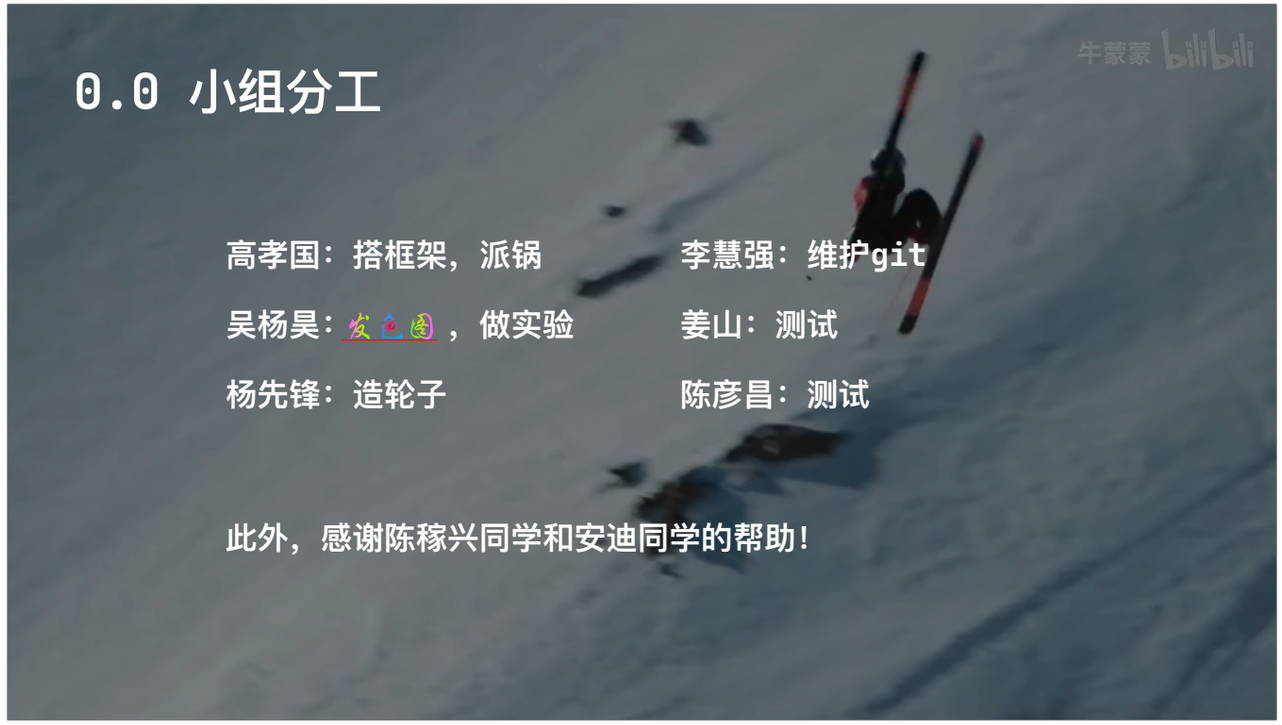
大家好，我们是NMSL小组。
从未有过如此美妙的开局。

0.0 Intro



我们小组的分工是比较明确的，本人前期主要负责搭框架，画图，把需求提给杨先锋同学实现，造轮子，把设想好的功能交给吴杨昊同学去做实验，两位同学做出了很大贡献。李慧强同学负责创建git和维护，其他同学负责评估测评。
此外，感谢陈稼兴和安迪同学的帮助，在讨论中我们获益良多。

0.1 目前完成的功能

用户：用户注册、登录、退出、Token

命令：exit、pwd、cd(cd fileName cd .. cd . cd /)、ls、mkdir、rmdir、rm、puts、gets

文件：服务端日志功能、断点续传、文件拷贝优化、上传秒传、超时断开(环形队列)

优化：客户端长短命令分离，充会员

0.2 未完成功能

多点下载

目前完成的功能除了多点下载都基本实现了，并加入了自己的一些设想。

1.0 功能演示

2.0 程序设计

演示之后我们讲一下程序设计。

2.0 建表

UserInfo: UNIQUE KEY `u_name` (`u_name`)

```
mysql> desc UserInfo;
+---------------+------------+------+-----+---------+----------------+
| Field         | Type       | Null | Key | Default | Extra          |
+---------------+------------+------+-----+---------+----------------+
| u_id          | int(11)    | NO   | PRI | NULL    | auto_increment |
| u_name        | varchar(20)| NO   | UNI | NULL    |                |
| u_salt        | char(20)   | NO   |     | NULL    |                |
| u_cryptpasswd | varchar(20)| NO   |     | NULL    |                |
| u_pwd         | varchar(20)| NO   |     | NULL    |                |
+---------------+------------+------+-----+---------+----------------+
5 rows in set (0.00 sec)
```

首先是建表

用户表设置了用户名唯一键

2.0 建表

FileTable: UNIQUE KEY `parent_id` (`parent_id`,`f_name`,`owner_id`,`f_type`)

```
mysql> desc FileTable;
+-----------+-------------+------+-----+---------+----------------+
| Field     | Type        | Null | Key | Default | Extra          |
+-----------+-------------+------+-----+---------+----------------+
| f_id      | int(11)     | NO   | PRI | NULL    | auto_increment |
| parent_id | int(11)     | NO   | MUL | NULL    |                |
| f_name    | varchar(20) | NO   |     | NULL    |                |
| owner_id  | int(11)     | NO   |     | NULL    |                |
| f_md5     | char(40)    | NO   |     | NULL    |                |
| f_size    | int(11)     | NO   |     | NULL    |                |
| f_type    | int(11)     | NO   |     | NULL    |                |
+-----------+-------------+------+-----+---------+----------------+
7 rows in set (0.01 sec)
```

文件表设置了父id，文件名，ownerid和文件类型四元组作为唯一键，保证在插入文件时就避免重复文件

2.0 建表

```
Log:CONSTRAINT `Log_ibfk_1` FOREIGN KEY (`u_id`)
REFERENCES `UserInfo` (`u_id`)
```

```
mysql> desc Log;
+---------+-------------+------+-----+---------+----------------+
| Field   | Type        | Null | Key | Default | Extra          |
+---------+-------------+------+-----+---------+----------------+
| l_id    | int(11)     | NO   | PRI | NULL    | auto_increment |
| u_id    | int(11)     | NO   | MUL | NULL    |                |
| u_name  | varchar(20) | NO   |     | NULL    |                |
| op_name | varchar(20) | NO   |     | NULL    |                |
| op_time | datetime    | NO   |     | NULL    |                |
+---------+-------------+------+-----+---------+----------------+
5 rows in set (0.00 sec)
```

日志表设置u_id作为外键

2.0 建表

```
UserToken:CONSTRAINT `u_id` FOREIGN KEY (`u_id`)
REFERENCES `UserInfo` (`u_id`)
```

```
mysql> desc UserToken;
+----------+--------------+------+-----+---------+----------------+
| Field    | Type         | Null | Key | Default | Extra          |
+----------+--------------+------+-----+---------+----------------+
| t_id     | int(11)      | NO   | PRI | NULL    | auto_increment |
| u_id     | int(11)      | NO   | MUL | NULL    |                |
| t_token  | varchar(300) | NO   |     | NULL    |                |
| t_expire | datetime     | NO   |     | NULL    |                |
+----------+--------------+------+-----+---------+----------------+
4 rows in set (0.00 sec)
```

Token表设置u_id作为外键

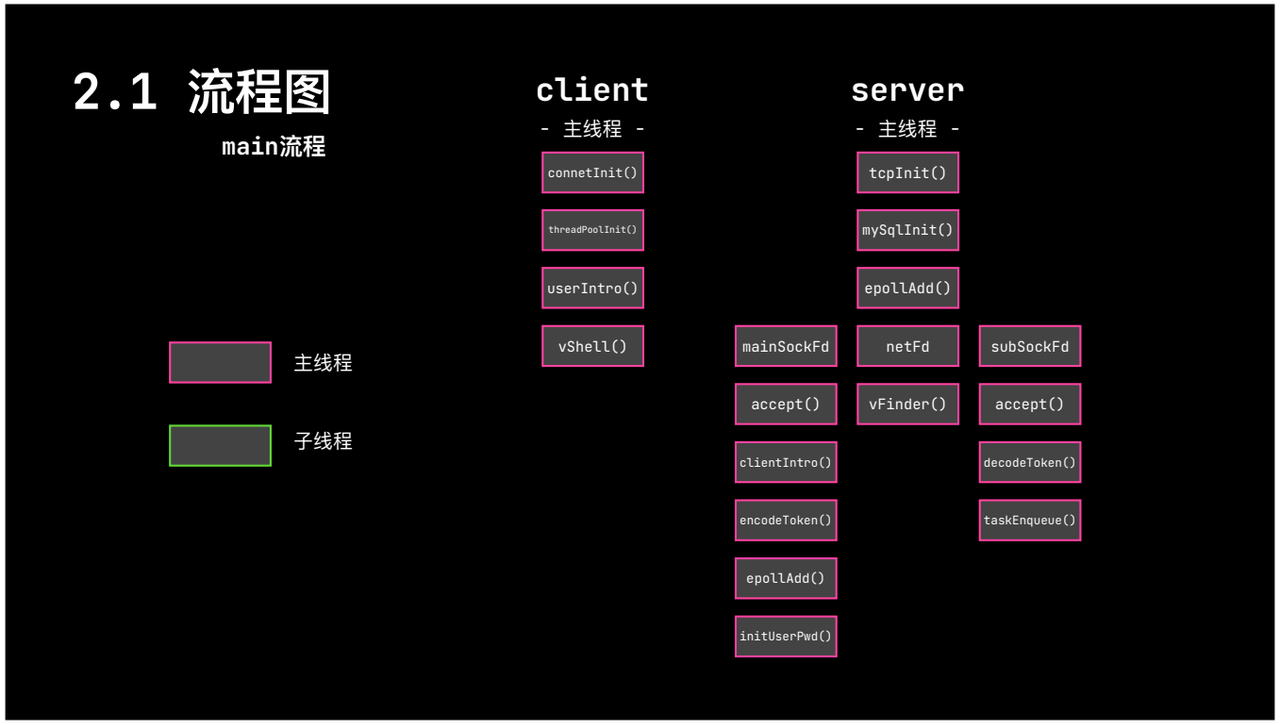
2.0 建表-轮子

```
// MYSQL
MYSQL* mySqlInit();

//用户表
int isUser(MYSQL* conn, const char* userName, char* salt, char* cryptpasswd);
int getUserId(MYSQL* conn, const char* userName);
void getRandStr(char* mystr);
int addUser(MYSQL* conn, const char* userName, const char* userPasswd);

//文件表
int addHomeDir(MYSQL* conn, char* userName);
int initUserPwd(MYSQL* conn, char* userName);
int updateUserPwd(MYSQL* conn, const char* userName, char* curPwd);
int myLastWd(char* buf, char* lines[]);
int getPwdFId(MYSQL* conn, const int userId);
int getFileInfo(MYSQL* conn, const int userId, const char* fileName, const int fileType, const int parent_id,
char* md5);
int isFileExistInPwd(MYSQL* conn, const char* fileName, int userId, int curF_id, int fileType);
int isFileExistInTable(MYSQL* conn, const char* fileName, char* md5);
int DelFileFromTable(MYSQL* conn, const int userId, const char* fileName, const int fileType, const int
parent_id);
int addFileToTable(MYSQL* conn, const int userId, const char* fileName, const char* md5, const int fileType,
const int parent_id);
```

键表的同时我们做了各类轮子，即之后程序实现过程中的查表行为

2.0 建表-轮子

```
//Token表
int encodeToken(char* userName, char* Token);
int decodeToken(char* userName, const char* JWT);
int InserIntoUserToken(MYSQL* conn, const int userId, const char* userToken);
int delFromUserToken(MYSQL* conn, const int userId);
int expireUserToken(MYSQL* conn, const int userId);

//日志表
int addToLog(MYSQL* conn, const int user_id, const char* user_name, const char* op_name);
```



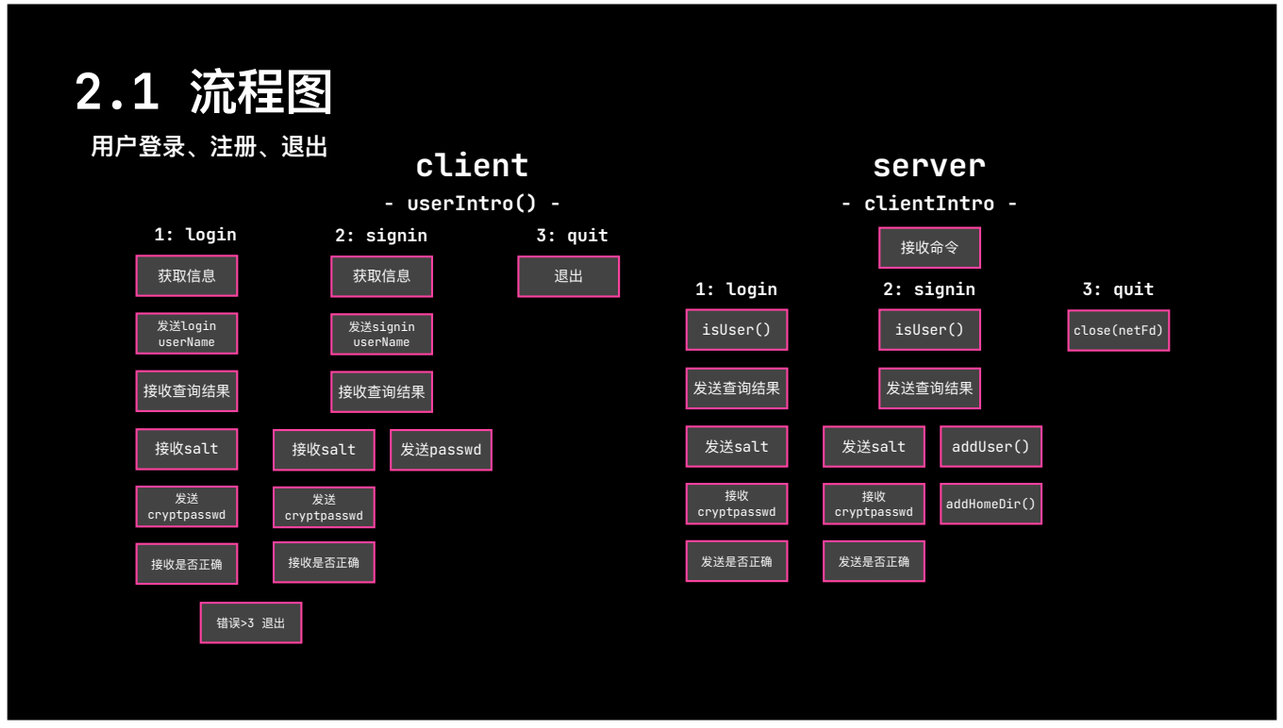
之后我们做了一系列的流程图，与老师的流程图不同的是，我们采取了线性的流程，方便程序实现。

如图，客户端的核心程序是userIntro和vShell，分别对应服务端的clientIntro和vFinder，实现客户登录系统和命令交互系统。

接下来的粉红色矩形代表主线程执行，绿色矩形代表子线程执行。

2.1 流程图

用户登录、注册、退出



接下来分别对这两个系统进行设计。

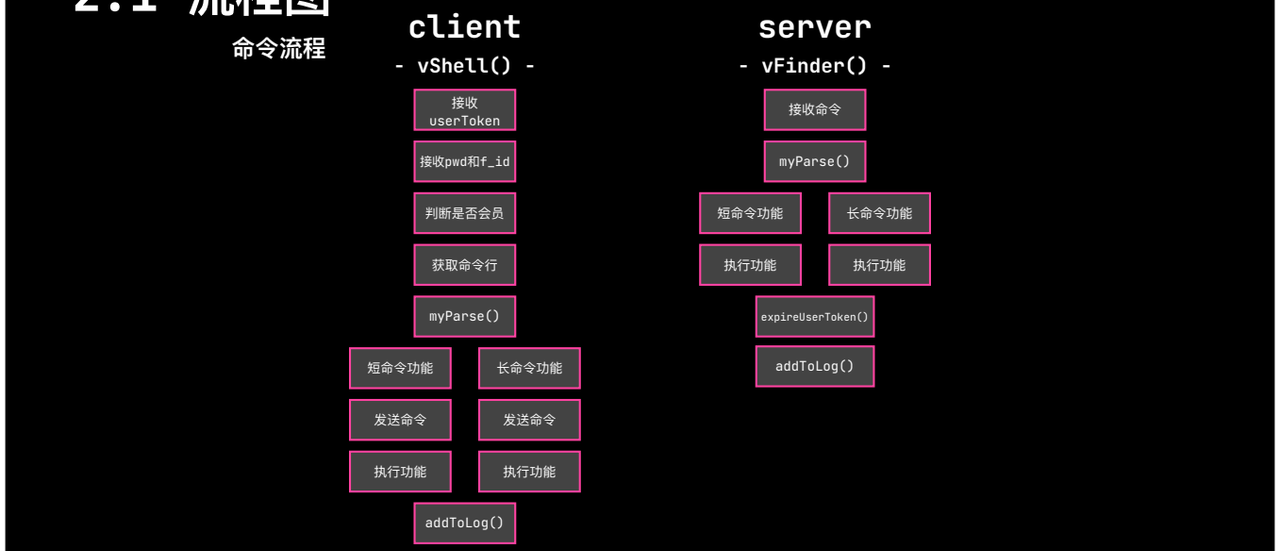
首先是用户登录系统，分为登录、注册和退出。

登录时，客户端发送用户名，接收salt，生成cryptpasswd发到服务端，服务端发送验证结果。验证3次失败后即自动退出。

注册时也是类似的过程，对于已有用户套用登录，对于新用户发送用户名和密码，服务器生成salt和cryptpasswd加入表中，同时用户进入登录状态。

2.1 流程图

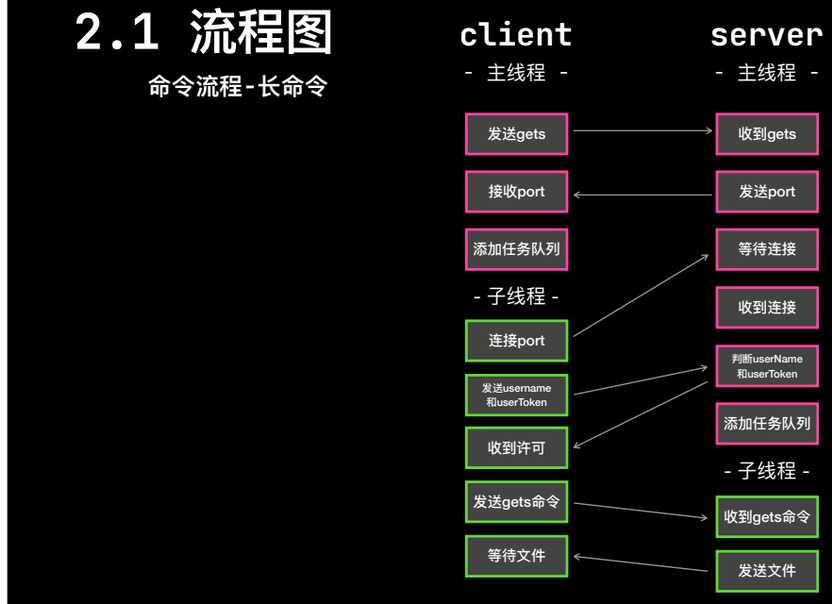
命令流程



无论是长命令还是短命令，客户端和服务端执行命令的流程是类似的，同样是传输命令、myParse分析命令，主线程自己或交给子线程执行，执行完毕后更新Token表和日志表。

2.1 流程图

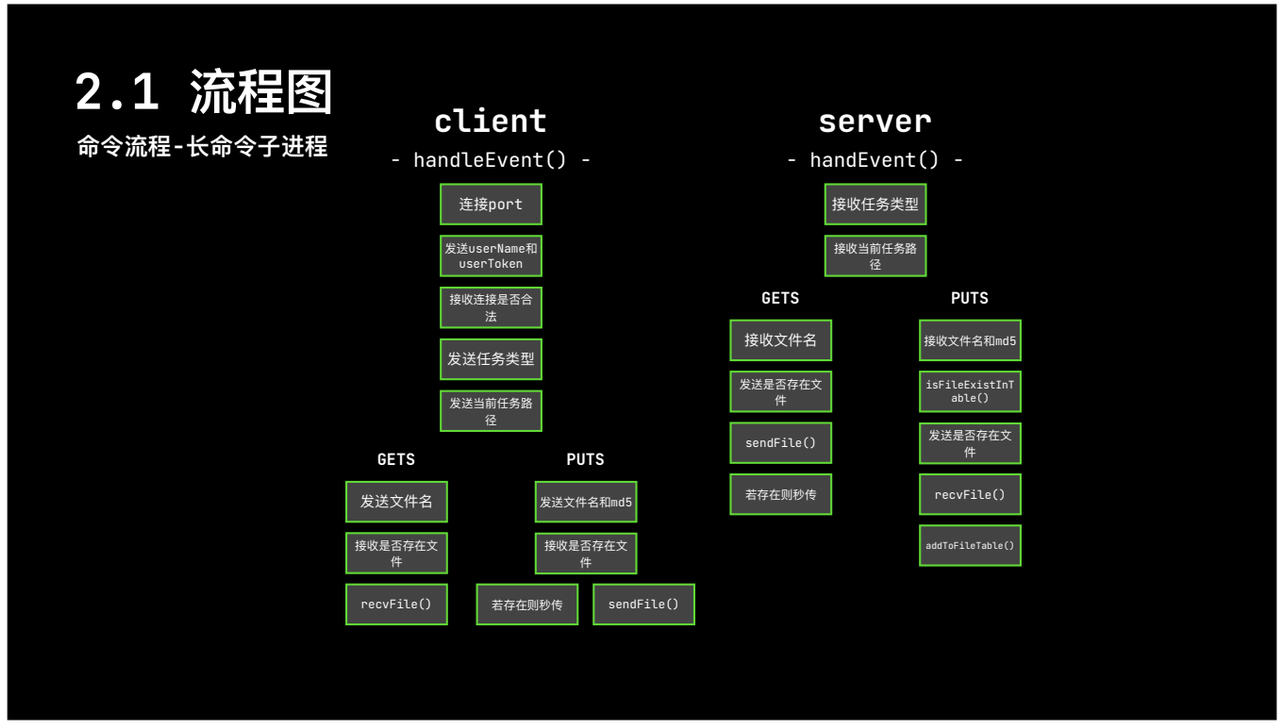
命令流程-长命令



首先分析长命令的执行，以gets为例，客户端发送gets命令后，从服务端接收端口，此时客户端创建子线程，令子线程尝试通过该端口连接服务端的主线程，并发送用户名和token，客户端验证通过后也创建子线程，客户端子线程再次发送gets命令和其他参数，和服务端子线程交互完成文件传输。

2.1 流程图

命令流程-长命令子进程



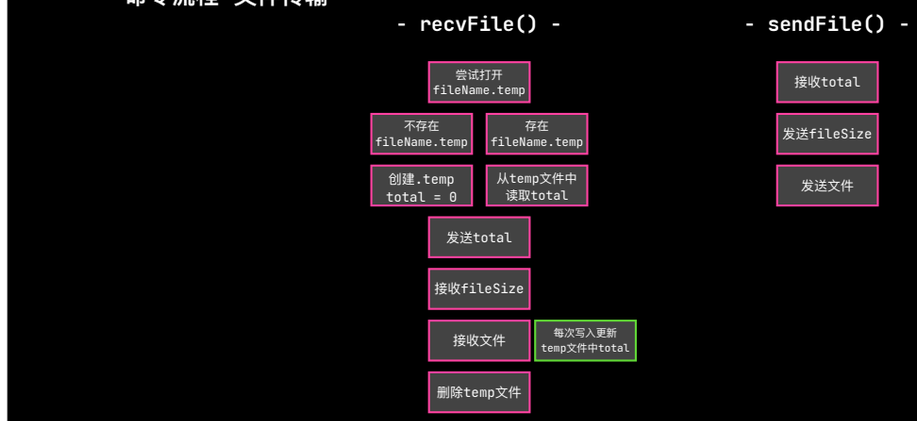
GETS和PUTS命令分别会进行不同的判断

下载时，客户端仅需发送文件名，通过建表时的四元组可以保证用户下载到唯一的文件。

上传时，客户端需要发送文件名和md5，服务端据此查询表中是否有同样的文件，有则秒传，并加入文件表

2.1 流程图

命令流程-文件传输



下载时，客户端会额外创建temp文件，保存下载进度，即指针。

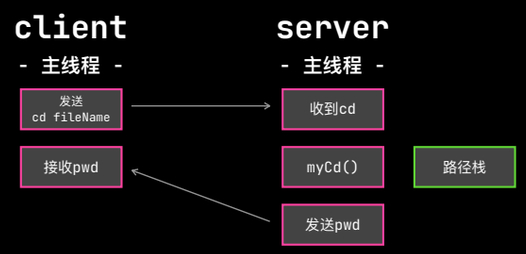
如果当前文件目录下存在temp文件，先读取temp文件内的指针，发给服务端同步指针，实现断点续传功能。

若不存在temp文件则创建fileName.temp

下载完毕后删除temp文件。

2.1 流程图

短命令流程



短命令的流程非常简单，一发一收即可。

值得一提的是，我们的cd命令采用了路径栈，方便用户频繁的切换目录和跳跃式的返回。

3.0 代码实现

3.0 实现-线程池

```c
/*任务结构体*/
typedef struct task_s {
    int port;  //传递连接端口
    char userName[BUFFERSIZE]; //传递userName
    char userToken[BUFFERSIZE]; //传递userToken
    int taskType;               //传递任务类型
    char fileName[BUFFERSIZE];  //传递文件名
    int curF_id;
    struct task_s* pNext;
} task_t;

/*线程池*/
typedef struct threadPool_s {
    pthread_t* tid;  //子线程的数组
    int threadNum;   //子线程数量
    taskQueue_t taskQueue;
    int exitFlag;
} threadPool_t;
```

```c
/*任务类型，GETS为0，PUTS为1*/
enum TASKTYPE {
    GETS,
    PUTS
};

/*任务队列链表*/
typedef struct taskQueue_s {
    task_t* pFront;  //队首
    task_t* pRear;   //队尾
    int size;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} taskQueue_t;
```

接下来是代码实现，线程池的数据结构包括任务结构体、线程池、任务类型和任务队列链表。
大家师出同门，基本都是采用了泥鳅老师提供的线程池模型，就不展开讲了。

3.0 实现-线程池

```
/*任务队列入队*/
int taskEnqueue(taskQueue_t* pTaskQueue, int netFd, MYSQL* conn, char* userName) {
    task_t* pTask = (task_t*)calloc(1, sizeof(task_t));

    pTask->netFd = netFd;
    pTask->conn = conn;
    strcpy(pTask->userName, userName);

    if (pTaskQueue->size == 0) {
        pTaskQueue->pFront = pTask;
        pTaskQueue->pRear = pTask;
    }
    else {
        pTaskQueue->pRear->pNext = pTask;
        pTaskQueue->pRear = pTask;
    }

    pTaskQueue->size++;
    return 0;
}
```

3.0 实现-线程池

```c
/*线程池初始化*/
int threadPoolInit(threadPool_t* pThreadPool, int workerNum){
    pThreadPool->threadNum = workerNum;
    pThreadPool->tid=(pthread_t*)calloc(workerNum,
sizeof(pthread_t));
    pThreadPool->taskQueue.pFront = NULL;
    pThreadPool->taskQueue.pRear = NULL;
    pThreadPool->taskQueue.size = 0;
    pthread_mutex_init(&pThreadPool->taskQueue.mutex, NULL);
    pthread_cond_init(&pThreadPool->taskQueue.cond, NULL);
    pThreadPool->exitFlag = 0;
}

/*任务队列出队*/
int taskDequeue(taskQueue_t* pTaskQueue){
    task_t* pCur = pTaskQueue->pFront;
    pTaskQueue->pFront = pCur->pNext;
    free(pCur);
    pTaskQueue->size--;
    return 0;
}
```

```c
/*任务队列入队*/
int taskEnqueue(taskQueue_t* pTaskQueue, int
netFd, MYSQL* conn, char* userName) {
    task_t* pTask = (task_t*)calloc(1,
sizeof(task_t));

    pTask->netFd = netFd;
    pTask->conn = conn;
    strcpy(pTask->userName, userName);

    if (pTaskQueue->size == 0) {
        pTaskQueue->pFront = pTask;
        pTaskQueue->pRear = pTask;
    }
    else {
        pTaskQueue->pRear->pNext = pTask;
        pTaskQueue->pRear = pTask;
    }

    pTaskQueue->size++;
    return 0;
}
```

3.1 环形队列

```c
/*用户信息结构体*/
typedef struct clientBox_s {
    int netFd;
    char userName[BUFFERSIZE];
    int userId;
    int slotIndex;
} clientBox_t;
```

```c
/*slot节点*/
typedef struct slotNode_s {
    int userId;
    int netFd;
    struct slotNode_s* next;
} slotNode_t;

/*环形队列slot链表*/
typedef struct slotList_s {
    slotNode_t* head;
    slotNode_t* tail;
    int size;
} slotList_t;
```

值得一讲的是环形队列，其实懂了之后就是一个相当简单的模型。

说是队列，其实只是一个链表数组。

方便起见，我创建了客户端Box结构体，存放每个连接的客户端所对应的用户名、id、netFd和接下来要用到的slotIndex。

每个链表数组中的链表结点存放了userId和netFd，其实只要放netFd就好，但我想用userId去查询。

3.1 环形队列

```c
/*初始化一个slotMap，slotMap是一个链表数组，每个元素都是一个链表的链表指针*/
int initSlotMap(slotList_t** slotMap, int timeOut) {
    for (int i = 0; i < timeOut; i++) {
        slotMap[i] = (slotList_t*)malloc(sizeof(slotList_t));
        slotMap[i]->head = NULL;
        slotMap[i]->tail = NULL;
        slotMap[i]->size = 0;
    }
}

/*将该timeIndex下的所有slotNode对应的netFd关闭*/
int clearSlot(int timeIndex, slotList_t** slotMap, clientBox_t* clients, int* pClientNum) {
    slotNode_t* curr = slotMap[timeIndex]->head;
    while (curr != NULL) {
        slotNode_t* next = curr->next;  // 保存curr后继结点
        printf("user %d 已超时，断开连接!\n", curr->userId);
        delFromClients(clients, curr->userId);
        free(curr);           //释放空间
        --(*pClientNum);
        curr = next;
    }
}
```

创建好数据结构之后，就应该造轮子了。

轮子的功能非常简单，就是普通的链表操作，initSlotMap初始化链表数组，clearSlot关闭某个链表中所有的netFd并释放结点。

3.1 环形队列

```c
/*在slotIndex处挂上slotNode，采用尾插法，成功返回0，失败返回-1*/
int attachToSlot(int userId, int netFd, int slotIndex, slotList_t** slotMap) {
    slotNode_t* newSlotNode = (slotNode_t*)malloc(sizeof(slotNode_t));
    if (newSlotNode == NULL) {
        printf("Error: malloc failed in add_before_head.\n");
        return -1;
    }

    //初始化节点
    newSlotNode->userId = userId;
    newSlotNode->netFd = netFd;

    //判断链表是否为空
    if (slotMap[slotIndex]->size == 0) {  //若为空
        slotMap[slotIndex]->head = newSlotNode;
        slotMap[slotIndex]->tail = newSlotNode;
    } else {  //非空
        slotMap[slotIndex]->tail->next = newSlotNode;
        slotMap[slotIndex]->tail = newSlotNode;
    }
    ++(slotMap[slotIndex]->size);
    return 0;
}
```

attachToSlot将某用户的netFd挂载到slotIndex对应的链表上，这里采用的是尾插法。

3.1 环形队列

```
/*更新当前client的slotIndex时也要将该client对应的slotNode挂到timeIndex之下，成功赶回-1，失败返回0*/
int removeFromSlot(int userId, int netFd, int slotIndex, slotList_t** slotMap) {
    //删除slotMap[oldIndex]中userId相等的节点
    slotNode_t* prev = NULL;
    slotNode_t* curr = slotMap[slotIndex]->head;
    while (curr != NULL && curr->userId != userId) {
        prev = curr;
        curr = curr->next;
    }
    // 若没有这样的元素
    if (curr == NULL) {
        return -1;
    }
    //删除第一个元素
    if (prev == NULL) {
        if (slotMap[slotIndex]->size == 1) {  //如果只有一个元素
            slotMap[slotIndex]->head = slotMap[slotIndex]->tail = NULL;
        } else {
            slotMap[slotIndex]->head = curr->next;
        }
        free(curr);
    } else {
        prev->next = curr->next;
        if (prev->next == NULL) {  //如果要删除尾节点
            slotMap[slotIndex]->tail = prev;
        }
        free(curr);
    }
    --(slotMap[slotIndex]->size);
    return 0;
}
```

removeFromSlot找到某用户的链表结点并摘下。

3.1 环形队列

```
/*初始化用户数组，成功返回0*/
int initClients(clientBox_t* clients, int* clientNum) {
    bzero(clients, CLIENTMAX * sizeof(clientBox_t));
    for (int i = 0; i < CLIENTMAX; i++) {
        clients[i].netFd = -1;
        clients[i].slotIndex = -1;
        // userId和userName已初始化为0
    }
    *clientNum = 0;
    return 0;
}
```

```
/*添加用户到用户数组，成功返回0，失败返回-1*/
int addToClients(clientBox_t* clients, int netFd, char*
userName, int userId, int timeIndex, slotList_t**
slotMap, int epfd, int* pClientNum) {
    int i;
    for (i = 0; i < CLIENTMAX; i++) {
        if (clients[i].netFd == -1) {
            break;
        }
    }
    if (i == CLIENTMAX) {
        printf("用户已满!\n");
        return -1;
    }
    clients[i].netFd = netFd;
    strcpy(clients[i].userName, userName);
    clients[i].userId = userId;
    clients[i].slotIndex = (TIMEOUT + timeIndex - 1) %
TIMEOUT;  //给slotIndex赋初值
    attachToSlot(userId, netFd, clients[i].slotIndex,
slotMap);  //加到slotMap中对应index位置
    epollAdd(clients[i].netFd, epfd);
    ++(*pClientNum);
    return 0;
}
```

客户端数组的维护和老师提到的类似，就不再赘述了。这里将用户加入用户数组同时在链表数组中挂载对应的结点

3.1 环形队列

```c
/*将用户从用户数组删除，成功返回0，失败返回-1*/
int delFromClients(clientBox_t* clients, int
userId) {
    int i;
    for (i = 0; i < CLIENTMAX; i++) {
        if (clients[i].userId == userId) {
            break;
        }
    }
    if (i == CLIENTMAX) {
        printf("不存在该用户!\n");
        return -1;
    }
    close(clients[i].netFd);
    bzero(&clients[i], sizeof(clientBox_t));
    clients[i].netFd = -1;
    clients[i].slotIndex = -1;
}
```

3.1 环形队列

```
/*客户端数组，存放客户端的netFd*/
clientBox_t clients[CLIENTMAX];
int clientNum;
initClients(clients, &clientNum);

/*初始化slotMap*/
int timeIndex = 0;
slotList_t* slotMap[TIMEOUT] = {0};
initSlotMap(slotMap, TIMEOUT);
```

在程序中，服务端首先需要创建客户端数组和slotMap，即链表数组。
令timeIndex为0。这个timeIndex代表了时针，每过一秒就走一格。
也正因为如此，链表数组的数组元素个数是和TIMEOUT对应的。

3.1 环形队列

```
//把netFd加入到客户端数组并监听
addToClients(clients, netFd, userName, userId, (TIMEOUT + timeIndex
- 1) % TIMEOUT, slotMap, epfd, &clientNum);


int readyNum = epoll_wait(epfd, readyArr, EPOLLNUM, 1000);
if (readyNum == 0 && clientNum > 0) {
   clearSlot(timeIndex, slotMap, clients, &clientNum);
   timeIndex = (timeIndex + 1) % TIMEOUT;
}
```

如果有用户登录进来，就将用户放入到客户端数组并监听，之类注意slotIndex传入的参数是(TIMEOUT + timeIndex - 1) % TIMEOUT，也就是时针指向的数组元素的上一个插槽（slot）。

Epoll监听时，每一秒都会执行一次clearSlot，也就是将时针指向的插槽中所有的链表结点含有的netFd关闭，断开与对应的客户端的连接，同时时针前进一格，为什么要这么做呢，有没有同学有疑问的。这里有个坑我debug了好久，才发现原来负数整除是会出问题的。

3.1 环形队列

```
if (readyArr[i].data.fd == clients[j].netFd) {
   vFinder(clients[j].netFd, conn, clients[j].userName, &threadPool);
   //从原来的slot移除
   removeFromSlot(clients[j].userId, clients[j].netFd, clients[j].slotIndex,
slotMap);
   //更新客户的slotIndex
   clients[j].slotIndex = (TIMEOUT + timeIndex - 1) % TIMEOUT;
   //挂到新的slot
   attachToSlot(clients[j].userId, clients[j].netFd, clients[j].slotIndex,
slotMap);
}
```

每当一个客户端的netFd就绪时，会将该netFd对应的链表结点从原来的插槽中取出并插入到时针指向的上一格。

这也解释了，为什么一直活跃的客户端不会被断开，因为他会跟在时针的后面，除非时针走过一圈遇到客户端还没跟上来，就把它断开，而时针走一圈的时间正好是TIMEOUT的时间。

3.2 传文件

```c
/*文件类型*/
enum FILETYPE {
    DIRFILE,
    COMMONFILE
};

/*文件协议*/
typedef struct train_s {
    int length;
    char buf[1000];
} train_t;
```

```c
/*文件信息结构体*/
typedef struct fileInfo_s {
    int parentId;
    char fileName[BUFFERSIZE];
    int ownerId;
    char md5[40];
    int fileSize;
    int fileType;
} fileInfo_t;
```

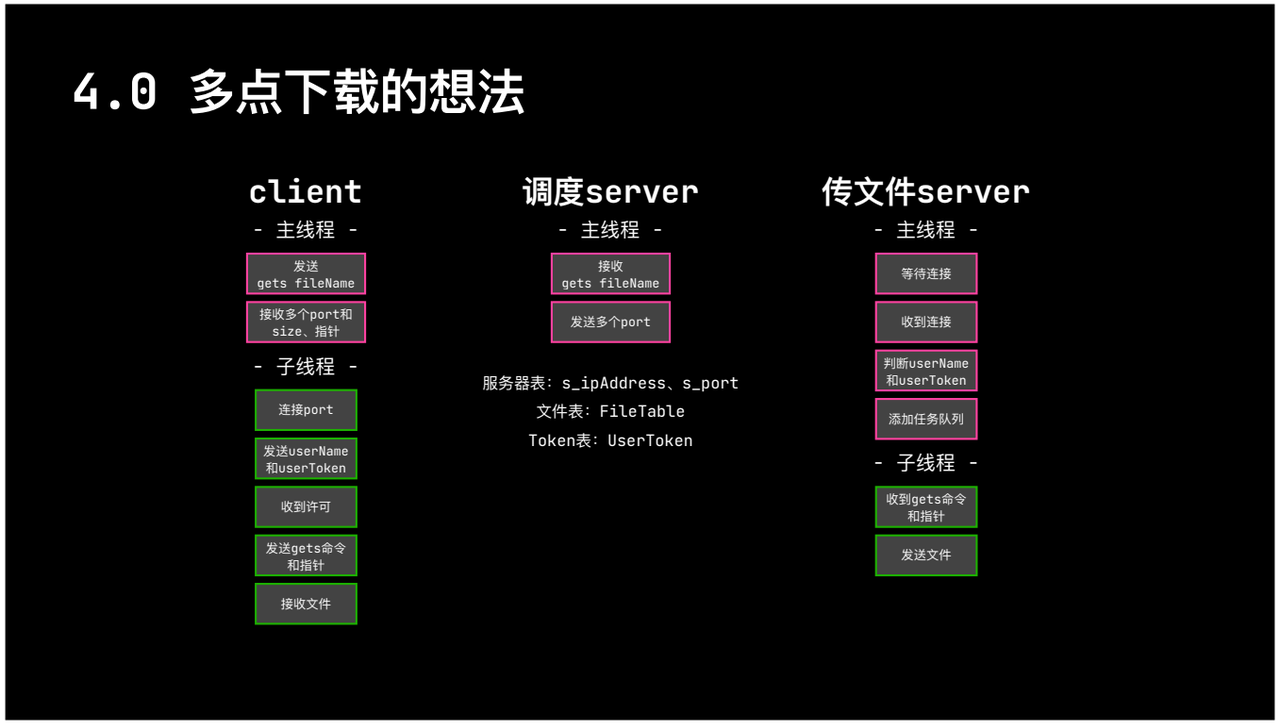
另一个值得讲的是我们的文件下载实现。

3.2 传文件

```c
int recvFile(int sockFd, char* fileName) {
    int total;

    // 尝试打开fileName.temp，看是否存在
    char tempFileName[BUFFERSIZE] = {0};
    sprintf(tempFileName, "%s.temp", fileName);
    int tempFd = open(tempFileName, O_RDWR);

    if (tempFd == -1) {  //若不存在temp文件，即便有fileName文件也需要重新下载并覆盖
        total = 0;
        tempFd = open(tempFileName, O_RDWR | O_CREAT);  //创建temp文件
    } else {                                            //若已存在temp文件
        //从temp中读取total
        char tempBuf[40] = {0};
        read(tempFd, tempBuf, sizeof(tempBuf));
        puts(tempBuf);
        total = atoi(tempBuf);
    }

    //发送total
    sendn(sockFd, &total, sizeof(int));

    int dataLength;

    int fd = open(fileName, O_RDWR | O_CREAT | O_TRUNC, 0666);
    ERROR_CHECK(fd, -1, "open");

    int fileSize;

    recvn(sockFd, &dataLength, sizeof(int));
    recvn(sockFd, &fileSize, dataLength);
    printf("fileSize = %d\n", fileSize);
    int doneSize = total;
    int lastSize = 0;
    int slice = fileSize / 10000;
    ftruncate(fd, fileSize);

    char* p = (char*)mmap(NULL, fileSize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    ERROR_CHECK(p, MAP_FAILED, "mmap");
    //映射pTotal

    time_t timeBeg, timeEnd;

    timeBeg = time(NULL);

    while (total < fileSize) {
        recvn(sockFd, &dataLength, sizeof(int));
        if (dataLength != 1000) {
            printf("dataLength = %d\n", dataLength);
            printf("100.00%%\n");
            remove(tempFileName);
            break;
        }
        doneSize += dataLength;
        if (doneSize - lastSize > slice) {
            printf("%5.2lf%%\r", 100.0 * doneSize / fileSize);
            fflush(stdout);
            lastSize = doneSize;
        }

        recvn(sockFd, p + total, dataLength);
        total += dataLength;

        //写入total
        char tempBuf[40] = {0};
        itoa(total, tempBuf);
        lseek(tempFd, 0, SEEK_SET);
        write(tempFd, tempBuf, strlen(tempBuf));

        usleep(sleepTime);
    }

    recvn(sockFd, p + total, dataLength);

    timeEnd = time(NULL);

    printf("totalTime = %lds\n", timeEnd - timeBeg);

    close(fd);
    munmap(p, fileSize);
}
```

在之前的流程中我们提到了temp文件，按照流程图构建了这个下载函数，保证了在任何时刻意外断连后，再次下载可以继续删次的断点。

4.0 感想



其他的功能其实实现了以后都觉得不难，只是老师在前天说的的内容导致我们需要重构非常多的内容，我们认为，与其在屎山上继续写多点下载功能，不如重构。
对于多点下载，我们画了以下流程图。
事实上这和单点下载没有太大区别，不过是多了个调度服务器。

4.0 多点下载的想法

fileName.temp

```
/*分段数*/
int x
/*每段已读指针*/
int p1
int p2
int p3
…
```

而temp文件也只需要从原有的一行分为多行，保存多个传输指针。
在这个流程上，我们能够实现最简单的多服务器下载。

4.0 多点下载的想法

TCP流量控制

传统的TCP流量探测机制有一个非常致命的缺陷：一旦检测到有丢包，立马将发送速率降为1/2。降速1/2后，如果没有丢包，将会在1/2速率的基础上，按照固定的增长值（线性增长），加大发送的速率。接下来就会一直按照这个节奏到达丢包的那一刻（实时可用带宽）为止。如果下一个检测周期依然有丢包现象，会在当前1/2速率的基础上继续降速1/2。循环往复，直到文件下载结束。

很显然指数级的降速、但是线性的增速；这最后造成的结果就是真实的传输速率远远小于实时可用带宽。

我们还做了一个实验，多线程同时从不同指针下载一个文件能提速，且不需要加锁。为什么能提速呢。
单点下载的过程中，如果丢包，会导致降速。

4.0 多点下载的想法

多线程下载

多线程下载时，由于多个线程在竞争实时可用带宽。尽管多线程逻辑上是并行的，但其实还是按时序的串行处理。所以每个线程处于的阶段并不一致。并且带宽资源是固定的。

比如使用3个线程来进行下载，因为处于不同的阶段，有的线程因为丢包直接降速1/2，有的线程处于线性增长阶段。通过多个线程的加权平均，最后得到的下载曲线是一条平滑的曲线，且这条曲线大多数应该处于单线程下载速率的上方。这也是为什么多线程下载大文件的速度更快的原因了。

而多线程下载，可以保证下载过程中大部分时间带宽利用率保持在一个比较高的水平，也就是说，多点下载时能够保证大部分时间的满速下载。

4.2 感悟

A：刚开始的项目和需求不明确，无从下手

B：感恩，热泪盈眶

C：写得慢无所谓，踏踏实实一个一个小功能完成就好了。（算是自我安慰把，听我说，谢谢你~）

D：线程池操作不熟练，小细节很多没处理好，如:返回值，函数类型设置

E：进程池传输文件较为复杂，MySQL操作经过调试更加熟练了

最后来谈一下感想，这里是我的小组成员的感想。我们小组在实现这个项目的过程中应该是，有了较大的进步。

4.2 感悟

不得不说星空老师在这个项目的过程中讲的东西都很实在很管用，让我茅塞顿开。但感恩之下我更多的是悔恨，以至于现在的积重难返，只能重新做。但这也不怪老师，看得出来老师已经很真诚了。自己写东西没有动力，说什么以后有时间写，其实都是骗自己，不管说什么，写了一星期学到了很多，原来觉得懵懵懂懂的东西现在看起来也就那么一回事。

我个人的感想是，不得不说星空老师在这个项目的过程中讲的东西都很实在很管用，让我茅塞顿开。但感恩之下我更多的是悔恨，以至于现在的积重难返，只能重新做。但这也不怪老师，看得出来老师已经很真诚了。自己写东西没有动力，说什么以后有时间写，其实都是骗自己，不管说什么，写了一星期学到了很多，原来觉得懵懵懂懂的东西现在看起来也就那么一回事。