# 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，

//--------------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



如果标准IO用不好，就用操作系统提供的write

# 4.5守护进程及信号处理实战

//一：守护进程功能的实现

//三章二节

//(1)拦截掉SIGHUP，那么终端窗口关闭，进程就不会跟着关闭

//(2)守护进程，三章七节，一运行就在后台，不会占着终端。

//创建守护进程ngx\_daemon（）；

//调用ngx\_daemon（）的时机： worker()子进程创建之前；

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

//(1)一个master,4个worker进程，状态S，表示休眠状态，但没有+,+号表示位于前台进程组，没有+说明我们这几个进程不在前台进程组；

//(2)master进程的ppid是1【老祖宗进程init】，其他几个worker进程的父进程都是master；

//(3)tt这列都为？，表示他们都脱离了终端，不与具体的终端挂钩了

//(4)他们的进程组PGRP都相同；

//结论：

//1)守护进程如果通过键盘执行可执行文件来启动，那虽然守护进程与具体终端是脱钩的，但是依旧可以往标准错误上输出内容，这个终端对应的屏幕上可以看到输入的内容；

//2)但是如果这个nginx守护进程你不是通过终端启动，你可能开机就启动，那么这个nginx守护进程就完全无法往任何屏幕上显示信息了，这个时候，要排错就要靠日志文件了；

//二：信号处理函数的进一步完善

//（2.1）避免子进程被杀掉时变成僵尸进程

//父进程要处理SIGCHILD信号并在信号处理函数中调用waitpid()来解决僵尸进程的问题；

//信号处理函数中的代码，要坚持一些书写原则：

//a)代码尽可能简单，尽可能快速的执行完毕返回；

//b)用一些全局量做一些标记；尽可能不调用函数；

//c)不要在信号处理函数中执行太复杂的代码以免阻塞其他信号的到来，甚至阻塞整个程序执行流程；

# 5.1 C/S，TCP/IP协议

//本节课是一些必须要讲解的基础知识；请大家认真倾听；

//一：客户端与服务器

//客户端：client，一般字母c表示

//服务器：server，一般字母s表示 ,所以c/s一般就是 ：客户端/服务器

//客户端：就是一个程序，

//服务器：也是 一个程序；

//（1.1）解析一个浏览器访问网页的过程





//（1.2）客户端服务器角色规律总结

//a)数据通讯总在两端进行，其中一端叫客户端，另一端叫服务器端；

//b)总有一方先泛起第一个数据包，这发起第一个数据包的这一端，就叫客户端【浏览器】；被动收到第一个数据包这端，叫服务器端【淘宝服务器】；

//c)连接建立起来，数据双向流动，这叫 双工【你可以发数据包给我，我也可以发数据包给你】

//d)既然服务器端是被动接收连接，那么客户端必须得能够找到服务器在哪里；



//我浏览器要访问淘宝网，我需要知道淘宝服务器的地址【ip地址：192.168.1.100 三个点分隔四个数】，

//以及淘宝服务器的姓名【端口号，这是 一个无符号数字，范围 0-65535之间的一个数字】

// 淘宝网服务器【nginx服务器】会调用listen()函数来监听80端口；

//在编写网络通讯程序时，你只需要指定淘宝服务器的ip地址和淘宝服务器的端口号，就能够跟淘宝服务器进行通讯；

//e)epoll

//二：网络模型

//（2.1）OSI七层网络模型:

//物【物理层】 链【数据链路层】 网【网络层】 传【传输层】 会【会话层】 表【表示层】 应【应用层】



//OSI(Open System Interconnect)：开放式系统互联；是ISO(国际标准化组织）在1985年研究的网络互联模型；

//把一个要发送出去的数据包从里到外裹了7层，就跟 一个人一样，穿了7件衣服，一件套一件 ；最终把包裹了7层的数据包发送都网络上去了；

//（2.2）TCP / IP协议四层模型

//Transfer Control Protocol[传输控制协议]/Internet Protocol[网际协议];

//tcp/ip实际是 一组 协议的代名词，而不仅仅是一个协议；

//tcp/ip协议，其实每一层都对应着一些协议；



//（2.3）TCP / IP协议的解释和比喻

//我们把人看成 要发送出去的数据包；人出门上街 ，我们把外边的街道，就看成网络，我们人出门上街，就等于把数据包发送到互联网是上去；

// 人 <=======> 数据包

//街道 <=======> 互联网

//人上街 <=======> 数据包发送到互联网上

//人不能光腚上街，人要先穿内衣内裤【TCP】；套一个衬衣衬裤【IP】,套个外衣外裤【以太网帧】，可以出门了；

// TCP 比喻成了 内衣内裤

// IP 比喻成了 衬衣衬裤

// 以太网帧 比喻成了 外衣外裤

//你要发送 abc 这三个字母出去到网络上；

//加个tcp头【abc套了个内衣内裤】

//加个IP头【abc套了个衬衣衬裤】

//加个以太网帧头/尾【abc套了个外衣外裤】

//加了这三个头一个尾之后，就认为这个数据包符合了TCP/IP协议，这个数据包能够被发送到网络上去了【人穿好了衣服可以出门了】；



//三：最简单的客户端和服务器程序实现代码

//a)客户端程序，服务器端程序；只具备演示价值，不具备商业价值。

//b)最终epoll技术实现商用的服务器程序；

//c)《Unix网络编程》第一卷；

//（3.1）套接字socket概念

//套接字(socket)：就是个数字，通过调用socket()函数来生成；这个数字具有唯一性；一直给你用，直到你调用close()函数把这个数字关闭；

//文件描述符；一切皆文件，咱们就把socket也看成是文件描述符，我们可以用socket来收发数据；send(),recv();

//（3.2）一个简单的服务器端通讯程序范例【看调用了哪些函数：面试官可能 会考】

//（3.3）IP地址简单谈

//192.168.1.100[IVP4]：理解成现实社会中的居住地址

//192.168.1.100[IVP4]：第四个版本的IP地址格式；

//发展处了新的IP地址版本【第六版】，IPV6



//我们写通讯程序代码时是否需要根据ipv4,ipv6来调整呢？

//a)写服务器程序，不用考虑ipv4,ipv6的问题，遵照ipv4规则写就行；

//b)写客户端程序，只演示ipv4版本的客户端范例。

//后续写项目老师会带着大家写同时兼容ipv4,ipv6【协议无关】客户端程序；

//（3.4）一个简单的客户端通讯程序范例

//c/s建立连接时双方彼此都要有 ip地址 /端口号；

//连接一旦建立起来，那么双方的通讯【双工收发】，就只需要用双方彼此对应的套接字即可；

//（3.5）客户端服务器程序综合演示和调用流程图

//服务器端程序要先运行；



//四：TCP和UDP的区别

//TCP(Transfer Control Protocol)：传输控制协议

//UDP(User Datagram Protocol)：用户数据报协议

//socket()

//TCP 比喻成 内衣内裤

//UDP 比喻成 内衣内裤

//a)TCP是大品牌内衣内裤，售后质量好；如果被偷取，厂家负责派人帮你找

//b)UDP小品牌内衣内裤，没有什么售后服务；

//TCP协议：可靠的面向连接的协议；数据包丢失的话操作系统底层会感知并且帮助你重新发送数据包；

//UDP协议：不可靠的，无连接的协议；

//优缺点：

//a)tcp：可靠协议，必然要耗费更多的系统资源确保数据传输的可靠；

//得到好处就是只要不断线，传输给对方的数据，一定正确的，不丢失，不重复，按顺序到达对端；

//b)udp：不可靠协议；发送速度特别快；但无法确保数据可靠性

//各自的用途：

//a)tcp：文件传输，收发邮件需要准确率高，但效率可以相对差；一般TCP比UDP用的范围和场合更广；

//b)udp：qq聊天信息；DNS。。。。。。估计随着网络的发展，网络性能更好，丢包率更低，那么udp应用范围更广；

# 5.2 C/S TCP三次握手详析、telnet，wireshark示范

//一：TCP连接的三次握手

//tcp：可靠的，面向连接的协议

//udp：不可靠的，无连接的协议

//大家必须要懂的TCP的三次握手，只有TCP有三次握手【UDP没有】

//（1.1）最大传输单元MTU

//MTU(Maximum Transfer Unit)：最大传输单元；

//MTU：每个数据包包含的数据最多可以有多少个字节；1.5K左右；

//你要发送100K，操作系统内部会把你这100K数据拆分成若干个数据包【分片】，每个数据包大概1.5K之内【大概拆解成68个包】；对端重组；

//我们只需要知道有 拆包，组包；

//这68个包各自传送的路径可能不同，每一个包可能因为路由器，交换机原因可能被再次分片；

//最终TCP/IP协议保证了我们收发数据的顺序性和可靠性；

//（1.2）TCP 包头结构

//a)源端口，目标端口

//b)关注syn位和ack位【开/关】

//c)一个tcp数据包，是可能没有包体，此时，总会设置一些标志位来达到传输 控制信息的目的；



//（1.3）TCP 数据包收发之前的准备工作

//回忆日志操作的步骤：

//a)打开日志文件

//b)多次，反复的往日志文件中写信息

//c)关闭日志文件

//TCP数据的收发是双工的：每端既可以收数据，又可以发数据；

//TCP数据包的收发也分三大步骤：

//a)建立TCP连接[connect：客户端] ，三次握手

//b)多次反复的数据收发[read/wirte]

//c)关闭TCP连接[close]

//UDP不存在三次握手来建立连接的问题。UDP数据包是直接发送出去，不用建立所谓的连接；

//（1.4）TCP 三次握手建立连接的过程

//客户端理解成一个人，服务器端理解成一个人，两个人要用电话通话:

//张三：你好，李四，我是张三 [syn] ,ip，端口

//李四：你好，张三，我是李四 [syn/ack]

//张三：你好，李四 [ack]

//聊正题......



//a)客户端给服务器发送 了一个SYN标志位置位的无包体TCP数据包，SYN被置位，就表示发起TCP链接，协议就这么定

//b)服务器收到了这个SYN标志位置位的数据包，服务器给客户端返回一个SYN和ACK标志位都被置位的无包体TCP数据包，协议就这么定的；

//c)客户端收到服务器发送回来的数据包之后，再次发送ACK置位的数据包，服务器端收到这个数据包之后，客户端和服务器端的TCP链接就正式建立；

//后续就可以进行数据收发了

//三次握手很大程度上是为了防止恶意的人坑害别人而引入的一种TCP连接验证机制；

//（1.5）为什么TCP握手是三次握手而不是二次

//网络诈骗

//110：客户端 ，你就是服务器

//伪造来电：110

//TCP之所以要三次握手，原因可能很多，但不管多少 原因，都是为了确保数据稳定可靠的收发；

//为什么要TCP三次握手最主要的原因之一：尽量减少伪造数据包对服务器的攻击；

//源ip，源端口 ---------------- 目的ip，目的端口

//syn------------->

//<--------syn/ack 【验证源ip和源端口真实存在】

//ack-------------->

//二：telnet工具使用介绍

//是一款命令行方式运行的客户端TCP通讯工具，可以连接到服务器端，往服务器端发送数据，也可以接收从服务器端发送过来的信息；

//类似nginx5\_1\_1\_client.c

//该工具能够方便的测试服务器端的某个TCP端口是否通，是否能够正常收发数据，所以是一个非常实用，重要，常用的工具，老师要求大家都会；

//telnet ip地址 端口号

//三：wireshark监控数据包

//wireshark是个软件:分析网络数据包

//https://www.wireshark.org/download.html

//automatically start the WinPcap driver at boot time;

//windows：192.168.1.100

//乌班图linux：192.168.1.126 9000端口

//希望wireshark抓 192.168.1.126 9000；

//host 192.168.1.126 and port 9000

//（3.1）TCP断开的四次挥手

// FIN,ACK 服务器->客户端

// ACK 客户端->服务器

// FIN,ACK 客户端->服务器

// ACK 服务器->客户端



# 5.3 TCP状态转换，TIME\_WAIT，SO\_REUSEADDR

//一：TCP状态转换

//同一个IP（INADDR\_ANY），同一个端口SERV\_PORT，只能被成功的bind()一次，若再次bind()就会失败，并且显示：Address already in use

//就好像一个班级里不能有两个人叫张三；

//结论：相同IP地址的相同端口，只能被bind一次；第二次bind会失败；

//介绍命令netstat：显示网络相关信息

//-a:显示所有选项

//-n:能显示成数字的内容全部显示成数字

//-p：显示段落这对应程序名

//netstat -anp | grep -E 'State|9000'

//我们用两个客户端连接到服务器，服务器给每个客户端发送一串字符"I sent sth to client!\n"，并关闭客户端;

//我们用netstat观察，原来那个监听端口 一直在监听【listen】，但是当来了两个连接之后【连接到服务器的9000端口】，

//虽然这两个连接被close掉了，但是产生了两条TIME\_WAIT状态的信息【因为你有两个客户端连入进来】

//只要客户端 连接到服务器，并且 服务器把客户端关闭，那么服务器端就会产生一条针对9000监听端口的 状态为 TIME\_WAIT 的连接；

//只要用netstat看到 TIME\_WAIT状态的连接，那么此时， 你杀掉服务器程序再重新启动，就会启动失败，bind()函数返回失败：

//bind返回的值为-1,错误码为:98，错误信息为:Address already in use

//TIME\_WAIT：涉及到TCP状态转换这个话题了；

//《Unix网络编程 第三版 卷1》有第二章第六节，2.6.4小节，里边就有一个TCP状态转换图；



//第二章第七节，专门介绍了 TIME\_WAIT状态；

//TCP状态转换图【11种状态】 是 针对“一个TCP连接【一个socket连接】”来说的；

//客户端： CLOSED ->SYN\_SENT->ESTABLISHED【连接建立，可以进行数据收发】

//服务端： CLOSED ->LISTEN->【客户端来握手】SYN\_RCVD->ESTABLISHED【连接建立，可以进行数据收发】

//谁主动close连接，谁就会给对方发送一个FIN标志置位的一个数据包给对方；【服务器端发送FIN包给客户端】

//服务器主动关闭连接：ESTABLISHED->FIN\_WAIT1->FIN\_WAIT2->TIME\_WAIT

//客户端被动关闭：ESTABLISHED->CLOSE\_WAIT->LAST\_ACK

//二：TIME\_WAIT状态

//具有TIME\_WAIT状态的TCP连接，就好像一种残留的信息一样；当这种状态存在的时候，服务器程序退出并重新执行会失败，会提示：

//bind返回的值为-1,错误码为:98，错误信息为:Address already in use

//所以，TIME\_WAIT状态是一个让人不喜欢的状态；



//连接处于TIME\_WAIT状态是有时间限制的（1-4分钟之间） = 2 MSL【最长数据包生命周期】；

//引入TIME\_WAIT状态【并且处于这种状态的时间为1-4分钟】 的原因：

//(1)可靠的实现TCP全双工的终止

//如果服务器最后发送的ACK【应答】包因为某种原因丢失了，那么客户端一定 会重新发送FIN，这样

//因为服务器端有TIME\_WAIT的存在，服务器会重新发送ACK包给客户端，但是如果没有TIME\_WAIT这个状态，那么

//无论客户端收到ACK包，服务器都已经关闭连接了，此时客户端重新发送FIN，服务器给回的就不是ACK包，

//而是RST【连接复位】包，从而使客户端没有完成正常的4次挥手，不友好，而且有可能造成数据包丢失；

//也就是说，TIME\_WAIT有助于可靠的实现TCP全双工连接的终止；

//（二.一）RST标志

//对于每一个TCP连接，操作系统是要开辟出来一个收缓冲区，和一个发送缓冲区 来处理数据的收和发；

//当我们close一个TCP连接时，如果我们这个发送缓冲区有数据，那么操作系统会很优雅的把发送缓冲区里的数据发送完毕，然后再发fin包表示连接关闭；

//FIN【四次挥手】，是个优雅的关闭标志，表示正常的TCP连接关闭；

//反观RST标志：出现这个标志的包一般都表示 异常关闭；如果发生了异常，一般都会导致丢失一些数据包；

//如果将来用setsockopt(SO\_LINGER)选项要是开启；发送的就是RST包，此时发送缓冲区的数据会被丢弃；

//RST是异常关闭，是粗暴关闭，不是正常的四次挥手关闭，所以如果你这么关闭tcp连接，那么主动关闭一方也不会进入TIME\_WAIT；

//(2)允许老的重复的TCP数据包在网络中消逝；

//三：SO\_REUSEADDR选项

//setsockopt（SO\_REUSEADDR）用在服务器端，socket()创建之后，bind()之前

//SO\_REUSEADDR的能力：

//（1）SO\_REUSEADDR允许启动一个监听服务器并捆绑其端口，即使以前建立的将端口用作他们的本地端口的连接仍旧存在；

//【即便TIME\_WAIT状态存在，服务器bind()也能成功】

//（2）允许同一个端口上启动同一个服务器的多个实例，只要每个实例捆绑一个不同的本地IP地址即可；

//（3）SO\_REUSEADDR允许单个进程捆绑同一个端口到多个套接字，只要每次捆绑指定不同的本地IP地址即可；

//（4）SO\_REUSEADDR允许完全重复的绑定：当一个IP地址和端口已经绑定到某个套接字上时，如果传输协议支持，

//同样的IP地址和端口还可以绑定到另一个套接字上；一般来说本特性仅支持UDP套接字[TCP不行]；

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//所有TCP服务器都应该指定本套接字选项，以防止当套接字处于TIME\_WAIT时bind()失败的情形出现；

//试验程序nginx5\_3\_2\_server.c

//（3.1）两个进程，绑定同一个IP和端口：bind()失败[一个班级不能有两个人叫张三]

//（3.2）TIME\_WAIT状态时的bind绑定：bind()成功

//SO\_REUSEADDR：主要解决TIME\_WAIT状态导致bind()失败的问题；

# 5.4 listen()队列剖析、阻塞非阻塞、同步异步

//一：listen()队列剖析

//listen()：监听端口，用在 TCP连接 中的 服务器端 角色；

//listen()函数调用格式：

//int listen(int sockfd, int backlog);

//要理解好backlog这个参数，我们需要先谈一谈 “监听套接字 队列”的话题；

//（1.1）监听套接字的队列

//对于一个调用listen()进行监听的套接字，操作系统会给这个套接字 维护两个队列；

//a)未完成连接队列 【保存连接用的】

//当客户端 发送tcp连接三次握手的第一次【syn包】给服务器的时候，服务器就会在未完成队列中创建一个 跟这个 syn包对应的一项，

//其实，我们可以把这项看成是一个半连接【因为连接还没建立起来呢】，这个半连接的状态会从LISTEN变成SYN\_RCVD状态，同时给客户端返回第二次握手包【syn,ack】

//这个时候，其实服务器是在等待完成第三次握手；

//b)已完成连接队列 【保存连接用的】

//当第三次握手完成了，这个连接就变成了ESTABLISHED状态，每个已经完成三次握手的客户端 都放在这个队列中作为一项；



//backlog曾经的含义：已完成队列和未完成队列里边条目之和 不能超过 backlog;





//(1)客户端这个connect()什么时候返回，其实是收到三次握手的第二次握手包（也就是收到服务器发回来的syn/ack）之后就返回了；

//(2)RTT是未完成队列中任意一项在未完成队列中留存的时间，这个时间取决于客户端和服务器；

//对于客户端，这个RTT时间是第一次和第二次握手加起来的时间；

//对于服务器，这个RTT时间实际上是第二次和第三次握手加起来的时间；

//如果这三次握手包传递速度特别快的话，大概187毫秒能够建立起来这个连接；这个时间挺慢，所以感觉建立TCP连接的成本挺高；【短连接游戏-挺恶心的】

//(3)如果一个恶意客户，迟迟不发送三次握手的第三个包。那么这个连接就建立不起来，那么这个处于SYN\_RCVD的这一项【服务器端的未完成队列中】，

//就会一致停留在服务器的未完成队列中，这个停留时间大概是75秒，如果超过这个时间，这一项会被操作系统干掉；

//（1.2）accept()函数

//accept()函数，就使用来 从 已完成连接队列 中 的队首【队头】位置取出来一项【每一项都是一个已经完成三路握手的TCP连接】，返回给进程；

//如果已完成连接队列是空的呢？那么咱们这个范例中accept()会一致卡在这里【休眠】等待，一直到已完成队列中有一项时才会被唤醒；

//所以，从编程角度，我们要尽快的用accept()把已完成队列中的数据【TCP连接】取走，大家必须有这个认识；

//accept()返回的是个套接字，这个套接字就代表那个已经用三次握手建立起来的那个tcp连接，因为accept()是从 已完成队列中取的数据；

//换句话来说，我们服务器程序，必须要严格区分两个套接字：

//a)监听9000端口这个套接字，这个东西叫“监听套接字【listenfd】”，只要服务器程序在运行，这个套接字就应该一直存在；

//b)当客户端连接进来，操作系统会为每个成功建立三次握手的客端再创建一个套接字【当然是一个已经连接套接字】，accept()返回的就是这种套接字；

//也就是从已完成连接队列中取得的一项。随后，服务器使用这个accept()返回的套接字和客户端通信的；

//思考题：

//(1)如果两个队列之和【已完成连接队列，和未完成连接队列】达到了listen()所指定的第二参数，也就是说队列满了；

//此时，再有一个客户发送syn请求，服务器怎么反应？

//实际上服务器会忽略这个syn，不给回应； 客户端这边，发现syn没回应，过一会会重发syn包；

//(2)从连接被扔到已经完成队列中去，到accept()从已完成队列中把这个连接取出这个之间是有个时间差的，如果还没等accept()从

//已完成队列中把这个连接取走的时候，客户端如果发送来数据，这个数据就会被保存再已经连接的套接字的接收缓冲区里，这个缓冲区有多大，

//最大就能接收多少数据量；

//（1.3）syn攻击【syn flood】:典型的利用TCP/IP协议涉及弱点进行坑爹的一种行为；

//拒绝服务攻击(DOS/DDOS)；

//backlog：进一步明确和规定了：指定给定套接字上内核为之排队的最大已完成连接数【已完成连接队列中最大条目数】；

//大家在写代码时尽快用accept()把已完成队列里边的连接取走，尽快 留出空闲为止给后续的已完成三路握手的条目用，那么这个已完成队列一般不会满；

//一般这个backlog值给300左右；

//二：阻塞与非阻塞I/O

//阻塞和非阻塞主要是指调用某个系统函数时，这个函数是否会导致我们的进程进入sleep()【卡在这休眠】状态而言的；

//a)阻塞I/O



//我调用一个函数，这个函数就卡在在这里，整个程序流程不往下走了【休眠sleep】，该函数卡在这里等待一个事情发生，只有这个事情发生了，这个函数才会往下走；

//这种函数，就认为是阻塞函数；accept();

//这种阻塞，并不好，效率很低；一般我们不会用阻塞方式来写服务器程序，效率低；

//b)非阻塞I/O：不会卡住，充分利用时间片，执行更高；



//非阻塞模式的两个鲜明特点：

//(1)不断的调用accept(),recvfrom()函数来检查有没有数据到来，如果没有，函数会返回一个特殊的错误标记来告诉你，这种标记可能是EWULDBLOCK，

//也可能是EAGAIN；如果数据没到来，那么这里有机会执行其他函数，但是也得不停的再次调用accept(),recvfrom()来检查数据是否到来，非常累；

//(2)如果数据到来，那么就得卡在这里把数据从内核缓冲区复制到用户缓冲区，所以复制这个阶段是卡着完成的；



//三：同步与异步I/O:这两个概念容易和 阻塞/非阻塞混淆；

//a)异步I/O：调用一个异步I/O函数时，我门要给这个函数指定一个接收缓冲区，我还要给定一个回调函数；

//调用完一个异步I/O函数后，该函数会立即返回。 其余判断交给操作系统，操作系统会判断数据是否到来，如果数据到来了，操作系统会把数据

//拷贝到你所提供的缓冲区里，然后调用你所指定的这个回调函数来通知你；

//很容易区别非阻塞和异步I/O的差别：

//（1）非阻塞I/O要不停的调用I/O函数来检查数据是否来，如果数据来了，就得卡在I/O函数这里把数据从内核缓冲区复制到用户缓冲区，然后这个函数才能返回；

//（2）异步I/O根本不需要不停的调用I/O函数来检查数据是否到来，只需要调用一次，然后就可以干别的事情去了；

//内核判断数据到来，拷贝数据到你提供的缓冲区，调用你的回调函数来通知你，你并没有被卡在那里的情况；



//b)同步I/O

//select/poll。epoll。

//1)调用select()判断有没有数据，有数据，走下来，没数据卡在那里；

//2)select()返回之后，用recvfrom()去取数据；当然取数据的时候也会卡那么一下；

//同步I/O感觉更麻烦，要调用两个函数才能把数据拿到手；

//但是同步I/O和阻塞式I/O比，就是所谓的 I/O复用【用两个函数来收数据的优势】 能力；

//（3.1）I / O复用

//所谓I/O复用，就是我多个socket【多个TCP连接】可以弄成一捆【一堆】，我可以用select这种同步I/O函数在这等数据；

//select()的能力是等多条TCP连接上的任意一条有数据来；，然后哪条TCP有数据来，我再用具体的比如recvfrom()去收。

//所以，这种调用一个函数能够判断一堆TCP连接是否来数据的这种能力，叫I/O复用，英文I/O multiplexing【I/O多路复用】

//很多资料把 阻塞I/O，非阻塞I/O，同步I/O归结为一类 ，因为他们多多少少的都有阻塞的行为发生；

//甚至有的资料直接就把 阻塞I/O，非阻塞I/O 都归结为同步I/O模型，这也是可以的】

//而把异步I/O单独归结为一类，因为异步I/O是真正的没有阻塞行为发生的；



//（3.2）思考题

//什么叫 用 异步的方法 去使用 非阻塞调用 ？

# 5.5 监听端口实战、epoll介绍及原理详析

//一：监听端口

//（1.1）开启监听端口

//二：epoll技术简介

//（2.1）epoll概述

//(1)I/O多路复用：epoll就是一种典型的I/O多路复用技术:epoll技术的最大特点是支持高并发；

//传统多路复用技术select,poll，在并发量达到1000-2000，性能就会明显下降；

//epoll,kquene(freebsd)

//epoll，从linux内核2.6引入的，2.6之前是没有的；

//(2)epoll和kquene技术类似：单独一台计算机支撑少则数万，多则数十上百万并发连接的核心技术；

//epoll技术完全没有这种性能会随着并发量提高而出现明显下降的问题。但是并发没增加一个，必定要消耗一定的内存去保存这个连接相关的数据；

//并发量总还是有限制的，不可能是无限的；

//(3)10万个连接同一时刻，可能只有几十上百个客户端给你发送数据，epoll只处理这几十上百个客户端；

//(4)很多服务器程序用多进程，每一个进程对应一个连接；也有用多线程做的，每一个线程对应 一个连接；

//epoll事件驱动机制，在单独的进程或者单独的线程里运行，收集/处理事件；没有进程/线程之间切换的消耗，高效

//(5)适合高并发，融合epoll技术到项目中，作为大家将来从事服务器开发工作的立身之本；

//写小demo非常简单，难度只有1-10，但是要把epoll技术融合到商业的环境中，那么难度就会骤然增加10倍；

//（2.2）学习epoll要达到的效果及一些说明

//(1)理解epoll的工作原理；面试考epoll技术的工作原理；

//(2)开始写代码

//(3)认可nginx epoll部分源码；并且能复用的尽量复用；

//(4)继续贯彻用啥讲啥的原则； 少就是多；

//三：epoll原理与函数介绍：三个函数，理解好久等于掌握了epoll技术的工作原理，以下内容务必认真听讲

//（3.1）课件介绍

//https://github.com/wangbojing

//a)c1000k\_test这里，测试百万并发的一些测试程序；一般以main();

//b)ntytcp：nty\_epoll\_inner.h，nty\_epoll\_rb.c

//epoll\_create();

//epoll\_ctl();

//epoll\_wait();

//epoll\_event\_callback();

//c)总结：建议学习完老师的epoll实战代码之后，再来学习 这里提到的课件代码，事半功倍；

//（3.2）epoll\_create()函数

//格式：int epoll\_create(int size);

//功能：创建一个epoll对象，返回该对象的描述符【文件描述符】，这个描述符就代表这个epoll对象，后续会用到；

//这个epoll对象最终要用close(),因为文件描述符/句柄 总是关闭的；

//size: >0;

//原理：

//a)struct eventpoll \*ep = (struct eventpoll\*)calloc(1, sizeof(struct eventpoll));

//b)rbr结构成员：代表一颗红黑树的根节点[刚开始指向空],把rbr理解成红黑树的根节点的指针；





//红黑树，用来保存 键【数字】/值【结构】，能够快速的通过你给key，把整个的键/值取出来；

//c)rdlist结构成员：代表 一个双向链表的表头指针；

//双向链表：从头访问/遍历每个元素特别快；next。



//d)总结：创建了一个eventpoll结构对象，被系统保存起来；

//rbr成员被初始化成指向一颗红黑树的根【有了一个红黑树】；

//rdlist成员被初始化成指向一个双向链表的根【有了双向链表】；

//（3.3）epoll\_ctl()函数

//格式：int epoll\_ctl(int efpd,int op,int sockid,struct epoll\_event \*event);

//功能：把一个socket以及这个socket相关的事件添加到这个epoll对象描述符中去，目的就是通过这个epoll对象来监视这个socket【客户端的TCP连接】上数据的来往情况;

//当有数据来往时，系统会通知我们；

//我们把感兴趣的事件通过epoll\_ctl（）添加到系统，当这些事件来的时候，系统会通知我们；

//efpd：epoll\_create()返回的epoll对象描述符；

//op：动作，添加/删除/修改 ，对应数字是1,2,3， EPOLL\_CTL\_ADD, EPOLL\_CTL\_DEL ,EPOLL\_CTL\_MOD

//EPOLL\_CTL\_ADD添加事件：等于你往红黑树上添加一个节点，每个客户端连入服务器后，服务器都会产生 一个对应的socket，每个连接这个socket值都不重复

//所以，这个socket就是红黑树中的key，把这个节点添加到红黑树上去；

//EPOLL\_CTL\_MOD：修改事件；你 用了EPOLL\_CTL\_ADD把节点添加到红黑树上之后，才存在修改；

//EPOLL\_CTL\_DEL：是从红黑树上把这个节点干掉；这会导致这个socket【这个tcp链接】上无法收到任何系统通知事件；

//sockid：表示客户端连接，就是你从accept()；这个是红黑树里边的key;

//event：事件信息，这里包括的是 一些事件信息；EPOLL\_CTL\_ADD和EPOLL\_CTL\_MOD都要用到这个event参数里边的事件信息；

//原理：

//a)epi = (struct epitem\*)calloc(1, sizeof(struct epitem));



//b)epi = RB\_INSERT(\_epoll\_rb\_socket, &ep->rbr, epi); 【EPOLL\_CTL\_ADD】增加节点到红黑树中

//epitem.rbn ，代表三个指针，分别指向红黑树的左子树，右子树，父亲；



//epi = RB\_REMOVE(\_epoll\_rb\_socket, &ep->rbr, epi);【EPOLL\_CTL\_DEL】，从红黑树中把节点干掉

//EPOLL\_CTL\_MOD，找到红黑树节点，修改这个节点中的内容；

//红黑树的节点是epoll\_ctl[EPOLL\_CTL\_ADD]往里增加的节点；面试可能考

//红黑树的节点是epoll\_ctl[EPOLL\_CTL\_DEL]删除的；

//总结：

//EPOLL\_CTL\_ADD：等价于往红黑树中增加节点

//EPOLL\_CTL\_DEL：等价于从红黑树中删除节点

//EPOLL\_CTL\_MOD：等价于修改已有的红黑树的节点

//当事件发生，我们如何拿到操作系统的通知；

//（3.4）epoll\_wait()函数

//格式：int epoll\_wait(int epfd,struct epoll\_event \*events,int maxevents,int timeout);

//功能：阻塞一小段时间并等待事件发生，返回事件集合，也就是获取内核的事件通知；

//说白了就是遍历这个双向链表，把这个双向链表里边的节点数据拷贝出去，拷贝完毕的就从双向链表里移除；

//因为双向链表里记录的是所有有数据/有事件的socket【TCP连接】；

//参数epfd：是epoll\_create()返回的epoll对象描述符；

//参数events：是内存，也是数组，长度 是maxevents，表示此次epoll\_wait调用可以手机到的maxevents个已经继续【已经准备好的】的读写事件；

//说白了，就是返回的是 实际 发生事件的tcp连接数目；

//参数timeout：阻塞等待的时长；

//epitem结构设计的高明之处：既能够作为红黑树中的节点，又能够作为双向链表中的节点；











//（3.5）内核向双向链表增加节点

//一般有四种情况，会使操作系统把节点插入到双向链表中；

//a)客户端完成三路握手；服务器要accept();

//b)当客户端关闭连接，服务器也要调用close()关闭；

//c)客户端发送数据来的；服务器要调用read(),recv()函数来收数据；

//d)当可以发送数据时；服务武器可以调用send(),write()；

//e)其他情况；写实战代码再说；

//（3.6）源码阅读额外说明

# 5.6 通讯代码精粹之epoll函数实战1

//一：一个更正，一个注意

//更正：kqueue

//注意：即将进入最重要，最核心的内容讲解；

//戒骄戒躁，代码精华。简单，容易理解；大家要认真学习老师给出来的代码；

//二：配置文件的修改

//增加worker\_connections项

//三：epoll函数实战

//epoll\_create(),epoll\_ctl(),epoll\_wait();系统提供的函数调用

//（3.1）ngx\_epoll\_init函数内容

//epoll\_create()：创建一个epoll对象，创建了一个红黑树，还创建了一个双向链表；

//连接池： 数组，元素数量就是worker\_connections【1024】，每个数组元素类型为 ngx\_connection\_t【结构】； ---结构数组；



//为什么要引入这个数组： 2个监听套接字， 用户连入进来，每个用户多出来一个套接字；

//把 套接字数字跟一块内存捆绑，达到的效果就是将来我通过这个套接字，就能够把这块内存拿出来；

//ngx\_get\_connection()重要函数：从连接池中找空闲连接；

//a)epoll\_create() \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//b)连接池（找空闲连接）

//c)ngx\_epoll\_add\_event（） \*\*\*\*\*\*\*\*\*\*\*\*

// epoll\_ctl();

//d)ev.data.ptr = (void \*)( (uintptr\_t)c | c->instance); 把一个指针和一个位 合二为一，塞到一个void \*中去，

//后续能够把这两个值全部取出来，如何取，取出来干嘛，后续再说；

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

//如下命令用root权限执行

//sudo su 获得root权限

//lsof -i:80 列出哪些进程在监听80端口

//netstat -tunlp | grep 80

//总结：

//a)epoll\_create();epoll\_ctl();

//b)连接池技巧ngx\_get\_connection（），ngx\_free\_connection（）；学习这种编程方法；

//c)同时传递 一个指针和一个二进制数字技巧；

//（3.2）ngx\_epoll\_init函数的调用（要在子进程中执行）

//四章，四节 project1.cpp：nginx中创建worker子进程；

//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) g\_socket.ngx\_epoll\_init(); //初始化epoll相关内容，同时 往监听socket上增加监听事件，从而开始让监听端口履行其职责

//(i) m\_epollhandle = epoll\_create(m\_worker\_connections);

//(i) ngx\_epoll\_add\_event((\*pos)->fd....);

//(i) epoll\_ctl(m\_epollhandle,eventtype,fd,&ev);

//(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