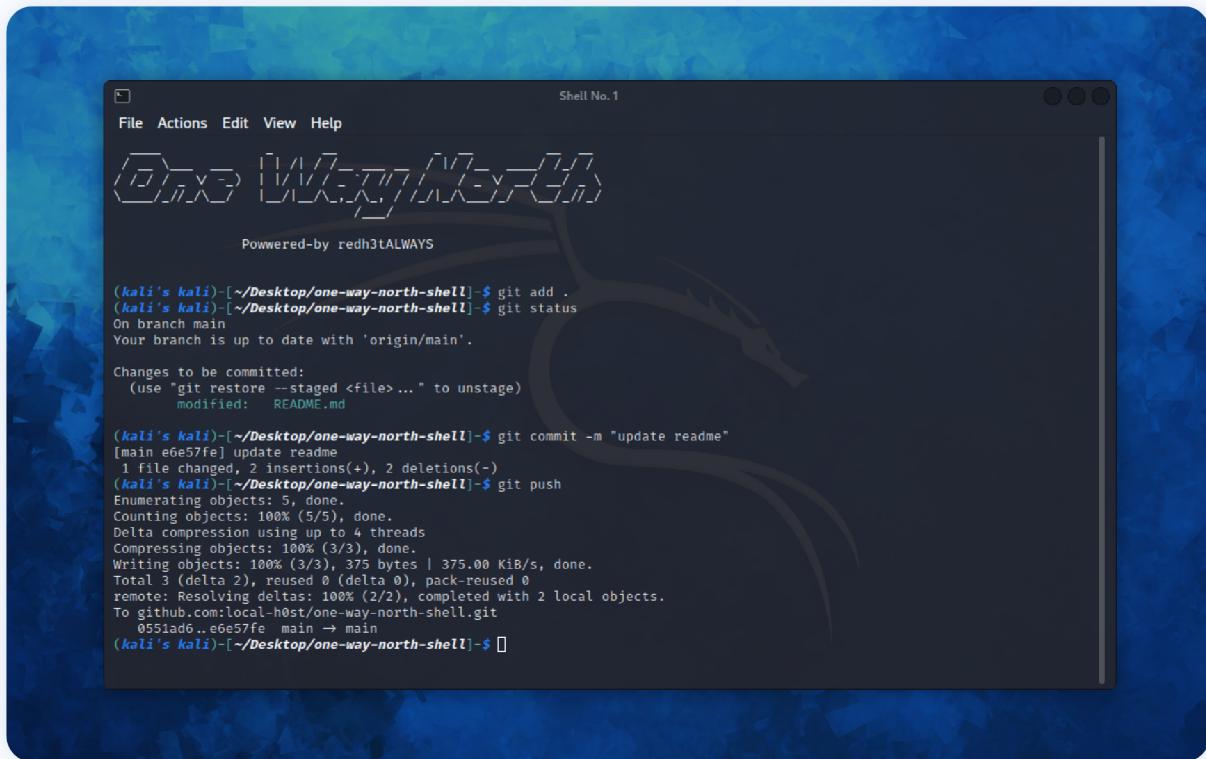


# nsh-shell设计与实现

## 效果预览



使用nsh能够正常完成git相关操作。one-way-north-shell实现代码开源在[我的Github仓库](#)。

## 设计框架速览

shell的核心流程是“获取用户输入-处理输入”的循环过程，在进入循环前shell需要进行初始化的工作，因此我的 `main()` 如下实现：

```
int main(void){
    // init
    State.command = (struct COMMAND_FRAG *)malloc(sizeof(struct COMMAND_FRAG));
    initCmdFragNull(State.command);
    printWelcome();
    while(1){
        flushAndPrompt();
        parseCommand();
        executeCommand();
        deleteFragAll(State.command);    // clear history here
    }
    return 0;
}
```

shell当前的状态保存在全局变量 `State` 中，方便其他函数随时读取和修改。`State` 定义如下：

```
struct REDIRECT_INFO{
    int direction;
    char *filename;
    int symbol_count;
    struct REDIRECT_INFO *next;
};

struct COMMAND_FRAG {
    char *binname;
    char *arg;
    struct COMMAND_FRAG *pipe_next;
    struct COMMAND_FRAG *arg_next;
    struct REDIRECT_INFO *redirect;
};

struct STATE_STRUCT {
    char input[STATE_INPUT_LEN];
    struct passwd *current_passwd;
    struct COMMAND_FRAG *command;
} State;
```

`State.input` 数组保存了当前用户的输入，是未经处理的 raw input；`State.current_passwd` 保存了当前用户 `/etc/passwd` 文件的内容，被用于权限判断、切换用户等操作中；`State.command` 是一张链表的头指针，`parseCommand()` 函数将解析 `State.input` 提权可执行文件名、传入参数、管道符、重定向信息等，并将解析结果保存到这张链表中。

解析后 `executeCommand()` 将根据 `State.command` 的内容执行命令，命令分为内部和外部，如果是 shell 内部支持的就会执行相关代码，如果不识别那么就将调用 `fork()` 和 `execvp()` 去执行外部程序并返回执行结果。单次循环的最后将清理 `State.command`，然后进行下一轮循环。

## A dive into the code

篇幅有限，函数亦有不同的重要程度，这里就看两个关键函数。

### parseCommand()

```
int parseCommand(){
    char temp[STATE_INPUT_LEN];
    int temp_index = 0;
    int is_binname = 2; // 2 the first, 1 after |, 0 the arg.
    struct COMMAND_FRAG *bin_head = NULL;
    struct COMMAND_FRAG *last_arg = NULL;
    int quot_count = 0; // only support "
    // first parse redirect as params, then parse redirect info, because redirect
    needs higher privilege
```

```

for(int i = 0; i < STATE_INPUT_LEN && State.input[i] != '\0'; i++){ // last is \n
and then is \0
    if (State.input[i] == '\"'){
        quot_count++;
        if(i>0)
            if(State.input[i-1] == '\\'){
                quot_count--;
                temp_index--;
            }
    }
    if ((!isspace(State.input[i]) && State.input[i] != '|' && quot_count%2 == 0)
|| quot_count%2 == 1){
        temp[temp_index] = State.input[i];
        temp_index++;
    }
    else{
        // encountered blank char or '|'
        if (temp_index == 0){
            if (State.input[i] == '|')
                is_binname = 1;
            continue; // ignore blank
        }
        temp[temp_index] = '\0';
        switch (is_binname) {
        case 2:
            State.command->binname = (char *)malloc((temp_index+1)*sizeof(char));
            strcpy(State.command->binname, temp);
            is_binname = 0;
            bin_head = State.command;
            last_arg = bin_head;
            break;
        case 1:
            bin_head->pipe_next = (struct COMMAND_FRAG *)malloc(sizeof(struct
COMMAND_FRAG));
            bin_head = bin_head->pipe_next;
            initCmdFragNull(bin_head);
            bin_head->binname = (char *)malloc((temp_index+1)*sizeof(char));
            strcpy(bin_head->binname, temp);
            is_binname = 0;
            last_arg = bin_head;
            break;
        default:
            // is_binname = 0, default is common arg.
            last_arg->arg_next = (struct COMMAND_FRAG *)malloc(sizeof(struct
COMMAND_FRAG));
            last_arg = last_arg->arg_next;
            initCmdFragNull(last_arg);
            last_arg->arg = (char *)malloc((temp_index+1)*sizeof(char));
            strcpy(last_arg->arg, temp);
            break;
        }
        temp_index =0;
    }
}

```

```

    }

    if (State.input[i] == '|' && quot_count%2 == 0)
        is_binname = 1;
}

// for every bin parse redirect info
struct COMMAND_FRAG *current_bin = State.command;
while (current_bin != NULL){
    parseRedirect(current_bin);
    current_bin = current_bin->pipe_next;
}
}

```

由于重定向的优先级要高于管道符，因此我采取的策略是，先将重定向符号以及重定向内容当作普通参数，先解析管道符。管道解析结束后再对每个参数判断是否是重定向，如果是，则新增重定向信息，并把此参数从真正的参数列表中删去。

如果存在管道符，那么所有管道上总共n个可执行文件的文件名将被保存在n个 `COMMAND_FRAG` 结构的 `binname` 属性中，然后这n个结构将由 `pipe_next` 属性串联起来。对于每一个可执行文件，其后的m个参数将被保存在m个 `COMMAND_FRAG` 结构的 `arg` 属性中，然后由 `arg_next` 属性串联起来。对于每一个可执行文件，如果存在重定向，那么其保存 `binname` 的 `COMMAND_FRAG` 结构将同时保存其重定向信息，如果存在多个重定向，那么也将以链表的形式串联起来。

举个例子，假设用户输入的命令是 `cat <README.md |grep "Some Thinking" >result.txt`。用户的输入被解析后，`State.command.binname` 保存第一个可执行文件名 `cat`，`State.command.arg_next` 指向其第一个参数，这里没有，因此为 `NULL`，由于有重定向，因此 `State.command.redirect` 指向第一个重定向信息（标准输入重定向到文件 `README.md`）。由于存在管道，因此 `State.command.pipe_next` 指向的 `COMMAND_FRAG` 结构将保存第二个可执行文件的信息，`State.command.pipe_next.binname = "grep"`，存在参数，因此 `State.command.pipe_next.arg_next.arg = "Some Thinking"`，由于存在重定向，因此 `State.command.pipe_next.redirect` 将保存“输出重定向到 `result.txt`”这一信息。

此外，在 `parseCommand()` 中可以看到变量 `quot_count`，这个参数使得nsh支持引号语法，也就是双引号之间的所有特殊字符，例如 `(blank)`，`|`，`<`，`>` 都不会被解析，而是直接当作用户的输入。如果要表示引号 `"`，就需要用 `\"` 来转义。

## executeCommand()

```

int executeOnce(struct COMMAND_FRAG *current_bin, int read_pipe_fd, int
write_pipe_fd){
    // build arg list first
    struct COMMAND_FRAG *temp = current_bin;
    int count = 1;
    while (temp->arg_next!=NULL){
        count++;
        temp = temp->arg_next;
    }
}

```

```

char **arg_list = (char **)malloc((count+1)*sizeof(char *));
arg_list[0] = current_bin->binname;
arg_list[count] = NULL;
temp = current_bin->arg_next;
count = 1;
while(temp!=NULL){
    if(temp->arg[0] == '\"'){
        temp->arg[strlen(temp->arg)-1] = '\0';
        char *str = (char *)malloc(strlen(temp->arg)*sizeof(char));
        strcpy(str,temp->arg+1);
        free(temp->arg);
        temp->arg = (char *)malloc((strlen(str)+1)*sizeof(char));
        strcpy(temp->arg,str);
        free(str);
    }
    arg_list[count] = temp->arg;
    count++;
    temp = temp->arg_next;
}

// call fork
pid_t pid = fork();
if (pid == 0){      // child process
    // pipe and redirect
    if (write_pipe_fd != -1){
        dup2(write_pipe_fd,STDOUT_FILENO);
        // dup2(write_pipe_fd,STDERR_FILENO); // pipe shouldn't handle stderr
        close(write_pipe_fd);
    }
    if(read_pipe_fd != -1){
        dup2(read_pipe_fd,STDIN_FILENO);
        close(read_pipe_fd);
    }
}

struct REDIRECT_INFO *r = current_bin->redirect;
while (r != NULL){
    if(r->direction == 1 && write_pipe_fd == -1){
        int w_fd;
        if(r->symbol_count==1){
            w_fd = open(r->filename, O_WRONLY|O_CREAT|O_TRUNC,
S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
        }
        else{
            w_fd = open(r->filename, O_WRONLY|O_CREAT|O_APPEND,
S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
        }
        dup2(w_fd,STDOUT_FILENO);
        close(w_fd);
    }
    else if(r->direction == 2 && read_pipe_fd == -1){
        int r_fd = open(r->filename, O_RDONLY);
        if (r_fd == -1){

```

```

        perror("redirect");
        exit(1);
    }
    dup2(r_fd, STDIN_FILENO);
    close(r_fd);
}
else{

}
r = r->next;
}
// execute
if (execvp(arg_list[0], arg_list) == -1){
    perror("child process: ");
    exit(1);
}
else if (pid > 0){ // parent
    wait(NULL);
}
else {
    perror("fork failed.");
    exit(1);
}

free(arg_list);
return 1;
}

int executeCommand(){
if (State.command->binname == NULL)
    return -1;
struct COMMAND_FRAG *current_bin = State.command;

int **pipes = (int **)malloc(2*sizeof(int *));
pipes[0] = (int *)malloc(2*sizeof(int));
pipes[1] = (int *)malloc(2*sizeof(int));
pipe(pipes[0]);
pipe(pipes[1]);

int index = 0;
int read_pipe = -1;
int write_pipe = 0;

while(1){
    if(current_bin->pipe_next != NULL)
        write_pipe = pipes[index%2][1];
    else{
        // last command, write_pipe is useless, close and ready to quit loop
        write_pipe = -1;
        close(pipes[index%2][0]);
        close(pipes[index%2][1]);
    }
}
}

```

```

}

switch (isBuildInCmd(current_bin->binname)){
// TODO add pipe support and redirect support for build-in commands.
case CMD_SH_HELP:
    printf("help document is not ready yet ~\n");
    break;
case CMD_CD:
    if (current_bin->arg_next->arg=NULL){
        printf("cd: need one argument.\n");
    }
    else{
        int success;
        if (current_bin->arg_next->arg[0] = '~'){
            success = chdir(getenv("HOME"));
            if (current_bin->arg_next->arg[1] = '/')
                success = chdir(current_bin->arg_next->arg+2);
        }
        else
            success = chdir(current_bin->arg_next->arg);
        if (success == -1)
            printf("cd: chdir() failed.");
    }
    break;
default:
    executeOnce(current_bin, read_pipe, write_pipe);
    break;
}

if(current_bin->pipe_next != NULL){
    current_bin = current_bin->pipe_next;
    read_pipe = pipes[index%2][0];
    close(pipes[(index+1)%2][0]);
    close(pipes[(index+1)%2][1]);
    free(pipes[(index+1)%2]);
    pipes[(index+1)%2] = NULL;
    pipes[(index+1)%2] = (int *)malloc(2*sizeof(int));
    pipe(pipes[(index+1)%2]);
    close(pipes[index%2][1]); // close write end to avoid stuck when read
    // should avoid use the same fd in two different pipes, so the order of
the code matters!
    index++;
}
else{
    close(read_pipe);
    break;
}

}

// all pipe fds are well closed, free() an return
free(pipes[0]);

```

```
    free(pipes[1]);
    free(pipes);
    return 1;
}
```

`executeCommand()` 识别内部和外部命令，如果nsh支持则直接执行相关代码，否则就调用 `executeOnce()` 去 `fork()` 和 `execvp()`。在 `executeCommand()` 的循环中，通过判断 `current_bin.pipe_next` 是否存在进行管道识别，并提前使用 `pipe()` 创建好管道以便接下来命令的执行。`executeOnce()` 通过传入 `COMMAND_FRAG *` 类型的变量 `current_bin`、管道的读写端 `fd`（如果没有就 `NULL`），在内部构建传参列表，使用 `dup2()` 按传入 `current_bin` 的信息设置好重定向，然后 `fork()` 和 `execvp()`。

我写的时候踩了三个坑，一个是 `pipe(pipe_fd)` 创建的管道是有读写方向的，写入端必须是 `pipe_fd[1]`，读取端必须是 `pipe_fd[0]`，否则会失败。另一个是读写 `pipe` 时，一端没有关闭将影响到另一端的正常操作，表现为写端不关闭会导致读取完毕后卡住不返回，读端不关闭会导致写端无法写入。还有一个是，两条 `pipe` 使用的4个 `fd` 一定不能有重合，否则会导致逻辑混乱。例如两条管道 `p1[2] = {3,4}, p1[2]`，关闭 `p1[1]` 释放 `fd = 4`，`p1[0]` 端没关闭，此时因为关闭而释放的 `fd = 4` 可以再次被 `pipe(p2)` 函数分配，得到 `p2[2] = {4,5}`，此时两条管道如下：`p1[0] = 3, p1[1] closed, p2[0] = 4, p2[1] = 5`，由于 `fd = 4` 曾经是管道 `p1` 的写端，而管道 `p1` 的读端仍在工作，但 `fd = 4` 却被分配给了另一条管道，这就会造成读写错误。

## 其他函数

其他函数诸如输出Prompt，垃圾回收等，相对没有那么核心，倘若放上来篇幅过长，因此报告中省略了，在[我的仓库](#)中可以看到相关内容。

## shell 功能测试

测试过程如下：

1. 测试常用单条命令
2. 测试单条命令，带重定向
3. 测试多条命令带管道
4. 测试多条命令带管道和重定向

### 常用单条命令

```
kali@kali: ~/Desktop/one-way-north-shell/bin
File Actions Edit View Help
Powered-by redh3tALWAYS

(kali's kali)-[~/Desktop/one-way-north-shell/bin]-$ cd
(kali's kali)-[~]-$ cd /etc
(kali's kali)-[/etc]-$ cd ~/Desktop
(kali's kali)-[~/Desktop]-$ cd ~/
(kali's kali)-[~]-$ cd ..
(kali's kali)-[/home]-$ 
(kali's kali)-[/home]-$ pwd
/home
(kali's kali)-[/home]-$ whoami
kali
(kali's kali)-[/home]-$ echo "hello this is redh3tALWAYS!"
hello this is redh3tALWAYS!
(kali's kali)-[/home]-$ ls
kali
(kali's kali)-[/home]-$ ls /proc
 1   106  1212  1584  195  26  398  517  653  708  83  930      diskstats      kpagecount      stat
 10  111  1217  1595  196  27  4   519  654  709  84  938      dma          kpageflags      swaps
 100 1110  1219  16   197  274  40  52  659  71  85  94      driver        loadavg       sys
 1005 1114  1235  1603  198  275  401  53  66  72  86  95      dynamic_debug  locks        sysrq-trigger
 101 1115  1247  1610  199  278  42  54  663  73  87  96      execdomains  meminfo      sysvipc
 1019 1116  1250  1615  2   28  43  55  664  74  88  97      fb           misc         thread-self
 1026 1117  1251  162   20  29  44  5588  67  75  89  98      filesystems  modules      timer_list
 1038 1118  1287  1769  200  3   45  56  677  76  90  99      fs            mounts      tty
 1049 112   13   18   201  30  46  6   678  77  903     acpi        interrupts    uptime
 1051 1125  14   186  202  31  47  63  68  78  909     asound      iomem        version
 1055 1129  1414  187  203  316  478  6304  69  79  91     buddyinfo   ioports      net         vmallocinfo
 106 1149  1422  189  204  317  48  6305  6904  8   910     bus          irq         pagetypeinfo  vmstat
 1061 1150  1435  19   205  35  483  6329  6915  80  92     cgroups     kallsyms    partitions  zoneinfo
 1085 1165  1443  190  21   36  49  6353  6929  8029  925    cmdline     kcore       pressure
 1089 1183  1466  191  2298  371  5   642  693  8070  926    consoles   keys        schedstat
 1095 1186  1481  192  23   38  50  646  6959  81   927    cpuinfo    key-users   self
 11 12   1495  193  24   388  51  65  7   82   928    crypto     kmsg        slabinfo
 1100 1202  15   194  25   39  5128  651  70   824  93     devices   kpagecgroup softirqs

(kali's kali)-[/home]-$ 
```

## 单条命令带重定向

```
(kali's kali)-[~/Desktop]-$ echo "aaaaaaaaaa redh3tALWAYS writing ..." >a.txt
(kali's kali)-[~/Desktop]-$ cat a.txt
aaaaaaaaaa redh3tALWAYS writing ...
(kali's kali)-[~/Desktop]-$ echo bbbbb >> a.txt
(kali's kali)-[~/Desktop]-$ cat a.txt
aaaaaaaaaa redh3tALWAYS writing ...
bbbbbb
(kali's kali)-[~/Desktop]-$ cat <a.txt
aaaaaaaaaa redh3tALWAYS writing ...
bbbbbb
(kali's kali)-[~/Desktop]-$ 
```

## 多条命令带管道

```
(kali's kali)-[~/Desktop]-$ cat "git proxy.md" |grep https
git config --global https.proxy socks5://192.168.56.1:7890
(kali's kali)-[~/Desktop]-$ ls /proc |wc -l
262
(kali's kali)-[~/Desktop]-$ 
```

## 多条命令带管道和重定向

```

(kali's kali)-[~/Desktop]-$ rm a.txt
(kali's kali)-[~/Desktop]-$ cat one-way-north-shell/README.md |grep "encountered a problem" >a.txt
(kali's kali)-[~/Desktop]-$ cat a.txt
I ever encountered a problem, that when I use pipe like `ls /proc | grep 2`, my shell will stuck after grep return its results. First I thought it might because what `ls` puts to pipe doesn't have a EOF flag, so `grep` thinks it haven't meet the end and it keeps on waiting for the EOF. So I test the execute function no longer in a loop but one step once. I already know pipe end need close, but I can't figure out when to close as the examples given by blogs are also hard to understand ( e.g they will close the write end after dup2, or in any other place they close the read end, I just don't know why the close the read/write end there, that is not a common place to close files ). When I was wondering when to close the pipe fd ( both read end and write end ), and I occasionally close the read end before execute `ls` , I found that even `ls` stuck. Then I realized the status of one end can influence the other end. Thus `grep` stuck on read pipe might because the pipe's write end is not closed, that means the unclosed write end will stuck read. It is later proved right with my futher tests.
(kali's kali)-[~/Desktop]-$ cat < one-way-north-shell/README.md | grep "they will close the write end" | wc >>a.txt
(kali's kali)-[~/Desktop]-$ cat a.txt
1      218     1102
(kali's kali)-[~/Desktop]-$ 

```

以上测试均符合预期，功能正常执行！

## 替换默认shell

修改 `/etc/passwd` 文件，将kali用户的默认shell改为nsh，并将编译后的nsh可执行文件放入 `/usr/bin` 目录下（我这里采用symbol link，这样编译之后默认shell直接能用新编译的nsh），重启kali，打开Terminal默认就是我的nsh了：

```

su root
cd /usr/bin
ln -s /home/kali/Desktop/one-way-north-shell/bin/nsh nsh

```

```

-rw-r--r-x 1 root root      3293 Mar  7  2023  nroff
-rw-r--r-x 1 root root    35368 Mar 23 06:02  nsenter
lrwxrwxrwx 1 root root        46 Sep 10 08:17  nsh → /home/kali/Desktop/one-way-north-shell/bin/nsh
-rw-r--r-x 1 root root   113400 Feb 10  2023  nslookup
-rw-r--r-x 1 root root   106952 Feb 25  2023  nstat

```

Shell No.1

File Actions Edit View Help

Powered-by redh3tALWAYS

```
(kali's kali)-[~/Desktop/one-way-north-shell]-$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
(kali's kali)-[~/Desktop/one-way-north-shell]-$
```