SFTP 源码浅析

1. 整体框架

当在终端通过 sftp 命令连接目标主机时，sftp 客户端程序便随之启动。客户端首先启动 ssh 程序连接目标主机完成身份认证等操作。此后客户端进入循环等待，等待用户命令交互，完成相关的文件传输任务。直到用户运行 exit 命令使程序退出。

在连接目标主机前，程序首先初始化日志系统，用以记录程序运行过程中的日志、错误、以及调试信息。此外，还将解析连接命令中的 URI 以及其他参数，获取目标主机的用户和主机名等信息。然后结合连接命令中的参数等信息构建参数列表 args.list。该参数列表将作为参数，启动 ssh 程序连接目标主机。

客户端调用 connect_to_server 函数完成与目标主机的连接。在 connect_to_server 函数中，使用使用 socketpair 创建套接字，并重定向至输入输出流。然后 fork 一个子进程，子进程中调用 execvp 函数覆盖子进程空间，并运行 ssh 程序连接目标主机（以 args.list 为命令行参数）。ssh 程序中完成了身份认证等一系列动作。目标主机上的守护进程 sshd 监听到这个连接后启动 sftp-server 进程响应客户端的后续请求。

结构体 sftp_conn 用以描述客户端到服务端的 sftp 连接。在实际连接建立后，父进程（即 sftp 客户端）将对该连接进行初始化，即初始化一个 sftp_conn 类型的结构体 conn。此后客户端进入循环交互等待（interactive_loop），根据后续的文件传输命令（get、put 等）完成相应的文件传输操作。此时，客户端存在 sftp 进程和 ssh 进程，服务端存在 sshd 进程和 sftp-server 进程。如图 1.1 所示。

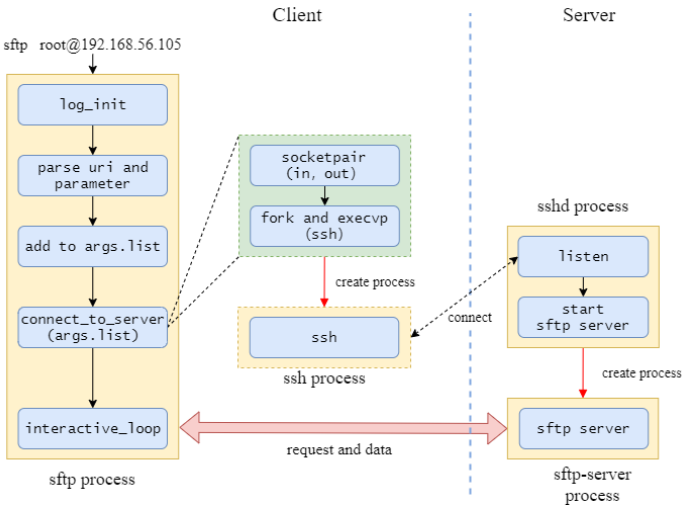


图 1.1：整体框架图

在 interactive_loop 中，程序从控制台读取后续的操作命令，并解析执行此调度命令（见 parse_dispatch_command 函数），执行完成后等待输入下一条命令，这个过程不断循环，直至 exit 命令使程序退出。

parse_dispatch_command 函数的函数名带有一些迷惑性，事实上函数不但对调度命令进行了解析（parse），还根据命令的类型（实际上是类型号）调用对应的函数进行执行。所谓解析，主要是从命令中获取操作的类型，如 put 操作或者 get 操作等；附加参数，如-a：断点续传、-r：操作文件夹；源文件（夹）地址和目标文件（夹）地址（如果有的话）等。

在客户端，每一种操作都被拆解为若干步骤，这些步骤以请求（request）的形式发送给服务端。网络另一边的服务端在收到请求后，根据请求的类型和内容作出响应，或写入数据、或回送状态、或传输数据等。服务端的原则是仅对客户端的请求作出响应，并且不对客户端的操作做任何假设。任务的执行逻辑等均由客户端决定。比如，数据的传输对于客户端来说是连续进行的（客户端连续切分数据），但在服务端的视角里，仅仅是根据客户端要求的偏移量和长度发送（或写入）指定数据。以 put 操作为例，如图 1.2 所示。

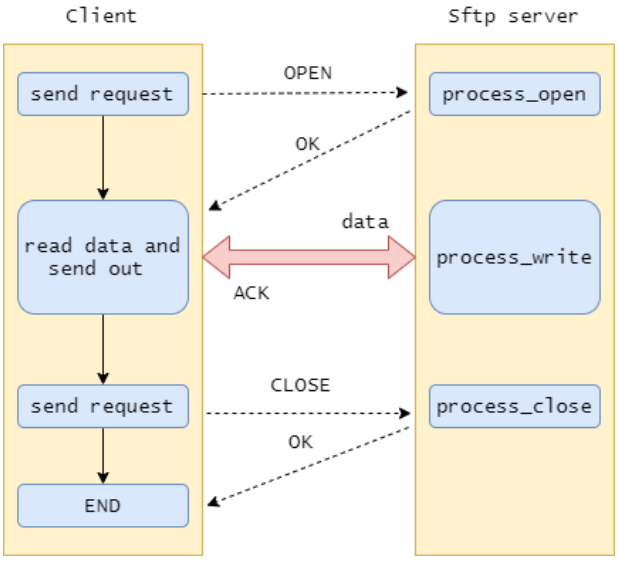


图 1.2：put 操作流程图

客户端首先发送一个 OPEN 请求给服务端，要求服务端打开（创建）文件，服务端执行操作后给客户端会送一个状态（status）。之后客户端不断读取数据并发送给服务端，服务端收到数据后将其写入相应的位置。这里需要说明的是，客户端并非是在收到服务端的确认后再逐一发送下一组数据（这种方式效率太低）。实际上，客户端维持了一个发送窗口（窗口大小 64），连续发送若干组数据并使用一个队列存储未确认的数据请求。当客户端数据发送完成后，便向服务端发

送 CLOSE 请求，令其关闭文件。任务完成后，客户端回到交互循环等待状态。

2. sftp-server 设计与实现

2.1 sftp_server_main 的整体设计

客户端在建立连接时，将创建好的 socket 套接字重定向至输入输出流。一端往输出流中写入数据，另一端便从输入流中读取数据，进而实现了数据的传输。当然，这是一个粗略的说法。但在 sftp 进程和 sftp-server 进程中，数据的发送只需要将数据写入输出流即可，接收数据与之相反。sftp-server_main 函数的核心代码:

```
struct sshbuf *iqueue;
struct sshbuf *oqueue;

int sftp_server_main(...){

for (;;) {

    FD_SET(in, rset);
    FD_SET(out, wset);
    ...
    if (select(max+1, rset, wset, NULL, NULL) == -1) {
      //TODO
    }

    ...
    /* copy stdin to iqueue */
    if (FD_ISSET(in, rset)) {
       len = read(in, buf, sizeof buf);
       //TODO
    }

    /* send oqueue to stdout */
    if (FD_ISSET(out, wset)) {
       len = write(out, sshbuf_ptr(oqueue), olen);
      //TODO
    }

    r = sshbuf_check_reserve(oqueue, SFTP_MAX_MSG_LENGTH);
    if (r == 0)
       process();
       ...
}    //end for
}    //end sftp_server_main
```

程序首先阻塞在 select 函数处，当阻塞模式的 select 函数检查到 rset 或 wset 中存在可读写的文件句柄时，即意味着有请求和数据需要读取或发送。服务端使用 sshbuf 结构体描述一个请求（实际上是一个字符串）。当收到数据时，程序从输入流中将数据读入到 buf 中，再将 buf 中的内容添加到 sshbuf 类型的结构体 iqueue 中。当发送数据时，服务端将数据添加到 oqueue 中，再将 oqueue 写入输出流。

2.2 sshbuf 的设计

sshbuf 结构体的定义如下。

```
struct sshbuf {
        u_char *d;      /* Data */
        const u_char *cd;   /* Const data */
        size_t off;     /* First available byte is buf->d + buf->off */
        size_t size;    /* Last byte is buf->d + buf->size - 1 */
        size_t max_size;    /* Maximum size of buffer */
        size_t alloc;       /* Total bytes allocated to buf->d */
        int readonly;       /* Refers to external, const data */
        int dont_free;      /* Kludge to support sshbuf_init */
        u_int refcount;     /* Tracks self and number of child buffers */
        struct sshbuf *parent;    /* If child, pointer to parent */
};
```

u_char *d 指向了 sshbuf 的数据存储区域，alloc 是已经分配给该 sshbuf 的空间大小，max_size 表示的是能分配给该 sshbuf 的最大空间。off 指明了有效数据的偏移位置。如图 2.1 所示。

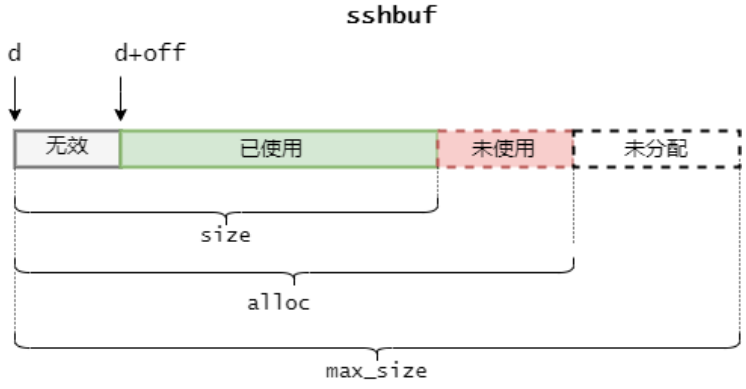


图 2.1：sshbuf 结构体示意图

process 函数负责对请求进行处理。当在一次循环中只是发送数据而不存在待处理的客户端请求时，process 函数会检查到 iqueue 中的 ssh_buf 长度小于 5，则表明这是一个“不完整的消息”，即没有可处理的请求，随即退出 process 函数，

进入下一次循环。

2.3 请求与响应

服务端收到的请求（request）被放置在 iqueue（sshbuf 类型的结构体）的数据存储区域（即指针 d+off 指向的内存空间）。process 函数中调用与 sshbuf 结构体相关的函数，如 sshbuf_get_u32、sshbuf_get_u8 等，对收到的数据进行解析，获取 type、id 和 suffix 等字段内容。客户端与服务端之间传输数据格式如图 2.2 所示。

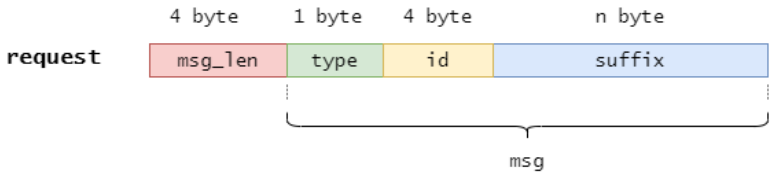


图 2.2：客户端请求数据格式

msg_len 表示后续整个 messge 部分的长度（即 type、id 和 data 三个字段）。type 表示客户端请求操作的类型，如 OPEN、WRITE、READ、CLOSE、STAT 等。id 表示请求的序号。data 部分作为后缀，其格式和长度由 type 的类型决定。

当识别出请求的类型后，接下来便可以调用对应的函数对请求进行响应。每一种请求对应着一个 sftp_handler，sftp_handler 的结构如下。name 表示 handler 的名称，type 表示所处理请求的类型，void（*handler）（u_int32_t）函数便是对应的处理函数。

```
struct sftp_handler {
        const char *name;   /* user-visible name for fine-grained perms */
        const char *ext_name;   /* extended request name */
        u_int type;       /* packet type, for non extended packets */
        void (*handler)(u_int32_t);
        int does_write;       /* if nonzero, banned for readonly mode */
};
```

随后构造一个 sftp_handler 类型的数组 handlers[]，handlers 中包含了所有非 extend 类型的请求（extend 请求用以向服务端发送附加请求，比如将前一次传输的数据立即从缓冲区写入磁盘等）。每次获取到请求的 type，只需在 handers 中遍历一次，找到对应的 sftp_handler，并调用对应的响应函数完成对该请求的处理。

```
for (i = 0; handlers[i].handler != NULL; i++) {
      if (type == handlers[i].type) {
            handlers[i].handler(id);
            break;
      }
```

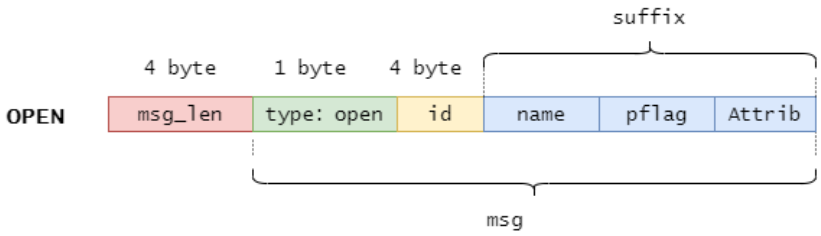
```
}
```

handlers 数组定义如下。

```
static const struct sftp_handler handlers[] = {
        /* NB. SSH2_FXP_OPEN does the readonly check in the handler itself */
        { "open", NULL, SSH2_FXP_OPEN, process_open, 0 },
        { "close", NULL, SSH2_FXP_CLOSE, process_close, 0 },
        { "read", NULL, SSH2_FXP_READ, process_read, 0 },
        { "write", NULL, SSH2_FXP_WRITE, process_write, 1 },
        { "lstat", NULL, SSH2_FXP_LSTAT, process_lstat, 0 },
        { "fstat", NULL, SSH2_FXP_FSTAT, process_fstat, 0 },
        { "setstat", NULL, SSH2_FXP_SETSTAT, process_setstat, 1 },
        { "fsetstat", NULL, SSH2_FXP_FSETSTAT, process_fsetstat, 1 },
        { "opendir", NULL, SSH2_FXP_OPENDIR, process_opendir, 0 },
        { "readdir", NULL, SSH2_FXP_READDIR, process_readdir, 0 },
        { "remove", NULL, SSH2_FXP_REMOVE, process_remove, 1 },
        { "mkdir", NULL, SSH2_FXP_MKDIR, process_mkdir, 1 },
        { "rmdir", NULL, SSH2_FXP_RMDIR, process_rmdir, 1 },
        { "realpath", NULL, SSH2_FXP_REALPATH, process_realpath, 0 },
        { "stat", NULL, SSH2_FXP_STAT, process_stat, 0 },
        { "rename", NULL, SSH2_FXP_RENAME, process_rename, 1 },
        { "readlink", NULL, SSH2_FXP_READLINK, process_readlink, 0 },
        { "symlink", NULL, SSH2_FXP_SYMLINK, process_symlink, 1 },
        { NULL, NULL, 0, NULL, 0 }
};
```

2.4 示例 1：process_open 与 process_write

process_open 函数处理的 OPEN 请求数据格式如下。name 字段表示待打开（创建）的文件名（其中前 4 个字节包含了 name 的长度），Attrib 表示文件的属性。



在从 iqueue 中获取到需要的字段内容后，便调用 open 函数打开（当文件不存在时创建）相应的文件。服务端使用一个 Handle 类型的结构体来描述一个打开的

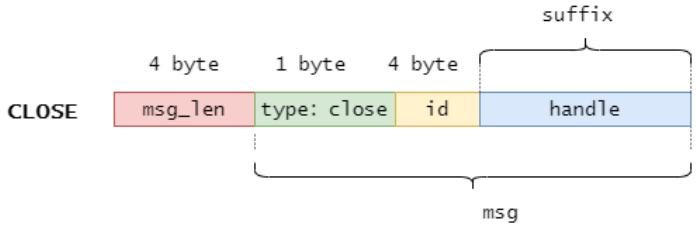
文件。并维持了一个链表 Handle 类型的动态数组 handles 存储已经打开的文件信息。process_open 中使用 open 函数创建一个文件后，便调用 handle_new 函数创建一个与该文件对应的 Handle，并将其添加到 handles 中。（这里有一个很让人费解的问题：针对文件和文件夹的 put 和 get 操作，都是逐一对文件进行操作的，即 handles 中始终只有一个 Handle 存在，这样的话貌似不需要设计这样的动态数组。）

Handle 结构体的定义如下。fd 表示所打开文件的句柄，flags 表示文件的各类标志。name 表示文件的名称，bytes_read 和 bytes_write 分别表示已读取和已写入的字节数。next_unused 表示下一个未使用的 Handle。文件创建成功后，调用 send_handle 函数将对应的 handle 发送给客户端。

```
struct Handle {
        int use;
        DIR *dirp;
        int fd;
        int flags;
        char *name;
        u_int64_t bytes_read, bytes_write;
        int next_unused;
};

//use  的类型
enum {
        HANDLE_UNUSED,
        HANDLE_DIR,
        HANDLE_FILE
};
```

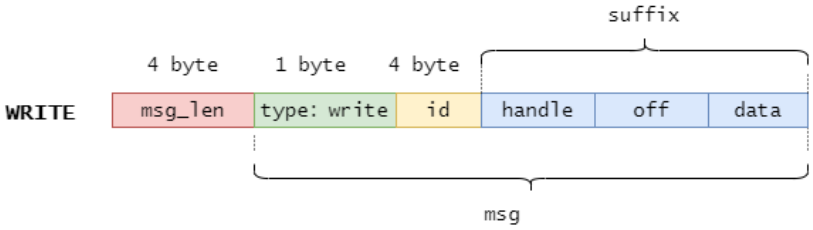
类似地，若服务端收到的是文件关闭的请求 CLOSE，则调用 process_close 函数进行处理。其数据格式如下。



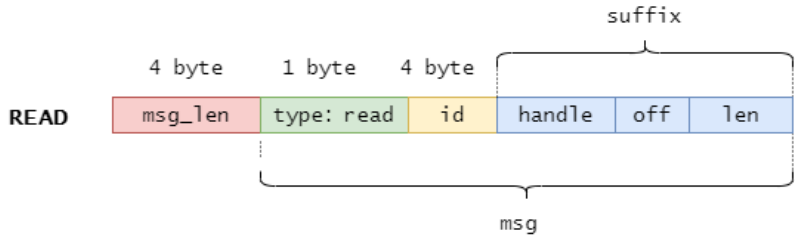
handle 字段表示了请求关闭的文件（所对应的 handle 值），然后以 handle 为参数调用 handle_close 函数，关闭对应的文件。

2.5 示例 2：process_write 与 process_read

process_write 函数用于向指定文件写入数据。其数据格式如下。通过 handle 指明了待写入数据的文件，通过 handle 可以获取已打开的对应的文件句柄。off 表示写入数据的位置，即文件的偏移位置。data 即为待写入的数据，该字段的前 4 个字节存储了数据的长度。当使用 lseek 函数设置好文件的读写指针后，便调用 write 函数将 data 写入文件，即 write(fd, data, len)。最后更新对应的 Handle，并向客户端发送一个操作成功的状态（send_status 函数）：SSH2_FX_OK。



类似地，process_read 函数用于从指定文件读取数据，并发送给客户端。其数据格式如下。suffix 字段的含义为：从 handle 对应的文件的 off 处，读取长度为 len 的数据发送给客户端。在获取这些参数后，程序调用 read(fd, buf, len)函数读取数据到 buf 中，然后调用 send_data 函数将数据发送给客户端。



2.6 消息传输

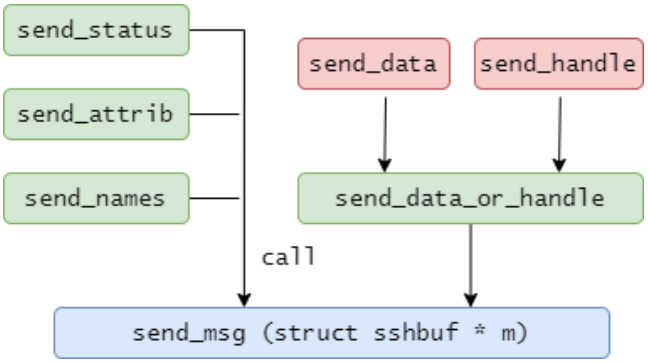
服务端在作出响应时，需要向客户端发送各类信息，比如客户端请求的数据、操作是否成功、文件对应的属性等。

服务端定义了一个 void send_msg (struct sshbuf *m)函数，该函数的功能是将 m 所指向的 sshbuf 放进 oqueue 中，sftp_server_main 在执行时将 oqueue 写入输出流。数据在发送前都按照 msg 的格式封装成 sshbuf 进行发送。部分函数如下。

函数名	函数功能
send_names	发送获取到的文件名称
send_attrib	发送文件的属性
send_data	发送读取的文件内容
send_handle	发送文件对应的 handle

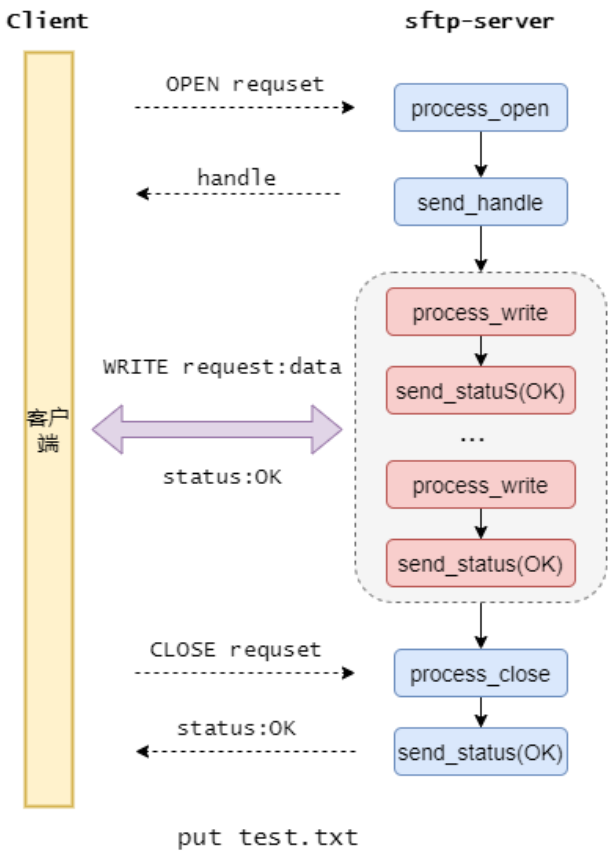
send_status	发送操作后的状态码

函数的调用关系如下。send_status、send_attrib 和 send_names 函数直接调用 send_msg 将封装成 sshbuf 的数据发送出去。send_data 和 send_handle 的实现很类似，为了增加相应的调试信息，并且将 send_handle 的参数由整型转换成字符串，所以新增加一个中间函数 send_data_or_handle。这里仅列举了部分与消息传输有关的函数。



2.7 put 与 get 操作的响应流程

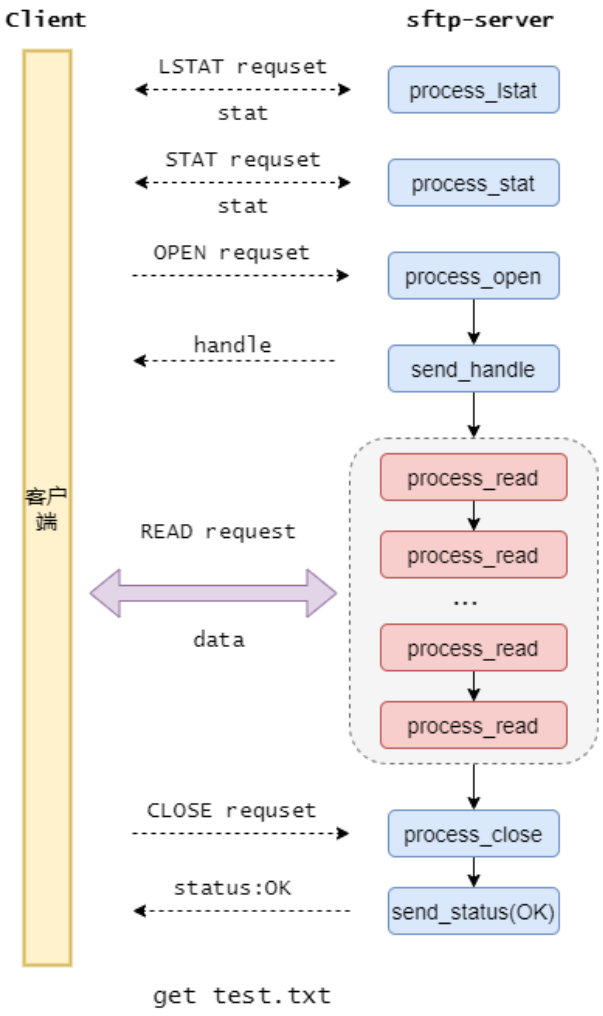
客户端使用 put 操作命令将 test.txt 上传到服务端的 root 文件夹中。客户端将这项操作划分成若干步骤，并向服务端发送请求，服务端的任务只是响应客户端的请求，不对客户端的任务执行逻辑做任何假设，比如数据传输是否连续。



首先，客户端发送 OPEN 请求令服务端打开指定文件（如果文件不存在时创建该文件），服务端打开文件成功之后，将该文件所对应的 Handle 结构体（以整型 handle 标识）回送给客户端，send_handle 实际上是在 process_open 函数中被调用。之后客户端不断发送 WRITE 请求和数据，要求服务端在指定位置将数据写入。服务端写入数据后同样向客户端回送一个确认。

当客户端读到 EOF 时，表示数据传输结束，随后向服务端发送 CLOSE 请求，要求服务端关闭文件。服务端关闭文件后，同样向客户端发送一个状态码（status）。至此文件传输完成，服务端回到 select 函数处阻塞，等待客户端下一轮请求。注：若 put 操作命令中包含目标路径时，在客户端在发送 OPEN 请求前会发送 LSTAT 请求让服务端检查这个路径是否存在。

类似地，当客户端使用 get 命令获取一个文件时，服务端的响应流程如下。客户端首先发送 LSTAT 请求和 STAT 请求，检查文件是否存在并获取文件相关属性。之后再发送 READ 请求获取文件内容。客户端每次请求的数据默认为 32KB，且可以通过参数修改，但服务端对这个大小会进行检查，最大不超过 64KB。当使用 put 或 get 命令传输文件夹或带有其他参数时，处理流程会略有不同。



3. sftp-client 设计与实现

4. 心慕手追：starFTP 的设计与实现

5. 后记