# 写一个daemon进程

<mnstory.net>

linux API提供int daemon(int nochdir, int noclose)函数，功能相对简单，有的地方考虑并不周全，比如文件描述符继承且没有close，raf@raf.org的<http://libslack.org/> 里有daemon的实现（也是shell里面daemon命令的源码），逻辑严谨，注释清晰，先欣赏一下他的源码，如果能看明白，这篇文章就不用看了，如果看不明白，请继续。

*int* daemon\_init(const *char* \*name)

{

*pid\_t* pid;

*long* nopen;

*int* fd;

/\* Don't setup a daemon-friendly process context if started by init(8) or inetd(8). \*/

    if (!(daemon\_started\_by\_init() || daemon\_started\_by\_inetd()))

    {

/\* Background the process. Lose process session/group leadership. \*/

        if ((pid = fork()) == -1)

            return -1;

        if (pid)

            exit(EXIT\_SUCCESS);

/\* Become a process session leader. This can only fail when we're already a session leader. \*/

        setsid();

#ifndef NO\_EXTRA\_SVR4\_FORK

#ifdef SVR4

/\* Ignore SIGHUP because when the session leader terminates (which is about to happen), all processes in the foreground process group are sent the SIGHUP signal (apparently). It is expected that clients will set their own SIGHUP handler after the call to daemon\_init() if necessary. \*/

        {

*struct* sigaction act[1];

            act->sa\_handler = SIG\_IGN;

            sigemptyset(&act->sa\_mask);

            act->sa\_flags = 0;

            if (sigaction(SIGHUP, act, NULL) == -1)

                return -1;

        }

/\* Lose process session leadership to prevent gaining a controlling terminal in SVR4. \*/

        if ((pid = fork()) == -1)

            return -1;

        if (pid)

            exit(EXIT\_SUCCESS);

#endif

#endif

    }

    /\* Enter the root directory to prevent hampering umounts. \*/

    if (chdir(ROOT\_DIR) == -1)

        return -1;

    /\* Clear umask to enable explicit file modes. \*/

    umask(0);

/\* We need to close all open file descriptors. Check how many file descriptors we have (If indefinite, a usable number (1024) will be returned). Flaw: If many files were opened and then this limit was reduced to below the highest file descriptor, we may not close all file descriptors. \*/

    if ((nopen = limit\_open()) == -1)

        return -1;

/\* Close all open file descriptors. If started by inetd, we don't close stdin, stdout and stderr. Don't forget to open any future tty devices with O\_NOCTTY so as to prevent gaining a controlling terminal (not necessary with SVR4). \*/

    if (daemon\_started\_by\_inetd())

    {

        for (fd = 0; fd < nopen; ++fd)

        {

            switch (fd)

            {

                case STDIN\_FILENO:

                case STDOUT\_FILENO:

                case STDERR\_FILENO:

                    break;

                default:

                    close(fd);

            }

        }

    }

    else

    {

        for (fd = 0; fd < nopen; ++fd)

            close(fd);

/\* Open stdin, stdout and stderr to /dev/null just in case some code buried in a library somewhere expects them to be open. \*/

        if ((fd = open("/dev/null", O\_RDWR)) == -1)

            return -1;

/\* This is only needed for very strange (hypothetical) POSIX implementations where STDIN\_FILENO != 0 or STDOUT\_FILE != 1 or STDERR\_FILENO != 2 (yeah, right). \*/

        if (fd != STDIN\_FILENO)

        {

            if (dup2(fd, STDIN\_FILENO) == -1)

                return -1;

             close(fd);

        }

        if (dup2(STDIN\_FILENO, STDOUT\_FILENO) == -1)

            return -1;

        if (dup2(STDIN\_FILENO, STDERR\_FILENO) == -1)

            return -1;

    }

     return 0;

}

要写一个daemon进程，需要了解如下知识。

## 终端设备

终端设备(terminal or tty devices) 是一类特定的字符设备，他可以作为session的控制终端（为session提供输入输出界面），包含如下三类：

1. 虚拟控制台(virtual consoles)

为什么要有控制台？

进程输出一些信息，用户要输入一些信息，例如单模式下提示用户输入用户名密码等，很多情况下离不开“人机交互界面”，这就是控制台，简单点想，就是你接在主机上的显示器，你也可以不接显示器，它也一样存在，只是你就要摸黑了。

控制台设备为/dev/console，有了控制台就可以供人机交互，但当你和其他人共用一台机器的时候，用同一个显示器进入，大家敲的命令，执行的输出都搅在一起，谁都感觉不爽，于是虚拟控制台的概念出来了。

虚拟控制台命名为/dev/tty[1-N]，其中/dev/tty0表示当前虚拟控制台。这里的每一个/dev/tty#就代表一个虚拟控制台，有了这玩意，按一下alt+F[1-6]就可以自由切换多个虚拟控制台，每个虚拟控制台的输入输出和权限可以相互独立，互不干涉。

1. 串口(serial ports)

既然你桌上的显示器是用来做人机交互的，类似你桌上的显示器的硬件还很多，而且形形色色，通过串口走的其他字符设备，也能做终端，于是也叫/dev/ttyXXX，那XXX怎么规定好呢，以前的设备种类比较少，于是就用了一个字符来区分设备种类，剩下的就是序号，所以你可以看到串口设备在系统里面是/dev/ttyS#或/dev/ttyC#这样的。

1. 伪终端(pseudoterminals or PTYs)

后来有了网络，绝大多数时候，我们都不用跑到机房打开显示器去使用服务器，而是用ssh等软件远程连接到服务器，客户端软件的输入窗口就变成了类似控制台的功能，可以输入，也可以输出。

那么，如何让远程连接的程序也能和系统里的进程做人机交互呢？

client server

+-------+ internet +-------------------------+

| putty |-------------> | sshd |

+-------+ | | |

| +- spawn 'vi main.c' |

| +- spawn 'dmesg' |

+-------------------------+

类似上图，我们可以让sshd将putty接收到的数据转发到vi进程，也可以让dmesg输出的数据转发到sshd再由sshd转发到putty，当然，系统为这部分转发做了抽象，抽象成为伪终端。

伪终端有server和client的概念，类似sshd进程，只需要打开server端，类似vi和dmesg只需要打开client端，他们之间即可传递输入输出数据。在linux下，这个server端是/dev/ptmx，client端是/dev/pts/#，不同的/dev/pts/#之间做隔离互不影响，这样可以同时支持多个伪终端。

如下，两个sshd进程打开/dev/ptmx各三次，生成两个伪终端pts/0 pts/1：

# lsof /dev/ptmx

COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME

sshd 125860 root 7u CHR 5,2 0t0 12660 /dev/ptmx

sshd 125860 root 9u CHR 5,2 0t0 12660 /dev/ptmx

sshd 125860 root 10u CHR 5,2 0t0 12660 /dev/ptmx

sshd 230540 root 7u CHR 5,2 0t0 12660 /dev/ptmx

sshd 230540 root 9u CHR 5,2 0t0 12660 /dev/ptmx

sshd 230540 root 10u CHR 5,2 0t0 12660 /dev/ptmx

# ps xf -o pid,pgid,ppid,sid,tty,start,command

PID PGID PPID SID TT STARTED COMMAND

10757 10757 1 10757 ? Oct 11 /usr/sbin/sshd

125860 125860 10757 125860 ? 09:22:17 \\_ sshd: root@pts/0

126015 126015 125860 126015 pts/0 09:22:18 | \\_ -bash

136159 136159 126015 126015 pts/0 09:22:32 | \\_ /usr/bin/perl

230540 230540 10757 230540 ? 09:50:58 \\_ sshd: root@pts/1

230665 230665 230540 230665 pts/1 09:50:58 \\_ -bash

154533 154533 230665 230665 pts/1 11:08:12 \\_ ps xf -o pid,pgid

类似printf，最终将数据输出到终端/dev/tty，你也可以，打开两个ssh感受下：

echo "hello, i am pts0" > /dev/pts/1

图形界面下也有字符输入输出，dabian下x-window控制台一般是/dev/tty7。

## 进程组(GROUP)与会话(SESSION)

我们知道进程的概念，每个进程他有自己的父进程，也有自己所在的进程组，进程组之于进程正如文件夹之于文件，主要是方便统一管理，例如，你可以使用"kill -SIGNUMBER -PGID"向进程组ID发送信号，则进程组内所有的进程都会收到相同信号。

每个进程组有进程组ID(group leader id)，该ID就是进程组组长的PID，可调用setpgid函数来设置组ID。

如果一个进程使用open tty打开一个终端设备，使得进程可以和终端设备进行人机交互，那么这个进程就叫前台进程。

理论上来说，一个终端设备只能同时关联到一个进程，但是我们可能有多个进程都要同时和终端交互的需求，比如使用 dmesg | grep error等命令，创建了一组进程，这组进程都可以输出到终端并接受到ctrl+c发出的SIGINT，比如使用gdb调试进程，父子进程都可以交叉接收你的输入，所以和终端设备交互的单位可以认为是进程组，能够关联控制终端的进程组叫做前台进程组(foreground process group)，因为控制终端一次只能服务于一个进程组，那些当前没获得控制终端的，叫做后台进程组(background process group)。

你可以在命令行后面加&，使得一个进程变成后台进程，也可以在输入ctrl+z将当前运行的前台进程变成后台进程，然后通过jobs可以查看当前的后台进程列表，通过 fg %x进后台进程调为前台进程，x表示jobs里面显示的工作号。

# sleep 33

^Z

[1]+ Stopped sleep 33

# jobs

[1]+ Stopped sleep 33

# fg %1

sleep 33

^C

但是ctrl+z和&还有一些区别，比如ctrl+z变成后台进程后，进程同时处于stopped状态，而&是running状态：

# sleep 30

^Z

[1]+ Stopped sleep 30

# sleep 33 &

[2] 4351

# jobs

[1]+ Stopped sleep 30

[2]- Running sleep 33 &

一个session可以有多个进程组，这些进程组可以由一个前台进程组(forkground process group)和多个后台进程组(background process group)组成。

在控制终端输入的中断键、退出键会转换为信号发送到前端进程组，所以，你会发现Ctrl+c可以停止一组进程，控制终端还会产生一些其他信号，一个健壮的程序都应该明确知道如何处理：

输入CTRL+c，产生SIGINT信号。

输入CTRL+\，产生SIGQUIT信号。

输入CTRL+z，产生SIGTSTP信号。

后台进程尝试读terminal的时候，产生SIGTTIN信号。

后台进程尝试写terminal的时候，产生SIGTTOU信号。

断开终端，产生SIGHUP信号，发送给session leader（有人说也发送给孤儿进程组，但是我测试发现孤儿进程没反应）。

我们可以调用setsid是来创建一个新的session，该函数内部调用setpgid（表明既是新的session又是group leader）然后调用ioctl(TIOCNOTTY)脱离控制终端。

## 守护进程

### 后台进程

一般来说，我们理解的后台进程就是在命令后面加&操作符，终端进行了一次fork操作，让命令走新的fork分支，而不阻塞终端窗口输入。此后台进程没有获得控制终端，不能接收输入（但是可以往终端输出信息）。

在终端窗口发送的控制键(例如ctrl+c)不能作用于后台进程，所以不会导致后台进程退出， 但是，以远程ssh为例，关闭伪终端的时候，会发送SIGHUP给session leader process，对于leader进程而言，收到这个信号有自己不同的处理方式。

以ubuntu的bash为例，它会清理所有jobs，会往"所有在当前session的进程发送sighup"(这里我个人理解，没有找到相关资料，测试出来的现象并不能跨session，但是如果是当前session，那么setsid后，还有必要ignore sighup吗？) 不管是运行状态还是非运行状态(非运行的会先cont再sighup)，默认情况下，进程收到sighup都会退出，所以，如果你想让自己的进程不至于在关闭终端窗口后就退出，你需要屏蔽sighup信号（nohup命令执行效果相同），详情参考 <https://www.gnu.org/software/bash/manual/html_node/Signals.html#Signals>。

此外，并非所有session leader都使用类似处理方式，基于不同实现和配置，有的处理只给前台进程发送sighup，后台进程在终端退出后会变成孤儿。

### 守护进程

后台进程已经可以在一定程度上不受控制终端约束，可达到控制终端退出后自己仍然运行的目的，但是还有几个问题：

1. 上面讲了关闭终端后的sighup问题，可能导致后台进程退出。

2. 控制终端退出了，后台进程之前的输出怎么办？所以守护进程应该是无控制终端的。

3. LINUX的进程会继承多父进程的很多特性，例如文件mask，当前目录，文件句柄，信号处理特征等，可能导致进程本身和环境依赖相关，理应重置。

## 守护进程的实现

1. 为了防止守护进程变成僵尸，我们第一步是让进程变成一个孤儿进程。

/\* step1: first time fork and exit parent, make child as a orphan \*/

    if (-1 == (pid = fork())) {

        lerror("first time fork failed %d", errno);

        return -1;

    }

    if (0 != pid) {

        exit(0);

    }

2. 要脱离控制终端的控制，就要和控制终端所在的session分开

/\* step2: child as a new session and a group leader. This can only fail when we're aleady a session leader. \*/

setsid();

setsid函数，会让当前进程变成新的进程组组长，并设置新session，这样，当前进程所在的session就和控制终端所在的session隔离开了，控制终端所在session接收到的输入或信号，不会影响到当前进程。我们可以看一下session源码：

/\* Create a new session with the calling process as its leader. The process group IDs of the session and the calling process are set to the process ID of the calling process, which is returned. \*/

*int* \_\_setsid ()

{

*pid\_t* pid = getpid ();

*int* tty;

*int* save = errno;

    if (\_\_getpgid (pid) == pid)

    {

        /\* Already the leader.  \*/

        \_\_set\_errno (EPERM);

        return -1;

    }

    //设置当前进程为新进程组组长

    if (setpgid (pid, pid) < 0)

        return -1;

    //使用 IOCTL 脱离 控制终端

    tty = open ("/dev/tty", 0);

    if (tty < 0)

    {

        \_\_set\_errno (save);

        return 0;

    }

    (*void*) \_\_ioctl (tty, TIOCNOTTY, 0);

    (*void*) \_\_close (tty);

    \_\_set\_errno (save);

    return 0;

}

3. 屏蔽掉sighup信号

按照我上面的疑问，如果session leader不会跨session发送sighup，这里似乎也不那么必要，如果这里设置了ignore，子进程也会继承。

/\* step3: Ignore SIGHUP because when the session leader terminates, all processes in the FOREGROUND process group are sent the SIGHUP signal (apparently). It is expected that clients will set their own SIGHUP handler after the call todaemon\_init() if necessary.

    \*/

    {

*struct* sigaction act[1];

        act->sa\_handler = SIG\_IGN;

        sigemptyset(&act->sa\_mask);

        act->sa\_flags = 0;

        if (sigaction(SIGHUP, act, NULL) == -1) {

            lerror("set SIGHUP failed %d", errno);

            return -1;

        }

    }

4. 二次fork，防止后续有误操作再次关联控制终端

这里二次fork的目的，是防止后续误操作再次关联控制终端，因为第二步我们已经设置了当前进程为session leader，而session leader是能够open tty，虽然自己写的代码很清楚不会这么干，但是不排除后续调用别人的代码会有如此操作，于是这里就先fork一个子进程，子进程不是session leader，父进程(原来的session leader)退出，子进程变成当前进程，后续即使有人想关联控制终端，也必然关联失败。

/\* step4: Lose process session leadership to prevent gaining a controlling terminal in SVR4.

    \*/

    if (-1 == (pid = fork())) {

        lerror("second time fork failed %d", errno);

        return -1;

    }

    if (0 != pid) {

        exit(0);

    }

5. 重置一些继承过来的变量

上面说过，文件mask，当前目录，文件句柄，信号处理特征等都会继承，这里只处理当前目录和文件mask。

/\* step5: reset inherited env \*/

    /\* Enter the root directory to prevent hampering umounts. \*/

    if (-1 == chdir("/")) {

        lerror("chdir to / %d", errno);

        return -1;

    }

    /\* Clear umask to enable explicit file modes. \*/

umask(0);

6. 关闭继承描述符并设置输入输出到/dev/null

本来可以和5合在一起说，但是这个解释有点多，所以单独出来。

为什么要关闭继承句柄？请参见实际工程中遇到的BUG《定位Python执行命令僵尸卡死》一文。

因为守护进程和控制中断失去了关联，输人输出都不能再继续，所以重置输入输出为/dev/null，这样没有了输入输出，也不会有ctrl+z之类的触发信号，所以上述讲到的哪些信号屏蔽也不需要做。

/\* step6: We need to close all open file descriptors, and redirect /dev/null \*/

    closeAllfds(1);

    if(-1 == redirectDftFD("/dev/null")) {

        lerror("can't redirect default fd to /dev/null");

        return -1;

}

两函数实现为：

*int* closeAllfds(*int* bIngoreDftFD) {

*struct* rlimit rl;

*int* closeCnt = 0;

    if(-1 == getrlimit(RLIMIT\_NOFILE, &rl)) {

        lerror("getrlimit RLIMIT\_NOFILE failed %d:%s\n", errno, strerror(errno));

        return -1;

    }

    if(rl.rlim\_max == RLIM\_INFINITY) {

//If many files were opened and then this limit was reduced to 1024, we may not close all file descriptors.

        rl.rlim\_max = 1024;

    }

*int* fd = 0;

    while(fd < (*int*)rl.rlim\_max) {

        if(!bIngoreDftFD || (fd != STDIN\_FILENO && fd != STDOUT\_FILENO && fd != STDERR\_FILENO)) {

            if(-1 == close(fd)) {

                if(EINTR == errno) {

                    continue; //try again

                }

                if(EBADF != errno) {

                    lerror("close fd %d failed %d:%s\n", fd, errno, strerror(errno));

                }

            } else {

                ++closeCnt;

                lerror("close fd %d, total count %d\n", fd, closeCnt);

            }

        }

        ++fd;

    }

    return closeCnt;

}

*int* redirectDftFD(const *char*\* toPath) {

*int* fd;

    if(NULL == toPath || '\0' == toPath[0]) {

        lerror("invalid redirect dest %p", toPath);

        return -1;

    }

    fd = open(toPath, O\_RDWR);

    if(-1 == fd) {

        lerror("open(%s) failed %d:%s\n", toPath, errno, strerror(errno));

        return -1;

    }

    if (STDIN\_FILENO != fd) {

        if (-1 == dup2(fd, STDIN\_FILENO)) {

            lerror("dup2(%d, %d) failed %d:%s\n", fd, STDIN\_FILENO, errno, strerror(errno));

            close(fd);

            return -1;

        }

        close(fd);

    }

    //if error occur, can't rollback

    if (-1 == dup2(STDIN\_FILENO, STDOUT\_FILENO)) {

        lerror("dup2(%d, %d) failed %d:%s\n", STDIN\_FILENO, STDOUT\_FILENO, errno, strerror(errno));

        return -1;

    }

    if (-1 == dup2(STDIN\_FILENO, STDERR\_FILENO)) {

        lerror("dup2(%d, %d) failed %d:%s\n", STDIN\_FILENO, STDERR\_FILENO, errno, strerror(errno));

        return -1;

    }

    return 0;

}

2016/11/3