

## 4-2 加餐 - 进程间关系与守护进程

### 1. 进程组

#### 1-1 什么是进程组

之前我们提到了进程的概念，其实每一个进程除了有一个进程 ID(PID)之外 还属于一个进程组。进程组是一个或者多个进程的集合，一个进程组可以包含多个进程。每一个进程组也有一个唯一的进程组 ID(PGID)，并且这个 PGID 类似于进程 ID，同样是一个正整数，可以存放在 pid\_t 数据类型中。

C++

```
$ ps -eo pid,pgid,ppid,comm | grep test
```

#结果如下

| PID  | PGID | PPID | COMMAND |
|------|------|------|---------|
| 2830 | 2830 | 2259 | test    |

# -e 选项表示 every 的意思，表示输出每一个进程信息

# -o 选项以逗号操作符(,)作为定界符，可以指定要输出的列

#### 1-2 组长进程

每一个进程组都有一个组长进程。组长进程的 ID 等于其进程 ID。我们可以通过 ps 命令看到组长进程的现象：

Shell

```
[node@localhost code]$ ps -o pid,pgid,ppid,comm | cat
```

# 输出结果

| PID  | PGID | PPID | COMMAND |
|------|------|------|---------|
| 2806 | 2806 | 2805 | bash    |
| 2880 | 2880 | 2806 | ps      |
| 2881 | 2880 | 2806 | cat     |

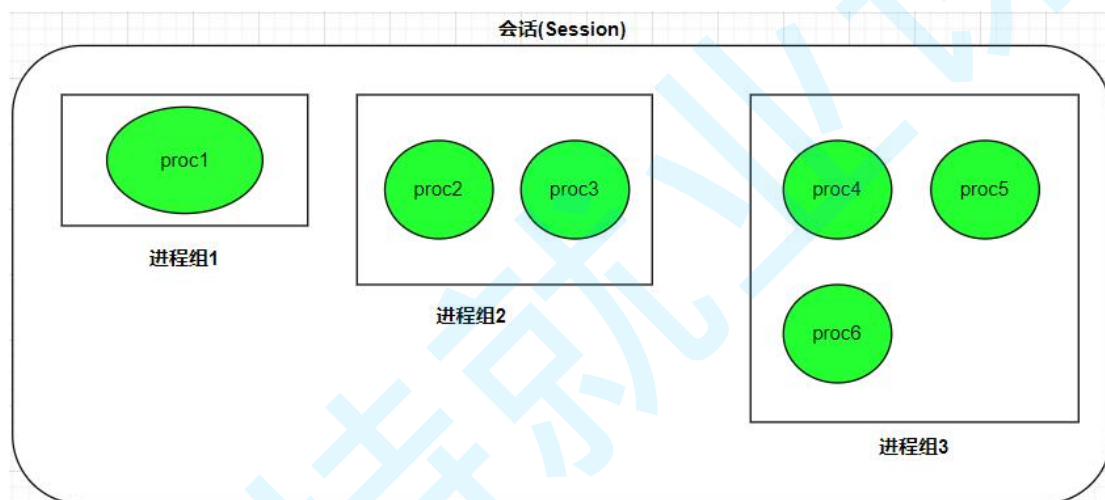
从结果上看 ps 进程的 PID 和 PGID 相同，那也就是说明 ps 进程是该进程组的组长进程，该进程组包括 ps 和 cat 两个进程。

- 进程组组长的作用：进程组组长可以创建一个进程组或者创建该组中的进程
- 进程组的生命周期：从进程组创建开始到其中最后一个进程离开为止。**注意：**主要某个进程组中有一个进程存在，则该进程组就存在，这与其组长进程是否已经终止无关。

## 2. 会话

### 2-1 什么是会话

刚刚我们谈到了进程组的概念，那么会话又是什么呢？会话其实和进程组息息相关，会话可以看成是一个或多个进程组的集合，一个会话可以包含多个进程组。每一个会话也有一个会话 ID(SID)



通常我们都是使用管道将几个进程编成一个进程组。如上图的进程组 2 和进程组 3 可能是由下列命令形成的：

```
Shell
[node@localhost code]$ proc2 | proc3 &
[node@localhost code]$ proc4 | proc5 | proc6 &
# &表示将进程组放在后台执行
```

我们举一个例子观察一下这个现象：

```
Shell
# 用管道和 sleep 组成一个进程组放在后台运行
[node@localhost code]$ sleep 100 | sleep 200 | sleep 300 &

# 查看 ps 命令打出来的列描述信息
[node@localhost code]$ ps axj | head -n1
```

```
# 过滤 sleep 相关的进程信息
[node@localhost code]$ ps axj | grep sleep | grep -v grep
# a 选项表示不仅列当前用户的进程，也列出所有其他用户的进程
# x 选项表示不仅列有控制终端的进程，也列出所有无控制终端的进程
# j 选项表示列出与作业控制相关的信息， 作业控制后续会讲
# grep 的 -v 选项表示反向过滤， 即不过滤带有 grep 字段相关的进程

# 结果如下
```

| PPID | PID  | PGID | SID  | TTY   | TPGID | STAT | UID  | TIME | COMMAND   |
|------|------|------|------|-------|-------|------|------|------|-----------|
| 2806 | 4223 | 4223 | 2780 | pts/2 | 4229  | S    | 1000 | 0:00 | sleep 100 |
| 2806 | 4224 | 4223 | 2780 | pts/2 | 4229  | S    | 1000 | 0:00 | sleep 200 |
| 2806 | 4225 | 4223 | 2780 | pts/2 | 4229  | S    | 1000 | 0:00 | sleep 300 |

从上述结果来看 3 个进程对应的 PGID 相同，即属于同一个进程组。

## 2-2 如何创建会话

可以调用 `setseid` 函数来创建一个会话，前提是调用进程不能是一个进程组的组长。

```
C
#include <unistd.h>
/*
 *功能：创建会话
 *返回值：创建成功返回 SID，失败返回-1
 */
pid_t setsid(void);
```

该接口调用之后会发生：

- 调用进程会变成新会话的**会话首进程**。此时，新会话中只有唯一的一个进程
- 调用进程会变成进程组组长。新进程组 ID 就是当前调用进程 ID
- 该进程没有控制终端。如果在调用 `setsid` 之前该进程存在控制终端，则调用之后会切断联系

需要注意的是：这个接口如果调用进程原来是进程组组长，则会报错，为了避免这种情况，我们通常的使用方法是先调用 `fork` 创建子进程，父进程终止，子进程继续执行，因为子进程会继承父进程的进程组 ID，而进程 ID 则是新分配的，就不会出现

错误的情况。

## 2-3 会话 ID(SID)

上边我们提到了会话 ID，那么会话 ID 是什么呢？我们可以先说一下会话首进程，会话首进程是具有唯一进程 ID 的单个进程，那么我们可以将会话首进程的进程 ID 当做是会话 ID。注意：会话 ID 在有些地方也被称为会话首进程的进程组 ID，因为会话首进程总是一个进程组的组长进程，所以两者是等价的。

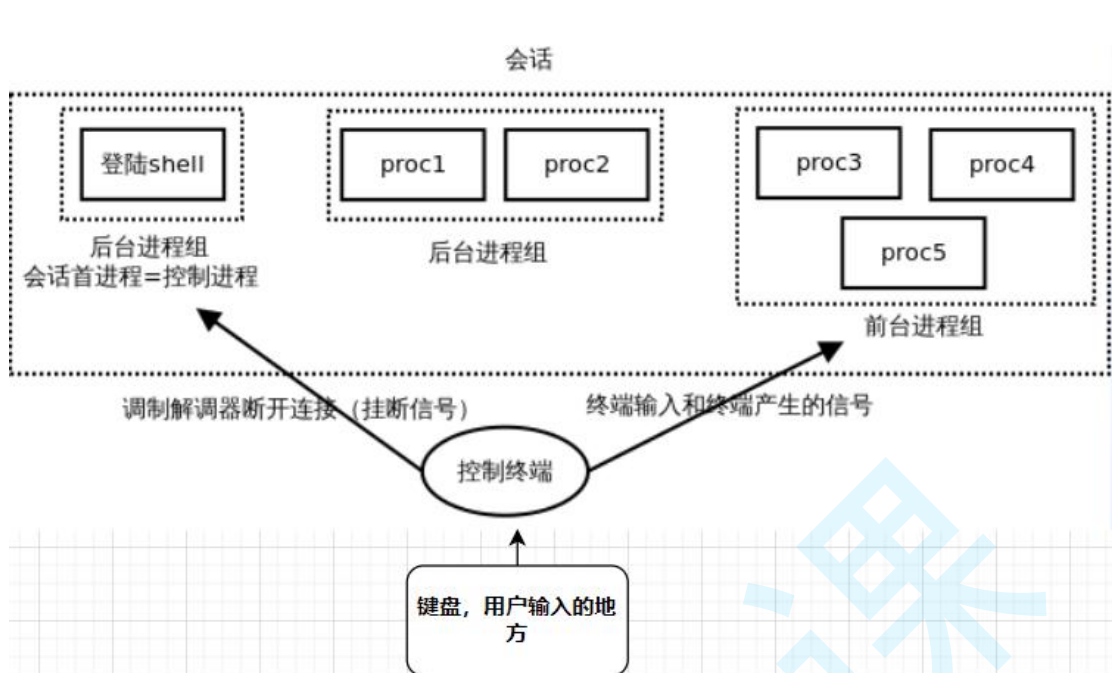
## 3. 控制终端

先说一下什么是控制终端？

在 UNIX 系统中，用户通过终端登录系统后得到一个 **Shell** 进程，这个终端成为 **Shell** 进程的控制终端。控制终端是保存在 **PCB** 中的信息，我们知道 **fork** 进程会复制 **PCB** 中的信息，因此由 **Shell** 进程启动的其它进程的控制终端也是这个终端。默认情况下没有重定向，每个进程的标准输入、标准输出和标准错误都指向控制终端，进程从标准输入读也就是读用户的键盘输入，进程往标准输出或标准错误输出写也就是输出到显示器上。另外会话、进程组以及控制终端还有一些其他的关系，我们在下边详细介绍一下：

- 一个会话可以有一个控制终端，通常会话首进程打开一个终端（终端设备或伪终端设备）后，该终端就成为该会话的控制终端。
- 建立与控制终端连接的会话首进程被称为**控制进程**。
- 一个会话中的几个进程组可被分成一个**前台进程组**以及一个或者多个**后台进程组**。
- 如果一个会话有一个控制终端，则它有一个前台进程组，会话中的其他进程组则为后台进程组。
- 无论何时进入终端的中断键（**ctrl+c**）或退出键（**ctrl+\**），就会将中断信号发送给前台进程组的所有进程。
- 如果终端接口检测到调制解调器（或网络）已经断开，则将挂断信号发送给控制进程（会话首进程）。

这些特性的关系如下图所示：



## 4. 作业控制

### 4-1 什么是作业(job)和作业控制(Job Control)?

**作业**是针对用户来讲，用户完成某项任务而启动的进程，一个作业既可以只包含一个进程，也可以包含多个进程，进程之间互相协作完成任务，通常是一个进程管道。

Shell 分前后台来控制的不是进程而是作业 或者进程组。一个前合作业可以由多个进程组成，一个后合作业也可以由多个进程组成，Shell 可以同时运行一个前合作业和任意多个后合作业，这称为**作业控制**。

例如下列命令就是一个作业，它包括两个命令，在执行时 Shell 将在前台启动由两个进程组成的作业：

```
Shell
[node@localhost code]$ cat /etc/filesystems | head -n 5
```

运行结果如下所示：

```
Shell
xfs
ext4
ext3
ext2
nodev proc
```

## 4-2 作业号

放在后台执行的程序或命令称为后台命令，可以在命令的后面加上`&`符号从而让 Shell 识别这是一个后台命令，后台命令不用等待该命令执行完成，就可立即接收新的命令，另外后台进程执行完后会返回一个作业号以及一个进程号（PID）。

例如下面的命令在后台启动了一个作业，该作业由两个进程组成，两个进程都在后台运行：

```
Shell
[node@localhost code]$ cat /etc/filesystems | grep ext &
```

执行结果如下：

```
Shell
[1] 2202
ext4
ext3
ext2
# 按下回车
[1]+  完成                  cat /etc/filesystems | grep --
color=auto ext
```

- 第一行表示作业号和进程 ID，可以看到作业号是 1，进程 ID 是 2202
- 第 3-4 行表示该程序运行的结果，过滤 `/etc/filesystems` 有关 `ext` 的内容
- 第 6 号分别表示作业号、默认作业、作业状态以及所执行的命令

关于默认作业：对于一个用户来说，只能有一个默认作业（+），同时也只能有一个即将成为默认作业的作业（-），当默认作业退出后，该作业会成为默认作业。

- +：表示该作业号是默认作业
- ：表示该作业即将成为默认作业
- 无符号：表示其他作业

## 4-3 作业状态

常见的作业状态如下表所示：

| 作业状态              | 含义                 |
|-------------------|--------------------|
| 正在运行【Running】     | 后台作业（&），表示正在执行     |
| 完成【Done】          | 作业已完成，返回的状态码为0     |
| 完成并退出【Done(code)】 | 作业已完成并退出，返回的状态码为非0 |
| 已停止【Stopped】      | 前台作业，当前被Ctrl+Z挂起   |
| 已终止【Terminated】   | 作业被终止              |

点击图片可查看完整电子表格

## 4-4 作业的挂起与切回

### (1) 作业挂起

我们在执行某个作业时，可以通过 **Ctrl+Z** 键将该作业挂起，然后 Shell 会显示相关的作业号、状态以及所执行的命令信息。

例如我们运行一个死循环的程序，通过 **Ctrl+Z** 将该作业挂起，观察一下对应的作业状态：

```
C
#include <stdio.h>

int main()
{
    while (1)
    {
        printf("hello\n");
    }
    return 0;
}
```

下面我运行这个程序，通过 **Ctrl+Z** 将该作业挂起：

```
Shell
# 运行可执行程序
[node@localhost code]$ ./test
#键入 Ctrl + Z 观察现象
```

运行结果如下：

```
Shell
# 结果依次对应作业号 默认作业 作业状态 运行程序信息
```

```
[1]+ 已停止                  ./test7
```

可以发现通过 **Ctrl+Z** 将作业挂起，该作业状态已经变为了停止状态

## (2) 作业切回

如果想将挂起的作业切回，可以通过 **fg** 命令，**fg** 后面可以跟作业号或作业的命令名称。如果参数缺省则会默认将作业号为 1 的作业切到前台来执行，若当前系统只有一个作业在后台进行，则可以直接使用 **fg** 命令不带参数直接切回。具体的参数参考如下：

| 参数       | 含义              |
|----------|-----------------|
| %n       | n为正整数，表示作业号     |
| %string  | 以字符串开头的命令所对应的作业 |
| %?string | 包含字符串的命令所对应的作业  |
| %+或%%    | 最近提交的一个作业       |
| %-       | 倒数第二个提交的作业      |

点击图片可查看完整电子表格

例如我们把刚刚挂起来的 **./test** 作业切回到前台：

```
Shell
[node@localhost code]$ fg %%
```

运行结果为开始无限循环打印 **hello**，可以发现该作业已经切换到前台了。

**注意：**当通过 **fg** 命令切回作业时，若没有指定作业参数，此时会将默认作业切到前台执行，即带有“+”的作业号的作业

## 4-5 查看后台执行或挂起的作业

我们可以直接通过输入 **jobs** 命令查看本用户当前后台执行或挂起的作业

- 参数 **-l** 则显示作业的详细信息
- 参数 **-p** 则只显示作业的 PID

例如，我们先在后台及前台运行两个作业，并将前作业挂起，来用 **jobs** 命令查看作业相关的信息：

```
Shell
# 在后台运行一个作业 sleep
[node@localhost code]$ sleep 300 &
```



```
# 运行刚才的死循环可执行程序
[node@localhost code]$ ./test
# 键入 Ctrl + Z 挂起作业
# 使用 jobs 命令查看后台及挂起的作业
[node@localhost code]$ jobs -l
```

运行结果如下所示：

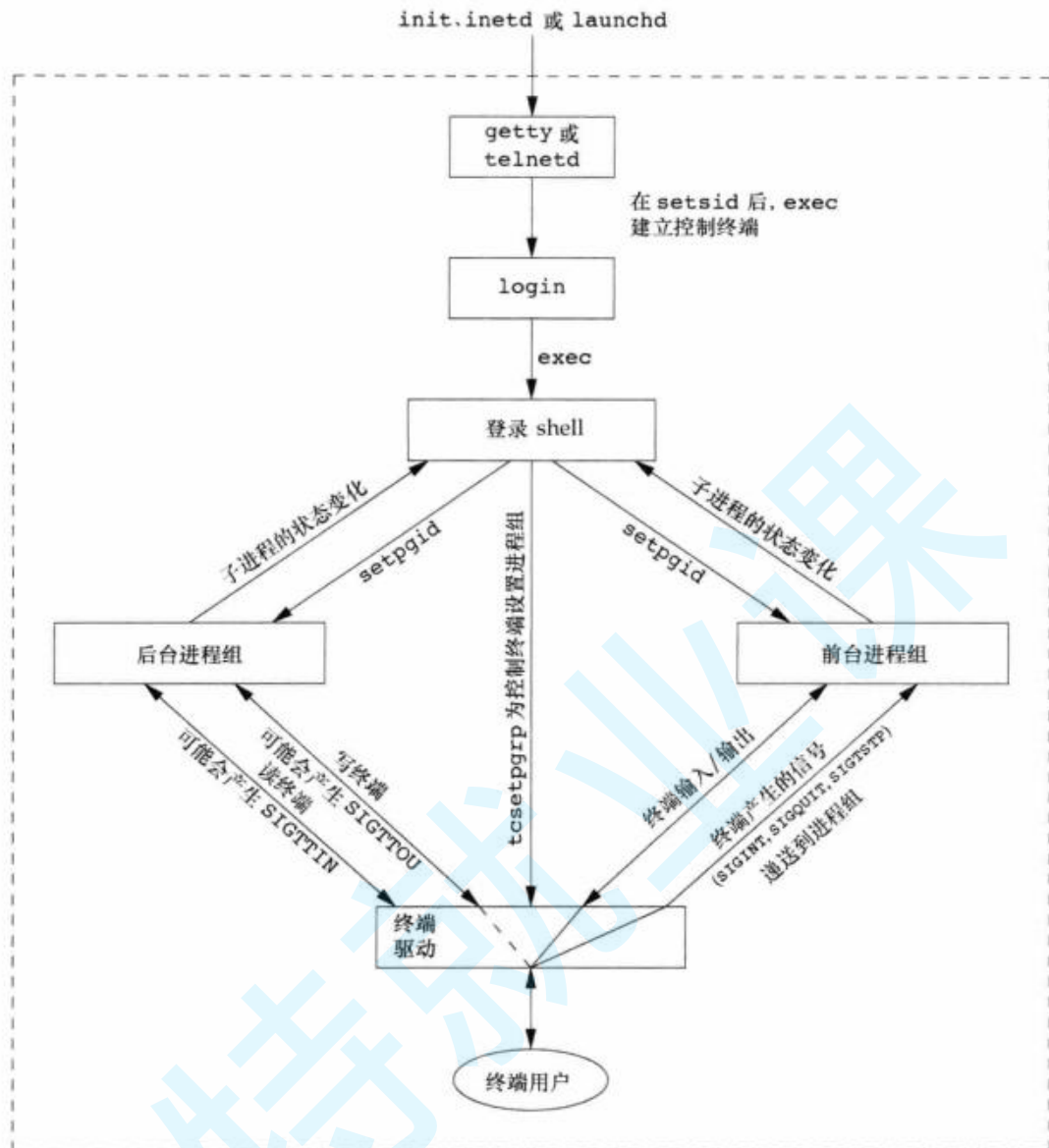
```
Shell
# 结果依次对应作业号 默认作业 作业状态 运行程序信息
[1]-  2265 运行中                sleep 300 &
[2]+  2267 停止                  ./test7
```

## 4-6 作业控制相关的信号

上面我们提到了键入 `Ctrl + Z` 可以将前台作业挂起，实际上是将 `STGTSTP` 信号发送至前台进程组作业中的所有进程，后台进程组中的作业不受影响。在 `unix` 系统中，存在 3 个特殊字符可以使得终端驱动程序产生信号，并将信号发送至前台进程组作业，它们分别是：

- `Ctrl + C`：中断字符，会产生 `SIGINT` 信号
- `Ctrl + \`：退出字符，会产生 `SIGQUIT` 信号
- `Ctrl + Z`：挂起字符，会产生 `STGTSTP` 信号

终端的 I/O(即标准输入和标准输出)和终端产生的信号总是从前台进程组作业连接打破实际终端。我们可以通过下体来看到作业控制的功能：



## 5. 守护进程

Daemon.hpp

```

C++
#pragma once

#include <iostream>
#include <cstdlib>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

```

```
const char *root = "/";
const char *dev_null = "/dev/null";

void Daemon(bool ischdir, bool isclose)
{
    // 1. 忽略可能引起程序异常退出的信号
    signal(SIGCHLD, SIG_IGN);
    signal(SIGPIPE, SIG_IGN);

    // 2. 让自己不要成为组长
    if (fork() > 0)
        exit(0);

    // 3. 设置让自己成为一个新的会话， 后面的代码其实是子进程在走
    setsid();

    // 4. 每一个进程都有自己的 CWD，是否将当前进程的 CWD 更改成为 /
    根目录
    if (ischdir)
        chdir(root);

    // 5. 已经变成守护进程啦，不需要和用户的输入输出，错误进行关联
    了
    if (isclose)
    {
        close(0);
        close(1);
        close(2);
    }
    else
    {
        // 这里一般建议就用这种
        int fd = open(dev_null, O_RDWR);
        if (fd > 0)
        {
            dup2(fd, 0);
            dup2(fd, 1);
            dup2(fd, 2);
            close(fd);
        }
    }
}
```

## 6. 如何将服务守护进程化

```
C++
// ./server port
int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        std::cout << "Usage : " << argv[0] << " port" <<
std::endl;
        return 0;
    }
    uint16_t localport = std::stoi(argv[1]);

    Daemon(false, false);
    std::unique_ptr<TcpServer> svr(new TcpServer(localport,
HandlerRequest));

    svr->Loop();

    return 0;
}
```