# 3.7 守护进程

一：普通进程运行观察

//ps -eo pid,ppid,sid,tty,pgrp,comm,stat,cmd | grep -E 'bash|PID|nginx'

//a)进程有对应的终端，如果终端退出，那么对应的进程也就消失了；它的父进程是一个bash

//b)终端被占住了，你输入各种命令这个终端都没有反应；

//二：守护进程基本概念

//守护进程 一种长期运行的进程：这种进程在后台运行，并且不跟任何的控制终端关联；

//基本特点：

//a)生存期长[不是必须，但一般应该这样做]，一般是操作系统启动的时候他就启动，操作系统关闭的时候他才关闭；

//b)守护进程跟终端无关联，也就是说他们没有控制终端，所以你控制终端退出，也不会导致守护进程退出；

//c)守护进程是在后台运行,不会占着终端，终端可以执行其他命令

//linux操作系统本身是有很多的守护进程在默默的运行，维持着系统的日常活动。大概30-50个；

//a)ppid = 0：内核进程，跟随系统启动而启动，声明周期贯穿整个系统；

//b)cmd列名字带[]这种，叫内核守护进程；

//c)老祖init：也是系统守护进程,它负责启动各运行层次特定的系统服务；所以很多进程的PPID是init。而且这个init也负责收养孤儿进程；

//d)cmd列中名字不带[]的普通守护进程（用户级守护进程）



//共同点总结：

//a)大多数守护进程都是以超级 用户特权运行的；

//b)守护进程没有控制终端，TT这列显示?

//内核守护进程以无控制终端方式启动

//普通守护进程可能是守护进程调用了setsid的结果（无控制端）；

//三：守护进程编写规则

//(1)调用umask(0);

//umask是个函数，用来限制（屏蔽）一些文件权限的。

//(2)fork()一个子进程(脱离终端)出来,然后父进程退出( 把终端空出来，不让终端卡住)；固定套路

//fork()的目的是想成功调用setsid()来建立新会话，目的是

//子进程有单独的sid；而且子进程也成为了一个新进程组的组长进程；同时，子进程不关联任何终端了；

//--------------讲解一些概念

//（3.1）文件描述符：正数，用来标识一个文件。

//当你打开一个存在的文件或者创建一个新文件，操作系统都会返回这个文件描述符（其实就是代表这个文件的），

//后续对这个文件的操作的一些函数，都会用到这个文件描述符作为参数；

//linux中三个特殊的文件描述符，数字分别为0,1,2

//0:标准输入【键盘】，对应的符号常量叫STDIN\_FILENO

//1:标准输出【屏幕】，对应的符号常量叫STDOUT\_FILENO

//2:标准错误【屏幕】，对应的符号常量叫STDERR\_FILENO

//类Unix操作系统，默认从STDIN\_FILENO读数据，向STDOUT\_FILENO来写数据，向STDERR\_FILENO来写错误；

//类Unix操作系统有个说法：一切皆文件,所以它把标准输入，标准输出，标准错误 都看成文件。

//与其说 把 标准输入，标准输出，标准错误 都看成文件 到不如说

//象看待文件一样看待 标准输入，标准输出，标准错误

//象操作文件一样操作 标准输入，标准输出，标准错误

//同时，你程序一旦运行起来，这三个文件描述符0,1,2会被自动打开(自动指向对应的设备)；

//文件描述符虽然是数字，但是，如果我们把文件描述符直接理解成指针（指针里边保存的是地址——地址说白了也是个数字）；



//write(STDOUT\_FILENO,"aaaabbb",6);

//（3.2）输入输出重定向

//输出重定向：我标准输出文件描述符，不指向屏幕了，假如我指向（重定向）一个文件



//重定向，在命令行中用 >即可；

//输入重定向 <





//（3.3）空设备（黑洞）

// /dev/null ：是一个特殊的设备文件，它丢弃一切写入其中的数据（象黑洞一样）；

//----

//守护进程虽然可以通过终端启动，但是和终端不挂钩。

//守护进程是在后台运行，它不应该从键盘上接收任何东西，也不应该把输出结果打印到屏幕或者终端上来

//所以，一般按照江湖规矩，我们要把守护进程的 标准输入，标准输出，重定向到 空设备（黑洞）；

//从而确保守护进程不从键盘接收任何东西，也不把输出结果打印到屏幕；

//int fd;

//fd = open("/dev/null",O\_RDWR) ;//打开空设备

//dup2(fd,STDIN\_FILENO); //复制文件描述符 ，像个指针赋值,把第一个参数指向的内容赋给了第二个参数；

//dup2(fd,STDOUT\_FILENO);

//if(fd > STDERR\_FILENO)

// close(fd); //等价于fd = null;



//（3.4）实现范例

//守护进程可以用命令启动，如果想开机启动，则需要借助 系统初始化脚本来启动。

//四：守护进程不会收到的信号：内核发给你，另外的进程发给你的；

//（4.1）SIGHUP信号

//守护进程不会收到来自内核的 SIGHUP 信号； 潜台词就是 如果守护进程收到了 SIGHUP信号，那么肯定是另外的进程发给你的；

//很多守护进程把这个信号作为通知信号，表示配置文件已经发生改动，守护进程应该重新读入其配置文件；

//4.2）SIGINT、SIGWINCH信号

//守护进程不会收到来自内核的 SIGINT（ctrl+C),SIGWINCH(终端窗口大小改变) 信号；

//五：守护进程和后台进程的区别

//(1)守护进程和终端不挂钩；后台进程能往终端上输出东西(和终端挂钩)；

//(2)守护进程关闭终端时不受影响，后台进程会随着终端的退出而退出；

//(3)......其他的，大家自己总结；

# 4.1 服务器程序目录规划、makefile编写

//一：信号高级认识范例

//ps -eo pid,ppid,sid,tty,pgrp,comm,stat,cmd | grep -E 'bash|PID|nginx'

//用kill 发送 USR1信号给进程

//（1）执行信号处理函数被卡住了10秒，这个时候因为流程回不到main()，所以main中的语句无法得到执行；

//（2）在触发SIGUSR1信号并因此sleep了10秒种期间，就算你多次触发SIGUSR1信号，也不会重新执行SIGUSR1信号对应的信号处理函数,

//而是会等待上一个SIGUSR1信号处理函数执行完毕才 第二次执行SIGUSR1信号处理函数；

//换句话说：在信号处理函数被调用时，操作系统建立的新信号屏蔽字（sigprocmask()）,自动包括了正在被递送的信号，因此，

//保证了在处理一个给定信号的时候，如果这个信号再次发生，那么它会阻塞到对前一个信号处理结束为止；

//（3）不管你发送了多少次kill -usr1信号，在该信号处理函数执行期间，后续所有的SIGUSR1信号统统被归结为一次。

//比如当前正在执行SIGUSR1信号的处理程序但没有执行完毕，这个时候，你又发送来了5次SIGUSR1信号，那么当SIGUSR1信号处理程序

//执行完毕（解除阻塞），SIGUSR1信号的处理程序也只会被调用一次（而不会分别调用5次SIGUSR1信号的处理程序）。

//kill -usr1,kill -usr2

//（1）执行usr1信号处理程序，但是没执行完时，是可以继续进入到usr2信号处理程序里边去执行的，这个时候，

//相当于usr2信号处理程序没执行完毕,usr1信号处理程序也没执行完毕；此时再发送usr1和usr2都不会有任何响应；

//（2）既然是在执行usr1信号处理程序执行的时候来了usr2信号，导致又去执行了usr2信号处理程序，这就意味着，

//只有usr2信号处理程序执行完毕，才会返回到usr1信号处理程序，只有usr1信号处理程序执行完毕了，才会最终返回到main函数主流程中去继续执行；

//思考：如果我希望在我处理SIGUSR1信号，执行usr1信号处理程序的时候，如果来了SIGUSR2信号，我想堵住（屏蔽住），

//不想让程序流程跳到SIGUSR2信号处理中去执行，可以做到的；我们后续会讲解其他的如何屏蔽信号的方法；

//二：服务器架构初步

//老师，要带着大家，从无到有产生这套 通讯架构源代码【项目/工程】

//项目 肯定会有多个源文件，头文件，会分别存放到多个目录；我们要规划项目的目录结构；

//（2.1）目录结构规划（make编译）

//特别注意：不固安是目录还是文件，文件名中一律不要带空格，一律不要用中文，最好的方式：字母，数字，下划线；

//不要给自己找麻烦，远离各种坑

//主目录名nginx

//a)\_include目录：专门存放各种头文件； 如果分散:#include "sfaf/sdafas/safd.h"

//b)app目录：放主应用程序.c(main()函数所在的文件)以及一些比较核心的文件；

//b.1)link\_obj：临时目录：会存放临时的.o文件，这个目录不手工创建，后续用makefile脚本来创建

//b.2)dep：临时目录，会存放临时的.d开头的依赖文件，依赖文件能够告知系统哪些相关的文件发生变化，需要重新编译，后续用makefile脚本来创建

//b.3)nginx.c：主文件，main()入口函数就放到这里；

//b.4)ngx\_conf.c ，普通的源码文件，跟主文件关系密切，又不值得单独放在 一个目录；

//c)misc目录：专门存放各种杂合性的不好归类的1到多个.c文件；暂时为空

//d)net目录：专门存放和网络处理相关的1到多个.c文件，暂时为空

//e)proc目录：专门存放和进程处理有古安的1到多个.c文件，暂时为空

//f)signal目录：专门用于存放和信号处理 有古安的1到多个.c文件；

//ngx\_signal.c

//linux上用tree看一下目录结构

//（2.2）编译工具make的使用概述（编译出可执行文件）

//我们 要用传统的 ，经过验证没有问题的方式来编译我们的项目，最终生成可执行文件

//每个.c生成一个.o，多个.c生成多个.o，最终这些.o被链接到一起，生成一个可执行文件



//gcc -o nginx ng1.c

//gcc -o nginx ng1.c ng2.c

//a)我们要借助make的命令来编译：能够编译，链接。。。。最终生成可执行文件，大型项目一般都用make来搞；

//b)make命令的工作原理，就去当前目录读取一个叫做makefile的文件（文本文件），根据这个makefile文件里的

//规则把咱们的源代码编译成可执行文件；咱们开发者的任务就是要把这个makefile文件写出来；

//这个makefile里边就定义了我们怎么去编译整个项目的编译、链接规则

//【实际上makefile文件就是一个我们编译工程要用到的各种源文件等等的一个依赖关系描述】

//有类似autotools自动生成makefile，这里不讲

//c)makefile文件：文本文件，utf8编码格式，没有扩展名，一般放在根目录下[也会根据需要放在子目录]（这里nginx)

//不同的程序员写的makefile代码也会千差万别；但不管怎么说，最终都要把可执行文件给我生成出来；

//老师挑选 灵活性、通用性比较好的一种makefile写法，介绍给大家；

//规划一下makefile文件的编写

//a)nginx根目录下我会放三个文件：

//a.1)makefile：是咱们编译项目的入口脚本，编译项目从这里开始，起总体控制作用；

//a.2）config.mk：这是个配置脚本，被makefile文件包含；单独分离出来是为了应付一些可变的东西,所以，一般变动的东西都往这里搞；

//a.3)common.mk：是最重要最核心的编译脚本，定义makefile的编译规则，依赖规则等，通用性很强的一个脚本，

//并且各个子目录中都用到这个脚本来实现对应子目录的.c文件的编译；

//b)每个子目录下（app,signal)都有一个叫做makefile的文件，每个这个makefile文件，都会包含根目录下的common.mk，

//从而实现自己这个子目录下的.c文件的编译

//现在的makefile不支持目录中套子目录，除非大家自己修改；

//c)其他规划，上边讲过；

//app/link\_obj临时目录，存放.o目标文件

//app/dep：存放.d开头的依赖关系文件；

//（2.3）makefile脚本用法介绍

//a)编译项目，生成可执行文件

//make

//make clean



//（2.4）makefile脚本具体实现讲解

//从common.mk讲起

//将来增加新目录时：

//a)修改根目录下的config.mk来增加该目录

//b)在对应的目录下放入makefile文件，内容参考signal目录ixa的makefile文件即可

# 4.2 读配置文件、查泄漏、设置标题实战

//一：基础设施之配置文件读取

//（1.1）前提内容和修改

//使用配置文件，使我们的服务器程序有了极大的灵活性,是我们作为服务器程序开发者，必须要首先搞定的问题；

//配置文件：文本文件，里边除了注释行之外不要用中文，只在配置文件中使用字母，数字下划线

//以#号开头的行作为注释行(注释行可以有中文)

//我们这个框架（项目），第一个要解决的问题是读取配置文件中的配置项（读到内存中来）；

//（1.2）配置文件读取功能实战代码

//写代码要多顾及别人感受，让别人更容易读懂和理解，不要刻意去炫技；这种炫技的人特别讨厌；

//该缩进的必须要缩进，该对齐的要对齐，该注释的要注释，这些切记

//二：内存泄漏的检查工具

//Valgrind：帮助程序员寻找程序里的bug和改进程序性能的工具集。擅长是发现内存的管理问题；

//里边有若干工具，其中最重要的是Memcheck(内存检查）工具，用于检查内存的泄漏；

//（2.1）memcheck的基本功能，能发现如下的问题；

//a)使用未初始化的内存

//b)使用已经释放了的内存

//c)使用超过malloc()分配的内存

//d)对堆栈的非法访问

//e)申请的内存是否有释放\*\*\*\*\*

//f)malloc/free,new/delete申请和释放内存的匹配

//g)memcpy()内存拷贝函数中源指针和目标指针重叠；

//（2.2）内存泄漏检查示范

//所有应该释放的内存，都要释放掉，作为服务器程序开发者，要绝对的严谨和认真

//格式：

//valgrind --tool=memcheck 一些开关 可执行文件名

//--tool=memcheck ：使用valgrind工具集中的memcheck工具

//--leak-check=full ： 指的是完全full检查内存泄漏

//--show-reachable=yes ：是显示内存泄漏的地点

//--trace-children = yes ：是否跟入子进程

//--log-file=log.txt：讲调试信息输出到log.txt，不输出到屏幕

//最终用的命令：

//valgrind --tool=memcheck --leak-check=full --show-reachable=yes ./nginx

//查看内存泄漏的三个地方：

//(1) 9 allocs, 8 frees 差值是1，就没泄漏，超过1就有泄漏

//(2)中间诸如： by 0x401363: CConfig::Load(char const\*) (ngx\_c\_conf.cxx:77)和我们自己的源代码有关的提示，就要注意；

//(3)LEAK SUMMARY:definitely lost: 1,100 bytes in 2 blocks

//三：设置可执行程序的标题（名称）

//（3.1）原理和实现思路分析

//argc:命令行参数的个数

//argv:是个数组，每个数组元素都是指向一个字符串的char \*，里边存储的内容是所有命令行参数；

//./nginx -v -s 5

//argc = 4

//argv[0] = ./nginx ----指向的就是可执行程序名： ./nginx

//argv[1] = -v

//argv[2] = -s

//argv[3] = 5

//比如你输入 ./nginx -12 -v 568 -q gess



//argv内存之后，接着连续的就是环境变量参数信息内存【是咱们这个可执行程序执行时有关的所有环境变量参数信息】



//可以通过一个全局的environ[char \*\*]就可以访问

//environ内存和argv内存紧紧的挨着

//修改可执行程序的实现思路：

//(1)重新分配一块内存，用来保存environ中的内容；

//(2)修改argv[0]所指向的内存；

//（3.2）设置可执行程序的标题实战代码



# 4.3 日志打印实战，优化main函数调用顺序

//一：基础设施之日志打印实战代码一

//1-3万行代码，想收获多少就要付出多少，平衡

//注意代码的保护，私密性

//日志的重要性：供日后运行维护人员去查看、定位和解决问题；

//新文件：ngx\_printf.cxx以及ngx\_log.cxx。

//ngx\_printf.cxx：放和打印格式相关的函数；

//ngx\_log.cxx：放和日志相关的函数；

//ngx\_log\_stderr() :三个特殊文件描述符【三章七节】，谈到了标准错误 STDERR\_FILENO，代表屏幕

//ngx\_log\_stderr()：往屏幕上打印一条错误信息；功能类似于printf

//printf("mystring=%s,myint=%d，%d","mytest",15,20);

//(1)根据可变的参数，组合出一个字符串:mystring=mytest,myint=15，20

//(2)往屏幕上显示出这个组合出来的字符串；

//讲解ngx\_log\_stderr()函数的理由：

//(1)提高大家编码能力；

//(2)ngx\_log\_stderr()：可以支持任意我想支持的格式化字符 %d， %f,对于扩展原有功能非常有帮助

//(i)void ngx\_log\_stderr(int err, const char \*fmt, ...)

//(i) p = ngx\_vslprintf(p,last,fmt,args); //实现了自我可定制的printf类似的功能

//(i) buf = ngx\_sprintf\_num(buf, last, ui64, zero, hex, width);

//(i) p = ngx\_log\_errno(p, last, err);

//二：设置时区67

//我们要设置成CST时区,以保证日期，时间显示的都正确

//我们常看到的时区，有如下几个：

//a)PST【PST美国太平洋标准时间】 = GMT - 8;

//b)GMT【格林尼治平均时间Greenwich Mean Time】等同于英国伦敦本地时间

//c)UTC【通用协调时Universal Time Coordinated】 = GMT

//d)CST【北京时间：北京时区是东八区，领先UTC八个小时】

//三：基础设施之日志打印实战代码二

//（3.1）日志等级划分

//划分日志等级，一共分8级，分级的目的是方便管理，显示，过滤等等；

//日志级别从高到低，数字最小的级别最高，数字最大的级别最低；

//（3.2）配置文件中和日志有关的选项

//继续介绍void ngx\_log\_init();打开/创建日志文件

//介绍ngx\_log\_error\_core()函数：写日志文件的核心函数

//ngx\_slprintf

// ngx\_vslprintf

//四：捋顺main函数中代码执行顺序

# 4.4信号，子进程实战，文件IO详谈

//一：信号功能实战

//signal()：注册信号处理程序的函数；

//商业软件中，不用signal()，而要用sigaction();

//二：nginx中创建worker子进程

//官方nginx ,一个master进程，创建了多个worker子进程；

// master process ./nginx

// worker process

//(i)ngx\_master\_process\_cycle() //创建子进程等一系列动作

//(i) ngx\_setproctitle() //设置进程标题

//(i) ngx\_start\_worker\_processes() //创建worker子进程

//(i) for (i = 0; i < threadnums; i++) //master进程在走这个循环，来创建若干个子进程

//(i) ngx\_spawn\_process(i,"worker process");

//(i) pid = fork(); //分叉，从原来的一个master进程（一个叉），分成两个叉（原有的master进程，以及一个新fork()出来的worker进程

//(i) //只有子进程这个分叉才会执行ngx\_worker\_process\_cycle()

//(i) ngx\_worker\_process\_cycle(inum,pprocname); //子进程分叉

//(i) ngx\_worker\_process\_init();

//(i) sigemptyset(&set);

//(i) sigprocmask(SIG\_SETMASK, &set, NULL); //允许接收所有信号

//(i) ngx\_setproctitle(pprocname); //重新为子进程设置标题为worker process

//(i) for ( ;; ) {}. .... //子进程开始在这里不断的死循环

//(i) sigemptyset(&set);

//(i) for ( ;; ) {}. //父进程[master进程]会一直在这里循环

//kill -9 -1344 ，用负号 -组id，可以杀死一组进程

//（2.1）sigsuspend()函数讲解

//a)根据给定的参数设置新的mask 并 阻塞当前进程【因为是个空集，所以不阻塞任何信号】

//b)此时，一旦收到信号，便恢复原先的信号屏蔽【我们原来的mask在上边设置的，阻塞了多达10个信号，从而保证我下边的执行流程不会再次被其他信号截断】

//c)调用该信号对应的信号处理函数

//d)信号处理函数返回后，sigsuspend返回，使程序流程继续往下走

//三：日志输出重要信息谈

//（3.1）换行回车进一步示意

//\r：回车符,把打印【输出】信息的为止定位到本行开头

//\n：换行符，把输出为止移动到下一行

//一般把光标移动到下一行的开头，\r\n

//a)比如windows下，每行结尾 \r\n

//b)类Unix，每行结尾就只有\n

//c)Mac苹果系统，每行结尾只有\r

//结论：统一用\n就行了

//（3.2）printf()函数不加\n无法及时输出的解释

//printf末尾不加\n就无法及时的将信息显示到屏幕 ，这是因为 行缓存[windows上一般没有，类Unix上才有]

//需要输出的数据不直接显示到终端，而是首先缓存到某个地方，当遇到行刷新表指或者该缓存已满的情况下，菜会把缓存的数据显示到终端设备；

//ANSI C中定义\n认为是行刷新标记，所以，printf函数没有带\n是不会自动刷新输出流，直至行缓存被填满才显示到屏幕上；

//所以大家用printf的时候，注意末尾要用\n；

//或者：fflush(stdout);

//或者：setvbuf(stdout,NULL,\_IONBF,0); //这个函数. 直接将printf缓冲区禁止， printf就直接输出了。

//标准I/O函数，后边还会讲到

//四：write()函数思考

//多个进程同时去写一个文件,比如5个进程同时往日志文件中写，会不会造成日志文件混乱。

//多个进程同时写 一个日志文件，我们看到输出结果并不混乱，是有序的；我们的日志代码应对多进程往日志文件中写时没有问题；

//《Unix环境高级编程 第三版》第三章：文件I/O里边的3.10-3.12，涉及到了文件共享、原子操作以及函数dup,dup2的讲解；

//第八章：进程控制 里庇安的8.3，涉及到了fork()函数；

//a)多个进程写一个文件，可能会出现数据覆盖，混乱等情况

//b)ngx\_log.fd = open((const char \*)plogname,O\_WRONLY|O\_APPEND|O\_CREAT,0644);

//O\_APPEND这个标记能够保证多个进程操作同一个文件时不会相互覆盖；

//c)内核wirte()写入时是原子操作；

//d)父进程fork()子进程是亲缘关系。是会共享文件表项，109

//--------------关于write()写的安全问题，是否数据成功被写到磁盘；

//e)write()调用返回时，内核已经将应用程序缓冲区所提供的数据放到了内核缓冲区，但是无法保证数据已经写出到其预定的目的地【磁盘 】；

//的确，因为write()调用速度极快，可能没有时间完成该项目的工作【实际写磁盘】，所以这个wirte()调用不等价于数据在内核缓冲区和磁盘之间的数据交换

//f)打开文件使用了 O\_APPEND，多个进程写日志用write()来写；

//（4.1）掉电导致write()的数据丢失破解法

//a)直接I/O：直接访问物理磁盘：

//O\_DIRECT：绕过内核缓冲区。用posix\_memalign

//b)open文件时用O\_SYNC选项：

//同步选项【把数据直接同步到磁盘】,只针对write函数有效，使每次write()操作等待物理I/O操作的完成；

//具体说，就是将写入内核缓冲区的数据立即写入磁盘，将掉电等问题造成的损失减到最小；

//每次写磁盘数据，务必要大块大块写，一般都512-4k 4k的写；不要每次只写几个字节，否则会被抽死；\*\*\*\*\*\*\*\*\*\*\*\*

//c)缓存同步：尽量保证缓存数据和写道磁盘上的数据一致；

//sync(void)：将所有修改过的块缓冲区排入写队列；然后返回，并不等待实际写磁盘操作结束，数据是否写入磁盘并没有保证；

//fsync(int fd)：将fd对应的文件的块缓冲区立即写入磁盘，并等待实际写磁盘操作结束返回；\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//fdatasync(int fd)：类似于fsync，但只影响文件的数据部分。而fsync不一样，fsync除数据外，还会同步更新文件属性；

//write(4k),1000次之后，一直到把这个write完整[假设整个文件4M]。

//fsync(fd) ,1次fsync [多次write,每次write建议都4k，然后调用一次fsync()，这才是用fsync()的正确用法\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*]

//五：标准IO库

//fopen,fclose

//fread,fwrite

//fflush

//fseek

//fgetc,getc,getchar

//fputc,put,putchar

//fgets,gets

//printf,fprintf,sprintf

//scanf,fscan,sscanf

//fwrite和write有啥区别；

//fwrite()是标准I/O库一般在stdio.h文件

//write()：系统调用；

//有一句话：所有系统调用都是原子性的