使用IO多路复用机制epoll监听管道，这样一旦管道就绪，子进程的主线程就可以得知程序将要被终止的信息，随后即可依次关闭子线程。

```
int exitPipe[2];

void sigHandler(int num)
{
    printf("\n sig is coming.\n");
    //激活管道，往管道中写一个1
    int one = 1;
    write(exitPipe[1], &one, sizeof(one));
}

int main(int argc, char ** argv)
{   //ip, port, threadNum
    ARGS_CHECK(argc, 4);
    //创建匿名管道
    pipe(exitPipe);
    //fork之后，将创建了子进程
    pid_t pid = fork();
    if(pid > 0) {//父进程
        close(exitPipe[0]);             //父进程关闭读端
        signal(SIGUSR1, sigHandler);  //只在父进程中处理10号信号
        wait(NULL);                     //等待子进程退出，回收其资源
        close(exitPipe[1]);
        printf("\nparent process exit.\n");
        exit(0);//父进程退出
    }
    //子进程
    close(exitPipe[1]);//子进程关闭写端

    addEpollReadfd(epfd, exitPipe[0]);

    struct epoll_event * pEventArr = (struct epoll_event*)
        calloc(EPOLL_ARR_SIZE, sizeof(struct epoll_event));
    while(1) {
        int nready = epoll_wait(epfd, pEventArr, EPOLL_ARR_SIZE, -1);
        if(nready == -1 && errno == EINTR) {
            continue;
        } else if(nready == -1) {
            ERROR_CHECK(nready, -1, "epoll_wait");
        } else {
            //大于0
            for(int i = 0; i < nready; ++i) {
                int fd = pEventArr[i].data.fd;
                if(fd == listenfd) {//对新连接进行处理
                    int peerfd = accept(listenfd, NULL, NULL);
                    //需要将peerfd加入到任务队列中
                    taskEnque(&threadpool.que, peerfd);
                } else if(fd == exitPipe[0]) {
                    //线程池要退出
                    int howmany = 0;
                    //对管道进行处理
                    read(exitPipe[0], &howmany, sizeof(howmany));
                    //主线程通知所有的子线程退出
                    threadpoolStop(&threadpool);
                    //子进程退出前，回收资源
```

```
                    threadpoolDestroy(&threadpool);
                    close(listenfd);
                    close(epfd);
                    close(exitPipe[0]);
                    printf("\nchild process exit.\n");
                    exit(0);
                }
            }
        }
    }
    return 0;
}
```

当epoll监听到exitPipe[0]的读事件时，表示线程池要退出了，其中要调用threadpoolStop函数

```
int threadpoolStop(threadpool_t * pthreadpool)
{
    //如果任务队列中还有任务，先等待一下，所有任务执行完毕之后
    //再发广播，退出
    while(!queueIsEmpty(&pthreadpool->que)) {
        sleep(1);//每一个线程都可以sleep
    }

    //发广播，通知所有的子线程退出
    broadcastALL(&pthreadpool->que);
    //回收所有子线程的资源
    for(int i = 0; i < pthreadpool->pthreadNum; ++i) {
        pthread_join(pthreadpool->pthreads[i], NULL);
    }
    return 0;
}
```

在threadpoolStop函数中，当所有的任务被执行完毕之后，由于每一个子线程都会阻塞在taskDeque函数中，等待任务的到来，如果想要该函数返回退出，则必须要通过broadcastALL函数，修改flag标志位的值，之后再唤醒条件变量。

```
//主线程调用
int broadcastALL(task_queue_t * que)
{
    //先修改要退出的标识位
    que->flag = 0;
    int ret = pthread_cond_broadcast(&que->cond);
    THREAD_ERROR_CHECK(ret, "pthread_cond_broadcast");

    return 0;
}
```

我们再来看看taskDeque函数的实现。

```
int taskDeque(task_queue_t * que)
{
    int ret = pthread_mutex_lock(&que->mutex);
    THREAD_ERROR_CHECK(ret, "pthread_mutex_lock");
    int peerfd = -1;
    while(que->flag && queueIsEmpty(que)) {//虚假唤醒
        pthread_cond_wait(&que->cond, &que->mutex);
    }
    if(que->flag) {
        peerfd = que->pFront->peerfd;
        //元素出队操作
        task_t * pDelete = que->pFront;
        if(taskSize(que) == 1) {
            que->pFront = que->pRear = NULL;
        } else {
            que->pFront = que->pFront->pNext;
        }
        free(pDelete);
```

```
        que->queSize--;
    } else {
        peerfd = -1;
    }
    ret = pthread_mutex_unlock(&que->mutex);
    THREAD_ERROR_CHECK(ret, "pthread_mutex_unlock");
    return peerfd;
}
```

由于flag的标志位被置为0了，那子线程在被唤醒时，while循环的条件就不满足了，因此taskDeque函数会返回-1，此时子线程执行的函数体得到的peerfd就为-1了，接下来会走else部分，那就会退出while循环，子线程就正常退出了。

```
void * threadFunc(void* arg) {
    //不断地从任务队列中获取任务，并执行
    threadpool_t * pThreadpool = (threadpool_t*)arg;
    while(1) {
        int peerfd = taskDeque(&pThreadpool->que);
        if(peerfd > 0) {
            //发送文件的任务
            transferFile(peerfd);
        } else {//peerfd为-1的情况
            break;
        }
    }
    printf("sub thread %ld is exiting.\n", pthread_self());
    return NULL;
}
```

每一个子线程退出，线程池也就退出了。