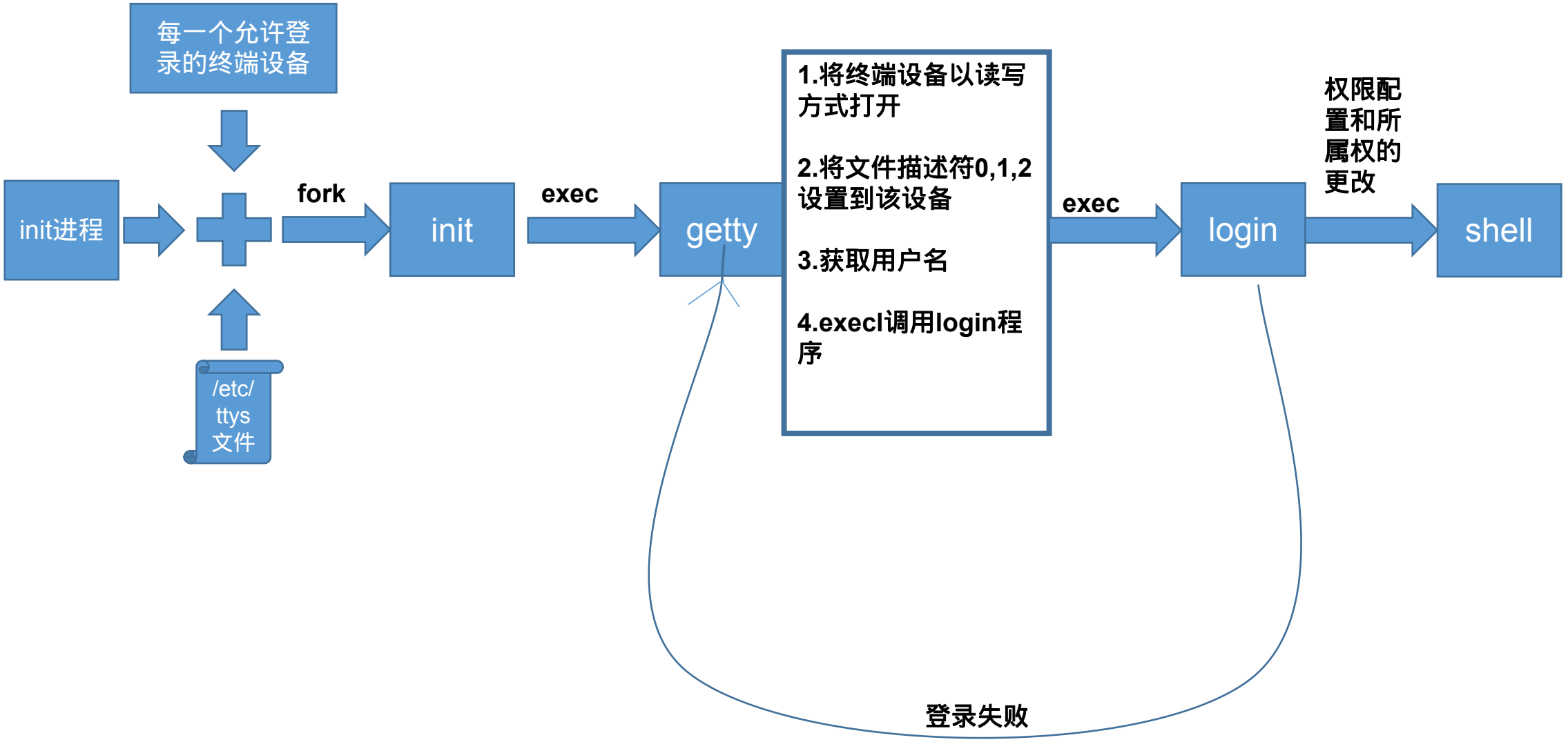
此处多次提到控制终端这个东西，究竟是什么东西， 运行机制是什么，还是不清楚，在19章会有完整的介绍

控制终端并不是shell

终端登录


每一个允许登录的终端设备
init进程
fork
init
exec
getty
1.将终端设备以读写方式打开
2.将文件描述符0,1,2设置到该设备
3.获取用户名
4.execl调用login程序
exec
login
权限配置和所属权的更改
shell
/etc/ttys文件
登录失败

进程组——基本概念

1．进程组：一个或者多个进程的集合

pid_t getpgrp(void); //返回调用进程的进程组id

pid_t getpgid(pid_t pid); //返回指定进程的进程组ID， 失败，返回-1．如果pid=0．相当于调用getpgrp()

进程组——进程组长

1. 进程组长：创建该进程组的进程，进程组长的进程ID等于该进程组的进程组ID

int setpgid(pid_t pid, pid_t pgid); //成功，返回0，失败，返回-1

1. 将pid进程的进程组ID设置为pgid
2. 如果pid==pgid，则pid指定的进程变为组长进程
3. 如果pid==0， 则pid默认为调用者的进程ID
4. 如果pgid为0，则pgid=pid
5. 一个进程只能为它自己或者它的子进程设置进程组ID，子进程调用exec后，就不能更改

```
zhaosong@zhaosong:~/WorkSpace/APUE/9$ ./9_1
parent: My pid: 23928, My gid: 23928
first child before: My pid: 23929, My gid: 23928
first child after: My pid: 23929, My gid: 23929
second child before: My pid: 23930, My gid: 23929
second child after: My pid: 23930, My gid: 23930
```

```c
void print_id(const char *str)
{
    printf("%s: My pid: %d, My gid: %d\n", str, getpid(), getpgid(0));
}

int main()
{
    print_id("parent");
    pid_t pid = fork();
    if(pid == 0)
    {
        print_id("first child before");
        sleep(2);
        print_id("first child after");
        pid = fork();
        if(pid == 0)
        {
            print_id("second child before");
            sleep(2);
            print_id("second child after");
        }
        else
        {
            sleep(1);
            setpgid(pid, 0);
            sleep(3);
        }
    }
    else
    {
        sleep(1);
        setpgid(pid, 0);
        sleep(10);
    }
    return 0;
}
```

进程组——进程组的生命周期

进程被创建直到其内部最后一个进程终止

会话

会话是一个或者多个进程组的集合

pid_t setsid(void)
　　//成功，返回进程组ID， 失败，返回-1
　　1.该函数只能让非进程组组长调用
　　2.该进程变为新会话的会话首进程（创建该会话的进程），此时，该进程是新会话中的唯一进程
　　3.该进程变为一个新进程组的组长进程，进程组id是该调用进程的进程ID
　　4.该进程丢弃控制终端
pid_t getsid(pid_t pid)
　　//成功，返回会话首进程的进程组ID，失败，返回-1
　　pid = 0. 则返回调用者的会话首进程ID，出于安全性考虑，如果pid和调用者不属于同一个会话，则出错

```c
void print_id(const char *str)
{
    printf("%s: my pid: %d, my gid: %d, my sid: %d\n", str, getpid(), getpgid(0), getsid(0))
}


int main()
{
    print_id("parent");
    pid_t pid = fork();
    if(pid == 0)
    {
        print_id("before setsid");
        pid = setsid();
        printf("pid = %d\n", pid);
        print_id("after setsid");
    }
    else
    {
        sleep(5);
    }
    return 0;
}
```

```
zhaosong@zhaosong:~/WorkSpace/APUE/9$ ./9_2
parent: my pid: 26920, my gid: 26920, my sid: 15191
before setsid: my pid: 26921, my gid: 26920, my sid: 15191
pid = 26921
after setsid: my pid: 26921, my gid: 26921, my sid: 26921
```

会话和进程组的高级概念

1.一个会话可以拥有一个控制终端， 通常是终端设备或者伪终端

2.建立与控制终端连接的会话首进程称为控制进程

3.一个会话中的进程组可被分为一个前台进程组和多个后台进程组

4.如果是一个拥有控制终端的会话，则其有一个前台进程组，其余的都是后台进程组

5.键入终端的中断键 、退出键触发的信号发送给 前台进程组的所有进程

6.如果终端断开连接，挂断信号将会被发送给前台进程组

```
pid_t tcgetpgrp(int fd); //返回前台进程组ID，出错，返回-1， fd为控制终端的文件描述符

int tcsetpgrp(int fd, pid_t pgrpid); //成功，返回0， 失败，返回-1， 将pgrpid指定的进程组设置位前台进程组

pid_t tcgetsid(int fd); //成功，返回该控制 终端对应的控制进程的ID，即会话首进程ID，失败，返回-1
```

作业控制

作业就是进程组

作业控制就是进程组控制，详细点就是：允许在一个终端上启动多个作业，它控制哪一个作业可以访问该终端以及那些作业在后台运行

`&` 命令， 将一个进程放在后台执行： `./1 &`

`nohup` 即使关闭了该终端，还可以运行：`nohup ./1 &`

`jobs` 查看后台运行的进程

`fg %`编号 将后台的进程调至前台执行

`ctrl + z` 将正在前台执行1的命令放在后台，并且处于暂停状态

`bg` 将一个在后台暂停的命令，变成在后台继续执行

作业控制——终端输入

我们可以有一个前台作业，若干个后台作业，当我们在终端进行输入输出的时候，谁对这一行为具有控制权？

1．只前台作业接收输入，如果后台作业试图读终端，终端驱动程序将会向后台发送一个SIGTTIN信号，该信号会停止该后台作业，此时shell也会回到前台，向用户发出通知“请把这个程序拿到前台来吧”， 然后用户使用fg将该进程转为前台进程组，并将继续信号SIGCONT发送给该进程组，从而得以继续执行。

2．后台作业输出到控制终端，是否禁止是可选的，当我们不禁止的时候，后台进程依旧可以输出到终端来。我们可以采用stty tostop 命令来禁止后台进程输出的终端， 此时，终端驱动程序向该进程发送SIGTTOU信号，该进程被阻塞，同时shell发出提醒”请把这个进程拿到前台来吧”。

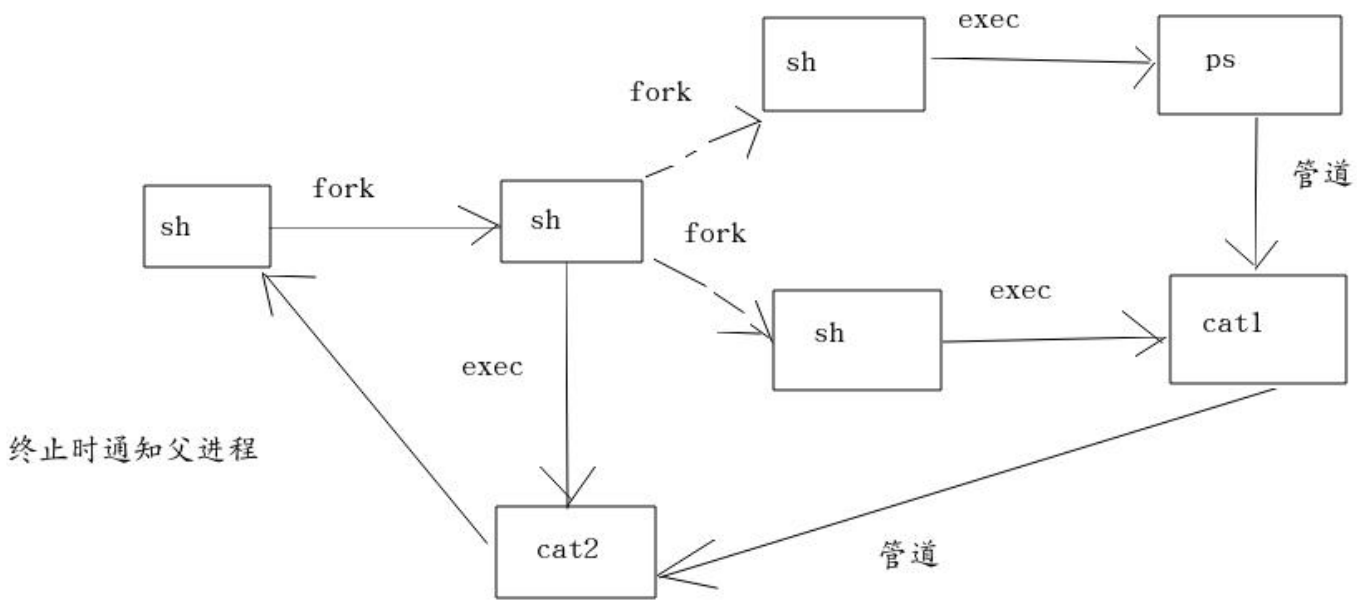
```
zhaosong@zhaosong:~/WorkSpace/APUE/9$ cat > log.txt &
[1] 7957
zhaosong@zhaosong:~/WorkSpace/APUE/9$ nihao
nihao: command not found

[1]+  Stopped                 cat > log.txt

[1]+  Stopped                 cat > log.txt
zhaosong@zhaosong:~/WorkSpace/APUE/9$ fg %1
cat > log.txt
nihao
```

shell执行程序

ps -o | cat1 | cat2



shell执行程序

```
zhaosong@zhaosong:~/WorkSpace/APUE/9$ ps -o pid,ppid,pgid,sid,tpgid,comm
  PID  PPID  PGID   SID TPGID COMMAND
 6602  6595  6602  6602 17389 bash
17389  6602 17389  6602 17389 ps
zhaosong@zhaosong:~/WorkSpace/APUE/9$ ps -o pid,ppid,pgid,sid,tpgid,comm &
[1] 17390
zhaosong@zhaosong:~/WorkSpace/APUE/9$   PID  PPID  PGID   SID TPGID COMMAND
 6602  6595  6602  6602  6602 bash
17390  6602 17390  6602  6602 ps
^C
[1]+  Done                    ps -o pid,ppid,pgid,sid,tpgid,comm
zhaosong@zhaosong:~/WorkSpace/APUE/9$ ps -o pid,ppid,pgid,sid,tpgid,comm | cat
  PID  PPID  PGID   SID TPGID COMMAND
 6602  6595  6602  6602 17391 bash
17391  6602 17391  6602 17391 ps
17392  6602 17391  6602 17391 cat
zhaosong@zhaosong:~/WorkSpace/APUE/9$ ps -o pid,ppid,pgid,sid,tpgid,comm | cat &
[1] 17394
zhaosong@zhaosong:~/WorkSpace/APUE/9$   PID  PPID  PGID   SID TPGID COMMAND
 6602  6595  6602  6602  6602 bash
17393  6602 17393  6602  6602 ps
17394  6602 17393  6602  6602 cat
^C
[1]+  Done                    ps -o pid,ppid,pgid,sid,tpgid,comm | cat
```

tpgid:前台进程组ID

孤儿进程组

什么是孤儿进程组？

该组中每个成员的父进程要么是该组的一个成员，要么不是该组所属会话的成员。

对于一个孤儿进程组，要求终端向孤儿进程组中处于停止状态的每一个进程发送挂断信号SIGHUP， 接着向其发送继续信号SIGCONT

对于处于后台的孤儿进程组来说，当其视图读取终端输入的时候，将会被SIGTTIN永远的停止，shell不会报告。

父进程终止后，如果该进程组变为孤儿进程组，将会自动转为后台进程组