

1.2 环境搭建

一：windows 下的虚拟机安装

装虚拟机软件（虚拟出一台电脑），然后我们就可以在这个虚拟出来的电脑上来安装一个 linux 操作系统；装虚拟机软件采用的 **VMware-workstation**；

二：虚拟机中安装 linux 操作系统

a)红帽子 Red hat，收费；

b)CentOS：被红帽子收购，免费的；

c)**Ubuntu**（乌班图），免费；

用户名：kdvictor 密码：123456

ctrl+alt 一起按，就能把鼠标显示出来；

三：配置固定 IP 地址

要修改配置文件 需要 **vim 编辑器**，乌班图要安装这个编辑器：**sudo apt-get install vim-gtk** **ESC+shift+:退出**

两台主机（windows,乌班图）

ip 地址不能相同，但是要在**同一个网段**中

主动发送数据包这一端 叫 “客户端”，另一端叫 “服务器端”

windows 电脑的网络信息用 **ipconfig** 来查看；

IPv4 地址: 192.168.1.119

子网掩码: 255.255.255.0

默认网关.....: fe80::1%8

192.168.1.1

所以这个乌班图 linux 的 ip 地址：192.168.1.88(**ping 一下，看有没有人用**)

linux 上查看网络信息是用 **ifconfig**,网卡叫 ens33

cd /etc ls cd network ls

sudo vim interface

#iface ens33 inet dhcp 注释掉

Iface ens33 inet static

Address 192.168.1.88

Gateway 192.168.1.1

Netmask 255.255.255.0

vim 编辑器分 文本输入状态，命令状态，从命令状态切换到文本输入状态，需要按字母 **i**；

从文本输入状态切换回命令状态，按键盘左上边的 **esc** 键盘

在命令状态下输入 **:wq!**（存盘退出），而输入**:q!**（不存盘退出）

修改一下 dns 8.8.8.8(谷歌域名解析服务器)

Cd /etc cd resolvconf cd resolv.conf.d sudo vim base

Nameserver 8.8.8.8.

Sudo reboot 重启使配置生效

Ping www.baidu.com 看域名解析服务器有没有配置对

四: 配置远程连接

(1)需要在 linux 上安装 ssh 服务;

先看一下有没有 ssh 服务: `ps -e | grep ssh`

安装 ssh: `sudo apt-get install openssh-server`

(2)远程连接工具, 推荐 xshell;

五: 安装编译工具 `gcc` (编译 c 程序.c) ,`g++`(编译 c++程序, 就是.cpp 程序) 等

`sudo apt-get install build-essential`

`sudo apt-get install gcc`

`sudo apt-get install g++`

六: 共享一个操作目录

vim 使用的坏习惯;

二:

samba 服务; 不采用

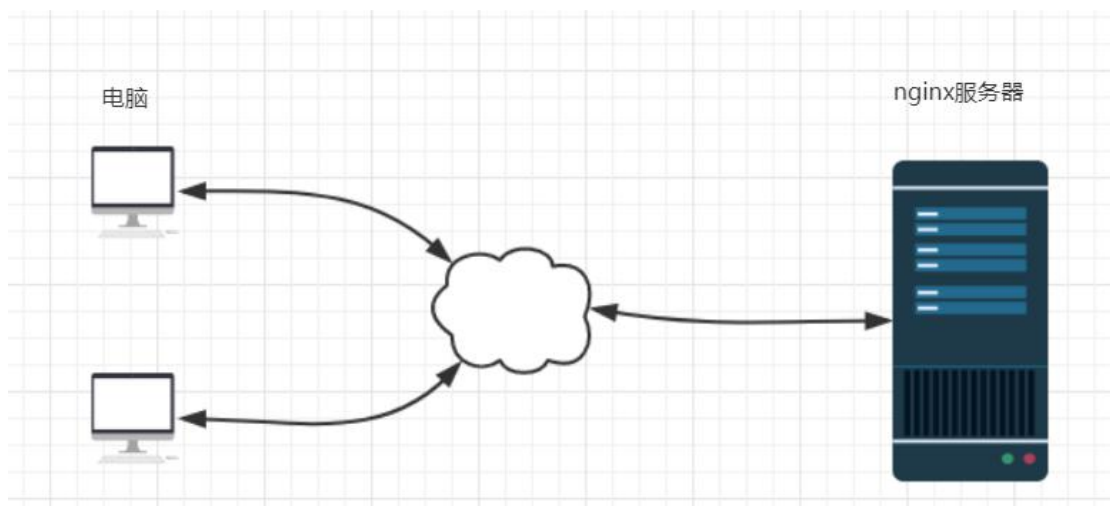
通过虚拟机, 把一个 windows 下的目录共享。让 linux 可以访问这个目录;

2.1 nginx 简介, 选择理由, 安装和使用

一: nginx 简介

nginx(2002 年开发,2004 年 10 才出现第一个版本 0.1.0):web 服务器, 市场份额, 排在第二位, Apache(1995)第一位;

`web 服务器`, 反向代理, 负载均衡, 邮件代理; 运行时需要的系统资源比较少, 所以经常被称呼为轻量级服务器;



是一个俄罗斯人 (Igor Sysoev), C 语言 (不是 c++) 开发的, 并且开源了;

nginx 号称并发处理百万级别的 TCP 连接, 非常稳定, 热部署 (运行的时候能升级), 高度模块化设计, 自由许可证。

很多人开发自己的模块来增强 nginx, 第三方业务模块 (c++开发); OpenResty;

linux epoll 技术; windows IOCP 技术 (高并发技术)

二: 为什么选择 nginx

单机 10 万并发, 而且同时能够保持高效的服务, epoll 这种高并发技术好处就是: 高并发只是占用更多内存就能 做到;

内存池, 进程池, 线程池, 事件驱动等等;

学习研究大师级的人写的代码, 是一个程序开发人员能够急速进步的最佳途径;

三: 安装 nginx, 搭建 web 服务器

(3.1) 安装前提

a)epoll,linux 内核版本为 2.6 或者以上; `uname -a`

b)gcc 编译器, g++编译器;

c)pcre 库: 函数库; 支持解析正则表达式; `sudo apt-get install libpcre3-dev`

d)zlib 库: 压缩解压缩功能; `sudo apt-get install libz-dev`

e)openssl 库: ssl 功能相关库, 用于网站加密通讯; `sudo apt-get install libssl-dev`

(3.2) nginx 源码下载以及目录结构简单认识

nginx 官网 <http://www.nginx.org>

nginx 的几种版本

(1)mainline 版本: 版本号中间数字一般为奇数。更新快, 一个月内就会发布一个新版本, 最新功能, bug 修复等, 稳定性差一点;

(2)stable 版本: 稳定版, 版本号中间数字一般为偶数。经过了长时间的测试, 比较稳定, 商业化环境中用这种版本; 这种版本发布周期比较长, 几个月;

(3)Legacy 版本: 遗产, 遗留版本, 以往的老版本;

安装, 现在有这种二进制版本: 通过命令行直接安装;

灵活: 要通过编译 nginx 源码手段才能把第三方模块弄进来;

`Mkdir ngxsourcecode`

`Cd ngxsourcecode`

`wget http://nginx.org/download/nginx-1.14.2.tar.gz` (.tar.gz:linux 下的压缩文件扩展名)

`tar -xzf nginx-1.14.2.tar.gz`

/*

Nginx 的目录结构如下:

`auto /`:编译相关的脚本, 可执行文件 `configure` 一会会用到这些脚本

`cc /`: 检查编译器的脚本

`lib /`: 检查依赖库的脚本

`os /`: 检查操作系统类型的脚本

`type /`: 检查平台类型的脚本

`CHANGES`: 修复的 bug, 新增加的功能说明

`CHANGES.ru`: 俄语版 `CHANGES`

`conf /`: 默认的配置文

`configure`: 编译 nginx 之前必须先执行本脚本以生成一些必要的中间文件

`contrib /`: 脚本和工具, 典型的是 vim 高亮工具

`vim /`: vim 高亮工具 `cp -r contrib/vim ~/.vim`

`html /`: 欢迎界面和错误界面相关的 html 文件

man / : nginx 帮助文件目录

src / : nginx 源码目录

core : 核心代码

event : event(事件)模块相关代码

http : http(web 服务)模块相关代码

mail : 邮件模块相关代码

os : 操作系统相关代码

stream : 流处理相关代码

objs/:执行了 configure 生成的中间文件目录

ngx_modules.c: 内容决定了我们一会编译 nginx 的时候有哪些模块会被编译到 nginx 里边来。

Makefile:执行了 configure 脚本产生的编译规则文件, 执行 make 命令时用到

*/

(3.3) nginx 的编译和安装

a)编译的第一步: 用 configure 来进行编译之前的配置工作

./configure

--prefix: 指定最终安装到的目录: 默认值 /usr/local/nginx

--sbin-path: 用来指定可执行文件目录: 默认的是 sbin/ nginx

--conf-path: 用来指定配置文件目录: 默认的是 conf/nginx.conf

b)用 make 来编译,生成了可执行文件 make

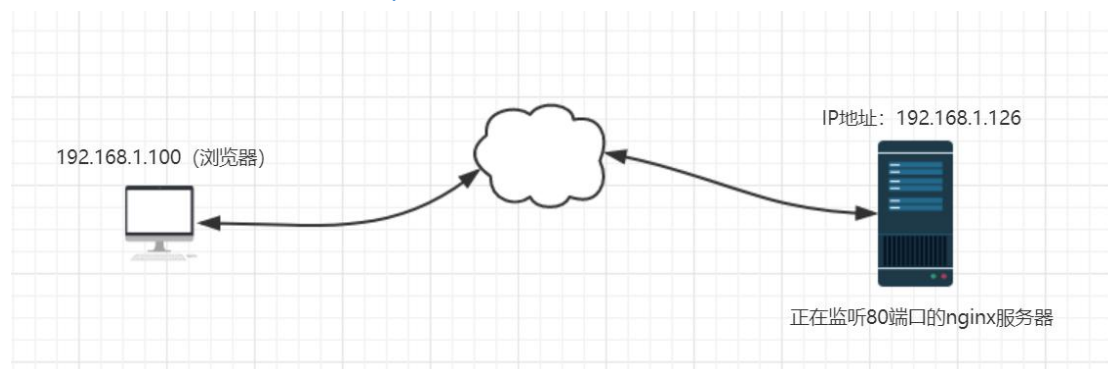
c)用 make 命令开始安装 sudo make install

四: nginx 的启动和简单使用

Ps -ef | grep nginx

启动: sudo ./nginx

乌班图: 192.168.1.88; [http:192.168.1.88/](http://192.168.1.88/), 默认监听 80 端口



百度: “服务器程序端口号”;

百度: “监听端口号”;

(4.1) 通讯程序基础概念

a)找个人: 这个人住哪 (IP 地址), 第二个事情是知道它叫什么 (端口号);

2.2 nginx 整体结构和进程模型

一：nginx 的整体结构

(1.1) master 进程和 worker 进程概览(父子关系)

启动 nginx，看到了一个 master 进程，一个 worker 进程

ps -ef 命令

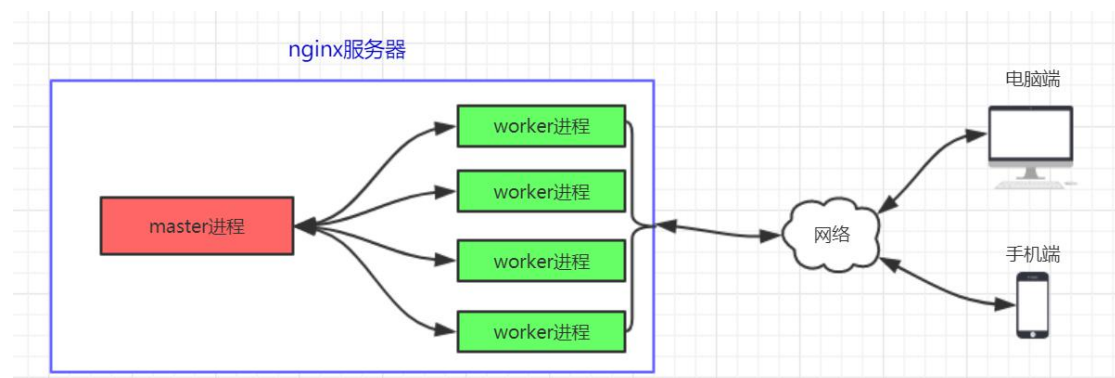
第一列：UID，进程所属的用户 id

第二列：进程 ID (PID),用来唯一的标识一个进程

第三列：父进程 ID (PPID)。 fork ()， worker 进程是被 master 进程通过 fork()创建出来的-worker 进程是 master 进程的子进程,master 是父进程

(1.2) nginx 进程模型

1 个 master 进程， 1 到多个 worker 进程 这种工作机制来对外服务的；这种工作机制保证了 nginx 能够稳定、灵活的运行；



a)master 进程责任：监控进程，不处理具体业务，专门用来管理和监控 worker 进程；master，角色是监工，比如清闲；

b)worker 进程：用来干主要的活的，(和用户交互)；

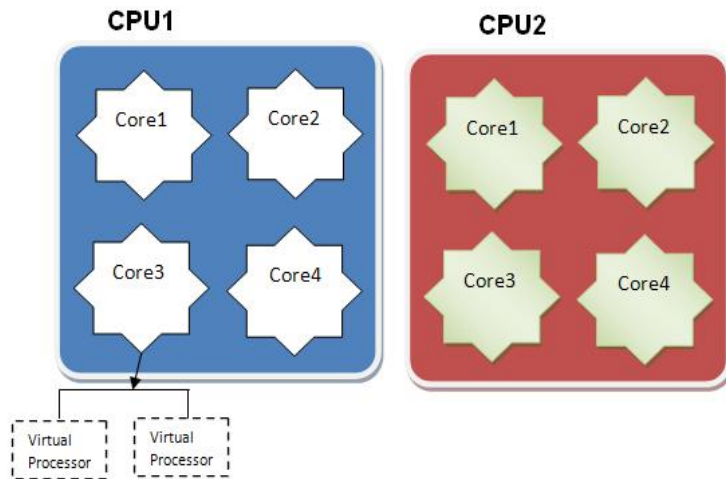
c)master 进程和 worker 进程之间要通讯，可以用 信号 ， 也可以用 共享内存 ；

d)稳定性，灵活性，体现之一： worker 进程 一旦挂掉，那么 master 进程会立即 fork()一个新的 worker 进程投入工作中去；

(1.3) 调整 worker 进程数量

worker 进程几个合适呢？公认的做法： 多核计算机，就让每个 worker 运行在一个单独的内核上，最大限度减少 CPU 进程切换成本，提高系统运行效率；

物理机： 8 核(8 个 processors)；



工作站: 2 个物理 cpu ,蓝色的一个 cpu, 红色的一个 cpu

每个物理 cpu 里边内核数量, 是 4 个; core1 --core4

每个 core 里边有两个逻辑处理器 (超线程技术/siblings)

16 个 processors(最细小的单位, 也就是平时大家说的处理器个数)

查看 linux 系统所在主机的 cpu processor: `grep -c processor /proc/cpuinfo`

重新配置虚拟机的处理器数量, 不超过所在物理电脑的 processor 数量为宜

将 nginx 的 worker 进程也配置成相应的个数:

`cd /usr/local/nginx`

`cd conf`

`sudo vim nginx.conf`

修改 `worker_processes`

二: nginx 进程模型细说

稳定 , 灵活

(2.1) nginx 重载配置文件: `sudo ./nginx -s reload`

可以发现 worker 进程的 pid 全变了, 也就是新的 worker 进程

`sudo kill 1059` 关闭一个 worker 进程, 会自动重新 `fork()` 一个新的 worker 进程

(2.2) nginx 热升级,热回滚

(2.3) nginx 的关闭

`./nginx -? / ./nginx -h` 看一看

`./nginx -s stop` 简单粗糙退出 `./nginx -s quit` 优雅退出

`sudo ./nginx -s quit`

(2.4) 总结

多进程, 多线程(windows 一般使用):

多线程模型的弊端: 共享内存,如果某个线程报错一定会影响到其他线程, 最终会导致整个服务器程序崩溃。

3.1 学习 nginx 之前的准备工作

二: nginx 源码查看工具

visual studio,source insight,visual studio Code.

采用 Visual Studio Code 来阅读 nginx 源码

Visual Studio Code:微软公司开发的一个跨平台的轻量级的编辑器 (不要混淆 vs2017:IDE 集成开发环境, 以编译器);

Visual Studio Code 在其中可以安装很多扩展模块;

1.30.0 版本, 免费的,多平台;

官方地址: <https://code.visualstudio.com>

<https://code.visualstudio.com/download>

为支持语法高亮, 跳转到函数等等, 可能需要安装扩展包;

三: nginx 源码入口函数定位

四: 创建一个自己的 linux 下的 c 语言程序

共享目录不见了, 一般可能是虚拟机自带的工具 VMWare tools 可能有问题;

VMWare-tools 是 VMware 虚拟机自带的一系列的增强工具, 文件共享功能就是 VMWare-tools 工具里边的

a)虚拟机->重新安装 VMware tools 塞了一块虚拟光盘

b)sudo mkdir /mnt/cdrom

c)sudo mount /dev/cdrom /mnt/cdrom 将光盘挂载到自己的 cdrom 目录

d)cd /mnt/cdrom

e)sudo cp VMwareTool....tar.gz ../

f)cd ..

g)sudo tar -zxvf VMwareToo.....tar.gz

h)cd vmware-tools-distrib

j)sudo ./vmware-install.pl

一路回车。

gcc 编译.c, g++编译 c++

.c 文件若很多, 都需要编译, 那么咱们就要写专门的 MakeFile 来编译了;

gcc -o:用于指定最终的可执行文件名

五: nginx 源码怎么讲

(1)讲与不讲, 是主观的;

(2)以讲解通讯代码为主。其他的也会涉及, 创建进程, 处理信号;

(3)有必要的老师带着大家看源码, 解释源码;

(4)把这些 nginx 中的精华的源码提取出来; 带着大家往新工程中增加新代码, 编译, 运行, 讲解; 入到自己的知识库, 这些是加薪的筹码

3.2 nginx 源码学习，终端进程之间的关系

一：nginx 源码学习方法

(1)简单粗暴，啃代码，比较艰苦，需要比较好的基础

(2)看书看资料,逃脱不了啃代码的过程

(3)跟着老师学，让大家用最少的力气掌握 nginx 中最有用的东西。

架构课程前期学习两个主要任务；

(1)泛读 nginx 中的一些重要代码；

(2)把 nginx 中最重要的代码提取出来作为我们自己知识库的一部分以备将来使用；

二：终端和进程的关系

(2.1) 终端与 bash 进程

ps -ef | grep bash: e(显示所有进程), f(以长格式显示),

pts(虚拟终端), 每连接一个虚拟终端到乌班图 linux 操作系统,

就会出现 一个 bash 进程 (shell[壳]),黑窗口, 用于解释用户输入的命令

bash = shell = 命令行解释器

开一个终端, 操作系统就会启动一个 bash

whereis bash

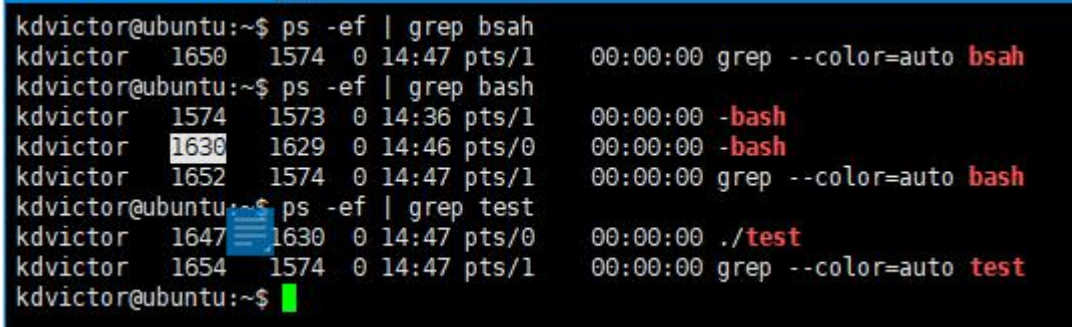
(2.2) 终端上的开启进程

ps -la

man ps

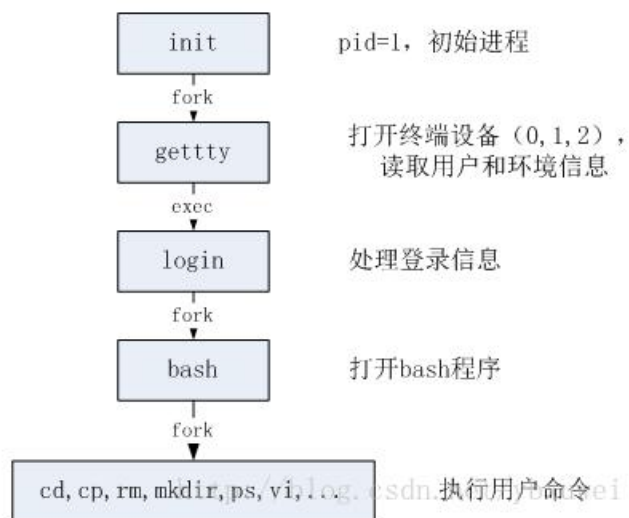
随着终端的退出, 这个终端上运行的进程 nginx 也退出了; 此 nginx 是自己写的程序, 只不过名字叫 nginx。

可执行程序 nginx 是 bash 的子进程;



```
kdvictor@ubuntu:~$ ps -ef | grep bsah
kdvictor  1650  1574  0 14:47 pts/1    00:00:00 grep --color=auto bsah
kdvictor@ubuntu:~$ ps -ef | grep bash
kdvictor  1574  1573  0 14:36 pts/1    00:00:00 -bash
kdvictor  1630  1629  0 14:46 pts/0    00:00:00 -bash
kdvictor  1652  1574  0 14:47 pts/1    00:00:00 grep --color=auto bash
kdvictor@ubuntu:~$ ps -ef | grep test
kdvictor  1647  1630  0 14:47 pts/0    00:00:00 ./test
kdvictor  1654  1574  0 14:47 pts/1    00:00:00 grep --color=auto test
kdvictor@ubuntu:~$
```

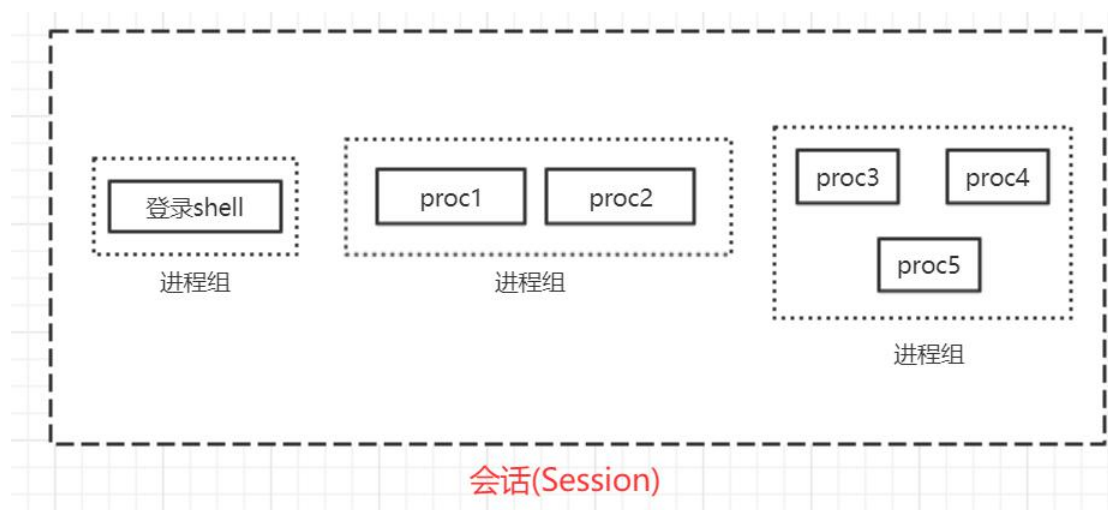

《unix 环境高级编程》第九章 进程关系:



(2.3) 进程关系进一步分析

每个进程还属于一个进程组:一个或者多个进程的集合, 每个进程组有一个唯一的进程组 ID, 可以调用系统 函数来创建进程组、加入进程组

“会话” (session): 是一个或者多个进程组的集合



一般, 只要不进行特殊的系统函数调用, 一个 bash(shell)上边运行的所有程序都属于一个会话, 而这个会话有一个 session leader;

那么这个 bash(shell)通常就是 session leader; 你可以调用系统功函数创建新的 session。

```
ps -eo pid,ppid,sid,tt,pgrp,comm | grep -E 'bash|PID|nginx'
```



```
or 'ps --help <sl|o|t|m|a>'
for additional help text.

For more details see ps(1).
kdvictor@ubuntu:~$ ps -eo pid,ppid,sid,ttty,pgrp,comm | grep -E 'bash|PID|test'
  PID  PPID   SID TT      PGRP COMMAND
  1574  1573   1574 pts/1    1574  bash
  1630  1629   1630 pts/0    1630  bash
  1647  1630   1630 pts/0    1647  test
kdvictor@ubuntu:~$ ps -eo pid,ppid,sid,ttty,pgrp,comm | grep -E 'bash|PID|test'
  PID  PPID   SID TT      PGRP COMMAND
  1574  1573   1574 pts/1    1574  bash
  1630  1629   1630 pts/0    1630  bash
  1647  1630   1630 pts/0    1647  test
  1689  1688   1689 pts/2    1689  bash
kdvictor@ubuntu:~$ sudo strace -e trace=signal -p 1630
[sudo] password for kdvictor:
strace: Process 1630 attached
--- SIGHUP {si_signo=SIGHUP, si_code=SI_KERNEL} ---
--- SIGCONT {si_signo=SIGCONT, si_code=SI_KERNEL} ---
rt_sigreturn(mask=[CHLD]) = -1 EINTR (Interrupted system call)
kill(4294965649, SIGHUP) = 0
rt_sigprocmask(SIG_BLOCK, [CHLD TSTP TTIN TTOU], [CHLD], 8) = 0
rt_sigprocmask(SIG_SETMASK, [CHLD], NULL, 8) = 0
rt_sigaction(SIGHUP, [{SIG_DFL, [], SA_RESTORER, 0x7fb3098004b0}, {0x45fa10, [HUP INT ILL TRAP ABRT BUS FPE USR1 S
EGV USR2 PIPE ALRM TERM XCPU XFSZ VTALRM SYS], SA_RESTORER, 0x7fb3098004b0}], 8) = 0
kill(1630, SIGHUP) = 0
--- SIGHUP {si_signo=SIGHUP, si_code=SI_USER, si_pid=1630, si_uid=1000} ---
+++ killed by SIGHUP +++
kdvictor@ubuntu:~$
```

192.168.1.88

```
New release '18.04.2 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Thu Aug 29 14:46:45 2019 from 192.168.1.119
kdvictor@ubuntu:~$ clear
kdvictor@ubuntu:~$ sudo strace -e trace = signal -p 1647
[sudo] password for kdvictor:
strace: invalid system call 'trace'
kdvictor@ubuntu:~$ sudo strace -e trace= signal -p 1647
strace: Can't stat 'signal': No such file or directory
kdvictor@ubuntu:~$ sudo strace -e trace=signal -p 1647
strace: Process 1647 attached
--- SIGHUP {si_signo=SIGHUP, si_code=SI_USER, si_pid=1630, si_uid=1000} ---
+++ killed by SIGHUP +++
kdvictor@ubuntu:~$
```

(2.5) 终端关闭时如何让进程不退出

设想

a)nginx 进程拦截 (忽略) SIGHUP(nginx 收到这个信号并告诉操作系统, 我不想死, 请不要把我杀死)信号, 是不是可以;

b)nginx 进程和 bash 进程不再同一个 session 里;

孤儿进程

```
  PID  PPID   SID TT      PGRP COMMAND
  1238  1237   1238 pts/1    1238  bash
  1278  1277   1278 pts/2    1278  bash
  1353    1   1324 ?        1353  test
kdvictor@ubuntu:~$
```

setsid 函数不适合进程组组长调用;

setsid 命令:启动一个进程, 而且能够使启动的进程在一个新的 session 中, 这样的话, 终端关闭时该进程就不会退出, 试试

setsid ./nginx

nohup(no hang up 不要挂断), 用该命令启动的进程跟上边忽略掉 SIGHUP 信号, 道理

相同

```
kdvictor@ubuntu:/mnt/hgfs/nginx$ nohup ./test
nohup: ignoring input and appending output to 'nohup.out'
```

该命令会把屏幕输出重新定位到当前目录的 nohup.out

```
01 -xl-xl-x 1 root root 4192 Aug 30 2019 ..
-rwxrwxrwx 1 root root 2048 Aug 30 15:21 nohup.out
-rwxrwxrwx 1 root root 8656 Aug 30 15:15 test
-rwxrwxrwx 1 root root 3072 Aug 30 15:21 nohup.out
```

```
ps -eo pid,ppid,sid,ty,pgpr,comm,cmd|grep -E 'bash|PID|nginx'
```

(2.6) 后台运行 & (./test &) 杀不掉，能执行其他命令

后台执行，执行这个程序的同时，你的终端能够干其他事情；你如果不用 后台执行，那么你执行这个程序后

你的终端就只能等这个程序完成后才能继续执行其他的操作；

fg 切换到前台

终端断开，后台进程一样会被杀掉，前后台执行不影响终端断开时杀掉进程

3.3 信号的概念，认识，处理动作

一：信号的基本概念

进程之间的常用通信手段：发送信号,kill, (杀掉一个 worker, master 会重启一个 worker) 第二章第二节讲过；

上节课讨论过 SIGHUP

信号：通知（事情通知），用来通知某个进程发生了某一个事情；

事情，信号都是突发事件，信号是异步发生的，信号也被称呼为“软件中断”

信号如何产生：

a) 某个进程发送给另外一个进程或者发送给自己；

b) 由内核(操作系统)发送给某个进程

b.1) 通过在键盘输入命令 ctrl+c[中断信号], kill 命令

b.2) 内存访问异常，除数为 0 等等，硬件都会检测到并且通知内核；

信号名字，都是以 SIG 开头，上节课 SIGHUP（终端断开信号）

UNIX 以及类（类似）UNIX 操作系统(linux, freebsd, solaris)；支持的信号数量各不相同。10-60 多个之间；

信号既有名字，其实也都是些数字，信号是一些正整数常量；信号就是宏定义（数字，从 1 开始）

```
#include <signal.h>(/usr/include/)
```

gcc 头文件搜索路径

头文件, 包含路径: /usr/local/include/ /usr/include/

库文件, 连接路径: /usr/local/lib/ /usr/lib

sudo find / -name "signal.h" | xargs grep -in "SIGHUP"

(/)根目录, (grep)文本搜索工具, (i)忽略大小写, (n)显示行号, (xargs)把内容搞到
gep 中去, 如果不用就不是内容了

```
kdvictor@ubuntu:~$ sudo find / -name "signal.h" | xargs grep -in "SIGHUP"
[sudo] password for kdvictor:
/usr/src/linux-headers-4.4.0-87/arch/xtensa/include/uapi/asm/signal.h:34:#define SIGHUP 1
/usr/src/linux-headers-4.4.0-87/arch/arm/include/uapi/asm/signal.h:17:#define SIGHUP 1
/usr/src/linux-headers-4.4.0-87/arch/cris/include/uapi/asm/signal.h:17:#define SIGHUP 1
/usr/src/linux-headers-4.4.0-87/arch/powerpc/include/uapi/asm/signal.h:20:#define SIGHUP 1
/usr/src/linux-headers-4.4.0-87/arch/m32r/include/uapi/asm/signal.h:19:#define SIGHUP 1
/usr/src/linux-headers-4.4.0-87/arch/h8300/include/uapi/asm/signal.h:17:#define SIGHUP 1
/usr/src/linux-headers-4.4.0-87/arch/ia64/include/uapi/asm/signal.h:12:#define SIGHUP 1
/usr/src/linux-headers-4.4.0-87/arch/x86/include/uapi/asm/signal.h:22:#define SIGHUP 1
/usr/src/linux-headers-4.4.0-87/arch/mn10300/include/uapi/asm/signal.h:27:#define SIGHUP 1
/usr/src/linux-headers-4.4.0-87/arch/parisc/include/uapi/asm/signal.h:4:#define SIGHUP 1
/usr/src/linux-headers-4.4.0-87/arch/avr32/include/uapi/asm/signal.h:24:#define SIGHUP 1
/usr/src/linux-headers-4.4.0-87/arch/m68k/include/uapi/asm/signal.h:17:#define SIGHUP 1
/usr/src/linux-headers-4.4.0-87/arch/s390/include/uapi/asm/signal.h:25:#define SIGHUP 1
/usr/src/linux-headers-4.4.0-87/arch/mips/include/uapi/asm/signal.h:24:#define SIGHUP 1
(POSIX). */
/usr/src/linux-headers-4.4.0-87/arch/sparc/include/uapi/asm/signal.h:11:#define SIGHUP 1
/usr/src/linux-headers-4.4.0-87/arch/alpha/include/uapi/asm/signal.h:22:#define SIGHUP 1
/usr/src/linux-headers-4.4.0-87/include/linux/signal.h:343: * | SIGHUP | terminate |
/usr/src/linux-headers-4.4.0-87/include/uapi/asm-generic/signal.h:10:#define SIGHUP 1
/usr/include/asm-generic/signal.h:10:#define SIGHUP 1
/usr/include/x86_64-linux-gnu/asm/signal.h:20:#define SIGHUP 1
kdvictor@ubuntu:~$
```

```
#define SIGHUP 1
#define SIGINT 2
#define SIGQUIT 3
#define SIGILL 4
#define SIGTRAP 5
#define SIGABRT 6
#define SIGIOT 6
#define SIGBUS 7
#define SIGFPE 8
#define SIGKILL 9
#define SIGUSR1 10
#define SIGSEGV 11
#define SIGUSR2 12
#define SIGPIPE 13
#define SIGALRM 14
#define SIGTERM 15
#define SIGSTKFLT 16
#define SIGCHLD 17
#define SIGCONT 18
#define SIGSTOP 19
#define SIGTSTP 20
#define SIGTTIN 21
#define SIGTTOU 22
#define SIGURG 23
#define SIGXCPU 24
#define SIGXFSZ 25
#define SIGVTALRM 26
#define SIGPROF 27
#define SIGWINCH 28
#define SIGIO 29
#define SIGPOLL SIGIO
/*
#define SIGLOST 29
*/
#define SIGPWR 30
#define SIGSYS 31
#define SIGUNUSED 31

/* These should not be considered constants
#define SIGRTMIN 32
#define SIGRTMAX _NSIG
*/
```


二: 通过 kill 命令认识一些信号

kill :kill 进程 id ,他的工作是发个信号给进程;

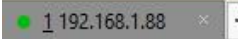
kill 能给进程发送多种信号;

ps -eo pid,ppid,sid,ttty,pgrp,comm | grep -E 'bash|PID|nginx'

sudo strace -e trace=signal -p 1184

a)如果你单纯的用 kill 进程 id, 那么就是往 进程发送 SIGTERM 信号 (终止信号)
等于 kill -15 id

```
Last login: Sun Sep 1 14:56:09 2019 from 192.168.1.119
kdvictor@ubuntu:~$ clear
kdvictor@ubuntu:~$ ps -eo pid,ppid,sid,ttty,pgrp,comm | grep -E 'bash|PID|test'
  PID  PPID  SID TT      PGRP COMMAND
  1194  1193  1194 pts/0    1194 bash
  1264  1263  1264 pts/1    1264 bash
  1298  1194  1194 pts/0    1298 test
kdvictor@ubuntu:~$ sudo strace -e trace=signal -p 1298
[sudo] password for kdvector:
strace: Process 1298 attached
--- SIGWINCH {si_signo=SIGWINCH, si_code=SI_KERNEL} ---
--- SIGTERM {si_signo=SIGTERM, si_code=SI_USER, si_pid=1323, si_uid=1000} ---
+++ killed by SIGTERM +++
kdvictor@ubuntu:~$
```




```
kdvictor@ubuntu:~$ ps -eo pid,ppid,sid,ttty,pgrp,comm | grep -E 'bash|PID|test'
  PID  PPID  SID TT      PGRP COMMAND
  1194  1193  1194 pts/0    1194 bash
  1264  1263  1264 pts/1    1264 bash
  1298  1194  1194 pts/0    1298 test
  1323  1322  1323 pts/2    1323 bash
kdvictor@ubuntu:~$ kill 1298
kdvictor@ubuntu:~$
```

kill -数字 进程 id, 能发出跟这个数字对应的信号 -1 进程 id, SIGHUP 信号去

b)如果我们用 kill -1 进程 id, 那么就是往进程 nginx 发送 SIGHUP 终止信号; 同时进程 nginx 就被终止掉了;

```
  PID  PPID  SID TT      PGRP COMMAND
  1194  1193  1194 pts/0    1194 bash
  1264  1263  1264 pts/1    1264 bash
  1323  1322  1323 pts/2    1323 bash
  1340  1194  1194 pts/0    1340 test
kdvictor@ubuntu:~$ sudo strace -e trace=signal -p 1340
strace: Process 1340 attached
--- SIGHUP {si_signo=SIGHUP, si_code=SI_USER, si_pid=1323, si_uid=1000} ---
+++ killed by SIGHUP +++
kdvictor@ubuntu:~$
```



```
kdvictor@ubuntu:~$ ps -eo pid,ppid,sid,ttty,pgrp,comm | grep -E 'bash|PID|test'
  PID  PPID  SID TT      PGRP COMMAND
  1194  1193  1194 pts/0    1194 bash
  1264  1263  1264 pts/1    1264 bash
  1298  1194  1194 pts/0    1298 test
  1323  1322  1323 pts/2    1323 bash
kdvictor@ubuntu:~$ kill 1298
kdvictor@ubuntu:~$ ps -eo pid,ppid,sid,ttty,pgrp,comm | grep -E 'bash|PID|test'
  PID  PPID  SID TT      PGRP COMMAND
  1194  1193  1194 pts/0    1194 bash
  1264  1263  1264 pts/1    1264 bash
  1323  1322  1323 pts/2    1323 bash
  1340  1194  1194 pts/0    1340 test
kdvictor@ubuntu:~$ kill -1 1340
kdvictor@ubuntu:~$
```

c)kill -2 进程 id, 发送 SIGINT 信号;
kill -数字 进程 id , 能发送出多种信号;
绝大多数的信号的缺省动作都是把进程杀死

kill 的参数	该参数发出的信号	操作系统缺省动作
-1	SIGHUP (连接断开)	终止掉进程 (进程没了)
-2	SIGINT (终端中断符, 比如 ctrl+c)	终止掉进程 (进程没了)
-3	SIGQUIT (终端退出符, 比如 ctrl+\)	终止掉进程 (进程没了)
-9	SIGKILL (终止)	终止掉进程 (进程没了)
-18	SIGCONT (使暂停的进程继续)	忽略 (进程依旧在运行不受影
-19	SIGSTOP (停止), 可用 SIGCONT 继续, 但任务被放到了后台	停止进程 (不是终止, 进程还
-20	SIGTSTP (终端停止符, 比如 ctrl+z), 但任务被放到了后台, 可用 SIGCONT 继续	停止进程 (不是终止, 进程还

三：进程的状态

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

```

1323 pts/2 bash
kdvictor@ubuntu:~$ ps -eo pid,ppid,sid,ttty,pgrp,comm,stat | grep -E 'bash|PID|test'
  PID  PPID  SID TT  PGRP COMMAND  STAT
 1194  1193  1194 pts/0  1194  bash    Ss
 1264  1263  1264 pts/1  1264  bash    Ss
 1323  1322  1323 pts/2  1323  bash    Ss+
 1359  1194  1194 pts/0  1359  test    S+
```

ps aux | grep -E 'bash|PID|nginx' aux 所谓 BSD 风格显示格式;
kill 只是发个信号, 而不是单纯的杀死的意思。

状态	含义
D	不可中断的休眠状态(通常是 I/O 的进程), 可以处理信号, 有 延迟
R	可执行状态&运行状态(在运行队列里的状态)
S	可中断的休眠状态之中 (等待某事件完成), 可以处理信号
T	停止或被追踪 (被作业控制信号所停止)
Z	僵尸进程
X	死掉的进程
<	高优先级的进程
N	低优先级的进程
L	有些页被锁进内存
s	Session leader (进程的领导者), 在它下面有子进程
t	追踪期间被调试器所停止
+	位于前台的进程组

四：常用的信号列举

信号名	信号含义
SIGHUP (连接断开)	是终端断开信号，如果终端接口检测到一个连接断开，发送此信号到该终端所在的会话首进程（前面讲过），缺省动作会导致所有相关的进程退出(上节课也重点讲了这个信号，xshell 断开就有这个信号送过来)； Kill -1 进程号也能发送此信号给进程；
SIGALRM (定时器超时)	一般调用系统函数 alarm 创建定时器，定时器超时了就会这个信号；
SIGINT (中断)	从键盘上输入 ctrl+C(中断键)【比如你进程正跑着循环干一个事】，这一 ctrl+C 就能打断你干的事，终止进程； 但 shell 会将后台进程对该信号的处理设置为忽略 (也就是说该进程若在后台运行则不会收到该信号)；
SIGSEGV (无效内存)	内存访问异常，除数为 0 等，硬件会检测到并通知内核；其实这个 SEGV 代表段违例 (segmentation violation)，你有的时候运行一个你编译出来的可执行的 c 程序，如果内存有问题，执行的时候就会出现这个提示；
SIGIO (异步 I/O)	通用异步 I/O 信号，咱们以后学通讯的时候，如果通讯套接口上有数据到达，或发生一些异步错误，内核就会通知我们这个信号；
SIGCHLD (子进程改变)	一个进程终止或者停止时，这个信号会被发送给父进程；(我们想象下 nginx, worker 进程终止时 master 进程应该会收到内核发出的针对该信号的通知)；
SIGUSR1,SIGUSR2 (都是用户定义信号)	用户定义的信号，可用于应用程序，用到再说；
SIGTERM (终止)	一般你通过在命令行上输入 kill 命令来杀一个进程的时候就会触发这个信号，收到这个信号后，你有机会退出前的处理，实现这种所谓优雅退出的效果；
SIGKILL (终止)	不能被忽略，这是杀死任意进程的可靠方法，不能被进程本身捕捉
SIGSTOP (停止)	不能被忽略，使进程停止运行，可以用 SIGCONT 继续运行，但进程被放入到了后台
SIGQUIT (终端退出符)	从键盘上按 ctrl+\ 但 shell 会将后台进程对该信号的处理设置为忽略 (也就是说该进程若在后台运行则不会收到该信号)；
SIGCONT (使暂停进程继续)	使暂停的进程继续运行
SIGTSTP (终端停止符)	从键盘上按 ctrl+z，进程被停止，并被放入后台，可以用 SIGCONT 继续运行

五：信号处理的相关动作

当某个信号出现时，我们可以按三种方式之一进行处理，我们称之为信号的处理或者与信号相关的动作；

(1)执行系统默认动作，绝大多数信号的默认动作是杀死你这个进程；

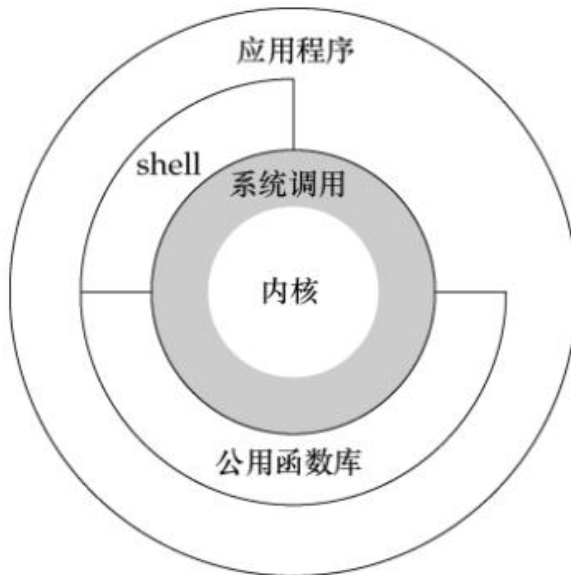
(2)忽略此信号(但是不包括 SIGKILL 和 SIGSTOP)

kill -9 进程 id，是一定能够把这个进程杀掉的；

(3)捕捉该信号：我写个处理函数，信号来的时候，我就用处理函数来处理；(但是不包括 SIGKILL 和 SIGSTOP)

3.4 信号编程初步

一：Unix/Linux 操作系统体系结构



类 Unix 操作系统体系结构分为两个状态 (1) 用户态, (2) 内核态

a)操作系统/内核：用来控制计算机硬件资源，提供应用程序运行的环境

我们写的程序，他要么运行在用户态，要么运行在内核态。一般来讲运行在用户态；

当程序要执行 一些特殊代码的时候，程序就可能切换到内核态，这种切换由操作系统控制，不需要人为介入；

换种角度理解：用户态：最外圈应用程序的活动空间；

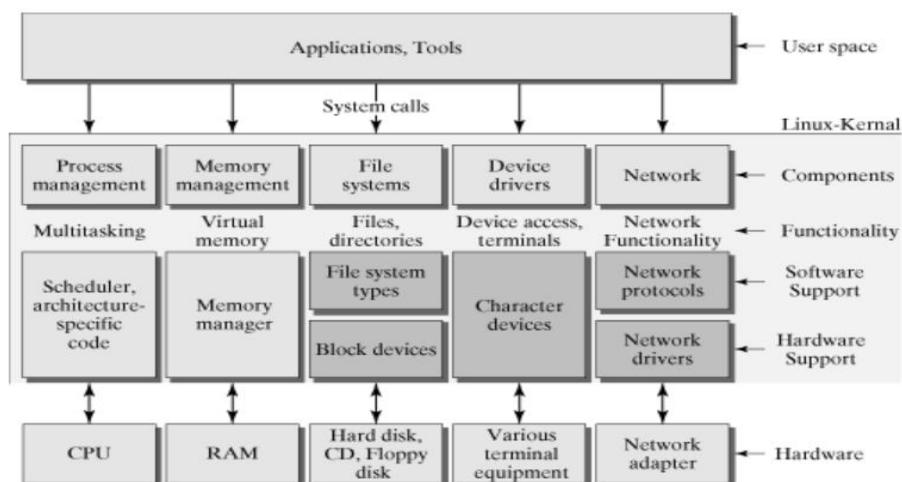
b)系统调用：就是一些函数（系统函数），你只需要调用这些函数；

c)shell： bash(borne again shell[重新装配的 shell]),它是 shell 的一种，linux 上默认采用的是 bash 这种 shell；

通俗一点理解，bash 是一个可执行程序；主要作用是：把用户输入的命令翻译给操作系统（命令解释器）；

```
kdvictor@ubuntu:~$ whereis bash
bash: /bin/bash /etc/bash.bashrc /usr/share/man/man1/bash.1.gz
kdvictor@ubuntu:~$ /bin/bash
kdvictor@ubuntu:~$ ps -ef | grep bash
kdvictor  1278  1277  0 14:49 pts/0    00:00:00 -bash
kdvictor  1294  1278  0 14:50 pts/0    00:00:00 /bin/bash
kdvictor  1305  1294  0 14:50 pts/0    00:00:00 [bash]
kdvictor@ubuntu:~$
```

分隔系统调用 和应用程序； 有胶水的感觉；



d) 用户态，内核态之间的切换

运行于用户态的进程可以执行的操作和访问的资源会受到极大限制（用户态权限小）；

而运行在内核态的进程可以执行任何操作并且在资源的使用上没有限制（内核态权限大）；

一个进程执行的时候，大部分时间是处于用户态下的，只有需要内核所提供的服务时才会切换到内核态，内核态做的事情完成后，

又转回到用户态；

`malloc();printf();` 这种状态在转换是操作系统干的，不需要我们介入；

疑问：为什么要区分用户态，内核态；

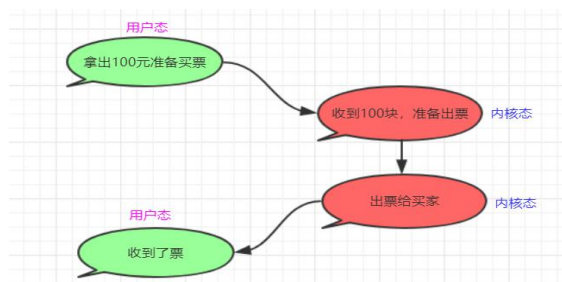
大概有两个目的：

(1)一般情况下，程序都运行在用户态状态，权限小，不至于危害到系统其它部分；当你干一些危险的事情的时候，系统给你提供接口，让你去干；

(2)既然这些接口是系统提供给你的，那么这些接口也是操作系统统一管理的；

资源是有限的，如果大家都来访问这些资源，如果不加以管理，一个是访问冲突，一个是被访问的资源如果耗尽，那系统还可能崩溃；

系统提供这些接口，就是为了减少有限的资源的访问以及使用上冲突；



那么什么时候从用户态切换到内核态去呢？

a) 系统调用，比如调用 `malloc();`

b) 异常事件，比如来了个信号；

c) 外围设备中断；

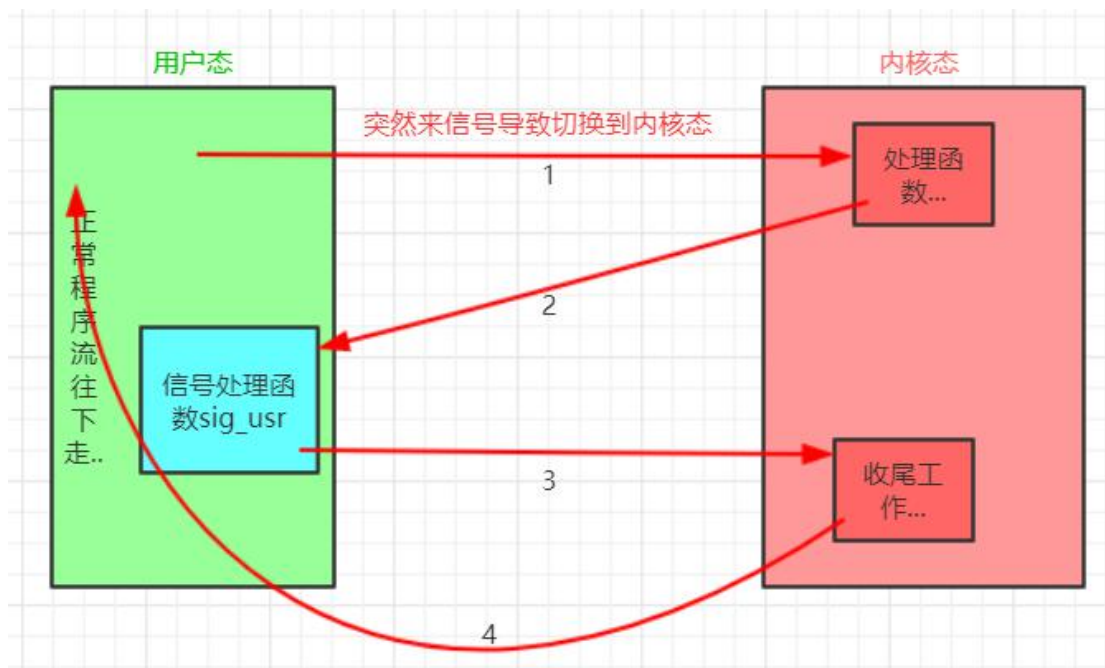
总结起来，大家只需要知道 用户态，内核态，两者根据需要自动切换

二: signal 函数范例

信号来了之后, 我们可以忽略, 可以捕捉, 咱们可以用 `signal` 函数来搞定这个事;

```
ps -eo pid,ppid,sid,tt,pg,comm | grep -E 'bash|PID|nginx'
```

你的进程收到了信号, 这个事 就会被内核注意到;



(2.1)可重入函数

严格意义: `muNEfunc()` 函数不应该是一个可重入函数

所谓的可重入函数: 就是我们在信号处理函数中 调用它 是安全的;

可重入函数: 在信号处理程序中保证调用安全的函数, 这些函数是可重入的并被称为异步信号安全的;

有一些大家周知的函数都是不可重入的, 比如 `malloc()`, `printf()`;

在写信号处理函数的时候, 要注意的事项:

a) 在信号处理函数中, 尽量使用简单的语句做简单的事情, 尽量不要调用系统函数以免引起麻烦; 老师认为这最好;

b) 如果必须要在信号处理函数中调用一些系统函数, 那么要保证在信号处理函数中调用的系统函数一定要是可重入的;

_exit & _Exit & exit*	abort*	accept	access	alo_error	alo_return
alo_suspend	alarm	bind	cfgetispeed	cfgetispeed	cfsetispeed
cfsetospeed	chdir	chmod	chown	clock_gettime	close
connect	creat	dup	dup2	execl	execve
fchmod	fchown	fcntl	fdatasync	fork	fpathconf
fstat	fsync	ftruncate	getgid	geteuid	getgid
getgroups	getpeername	getpgid	getpid	getppid	getsockname
getsockopt	getuid	kill	link	listen	longjmp*
lseek	lstat	mknod	mknod	open	pathconf
pause	pipe	poll	posix_trace_event	pselect	raise
read	readlink	recv	recvfrom	recvmsg	rename
rmdir	select	sem_post	send	sendmsg	sendto
setgid	setpgid	setsid	setsockopt	setuid	shutdown
sigaction	sigaddset	sigdelset	sigemptyset	sigfillset	sigismember
signal*	sigpause	sigpending	sigprocmask	sigqueue	sigset
sigsuspend	sleep	socket	socketpair	stat	symlink
sysconf	tcdrain	tcflow	tcflush	tcgetattr	tcgetpgrp
tcsetbreak	tcsetattr	tcsetpgrp	time	timer_getoverrun	timer_gettime
timer_settime	times	umask	uname	unlink	utime
wait	waitpid	write			

PS: 以上函数的可重入性也不是绝对的 (比如read, recv, ... 都会修改errno)

c)如果必须要在信号处理函数中调用那些可能修改 `errno` 值的可重入的系统函数，那么就事先备份 `errno` 值，

从信号处理函数返回之前，将 `errno` 值恢复；

(2.2)不可重入函数的错用演示

一旦在信号处理函数中用了不可重入函数，可能导致程序错乱，不正常.....

`signal` 因为兼容性，可靠性等等一些历史问题；不建议使用(我们的策略，坚决不用)，建议用 `sigaction()`函数代替；

高手：我们摸不清楚（有可能有坑）的东西（地方）我们主动回避，不去踩；

3.5 信号编程进阶，`sigprocmask` 范例

一：信号集

一个进程，必须能够记住 这个进程 当前阻塞了哪些信号

0000000000000000000000000000

我们需要 “信号集 ” 的这么一种数据类型(结构)，能够把这 60 多个信号都表示下（都装下）。

0000000000,0000000000,0000000000,00,0000000000,0000000000,0000000000,00

(64 个二进制位)

linux 是用 `sigset_t` 结构类型来表示信号集的；

```
typedef struct{
```

```
    unsigned long sig[2];
```

```
}sigset_t
```

信号集的定义：信号集表示一组信号的来（1）或者没来（0）

信号集相关的数据类型： `sigset_t`;

二：信号相关函数

a)`sigemptyset()`:把信号集中的所有信号都清 0，表示这 60 多个信号都没有来；

0000000000000000000000000000.....

b)`sigfillset()`;把信号集中的所有信号都设置为 1，跟 `sigemptyset()`正好相反；

1111111111111111111111111111.....

c)用 `sigaddset()`,`sigdelset()`就可以往信号集中增加信号，或者从信号集中删除特定信号；

d)`sigprocmask`,`sigmember`

一个进程，里边会有一个信号集，用来记录当前屏蔽（阻塞）了哪些信号；

如果我们把这个信号集中的某个信号位设置为 1，就表示屏蔽了同类信号，此时再来个同类信号，那么同类信号会被屏蔽，不能传递给进程；

如果这个信号集中有很多个信号位都被设置为 1，那么所有这些被设置为 1 的信号都是属于当前被阻塞的而不能传递到该进程的信号；

`sigprocmask()`函数，就能够设置该进程所对应的信号集中的内容；

三: sigprocmask 等信号函数范例演示

sleep()函数能够被打断:

(1)时间到达了;

(2)来了某个信号, 使 sleep()提前结束, 此时 sleep 会返回一个值, 这个值就是未睡够的时间;

```
kdvictor@ubuntu:/mnt/hgfs/nginx$ ./nginx
我要开始休息10秒了-----begin--, 此时我无法接收SIGQUIT信号!
^\\^\\^\\^\\我已经休息了10秒了-----end----!
SIGQUIT信号被屏蔽了!
SIGHUP信号没有被屏蔽!!!!!!
收到了SIGQUIT信号!
sigprocmask(SIG_SETMASK)成功!
SIGQUIT信号没有被屏蔽, 您可以发送SIGQUIT信号了, 我要sleep(10)秒钟!!!!!!
^\\收到了SIGQUIT信号!
sleep还没睡够, 剩余8秒
再见了!
kdvictor@ubuntu:/mnt/hgfs/nginx$
```

sigaction()函数;

3.6 fork 函数

一: fork()函数简单认识

创建进程;

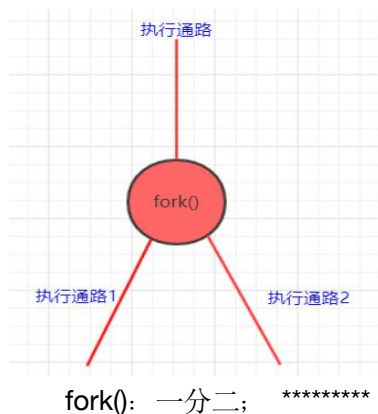
进程的概念: 一个可执行程序, 执行起来就是一个进程, 再执行起来一次, 它就又是一个进程 (多个进程可以共享同一个可执行文件)

文雅说法: 进程 定义为程序执行的一个实例;

在一个进程 (程序) 中, 可以用 fork()创建一个子进程, 当该子进程创建时, 它从 fork()指令的下一条(或者说从 fork()的返回处)开始执行与父进程相同的代码;

a)说白了: fork()函数产生了一个和当前进程完全一样的新进程, 并和当前进程一样从 fork()函数里返回;

原来一条执行通路 (父进程), 现在变成两条 (父进程+子进程)



(1.1) fork()函数简单范例

```
ps -eo pid,ppid,sid,ttty,pgrp,comm,stat | grep -E 'bash|PID|nginx'
```

fork()之后，是父进程 fork()之后的代码先执行还是子进程 fork()之后的代码先执行是不一定的；这个跟内核调度算法有关；

kill 子进程，观察父进程收到什么信号:SIGCHLD 信号，子进程变成了僵尸进程 Z

```
sleep 1 second, pid=1267!
sleep 1 second, pid=1267!
sleep 1 second, pid=1267!
sleep 1 second, pid=1267!
sleep 1 second, pid=1267!
sleep 1 second, pid=1267!
sleep 1 second, pid=1267!
[]

kdvictor@ubuntu:~$
kdvictor@ubuntu:~$ sudo strace -e trace=signal -p 1267
[sudo] password for kdvictor:
strace: Process 1267 attached
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_KILLED, si_pid=1268, si_uid=1000, si_status=SIGHUP, si_utime=0, si_stime=1} ---
--- SIGWINCH {si_signo=SIGWINCH, si_code=SI_KERNEL} ---
[]

kdvictor@ubuntu:~$
kdvictor@ubuntu:~$ ps -eo pid,ppid,sid,ttty,pgrp,comm,stat | grep -E 'bash|PID|demo'
  PID  PPID  SID TT      PGRP COMMAND  STAT
  1145  1144  1145 pts/0    1145  bash     Ss
  1238  1237  1238 pts/1    1238  bash     Ss
  1267  1145  1145 pts/0    1267  demo     S+
  1268  1267  1145 pts/0    1267  demo     S+
kdvictor@ubuntu:~$ kill -1 1268
kdvictor@ubuntu:~$ ps -eo pid,ppid,sid,ttty,pgrp,comm,stat | grep -E 'bash|PID|demo'
  PID  PPID  SID TT      PGRP COMMAND  STAT
  1145  1144  1145 pts/0    1145  bash     Ss
  1238  1237  1238 pts/1    1238  bash     Ss
  1267  1145  1145 pts/0    1267  demo     S+
  1268  1267  1145 pts/0    1267  demo <defunct> Z+
  1290  1289  1290 pts/2    1290  bash     Ss
kdvictor@ubuntu:~$
```

(1.2) 僵尸进程的产生、解决，SIGCHLD

僵尸进程的产生：在 Unix 系统中，一个子进程结束了，但是他的父进程还活着，
但该父进程没有调用(wait/waitpid)函数来进行额外的处置，那么这个子进程就会变成一个僵尸进程；

僵尸进程：已经被终止，不干活了，但是依旧没有被内核丢弃掉，因为内核认为父进程可能还需要该子进程的一些信息；

作为开发者，坚决不允许僵尸进程的存在；

如何干掉僵尸进程：

a)重启电脑

b)手工的把僵尸进程的父进程 kill 掉，僵尸进程就会自动消失；

SIGCHLD 信号：一个进程被终止或者停止时，这个信号会被发送给父进程；

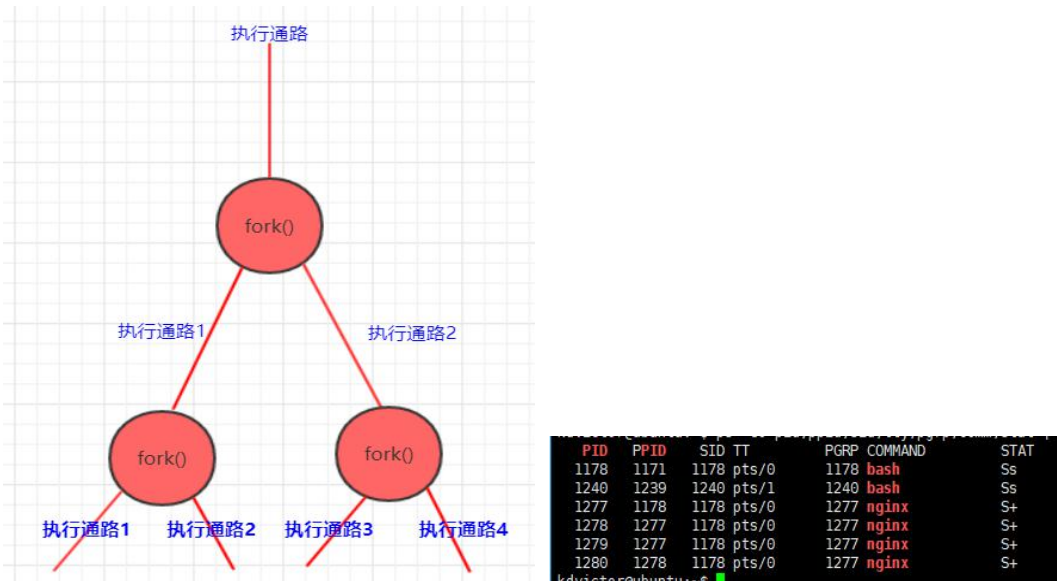
所以，对于源码中有 fork()行为的进程，我们 应该拦截并处理 SIGCHLD 信号；

waitpid();

二：fork()函数进一步认识

b)fork()产生新进程的速度非常快，fork()产生的新进程并不复制原进程的内存空间，而是和原进程（父进程）一起共享一个内存空间，但这个内存空间的特性是“写时复制”，也就是说：原来的进程和 fork()出来的子进程可以同时、自由的读取内存，但如果子进程（父进

程) 对内存进行修改的话, 那么这个内存就会复制一份给该进程单独使用, 以免影响到共享这个内存空间的其他进程使用;
两个 `fork()`



三: 完善一下 `fork()` 代码

`fork()` 会返回两次: 父进程中返回一次, 子进程中返回一次, 而且, `fork()` 在父进程中返回的值和在子进程中返回的值是不同的

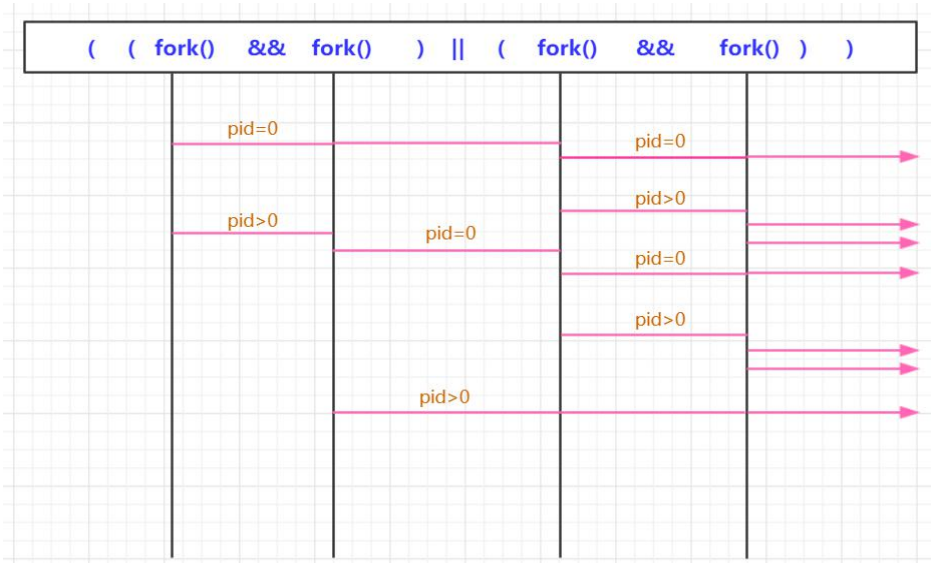
子进程的 `fork()` 返回值是 0;

父进程的 `fork()` 返回值是新建立的子进程的 ID, 因为全局量 `g_mygbttest` 的值发生改变, 导致主, 子进程内存被单独的分开, 所以每个的 `g_mygbttest` 值也不同;

(3.1) 一个和 `fork()` 执行有关的逻辑判断 (短路求值)

`||` 或: 有 1 出 1, 全 0 出 0;

`&&` 与: 全 1 出 1, 有 0 出 0;



```

root@ubuntu:~# ps -eo pid,ppid,sid,ty,pgrp,comm,stat | grep

```

PID	PPID	SID	TT	PGRP	COMMAND	STAT
1178	1171	1178	pts/0	1178	bash	Ss
1240	1239	1240	pts/1	1240	bash	Ss
1312	1178	1178	pts/0	1312	nginx	S+
1313	1312	1178	pts/0	1312	nginx	S+
1314	1312	1178	pts/0	1312	nginx	S+
1315	1313	1178	pts/0	1312	nginx	S+
1316	1313	1178	pts/0	1312	nginx	S+
1317	1314	1178	pts/0	1312	nginx	S+
1318	1314	1178	pts/0	1312	nginx	S+

四: `fork()`失败的可能性

a)系统中进程太多

缺省情况, 最大的 pid: 32767

b)每个用户有个允许开启的进程总数:

7788

3.7 守护进程

一: 普通进程运行观察

```
ps -eo pid,ppid,sid,ty,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)

```

huan@ubuntu:~$ ps -efj
UID      PID  PPID  PGID   SID  C STIME TTY      TIME CMD
root      1    0     1     1  0  14:15 ?        00:00:02 /sbin/init noprompt
root      2    0     0     0  0  14:15 ?        00:00:00 [kthreadd]
root      3    2     0     0  0  14:15 ?        00:00:00 [ksoftirqd/0]
root      4    2     0     0  0  14:15 ?        00:00:00 [kworker/0:0]
root      5    2     0     0  0  14:15 ?        00:00:00 [kworker/0:0H]
root      7    2     0     0  0  14:15 ?        00:00:00 [rcu_sched]
root      8    2     0     0  0  14:15 ?        00:00:00 [rcu_bh]
root      9    2     0     0  0  14:15 ?        00:00:00 [migration/0]
root     10    2     0     0  0  14:15 ?        00:00:00 [watchdog/0]
root     11    2     0     0  0  14:15 ?        00:00:00 [watchdog/1]
root     12    2     0     0  0  14:15 ?        00:00:00 [migration/1]
root     13    2     0     0  0  14:15 ?        00:00:00 [ksoftirqd/1]
root     15    2     0     0  0  14:15 ?        00:00:00 [kworker/1:0H]
root     16    2     0     0  0  14:15 ?        00:00:00 [watchdog/2]
root     17    2     0     0  0  14:15 ?        00:00:00 [migration/2]
root     18    2     0     0  0  14:15 ?        00:00:00 [ksoftirqd/2]
root     19    2     0     0  0  14:15 ?        00:00:00 [kworker/2:0]
root     20    2     0     0  0  14:15 ?        00:00:00 [kworker/2:0H]
root     21    2     0     0  0  14:15 ?        00:00:00 [watchdog/3]
root     22    2     0     0  0  14:15 ?        00:00:00 [migration/3]
root     23    2     0     0  0  14:15 ?        00:00:00 [ksoftirqd/3]
root     25    2     0     0  0  14:15 ?        00:00:00 [kworker/3:0H]
root     26    2     0     0  0  14:15 ?        00:00:00 [kdevtmpfs]
root     27    2     0     0  0  14:15 ?        00:00:00 [netns]
root     28    2     0     0  0  14:15 ?        00:00:00 [perf]
root     29    2     0     0  0  14:15 ?        00:00:00 [khungtaskd]
root     30    2     0     0  0  14:15 ?        00:00:00 [writeback]

```

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`

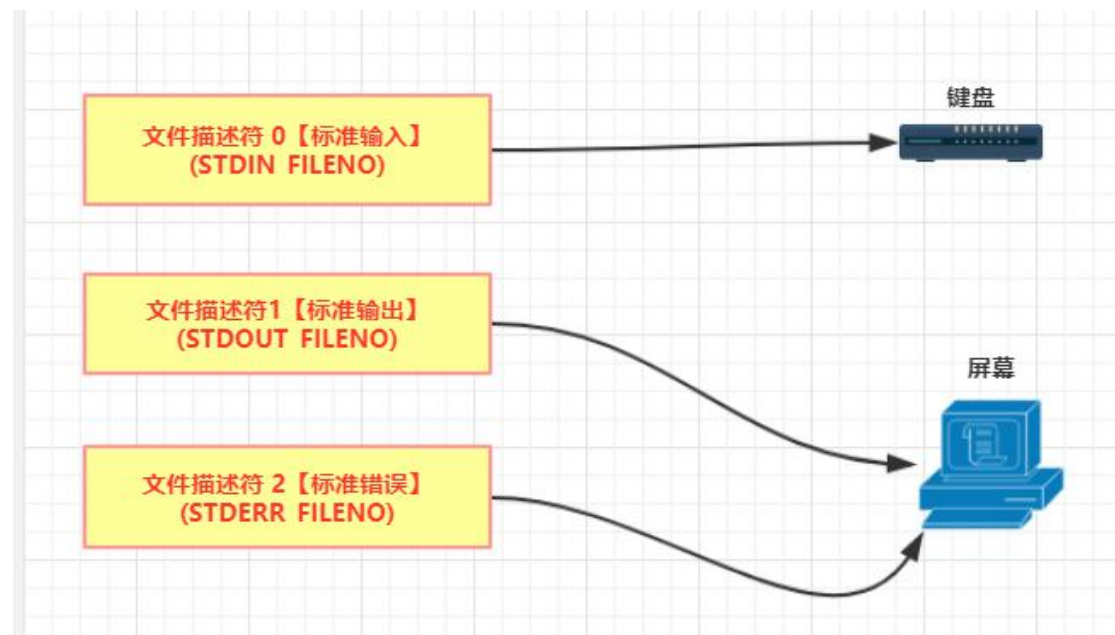
2:标准错误 【屏幕】, 对应的符号常量叫 `STDERR_FILENO`

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。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);

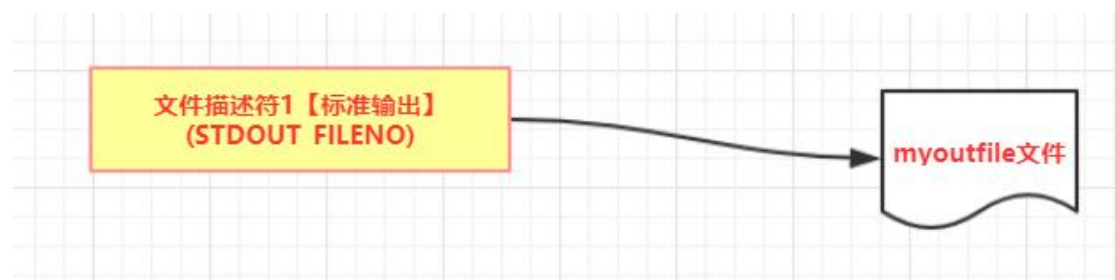
文件描述符虽然是数字，但是，如果我们把文件描述符直接理解成指针（指针里边保存的是地址——地址说白了也是个数字）；



```
write(STDOUT_FILENO,"aaaabbb",6);
```

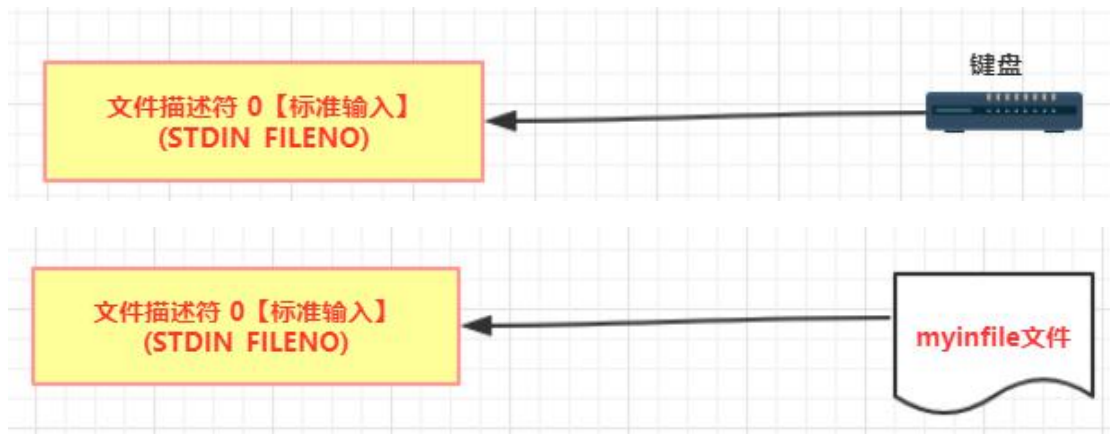
(3.2) 输入输出重定向

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；ls -la > myoutfile

输入重定向 < (vim myinfile, cat < myinfile)



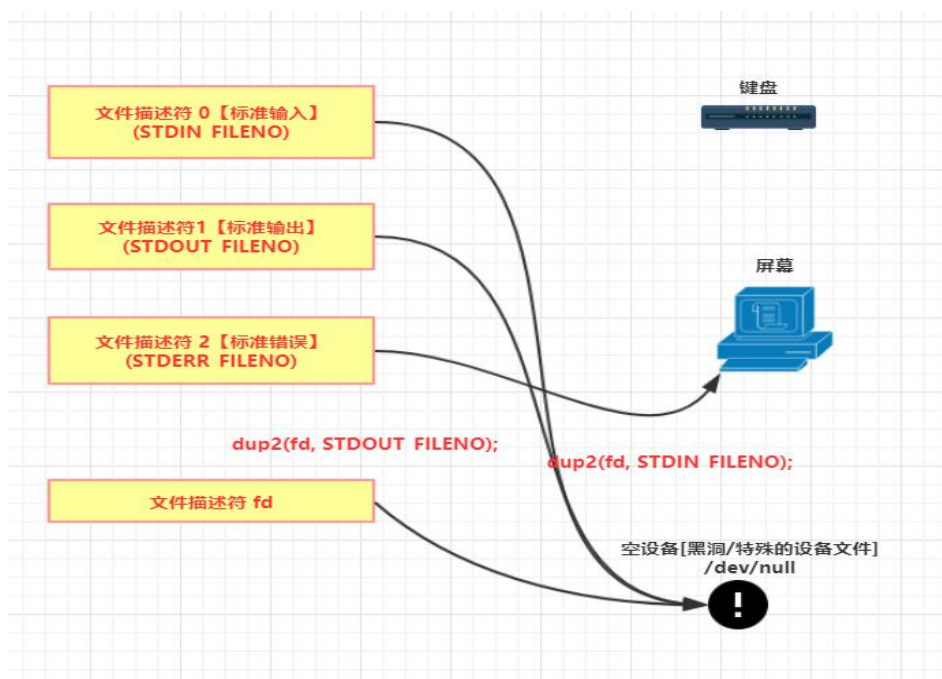
(3.3) 空设备（黑洞）

/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；

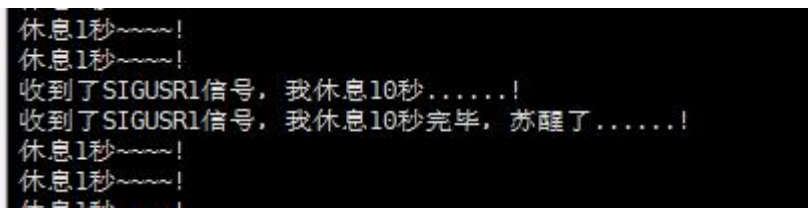
(3).....其他的，大家自己总结；

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,tt,pgrp,comm,stat,cmd | grep -E 'bash|PID|nginx'`

用 kill 发送 USR1 信号给进程



```
休息1秒~~~~!  
休息1秒~~~~!  
收到了SIGUSR1信号，我休息10秒.....!  
收到了SIGUSR1信号，我休息10秒完毕，苏醒了.....!  
休息1秒~~~~!  
休息1秒~~~~!  
休息1秒~~~~!
```

内核态与用户态切换

(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，

```
休息1秒~~~~~!
收到了SIGUSR1信号，我休息10秒.....!
收到了SIGUSR1信号，我休息10秒完毕，苏醒了.....!
收到了SIGUSR1信号，我休息10秒.....!
收到了SIGUSR1信号，我休息10秒完毕，苏醒了.....!
休息1秒~~~~~!
休息1秒~~~~~!
休息1秒~~~~~!
休息1秒~~~~~!
休息1秒~~~~~!
```

```
192.168.1.88 * +
usage: sudo -e [-AknS] [-r role] [-t type] [-C num]
kdvictor@ubuntu:~$ sudo -USR1 1288
sudo: the '-U' option may only be used with the '-l'
usage: sudo -h | -K | -k | -V
usage: sudo -v [-AknS] [-g group] [-h host] [-p prom]
usage: sudo -l [-AknS] [-g group] [-h host] [-p prom]
usage: sudo [-AbEHknPS] [-r role] [-t type] [-C num]
               [-il-s] [<command>]
usage: sudo -e [-AknS] [-r role] [-t type] [-C num]
kdvictor@ubuntu:~$ sudo kill -USR1 1288
[sudo] password for kdvictor:
kdvictor@ubuntu:~$ sudo kill -USR1 1288
kdvictor@ubuntu:~$ sudo kill -USR1 1288
kdvictor@ubuntu:~$ sudo kill -USR1 1288
kdvictor@ubuntu:~$ sudo kill -USR1 1288
kdvictor@ubuntu:~$ sudo kill -USR1 1288
```

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
```

```
休息1秒~~~~!  
收到了SIGUSR1信号，我休息10秒.....!  
收到了SIGUSR2信号，我休息10秒.....!  
收到了SIGUSR2信号，我休息10秒完毕，苏醒了.....!  
收到了SIGUSR1信号，我休息10秒完毕，苏醒了.....!  
休息1秒~~~~!  
休息1秒~~~~!  
休息1秒~~~~!  
休息1秒~~~~!  
休息1秒~~~~!  
休息1秒~~~~!  
休息1秒~~~~!  
  
  
kdvictor@ubuntu:~$ sudo kill -USR1 1288  
kdvictor@ubuntu:~$ sudo kill -USR2 1288  
kdvictor@ubuntu:~$ 
```

(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：不管是目录还是文件，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关的 1 到多个.c 文件，暂时为空

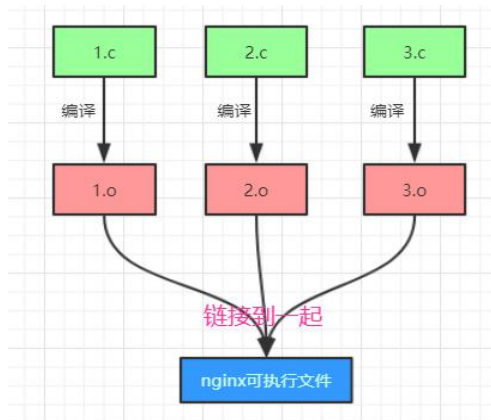
f)signal 目录：专门用于存放和信号处理 有关的 1 到多个.c 文件；

`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東西，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

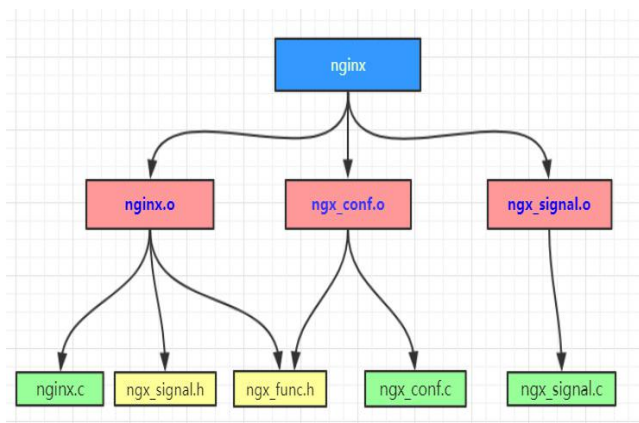
app/dep: 存放.d 开头的依赖关系文件;

(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的字母, 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;
`sudo apt-get install valgrind`

(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`

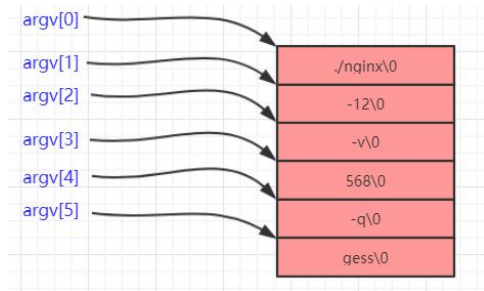
`argv[0] = ./nginx` ----指向的就是可执行程序名: `./nginx`

`argv[1] = -v`

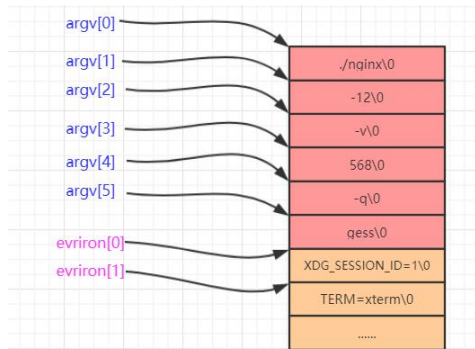
`argv[2] = -s`

`argv[3] = 5`

比如你输入 `./nginx -12 -v 568 -q gess`



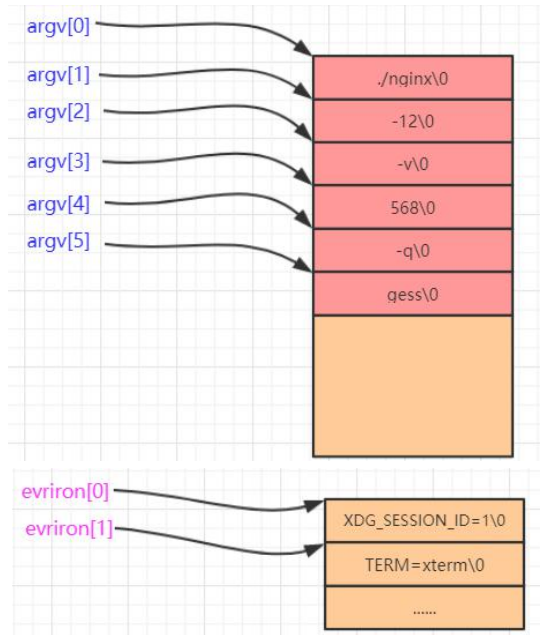
argv 内存之后，接着连续的就是环境变量参数信息内存【是咱们这个可执行程序运行时有关的所有环境变量参数信息】



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 tzselect sudo cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime

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

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, 才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 里底安的 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)`;

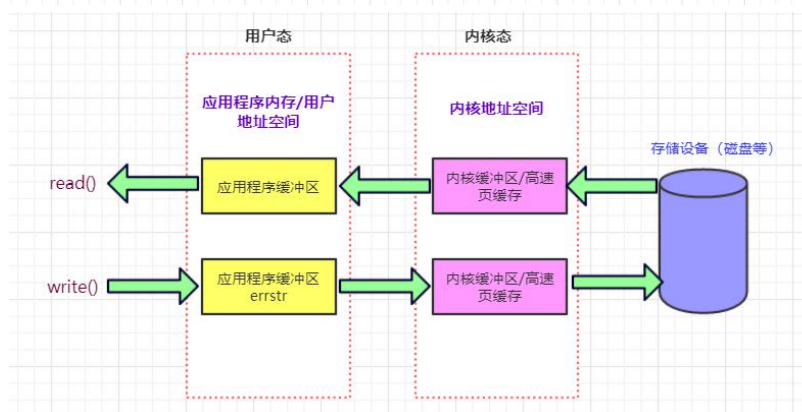
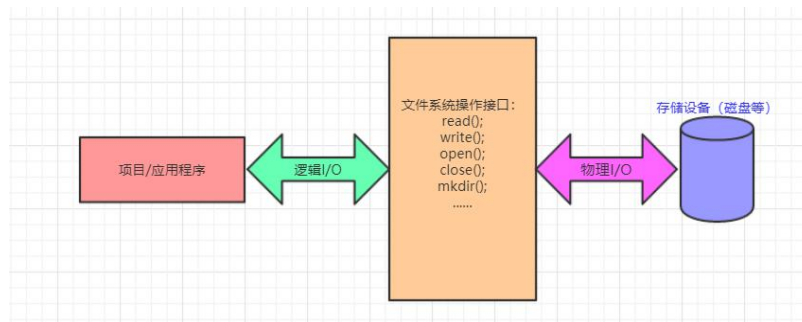
`O_APPEND` 这个标记能够保证多个进程操作同一个文件时不会相互覆盖;

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】；

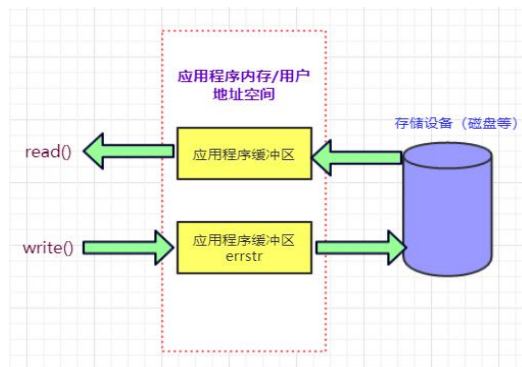


的确，因为 `write()`调用速度极快，可能没有时间完成该项目的工作【实际写磁盘】，所以这个 `wirte()`调用不等价于数据在内核缓冲区和磁盘之间的数据交换

f)打开文件使用了 `O_APPEND`，多个进程写日志用 `write()`来写；

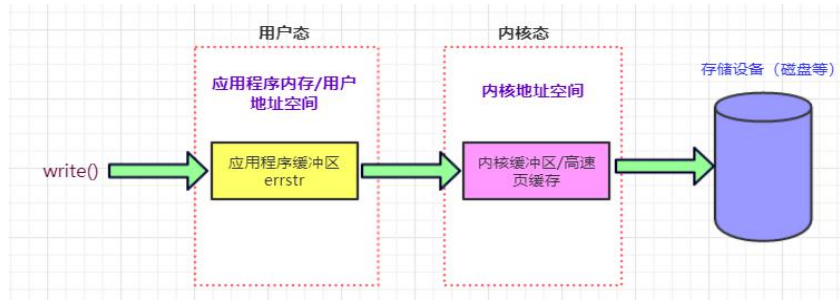
(4.1) 掉电导致 `write()`的数据丢失破解法

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`

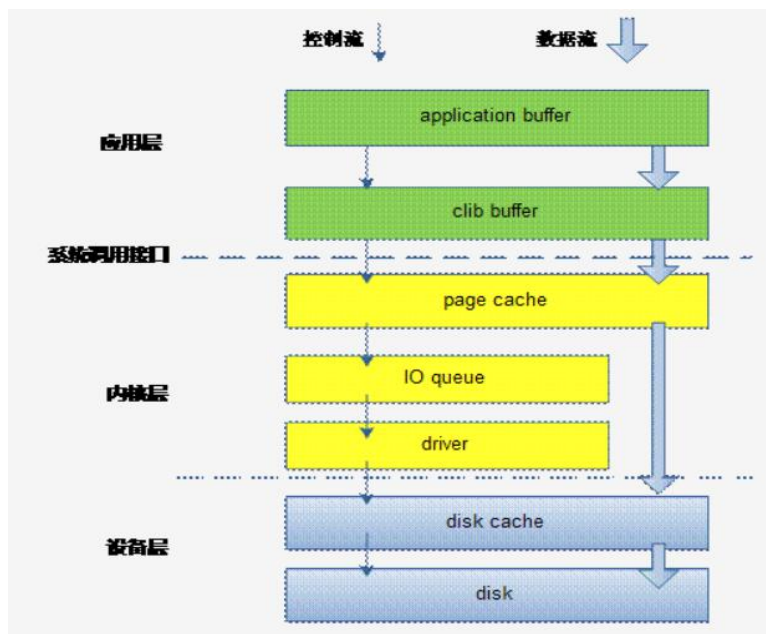
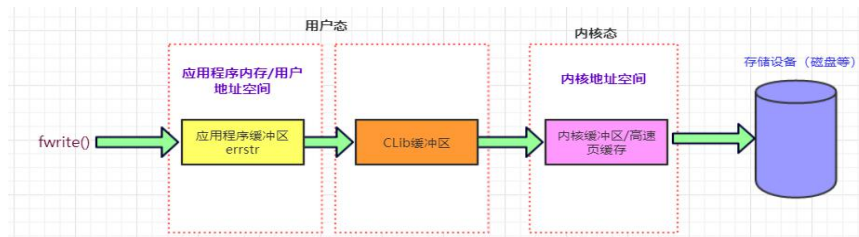
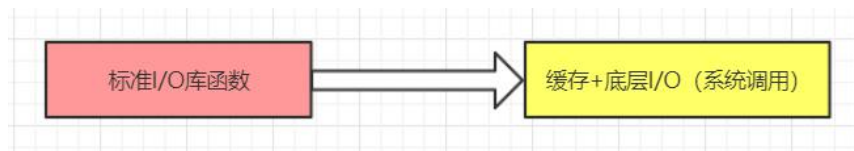
`printf,fprintf,sprintf`

`scanf,fscanf,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

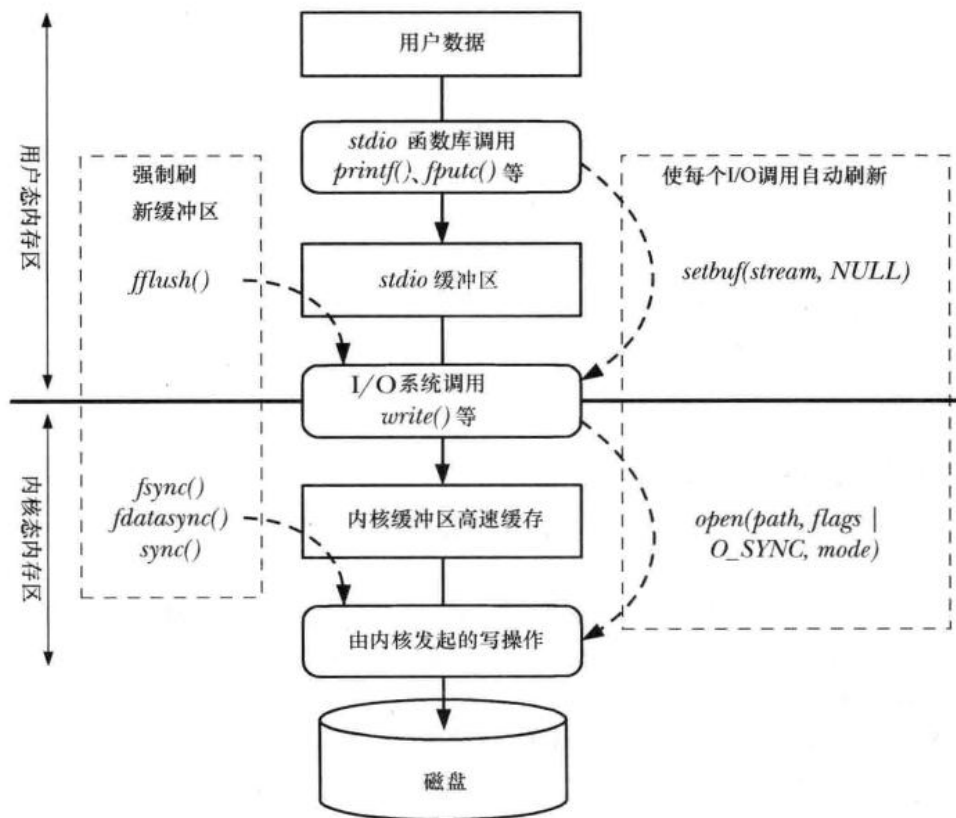


图 13-1: I/O 缓冲小结

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,tt,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 SIGCHLD 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

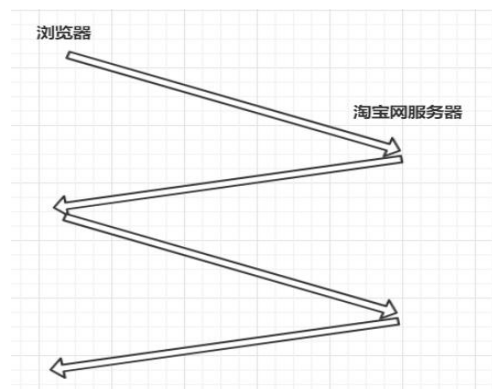
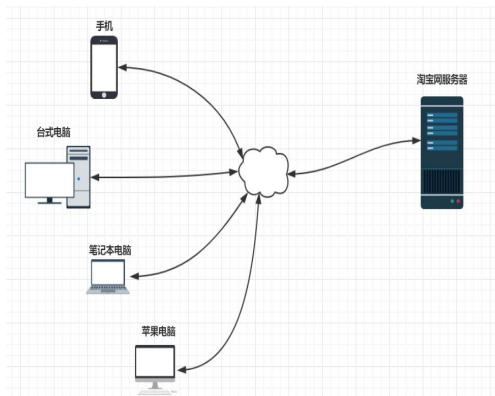
客户端：client，一般字母 c 表示

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



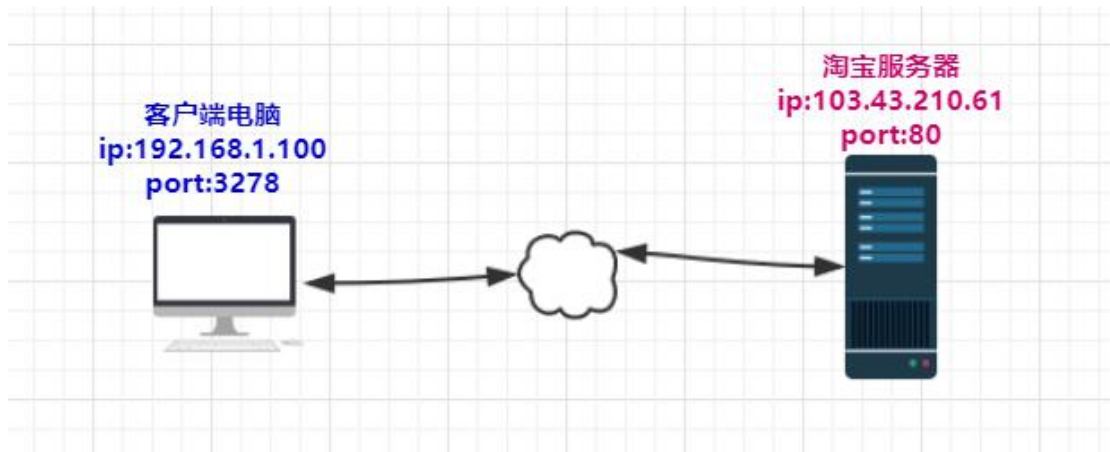
(1.2) 客户端服务器角色规律总结

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)总有一方先发起第一个数据包，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；

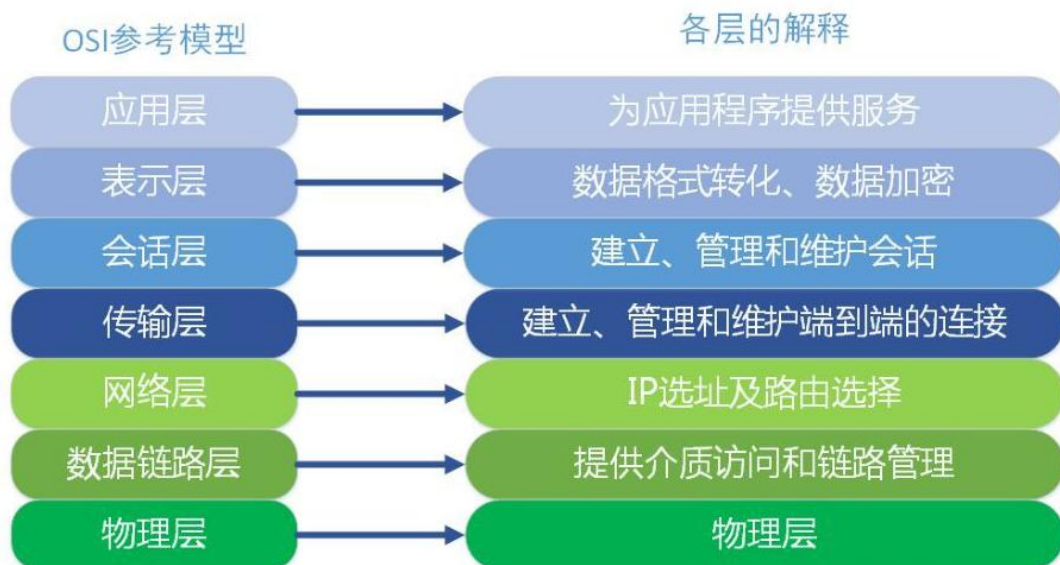
在编写网络通讯程序时，你只需要指定淘宝服务器的 ip 地址和淘宝服务器的端口号，就能够跟淘宝服务器进行通讯；

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】

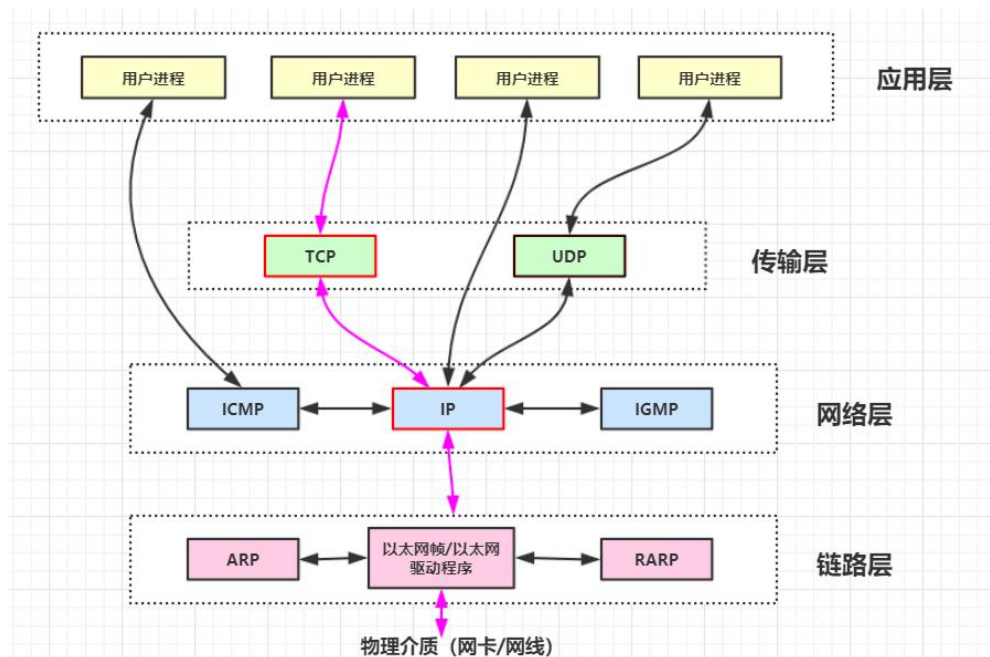
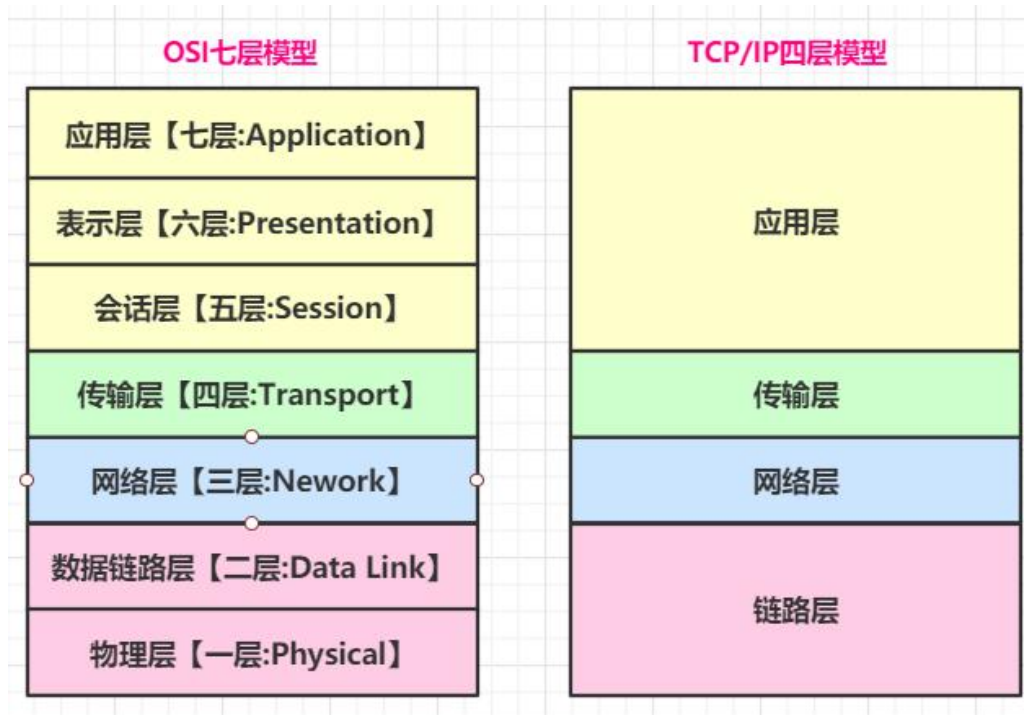


OSI(Open System Interconnect): 开放式系统互联；是 ISO(国际标准化组织) 在 1985 年研究的网络互联模型；

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

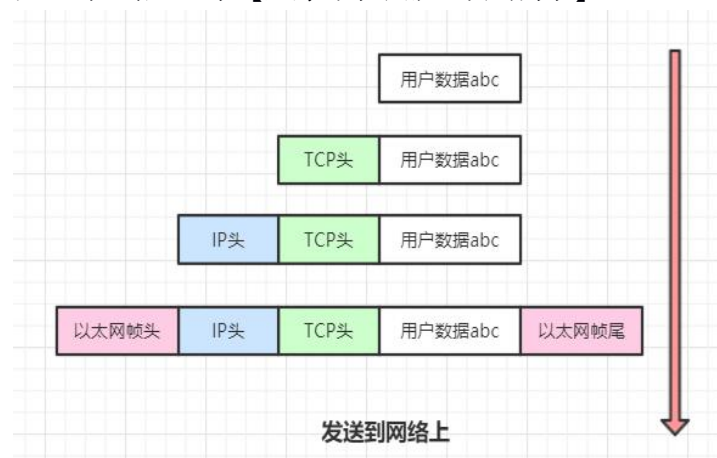
你要发送 abc 这三个字母出去到网络上；

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端程序实现代码

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[IPv4]：理解成现实社会中的居住地址

192.168.1.100[IPv4]：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

一个IPv6地址的字符表示：

ABCD:EF01:2345:6789:ABCD:EF01:2345:6789

带有子网前缀的IPv6地址表示：

ABCD:EF01:2345:6789:ABCD:EF01:2345:6789/64

带有端口的IPv6地址表示：

[ABCD:EF01:2345:6789:ABCD:EF01:2345:6789]:8080

IPv6地址表示法

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；

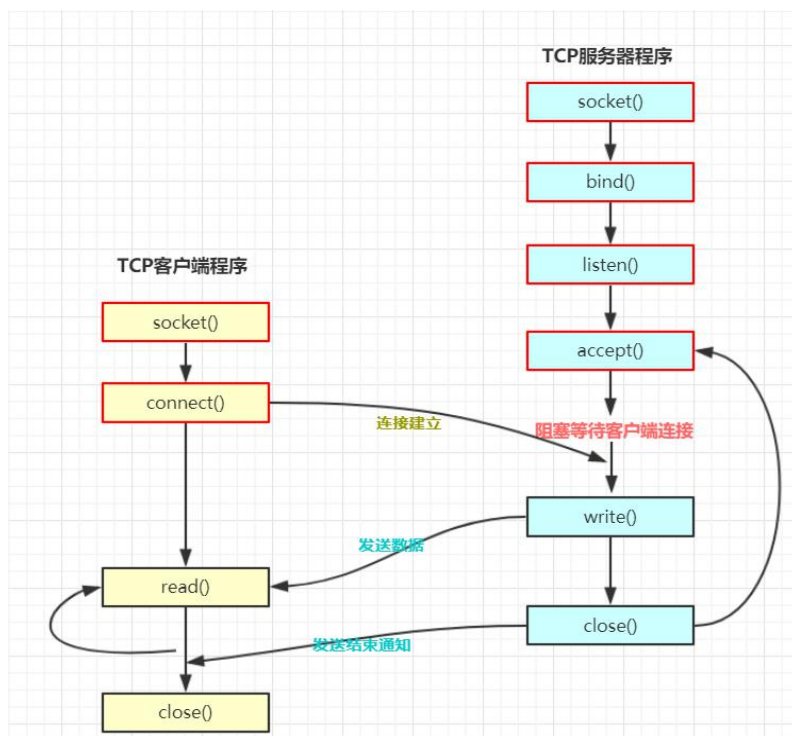
(3.4) 一个简单的客户端通讯程序范例

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；

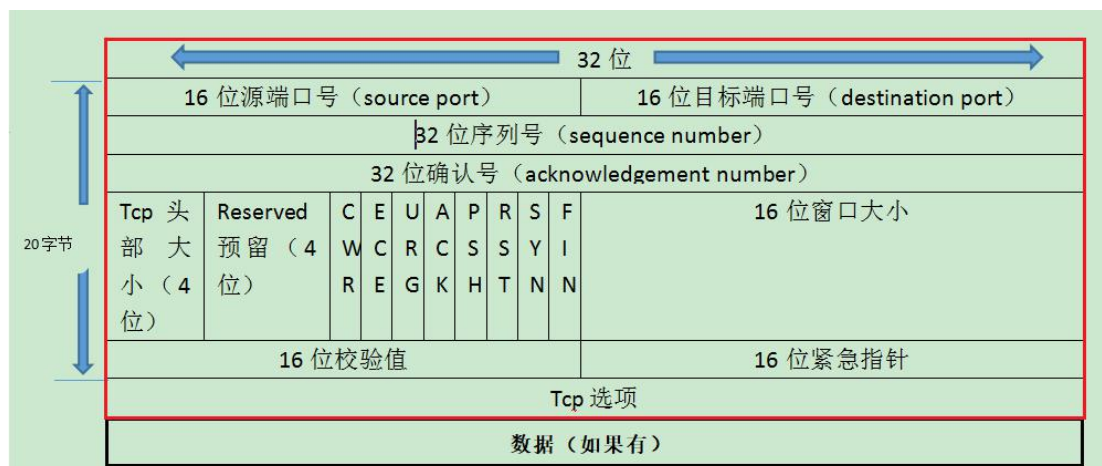
最终 TCP/IP 协议保证了我们收发数据的顺序性和可靠性;

(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

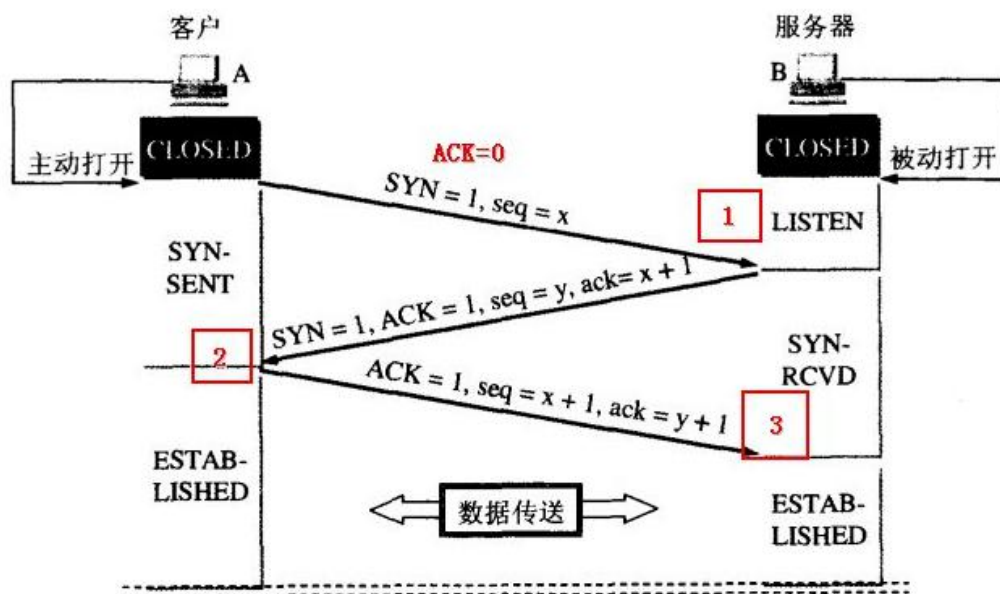
客户端理解成一个人, 服务器端理解成一个人, 两个人要用电话通话:

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.....



- 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
 - 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;
 - 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, 客户端和服务端端的 **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

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.1.119	192.168.1.88	TCP	66	9121 → 9000 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PER...
2	0.000208	192.168.1.88	192.168.1.119	TCP	66	9000 → 9121 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK...
3	0.000259	192.168.1.119	192.168.1.88	TCP	54	9121 → 9000 [ACK] Seq=1 Ack=1 Win=525568 Len=0
4	22.403235	192.168.1.119	192.168.1.88	TCP	55	9121 → 9000 [PSH, ACK] Seq=1 Ack=1 Win=525568 Len=1
5	22.403856	192.168.1.88	192.168.1.119	TCP	60	9000 → 9121 [ACK] Seq=1 Ack=2 Win=29312 Len=0
6	22.403978	192.168.1.88	192.168.1.119	TCP	75	9000 → 9121 [PSH, ACK] Seq=1 Ack=2 Win=29312 Len=21
7	22.404125	192.168.1.88	192.168.1.119	TCP	60	9000 → 9121 [FIN, ACK] Seq=22 Ack=2 Win=29312 Len=0
8	22.404151	192.168.1.119	192.168.1.88	TCP	54	9121 → 9000 [ACK] Seq=2 Ack=23 Win=525568 Len=0
9	22.406013	192.168.1.119	192.168.1.88	TCP	54	9121 → 9000 [FIN, ACK] Seq=2 Ack=23 Win=525568 Len=0
10	22.406274	192.168.1.88	192.168.1.119	TCP	60	9000 → 9121 [ACK] Seq=23 Ack=3 Win=29312 Len=0

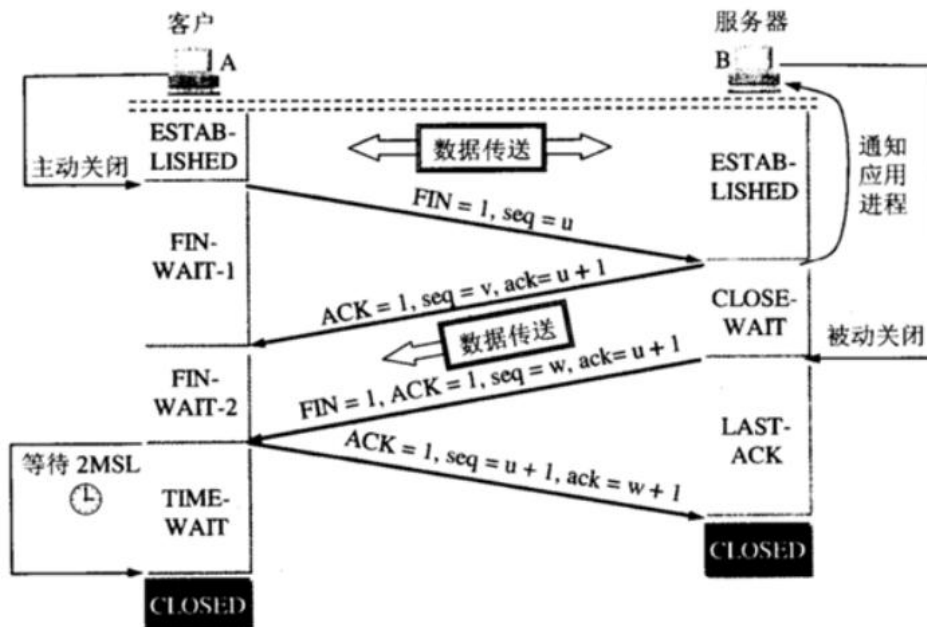
(3.1) TCP 断开的四次挥手

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】

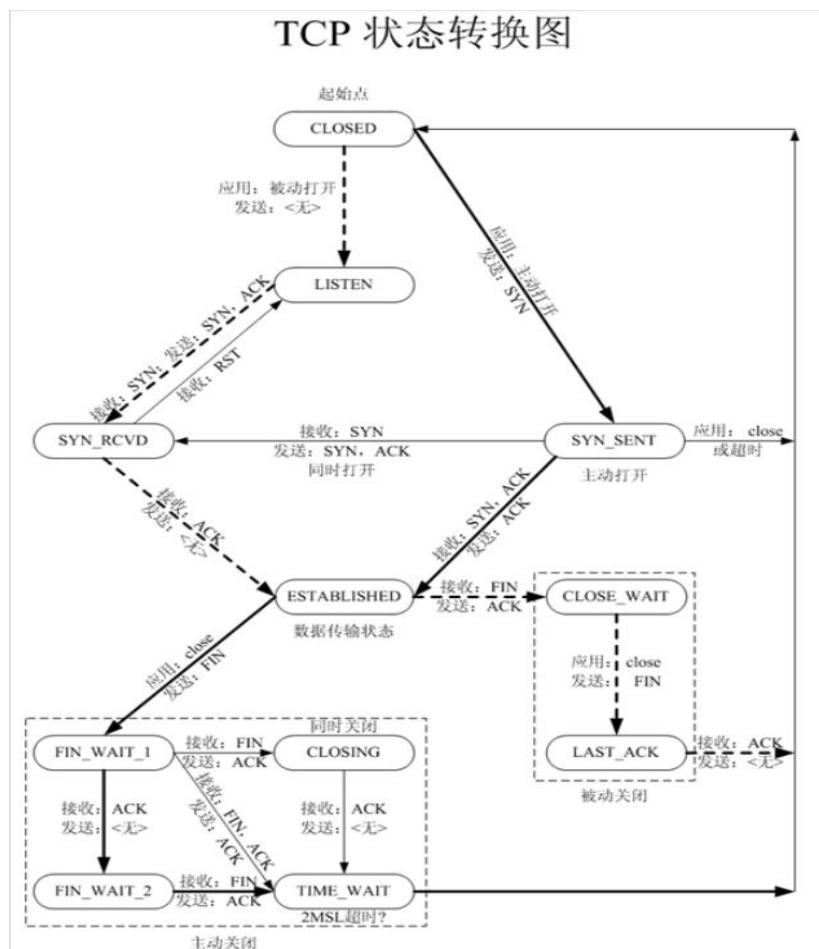
只要客户端 连接到服务器, 并且 服务器把客户端关闭, 那么服务器端就会产生一条针对 9000 监听端口的 状态为 TIME_WAIT 的连接;

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

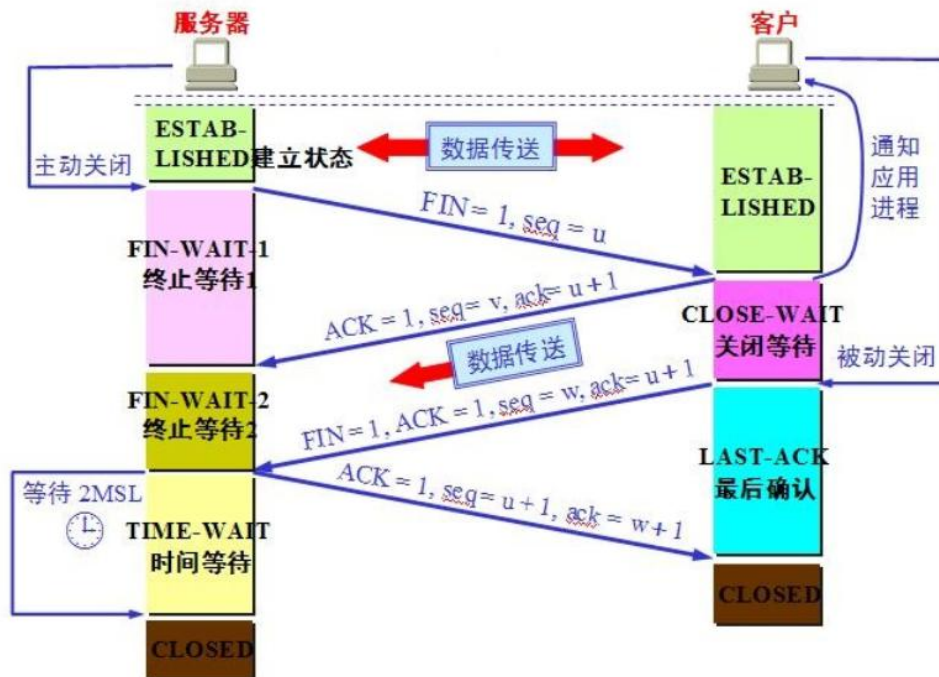
客户端被动关闭： ESTABLISHED->CLOSE_WAIT->LAST_ACK

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；

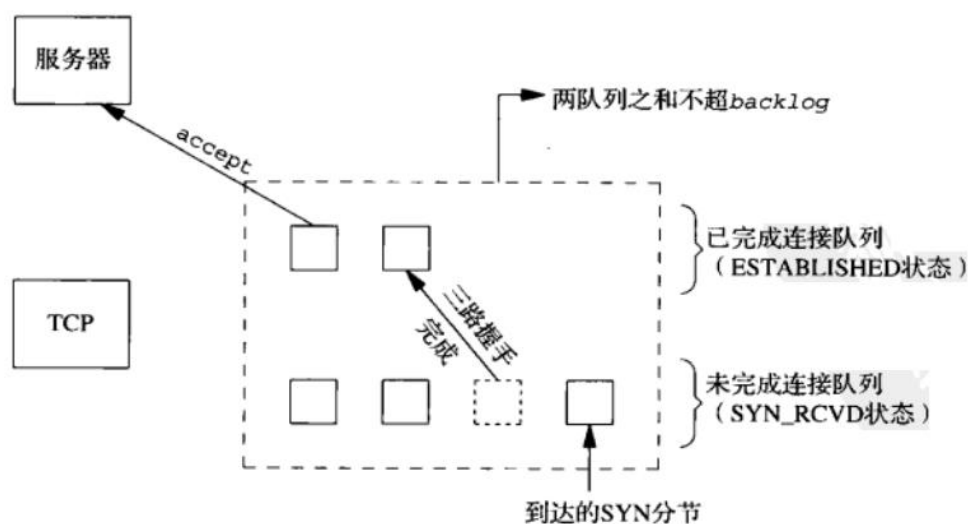
a)未完成连接队列 【保存连接用的】

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；

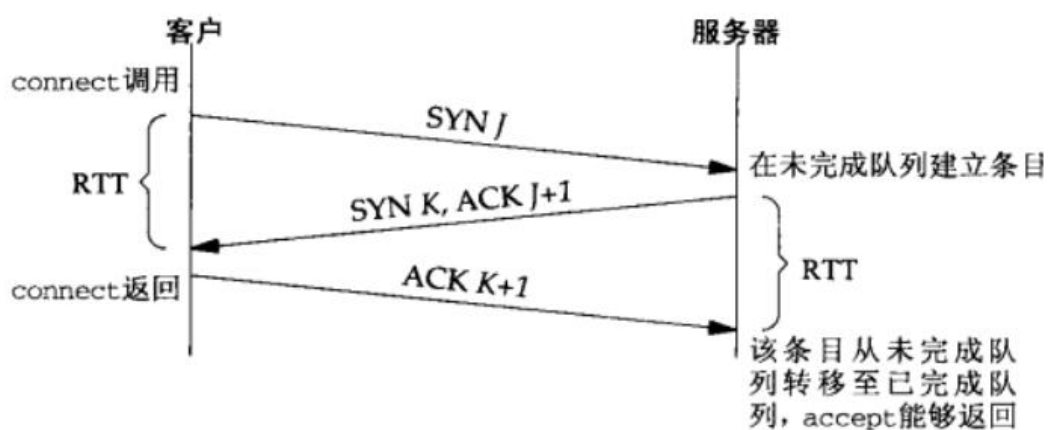
b)已完成连接队列 【保存连接用的】

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；

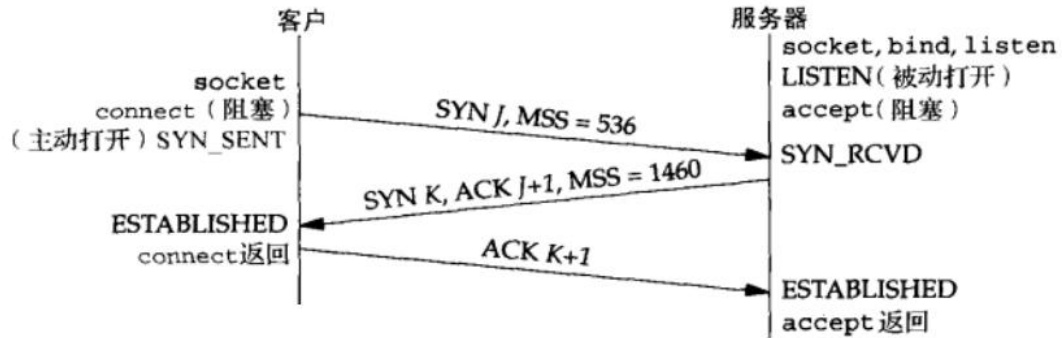


TCP为监听套接字维护的两个队列

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；



TCP三路握手和监听套接字的两个队列



(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端；

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户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】;

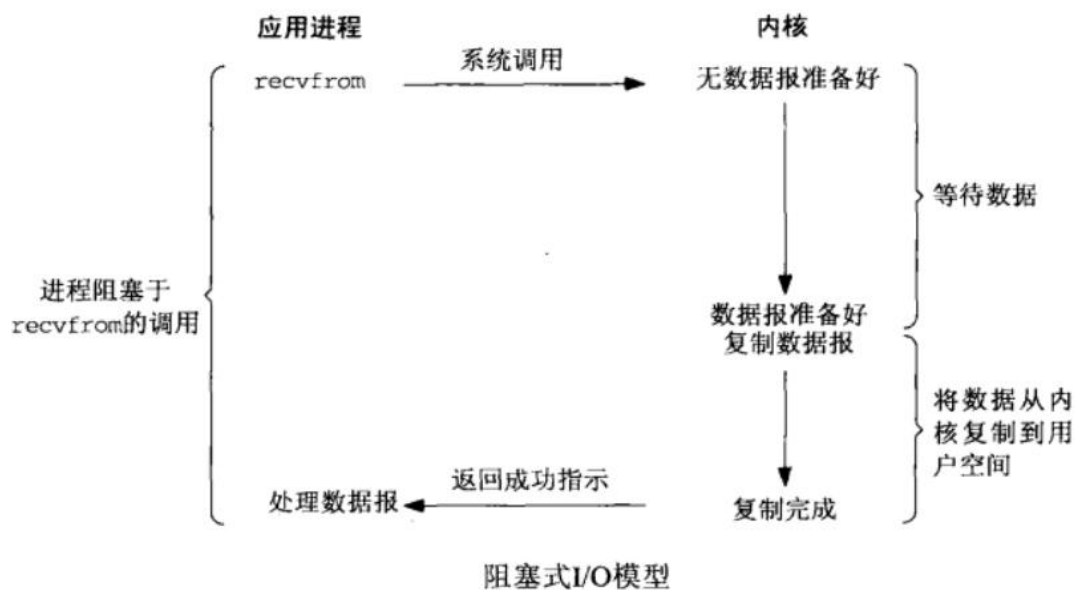
大家在写代码时尽快用 `accept()`把已完成队列里边的连接取走, 尽快 留出空闲为止给后续的已完成三路握手的条目用, 那么这个已完成队列一般不会满;

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

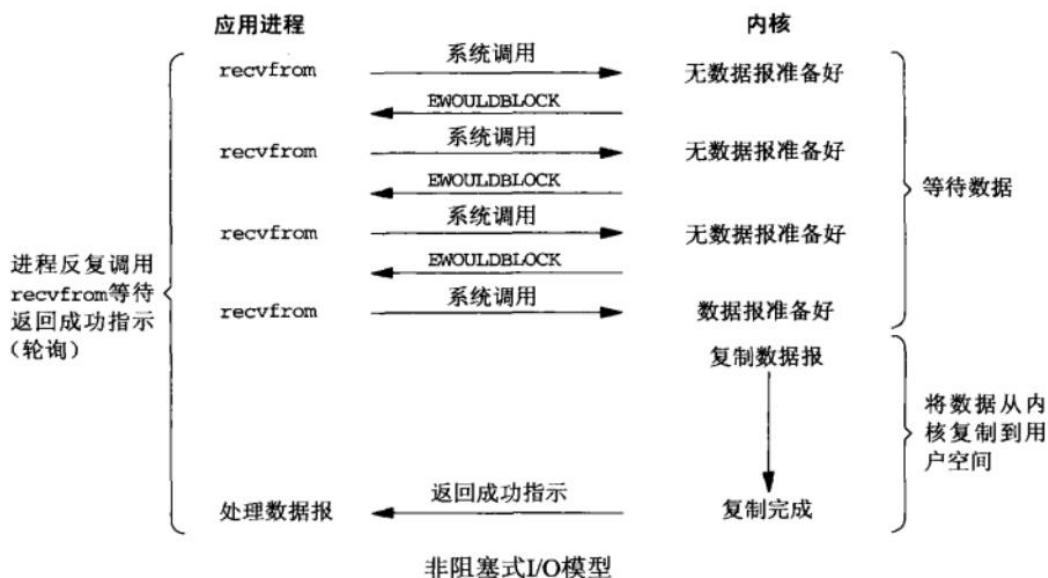


我调用一个函数, 这个函数就卡在在这里, 整个程序流程不往下走了【休眠 `sleep`】, 该函数卡在这里等待一个事情发生, 只有这个事情发生了, 这个函数才会往下走;

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;

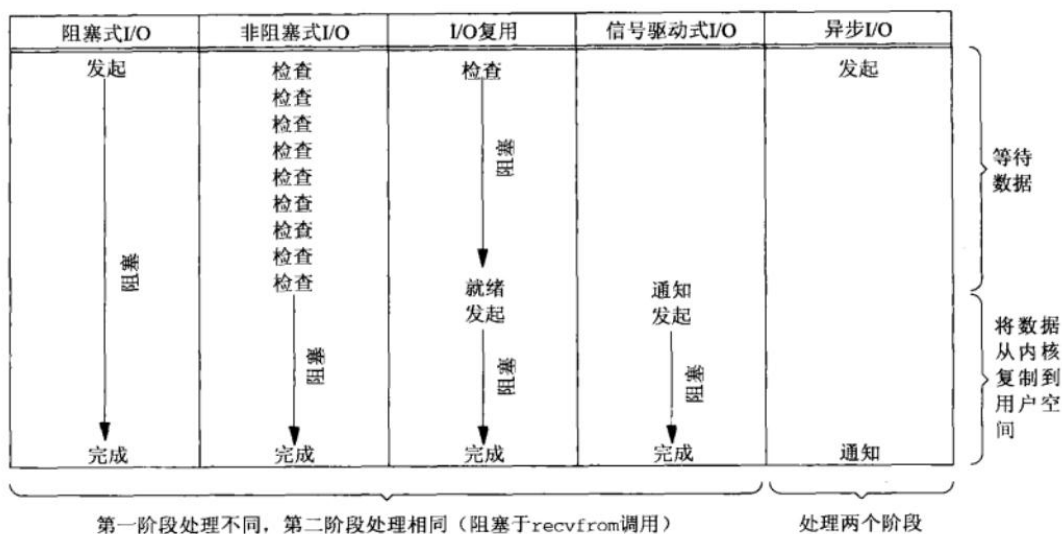


非阻塞模式的两个鲜明特点:

(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 `EWOULDBLOCK`,

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, 我们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;

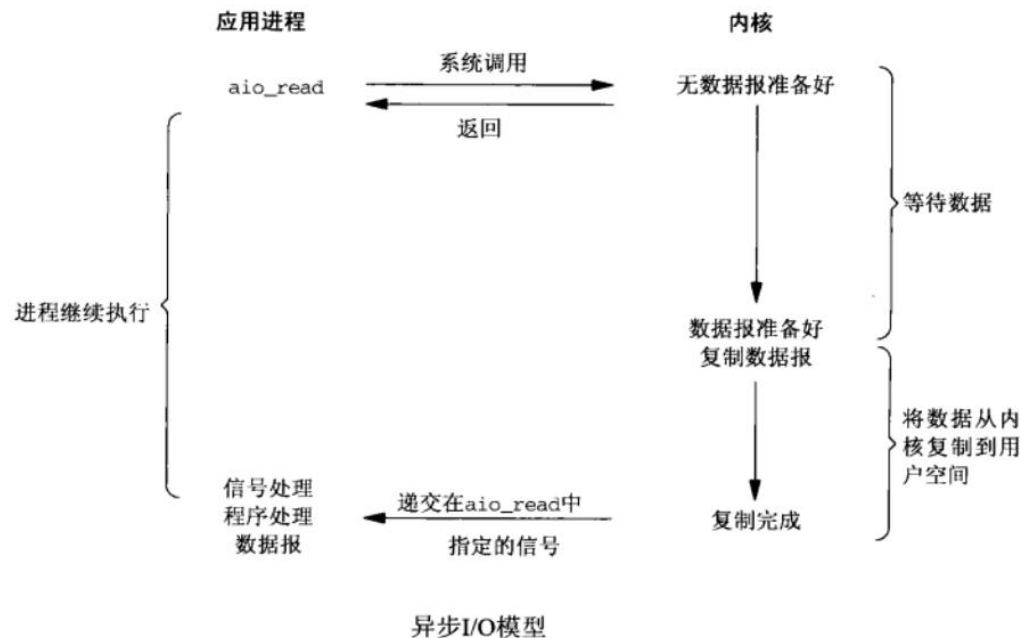
很容易区别非阻塞和异步 I/O 的差别:

(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；

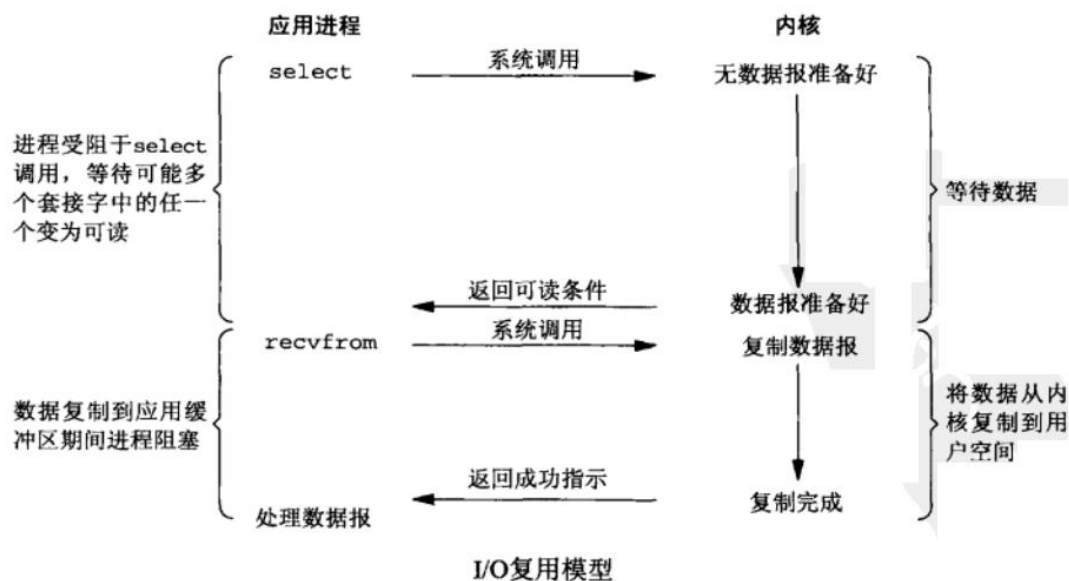
select()的能力是等多条 TCP 连接上的任意一条有数据来；，然后哪条 TCP 有数据来，我再用具体的比如 recvfrom()去收。

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,**kqueue**(freebsd)

epoll，从 linux 内核 2.6 引入的，2.6 之前是没有的；

(2)**epoll** 和 **kqueue** 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；

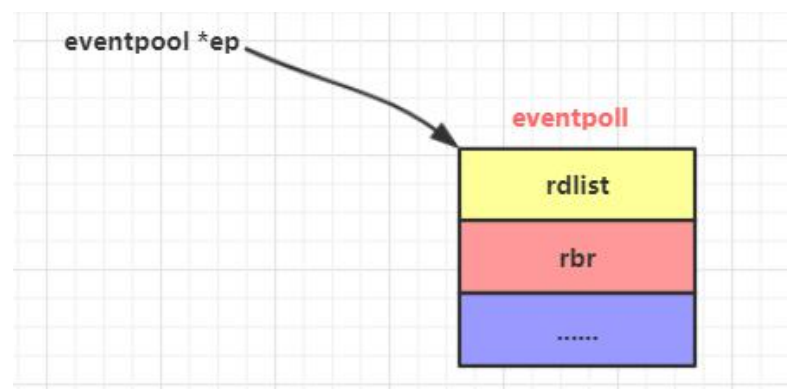
这个 **epoll** 对象最终要用 **close()**,因为文件描述符/句柄 总是关闭的；

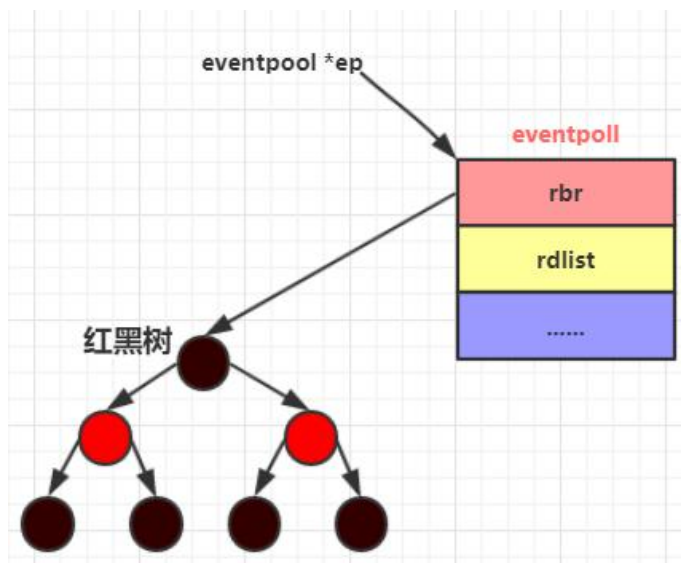
size: >0;

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；

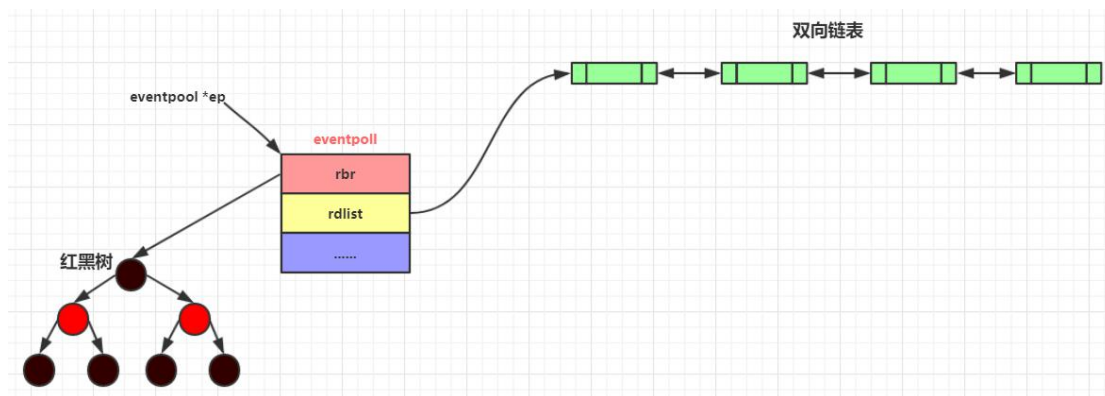




红黑树，用来保存 键【数字】/值【结构】，能够快速通过你给 key，把整个的键/值取出来；

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 efd,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，目的就是希望通过这个 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；

efd: 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；

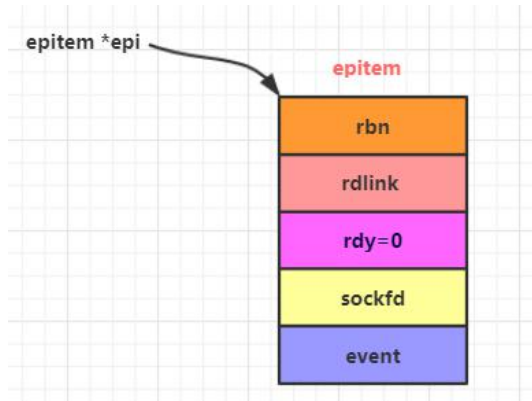
EPOLL_CTL_DEL：是从红黑树上把这个节点干掉；这会导致这个 **socket** 【这个 tcp 链接】上无法收到任何系统通知事件；

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；

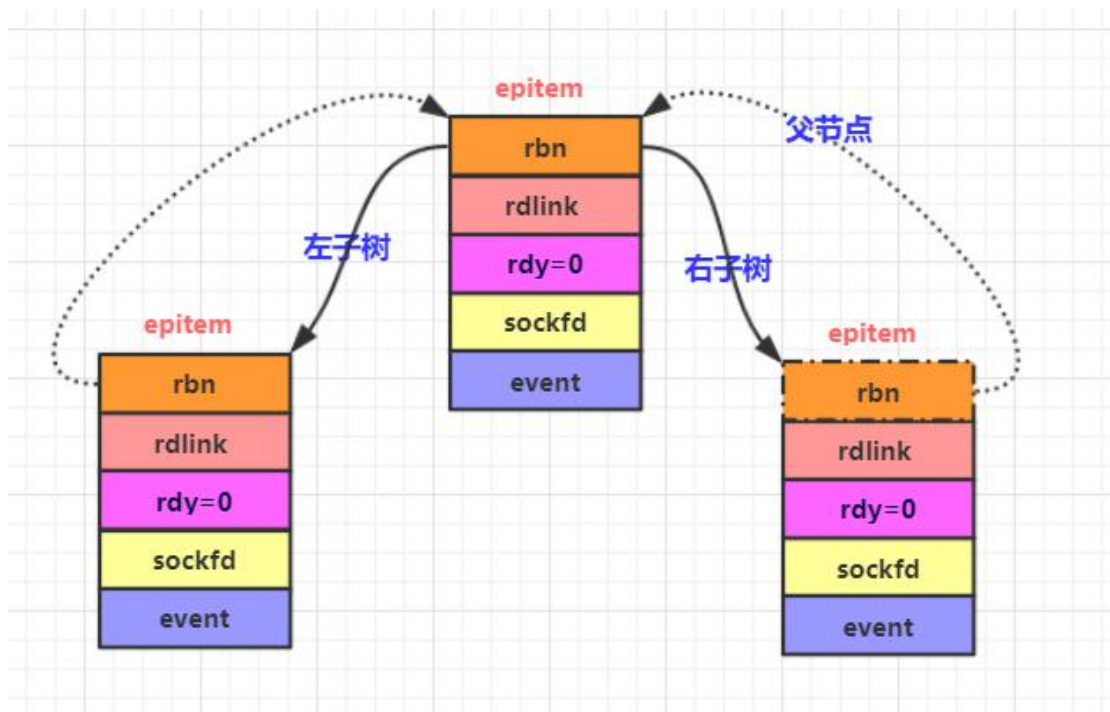
原理：

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】；

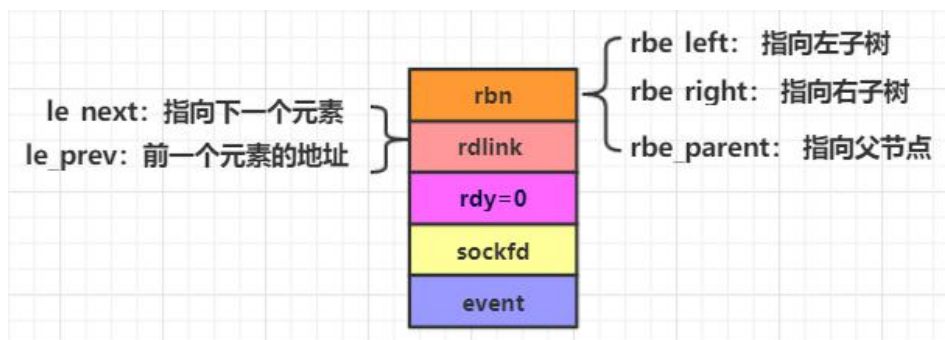
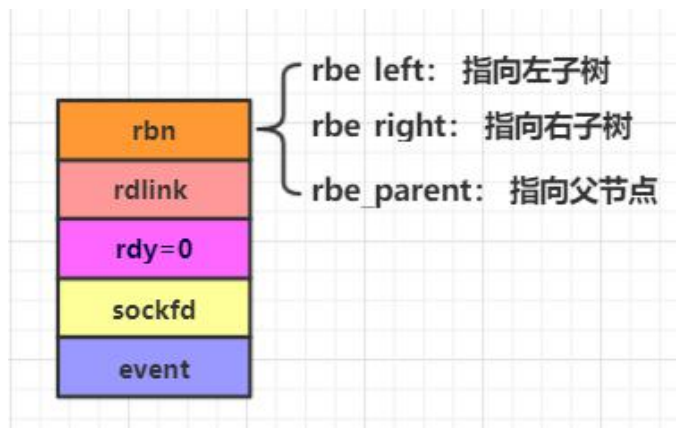
参数 `epfd`：是 `epoll_create()` 返回的 `epoll` 对象描述符；

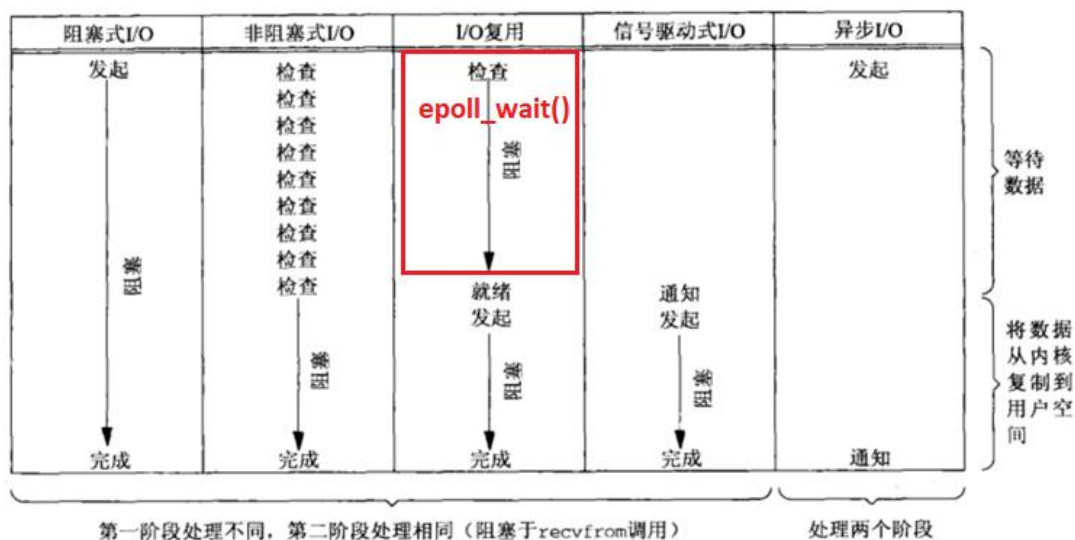
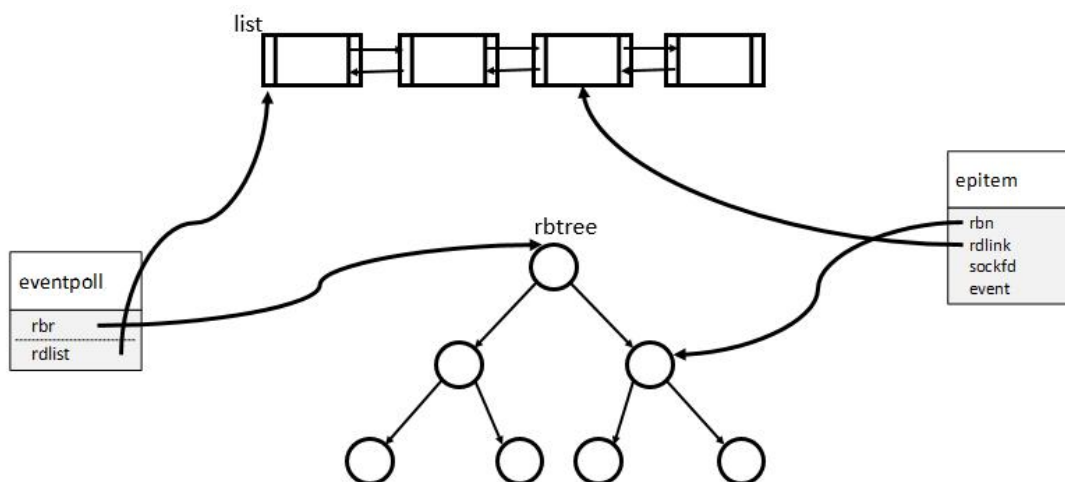
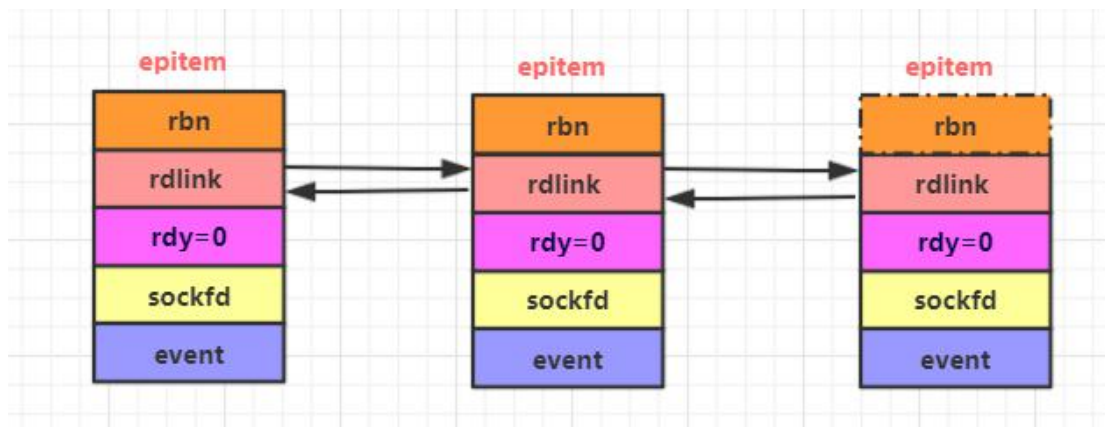
参数 `events`：是内存，也是数组，长度是 `maxevents`，表示此次 `epoll_wait` 调用可以手机到的 `maxevents` 个已经继续【已经准备好的】的读写事件；

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；

- 客户端完成三路握手；服务器要 `accept()`;
- 当客户端关闭连接，服务器也要调用 `close()` 关闭；
- 客户端发送数据来的；服务器要调用 `read()`, `recv()` 函数来收数据；
- 当可以发送数据时；服务器可以调用 `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

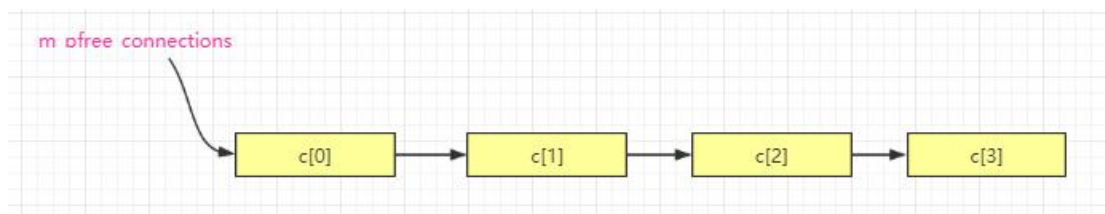
三：epoll 函数实战

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

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

5.7 通讯代码精粹之 epoll 函数实战 2

一: ngx_epoll_process_events 函数调用位置

上节课: epoll_create();epoll_ctl(); --我们目前已经做好准备 等待迎接客户端主动发起三次握手连入;

介绍 ngx_epoll_process_events();

```
(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 (;;)
(i)      {
(i)      子进程开始在这里不断的死循环
(i)      ngx_process_events_and_timers(); 处理网络事件和定
时器事件
(i)      g_socket.ngx_epoll_process_events(-1); -1 表示卡
着等待吧
(i)      }

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

从 ngx_epoll_process_events()的函数调用位置, 我们能够感觉到:

a)这个函数, 仍旧是在子进程中被调用;

b)这个函数，放在了子进程的 `for (;;)`，这意味着这个函数会被不断的调用；

二： `ngx_epoll_process_events` 函数内容

用户三次握手成功连入进来，这个“连入进来”这个事件对于我们服务器来讲，就是一个监听套接字上的可读事件；

(2.1) 事件驱动：官方 `nginx` 本身的架构也被称为“事件驱动架构”；拆开：“事件”，“驱动”；

“驱动”：汽车靠发动机驱动，人体的发动机就是心脏；

“驱动”：动力的来源；驱动的问题，就是探讨程序怎么来干活的问题；

“事件驱动”，无非就是通过获取事件，通过获取到的事件并根据这个事件来调用适当的函数从而让整个程序干活，无非就是这点事；

三： `ngx_event_accept` 函数内容

a) `accept4/accept`

b) `ngx_get_connection/setnonblocking`

c) `ngx_epoll_add_event`

(3.1) `epoll` 的两种工作模式：LT 和 ET 【面试可能问】

LT: level triggered, 水平触发，这种工作模式 低速模式（效率差） -----`epoll` 缺省用次模式

ET: edge triggered, 边缘触发/边沿触发，这种工作模式 高速模式（效率好）

现状：所有的监听套接字用的都是 水平触发； 所有的接入进来的用户套接字都是边缘触发

水平触发的意思：来 一个事件，如果你不处理它，那么这个事件就会一直被触发；

边缘触发的意思：只对非阻塞 `socket` 有用；来一个事件，内核只会通知你一次 【不管你是否处理，内核都不再次通知你】；

边缘触发模式，提高系统运行效率，编码的难度加大；因为只通知一次，所以接到通知后，你必须要保证把该处理的事情处理利索；

程序高手能够洞察到很多普通程序员所洞察不到的问题，并且能够写出更好的，更稳定的，更少出错误的代码；

四：总结和测试

(1) 服务器能够感知到客户端发送过来 `abc` 字符了；

(2) 来数据会调用 `ngx_wait_request_handler ()`

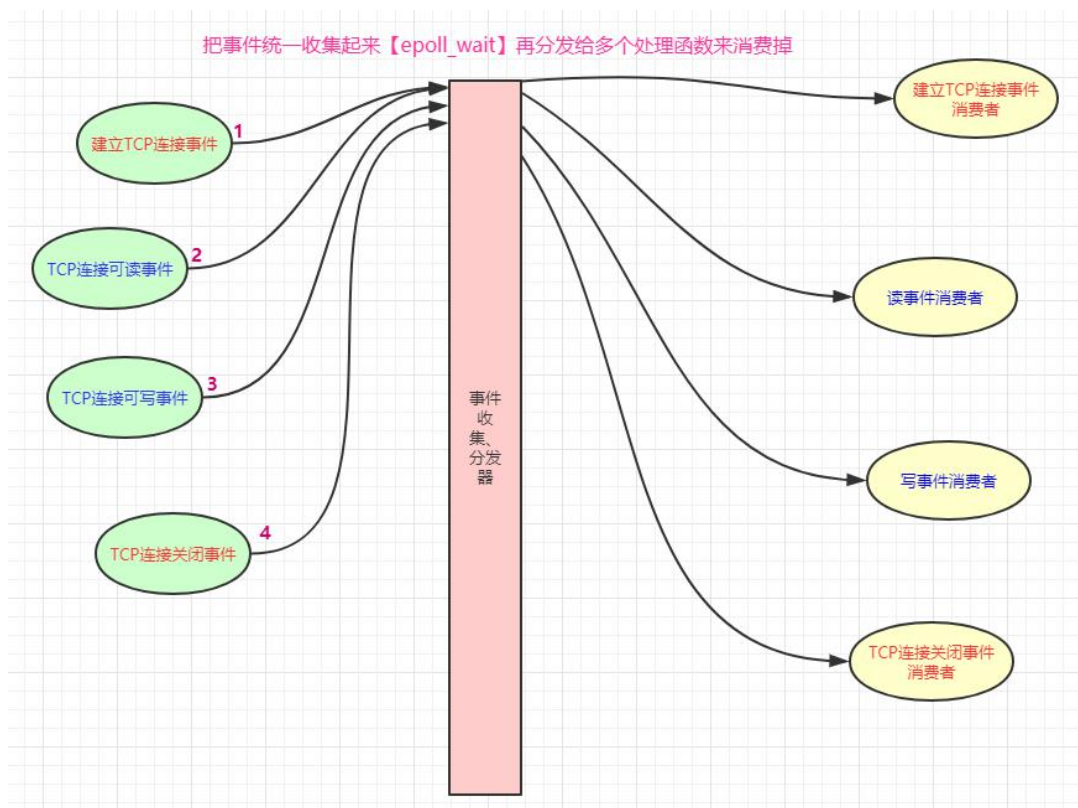
五：事件驱动总结： `nginx` 所谓的事件驱动框架 【面试可能问到】

我们的项目和官方 `nginx` 一样，都是事件驱动框架；

总结事件驱动框架/事件驱动架构

所谓事件驱动框架，就是由一些事件发生源 【三次握手内核通知，事件发生源就是客户端】，通过事件收集器来收集和分发事件 【调用函数处理】

【事件收集器： `epoll_wait()` 函数】 【`ngx_event_accept ()`， `ngx_wait_request_handler ()` 都属于事件处理器，用来消费事件】



六：一道腾讯后台开发的面试题

问题：使用 Linux epoll 模型，水平触发模式；当 socket 可写时，会不停的触发 socket 可写的事件，如何处理？

/*

答案：

第一种最普遍的方式：

需要向 socket 写数据的时候才把 socket 加入 epoll【红黑树】，等待可写事件。接收到可写事件后，调用 write 或者 send 发送数据。当所有数据都写完后，把 socket 移出 epoll。

这种方式的缺点是，即使发送很少的数据，也要把 socket 加入 epoll，写完后在移出 epoll，有一定操作代价。

一种改进的方式：

开始不把 socket 加入 epoll，需要向 socket 写数据的时候，直接调用 write 或者 send 发送数据。如果返回 EAGAIN，把 socket 加入 epoll，在 epoll 的驱动下写数据，全部数据发送完毕后，再移出 epoll。

这种方式的优点是：数据不多的时候可以避免 epoll 的事件处理，提高效率。

*/

5.8 ET、LT 深释，服务器设计、粘包解决

一：ET，LT 模式深入分析及测试

LT: 水平触发/低速模式, 这个事件没处理完, 就会被 一直触发;

ET: 边缘触发/告诉模式, 这个事件通知只会出现一次;

普遍认为 ET 比 LT 效率高一些, 但是 ET 编程难度比 LT 大一些;

ET 模式下, 如果没有数据可接收, 则 `recv` 会返回 -1

思考: 为什么 ET 模式事件只触发一次[事件被扔到双向链表中一次, 被 `epoll_wait` 取出后就干掉]

LT 模式事件会触发多次呢? [事件如果没有处理完, 那么事件会被多次往双向链表中扔]

如何选择 ET, 还是 LT

如果收发数据包有固定格式【后续会讲】, 那么老师建议采取 LT: 编程简单, 清晰, 写好了效率不见得低;

老师准备本项目中采用 LT 这种方法【固定格式的数据收发方式来写我们的项目】

如果收发数据包没有固定格式, 可以考虑采用 ET 模式;

二: 我们的服务器设计

(2.1) 服务器设计原则总述

我们写的是: 通用的服务框架:将来, 稍加改造甚至不用改造就可以把它直接应用在很多的具体开发工作中;

我们的工作重点就可以聚焦在业务逻辑上; 相当于你自带框架【自带源码】入职; 甚至你可以挑战高级程序员/主程序这种职业;

(2.2) 收发包格式问题提出

第一条命令出拳【1abc2】, 第二条加血【1def2|30】;

1abc21def2|30

(2.3) TCP 粘包、缺包

tcp 粘包问题

client 发送 abc,def,hij, 三个数据包发出去;

a)客户端粘包现象

客户端因为有一个 Nagle 优化算法;

`send("abc"); write()`也可以

`send("def");`

`send("hij");`

因为 Nagle 算法存在的, 这三个数据包被 Nagle 优化算法直接合并一个数据包发送出去; 这就属于客户端粘包;

如果你关闭 Nagle 优化算法, 那么你调用几次 `send()`就发送出去几个包; 那客户端的粘包问题就解决了;

b)服务器端粘包现象

不管你客户端是否粘包, 服务器端都存在粘包的问题: 就算你客户端不粘包, 但是, 仍然避免不了服务器端粘包的问题;

服务器端两次 `recv` 之间可能间隔 100 毫秒，那可能在这 100 毫秒内，客户端这三个包都到了，这三个包都被保存到了服务器端的

针对该 TCP 连接数据缓冲中 **【abcdefghij】**；你再次 `recv` 一次，就可能拿到了全部的“abcdefghij”，这就叫服务器端的 粘包；

再举一例：

`send("abc.....");` 8000 字节；这个可能被操作系统拆成 6 个包发送出去了；

网络可能出现延迟或者阻塞，

服务器端第一次 `recv() = "ab"`

`recv = "c..."`，

`recv.....`

`recv() = ".....de"....` [缺包]

(2.4) TCP 粘包、缺包解决：面试服务器岗位经常考，没听说过粘包或者不会解决 TCP 粘包这个问题，那么会立即被 **pass**；

粘包，要解决的就是吧这几个包拆出来，一个是一个；

解决粘包的方案很多；老师提供的方案是简单、严谨、有效的一种解决方案；

严谨

`abcdefhij` ,很多服务器程序员不考虑 恶意数据包；

服务器程序员不能假设收到的数据包都是善意的，合理的，构造畸形数据包 `abc#def-hij`

如何解决拆包问题：给收发的数据包定义一个统一的格式[规则]；**c/s** 都按照这个格式来，就能够解决粘包问题；

包格式： 包头+包体 的格式；其中 包头 是固定长度 **【10 字节】**，在包头中，有一个成员变量会记录整个包 **【包头+包体】** 的长度；

这样的话，先收包头，从包头中，我知道了整个包的长度，然后 用整个包的长度 - 10 个字节 = 包体的长度。

我再收 “包体的长度” 这么多的字节； 收满了包体的长度字节数，我就认为，一个完整的数据包 **【包头+包体】** 收完；

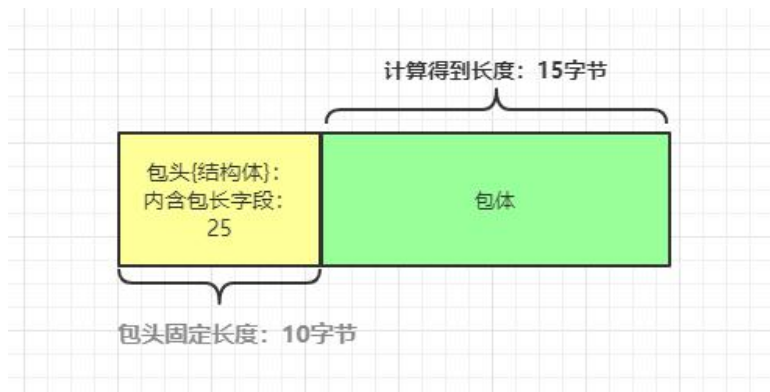
收包总结：

(1) 先收固定长度包头 10 字节；

(2) 收满后，根据包头中的内容，计算出包体的长度：整个长度-10

(3) 我再收包体长度这么多的数据，收完了，一个包就完整了；

我们就认为受到了一个完整的数据包；从而解决了粘包的问题；



大家要有一个认识:

官方的 nginx 的代码主要是用来处理 web 服务器【一种专用的服务器】, 代码写的很庞杂;

不太适合咱们这种固定数据格式【包头+包体】的服务器【通用性强的服务器, 可以应用于各种领域】。

所以, 后续服务器代码, 老师带着大家以老师自由发挥为主, 共同领略服务器开发的各种风景, 精彩继续;

5.9 通讯代码精粹之收包解包实战

一: 收包分析及包头结构定义

发包: 采用 包头+包体, 其中包头中记录着整个包【包头+包体】的长度;

包头: 就是一个结构;

a) 一个包的长度不能超过 30000 个字节, 必须要有最大值;

伪造恶意数据包, 他规定这里 300 亿, 我这个规定能够确保服务器程序不会处于非常危险的境地;

b) 开始定义包头结构: COMM_PKG_HEADER

c) *****大家千万注意这个问题, 不然会大错特错;

结构字节对齐问题; 为了防止出现字节问题, 所有在网络上传输的这种结构, 必须都采用 1 字节对齐方式 *****

二: 收包状态宏定义

收包: 粘包, 缺包;

收包思路: 先手包头->根据包头中的内容确定包体长度并收包体, 收包状态 (状态机);

定义几种收包的状态, 4 种: 0,1,2,3

三: 收包实战代码

聚焦在 ngx_wait_request_handler() 函数;

同时设置好各种收包的状态: `c->curStat = _PKG_HD_INIT;` `c->precvbuf = c->dataHeadInfo;` `c->irecvlen = sizeof(COMM_PKG_HEADER);`

我们要求, 客户端连入到服务器后, 要主动地【客户端有义务】给服务器先发送数据包; 服务器要主动收客户端的数据包;

服务器按照 包头 + 包体的格式来收包;

引入一个消息头【结构】STRUC_MSG_HEADER, 用来记录一些额外信息

服务器 收包时, 收到: 包头+包体, 我再额外附加一个消息头 ==》 消息头 + 包头 + 包体

再介绍一个分配和释放内存类 CMemory;

本项目中不考虑内存池; 内存池: 对于提高程序运行效率帮助有效; new 非常快;

内存池主要功能就是 频繁的分配小块内存时 内存池可以节省额外内存开销【代价就是代码更复杂】;

四: 遗留问题处理

inMsgRecvQueue, tmpoutMsgRecvQueue, clearMsgRecvQueue

五: 测试服务器收包避免推诿扯皮

验证 ngx_wait_request_handler()函数是否正常工作,准备写一个客户端程序;

windows vs2017, mfc 程序

老师强制大家要写测试程序来测试; 你可以写 linux 平台下的测试程序;

觉悟: 服务器主程序员【重担压肩】;

防止扯皮, 所以服务器端有必要自己书写一个客户端测试程序;

说明: windows vs2017 客户端测试代码, 非常简陋, 只用于演示目的, 不具备商业代码质量;

客户端的 SendData()函数值得学习;

核心代码;MFCApplcation3Dlg.cpp

6.1 业务逻辑之多线程、线程池实战

一: 学习方法

不但要学习老师编写程序的方法, 风格, 更要学习老师解决一个问题的思路。

编程语言、语法这种东西如果你不会, 可以通过学习来解决, 但是这种 解决问题的思路, 是一种只可意会难以言传的东西,

却恰恰能够决定你在开发道路上走多远的东西, 搞程序开发一定要培养自己非常清晰的逻辑思维, 不然, 这条程序开发之路

你会走的特别艰辛;

二: 多线程的提出

用 “线程” 来解决客户端发送过来的 数据包

一个进程 跑起来之后缺省 就自动启动了一个 “主线程”, 也就是我们一个 worker 进程一启动就等于只有一个 “主线程” 在跑;

我们现在涉及到了业务逻辑层面, 这个就要用多线程处理, 所谓业务逻辑: 充值, 抽卡, 战斗;

充值, 需要本服务器和专门的充值服务器通讯, 一般需要数秒到数十秒的通讯时间。此时, 我们必须采用多线程【100 个多线程】处理方式;

一个线程因为充值被卡住, 还有其他线程可以提供给其他玩家及时的服务;

所以，我们服务器端处理用户需求【用户逻辑/业务】的时候一般都会启动几十甚至上百个线程来处理，以保证用户的需求能够得到及时处理；

epoll, iocp(windows), 启动线程数 $\text{cpu} \times 2 + 2$;

主线程 往消息队列中用 `inMsgRecvQueue()` 扔完整包（用户需求），那么一堆线程要从这个消息对列中取走这个包，所在必须要用互斥；

互斥技术在《C++从入门到精通 C++98/11/14/17》的并发与多线程一章详细介绍过；
多线程名词

a)POSIX: 表示可移植操作系统接口 (Portable Operating System Interface of UNIX)。

b)POSIX 线程: 是 POSIX 的线程标准【大概在 1995 年左右标准化的】；它定义了创建和操纵线程的一套 API (Application Programming Interface: 应用程序编程接口)，

说白了 定义了一堆我们可以调用的函数，一般是以 `pthread_` 开头，比较成熟，比较好用；我们就用这个线程标准；

三：线程池实战代码

(3.1) 为什么引入线程池

我们完全不推荐用单线程的方式解决逻辑业务问题，我们推荐多线程开发方式；

线程池: 说白了 就是 我们提前创建好一堆线程，并搞一个雷来统一管理和调度这一堆线程【这一堆线程我们就叫做线程池】，

当来了一个任务【来了一个消息】的时候，我从这一堆线程中找一个空闲的线程去做这个任务【去干活/去处理这个消息】，

活干完之后，我这个线程里边有一个循环语句，我可以循环回来等待新任务，再有新任务的时候再去执行新的任务；

就好像这个线程可以回收再利用 一样；

线程池存在意义和价值；

a)实现创建好一堆线程，避免动态创建线程来执行任务，提高了程序的稳定性；有效的规避程序运行之中创建线程有可能失败的风险；

b)提高程序运行效率: 线程池中的线程，反复循环再利用；

大家有兴趣，可以百度 线程池；但是说到根上，用线程池的目的无非就两条：提高稳定性，提升整个程序运行效率，容易管理【使编码更清晰简单】

【pthread 多线程库】 gcc 末尾要增加 `-lpthread`;

`$(CC) - o $@ $^ -lpthread`

CThreadPool【线程池管理类】

讲解了 `Create()`, `ThreadFunc()`, `StopAll()`;

四：线程池的使用

(4.1) 线程池的初始化 : `Create()`;

(4.2) 线程池工作的激发,所谓激发，就是让线程池开始干活了；

激发的时机: 当我收到了一个完整的用户来的消息的时候，我就要激发这个线程池来获取消息开始工作；

那我激发代码放在哪里呢？

(4.3) 线程池完善和测试

- a)我只开一个线程【线程数量过少，线程池中只有一个线程】，我们需要报告；
- b)来多个消息会堆积，但是不会丢消息，消息会逐条处理；
- c)开两个线程,执行正常，每个线程，都得到了一个消息并且处理；表面看起来，正常；

程序执行流程【可能不太全，后续讲到哪里缺了再补充不着急】

```
(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_threadpool.Create(tmpthreadnums); 创建线程池中
线程
(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
```

6.2 业务逻辑之打通业务处理脉搏实战

一：线程池代码调整及补充说明

支撑线程池的运作主要靠两个函数：pthread_cond_signal(&m_pthreadCond); 触发
pthread_cond_wait(&m_pthreadCond, &m_pthreadMutex); 等待

《Unix 环境高级编程》 11 章 线程，11.6.6: 条件变量: m_pthreadCond:

条件变量，是线程可用的另一种同步机制，条件变量给多个线程提供了一个会合的场所，条件变量与互斥量一起使用时，允许线程以无竞争的方式等待特定的条件发生；

a) 条件本身 **【 while ((pThreadPoolObj->m_MsgRecvQueue.size() == 0) && m_shutdown == false)】** 是由互斥量保护的。

线程在改变条件状态之前必须首先锁住互斥量，其他线程在获取到互斥量之前不会觉察到这种改变，因为互斥量必须在锁定以后才能计算条件；

c++11，也有条件变量的说法 `my_cond.wait(...)`, `my_cond.notify_one(...)`

大家如果有兴趣可以用 c++11 多线程开发技术实现 自己的跨平台的线程池代码；

b) 传递给 `pthread_cond_wait` 的互斥量 `m_pthreadMutex` 对条件 **【 while ((pThreadPoolObj->m_MsgRecvQueue.size() == 0) && m_shutdown == false)】** 进行保护的，

调用者把锁住的互斥量传递给函数 `pthread_cond_wait`，函数 然后自动把调用线程放在 等待条件的 线程列表 上，对互斥量解锁，

这就关闭了 条件检查 和 线程进入休眠状态等待条件改变 这两个操作之间的时间通道，

这样线程就不会错过条件的任何变化。`pthread_cond_wait` 返回时，互斥量再次被锁定；

二：线程池实现具体业务之准备代码

(2.1) 一个简单的 crc32 校验算法介绍

CCRC32 类：主要目的是对收发的数据包进行一个简单的校验，以确保数据包中的内容没有被篡改过；

`Get_CRC()`：给你一段 `buffer`，也就是一段内存，然后给你这段内存长度，该函数计算出一个数字来(CRC32 值)返回来；

(2.2) 引入新的 CSocket 子类

真正项目中要把 `CSocket` 类当成父类使用，具体业务逻辑代码应该放在 `CSocket` 的子类中；

`threadRecvProcFunc()`收到消息之后的处理函数；

(2.3) 设计模式题外话

有很多人善于，乐于：抽象；把一个一个小功能封装成一个一个类；往设计模式上套 进行所谓的面向对象程序设计；

最能体现面向对象的 多态 **【虚函数】**；

写程序：每个人有每个人的喜好；老师最喜欢的就是简单粗暴有效的程序设计方式，完全不喜欢动不动就封装一个类的这种写法；

a)类太多，别人理解起来就非常困难，另外类太多，对程序效率影响很大；

b)几十万，上百万上代码，里边很多部件需要灵活调整，经常变动，不稳定的部分，才需要抽象出来，用虚函数，通过设计模式来灵活解决；

不要乱用设计模式，不要乱封装；

(2.4) 消息的具体设计

为了能够根据客户端发送过来的消息代码 迅速定位到要执行的函数，我们就把客户端发送过来的 消息代码直接当做 一个数组的下标来用；

最终认识：咱们的服务器开发工作【业务逻辑】，主要集中在三个文件中：
ngx_logiccomm.h, ngx_c_slogic.cxx, ngx_c_slogic.h

三：threadRecvProcFunc()函数讲解

四：整体测试工作的开展

服务器开发工作，公司 配备专门客户端开发人员来开发客户端工作；

c/s 配合工作，配合指定通讯协议；协议的制定一般是 服务器程序员来主导；

(1)确定通讯格式是 包头+包体，包头固定多少个字节的，这种规则是服务器端 来制定并在开发一个项目之前，要明确的 和客户端交代好；

要求客户端给服务器发送数据包时严格遵循这种格式；

(2)注册，登录，都属于具体的业务逻辑 命令；这种命令一般都是由服务器牵头来制定；

(4.1) 不做背锅侠

服务器开发难度往往比客户端大很多，责任也重很多；要求也高得多；

讲清楚：服务器端要负责通讯协议的制定工作，以免跟客户端推诿扯皮

服务器有能力站在客户端的角度去制定各种通讯协议；

商量，共同指定协议和数据结构；共同制定协议；

(4.2) 客户端测试代码的调整

服务器端有责任把 crc32 算法给到客户端；

6.3 发包，多线程资源回收深度思考

一：业务逻辑细节写法说明

_HandleRegister(),_HandleLogin()里边到底执行什么，是属于业务逻辑代码；写法，大家自己决定

100 元， 加血 90，加魔 80

发送数据代码下节实现

二：连接池中连接回收的深度思考

服务器 7*24 不间断，服务器稳定性是第一位的；一个服务器，如果不稳定，那么谈别的都是虚的；

稳定性：连接池连接回收问题；

如果客户端【张三】断线，服务器端立即回收连接，这个连接很可能被紧随其后连入的新客户端【李四】所使用，那么这里就很可能产生麻烦；

a)张三 funca(); ---执行 10 秒，服务器从线程池中找到一个线程来执行张三的任务；

b)执行到第 5 秒的时候，张三断线；

c)张三断线这个事情会被服务器立即感知到，服务器随后调用 ngx_close_connection 把原来属于张三这个连接池中的连接给回收了；

d)第 7 秒的时候，李四连上来了，系统会把刚才张三用过的连接池中的连接分配给李四【现在这个连接归李四使用】；

e)10 秒钟到了，这个线程很可能会继续操纵 连接【修改读数据】；很可能导致服务器

整个崩溃；这种可能性是非常有的；

一个连接，如果我们程序判断这个连接不用了；那么 不 应该把这个连接立即放到空闲队列里，而是 应该放到一个地方；

等待一段时间【60】，60 秒之后，我再真正的回收这个连接到 连接池/空闲队列 中去，这种连接才可以真正的分配给其他用户使用；

为什么要等待 60 秒，就是需要确保即使用户张三真断线了，那么我执行的该用户的业务逻辑也一定能在这个等待时间内全部完成；

这个连接不立即回收是非常重要的，有个时间缓冲非常重要；这个可以在极大程度上确保服务器的稳定；

服务器程序要常年累月的优化；

(2.1) 灵活创建连接池

(2.2) 连接池中连接的回收

a)立即回收【accept 用户没有接入时可以立即回收】 b)延迟回收【用户接入进来开始干活了】；

(2.2.1)立即回收 ngx_free_connection()；

(2.2.2)延迟回收 inRecyConnectQueue(), ServerRecyConnectionThread()

三：程序退出时线程的安全终止

多线程开发技术在我们这个项目中是全面开花的

四：epoll 事件处理的改造

(4.1) 增加 新 的 事 件 处 理 函 数 ， 引 入 ngx_epoll_oper_event() 函 数 取 代 ngx_epoll_add_event()；

(4.2) 调 整 对 事 件 处 理 函 数 的 调 用:ngx_epoll_init(),ngx_event_accept(),ngx_epoll_process_events()；

(5) 连接延迟回收的具体应用

recvproc()调用了 inRecyConnectQueue(延迟回收)取代了 ngx_close_connection(立即回收)

Initialize_subproc()子进程中干；

Shutdown_subproc()子进程中干

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
有信号
(i)          g_threadpool.Create(tmpthreadnums); 创建线程池中线程
线程
(i)          _socket.Initialize_subproc(); 初始化子进程需要具备的一些多线程能力相关的信息
一些多线程能力相关的信息
(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
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
标题为 worker process
(i)          for (;;) {
(i)          ngx_process_events_and_timers(); 处理网络事件和定时器事件
定时器事件
(i)          g_socket.ngx_epoll_process_events(-1); -1 表示卡着等待吧
着等待吧
(i)          epoll_wait();
(i)          }. .... 子进程开始在这里不断的死循环
(i)          g_threadpool.StopAll(); 考虑在这里停止线程池;
(i)          g_socket.Shutdown_subproc(); socket 需要释放的东西考虑释放;
释放;

(i)  sigemptyset(&set);
(i)  for (;;) {. 父进程[master 进程]会一直在这里循环

```

6.4 LT 发数据机制深释、gdb 调试浅谈

一：水平触发模式 (LT) 下发送数据深度解释

在水平触发模式下，发送数据有哪些注意事项；

a)一个问题

通过调用 ngx_epoll_oper_event(Epoll_CTL_MOD,Epollout),那么当 socket 可写的时候，

会触发 socket 的可写事件，我得到了这个事件我就可以发送数据了；

什么叫 socket 可写； 每一个 tcp 连接(socket)，都会有一个接收缓冲区 和 一个发送缓冲；

发送缓冲区缺省大小一般 10 几 k，接收缓冲区大概几十 k，setsocketopt()来设置；

`send()`,`write()`发送数据时，实际上这两个函数是把数据放到了发送缓冲区，之后这两个函数返回了；

客户端用 `recv()`,`read()`;

如果服务器端的发送缓冲区满了，那么服务器再调用 `send()`,`write()`发送数据的时候，那么 `send()`,`write()`函数就会返回一个 `EAGAIN`;

`EAGAIN` 不是一个错误，只是示意发送缓冲区已经满了，迟一些再调用 `send()`,`write()`来发送数据吧；

二: gdb 调试浅谈

当 `socket` 可写的时候【发送缓冲区没满】，会不停的触发 `socket` 可写事件【水平触发模式】，已经验证；

遇到程序崩溃问题，所以需要借助 `gdb` 调试来找到崩溃行；

好在：我们的错误能够重现[必现的错误，是很好找的]；

最怕的就是偶尔出现的 `bug`；有的时候运行三个小时就出现，有的时候运行两天也不出现；

a)编译时 `g++` 要带这个 `-g` 选项；

b)`su` 进入 `root` 权限，然后 `gdb nginx` 调试

c)`gdb` 缺省调试主进程，但是 `gdb 7.0` 以上版本可以调试子进程【我们需要调试子进程，因为干活的是 `worker process` 是子进程】；

命令 行下 `:gdb -v` 看版本

d)为了让 `gdb` 支持多进程调试，要设置一下 `follow-fork-mode` 选项，这是个调试多进程的开关；

取值可以是 `parent`[主] /`child`[子]，我们这里需要设置成 `child` 才能调试 `worker process` 子进程；

查看 `follow-fork-mode`: 在 `gdb` 下输入 `show follow-fork-mode`

输入 `set follow-fork-mode child`

(e) 还有个选项 `detach-on-fork`，取值为 `on/off`，默认是 `on`【表示只调试父进程或者子进程其中的一个】

调试是父进程还是子进程，由上边的 `follow-fork-mode` 选项说了算；

如果 `detach-on-fork = off`，就表示父子都可以调试，调试一个进程时，另外一个进程会被暂停；

查看 `show detach-on-fork`

输入 `set show detach-on-fork off`，如果设置为 `off` 并且 `follow-fork-mode` 选项为 `parent`，那么 `fork()`后的子进程并不运行，而是处于暂停状态；

(f)`b logic/nginx_c_slogic.cxx:198`

(g)`run` 运行程序运行到断点；

(h)`print...`打印变量值。这些调试手段,大家自己百度学习；

(i)`c` 命令，继续运行

针对 当 `socket` 可写的时候【发送缓冲区没满】，会不停的触发 `socket` 可写事件，我们提出两种解决方案【面试可能考试】；

b)两种解决方案，来自网络,意义在于我们可以通过这种解决方案来指导我们写代码；

b.1)第一种最普遍的解决方案：

需要向 `socket` 写数据的时候把 `socket` 写事件通知加入到 `epoll` 中, 等待可写事件, 当可写事件来时操作系统会通知咱们;

此时咱们可以调用 `write/send` 函数发送数据, 当发送数据完毕后, 把 `socket` 的写事件通知从红黑树中移除;

缺点: 即使发送很少的数据, 也需要把事件通知加入到 `epoll`, 写完后, 有需要把写事件通知从红黑树干掉, 对效率有一定的影响【有一定的操作代价】

b.2)改进方案:

开始不把 `socket` 写事件通知加入到 `epoll`, 当我需要写数据的时候, 直接调用 `write/send` 发送数据;

如果返回了 `EAGAIN`【发送缓冲区满了, 需要等待可写事件才能继续往缓冲区里写数据】, 此时, 我再把写事件通知加入到 `epoll`,

此时, 就变成了在 `epoll` 驱动下写数据, 全部数据发送完毕后, 再把写事件通知从 `epoll` 中干掉;

优点: 数据不多的时候, 可以避免 `epoll` 的写事件的增加/删除, 提高了程序的执行效率;

老师准备采用 **b.2)**改进方案来指导咱们后续发送数据的代码;

6.5 LT 发数据、信号量、并发、多线程综合实战

针对 当 `socket` 可写的时候【发送缓冲区没满】, 会不停的触发 `socket` 可写事件, 我们提出两种解决方案【面试可能考试】:

b)两种解决方案, 来自网络, 意义在于我们可以通过这种解决方案来指导我们写代码;

b.1)第一种最普遍的解决方案:

需要向 `socket` 写数据的时候把 `socket` 写事件通知加入到 `epoll` 中, 等待可写事件, 当可写事件来时操作系统会通知咱们;

此时咱们可以调用 `write/send` 函数发送数据, 当发送数据完毕后, 把 `socket` 的写事件通知从红黑树中移除;

缺点: 即使发送很少的数据, 也需要把事件通知加入到 `epoll`, 写完后, 有需要把写事件通知从红黑树干掉, 对效率有一定的影响【有一定的操作代价】

b.2)改进方案:

开始不把 `socket` 写事件通知加入到 `epoll`, 当我需要写数据的时候, 直接调用 `write/send` 发送数据;

如果返回了 `EAGAIN`【发送缓冲区满了, 需要等待可写事件才能继续往发送缓冲区里写数据】, 此时, 我再把写事件通知加入到 `epoll`,

此时, 就变成了在 `epoll` 驱动下写数据, 全部数据发送完毕后, 再把写事件通知从 `epoll` 中干掉;

优点: 数据不多的时候, 可以避免 `epoll` 的写事件的增加/删除, 提高了程序的执行效率;

老师准备采用 b.2)改进方案来指导咱们后续发送数据的代码；

一：发送数据指导思想

把要发送的数据放到一个队列中 [msgSend]，然后咱们专门创建一个线程 [ServerSendQueueThread]来统一负责数据发送；

二：发送数据代码实战

(2.1) 信号量:也是一种同步机制；他跟互斥量有什么不同呢/特殊？

互斥量：线程之间同步；

信号量：提供进程之间的同步，也能提供线程之间的同步；

《Unix 网络编程 卷 2—进程间通讯》第二版，第十章 清晰描述了信号量用法；

用之前调用 `sem_init()`初始化一下；信号量的初始值 我们给了 0 【这个值 0 有大用】；

用完后用 `sem_destroy()`释放信号量；

`sem_wait()`：测试指定信号量的值，如果该值>0，那么将该值-1 然后该函数立即返回；

如果该值 等于 0，那么该线程将投入睡眠中，一直到该值>0，这个时候 那么将该值-1 然后该函数立即返回；

`sem_post()`：能够将指定信号量值+1，即便当前没有其他线程在等待该信号量值也没关系；

(2.2) 数据发送线程 `ServerSendQueueThread` *****

(2.3) 可写通知到达后数据的继续发送

`ngx_write_request_handler()`； *****

(2.4) 发送数据的简单测试

发送缓冲区大概 10-几 10K，

如何把发送缓冲区撑满

(1) 每次服务器给客户端发送 65K 左右的数据，发送到第 20 次才出现服务器的发送缓冲区满；这时客户端收了一个包(65K)，

此时执行了 `ngx_write_request_handler()`；

(2) 我又发包，连续成功发送了 16 次，才又出现发送缓冲区满；我客户端再收包，结果连续收了 16 次包，服务器才又出现

`ngx_write_request_handler()`函数被成功执行，这表示客户端连续收了 16 次包，服务器的发送缓冲区才倒出地方来；

(3) 此后，大概服务器能够连续发送 16 次才再出现发送缓冲区满，客户端连续收 16 次，服务器端才出现 `ngx_write_request_handler()`被执行 【服务器的发送缓冲区有地方】；

测试结论：

(1) `ngx_write_request_handler ()` 逻辑正确；能够通过此函数把剩余的未成功发送的数据发送出去；

(2) LT 模式下，我们发送数据采用的 改进方案 是非常有效的，在很大程度上提高了效率；

(3) 发送缓冲区大概 10-几 10K,但是我们实际测试的时候，成功的发送出去了 1000 多 k 数据才报告发送缓冲区满；

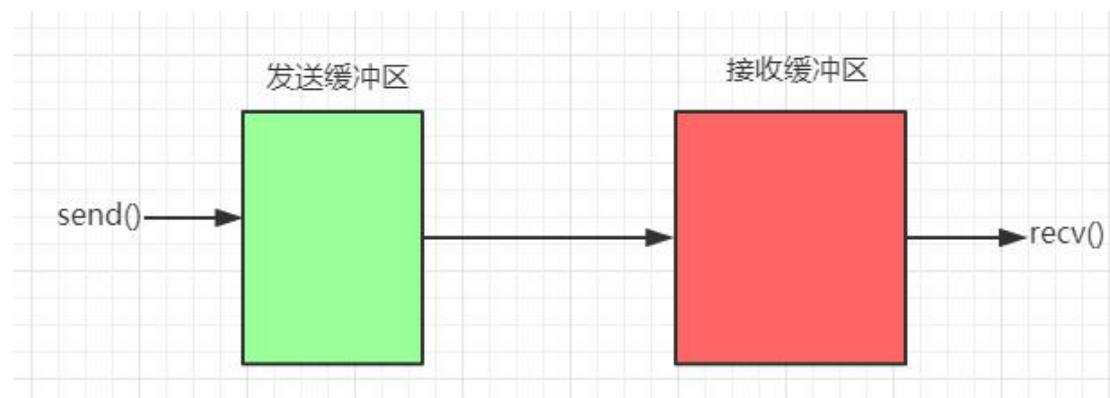
当我们发送端调用 `send()` 发送数据时，操作系统底层已经把数据发送到了 该连接的接收端 的接收缓存，这个接收缓存大概有几百 K，

千万不要认为发送缓冲区只有几十 K，所以我们 `send()` 几十 k 就能把发送缓冲区填满；

(4) 不管怎么说，主要对方不接收数据，发送方的发送缓冲区总有满的时候；

当发送缓冲满的时候，我们发送数据就会使用 `ngx_write_request_handler()` 来执行了，所以现在看起来，我们整个的服务器的发送数据的实现代码是正确的；

三：发送数据后续处理代码



7.1 过往总结、心跳包代码实战

一：前面学习的总结

核心架构浓缩总结实现的功能：

- (1) 服务器按照包头包体格式正确的接收客户端发送过来的数据包；
- (2) 根据手动的包的不同来执行不同的业务处理逻辑；
- (3) 把业务处理产生的结果数据包返回客户端；

咱们用到的主要技术

- (1) `epoll` 高并发通讯技术
- (2) 线程池技术来处理业务逻辑
- (3) 线程之间的同步技术包括互斥量、信号量

其他技术：信号，日志打印，`fork()`子进程，守护进程

借鉴了哪些官方 `nginx` 的精华代码

- (1) `master` 进程，多个 `worker` 子进程——进程框架；
`nginx`：热更新，`worker` 子进程挂了 `master` 能够重新启动 `worker`；重载配置文件

【卷 1 讲解的主要是框架——包括通讯框架、包括业务逻辑处理框架，这才是我们本门课程的核心和精粹】

(2) 借鉴了 `epoll` 的一些实现代码；官方 `nginx` 用的 ET【边缘触发模式】，咱们本项目中用的是水平触发模式 LT

- (3) 借鉴了接收数据包，以及发送数据包的核心代码；

官方 `nginx` 对我们项目的实现帮助不小，咱们借鉴的都是 `nginx` 中最核心，最值得借鉴的优秀代码；

哪些内容我们没有借鉴官方 **nginx** 呢？

(1) 比如 **epoll** 技术中我们采用 **LT** 模式来书写网络数据的接收和发送；

(2) 自己写一套线程池来处理业务逻辑，调用适当的业务逻辑处理函数，直至处理完毕把数据发送回客户端；

(3) 连接池 中 连接的 延迟 回收，以及专门处理数据发送的发送线程；

上面三条都是咱们这门课程的核心精粹代码；

只要学懂，最低都是一个高级程序员，在一个十几个人的小公司，做一个服务器开发主程序都没有问题；

良心与责任同行：扶上马，送一程，能送多远，老师会尽力；

目前为止：

a)咱们服务器代码有没有什么瑕疵 **bug**，导致比如说客户端发送恶意数据包能够把我们服务器攻击死呢？

b)有没有代码能够进一步完善，写的更好； 提出问题，解决问题：

二：心跳包概念：注意面试可能考

c/s 程序：都有责任把心跳包机制在程序代码中实现好，以确保程序能良好的工作以及应付意外的情形发生；

(2.1) 什么叫心跳包以及如何使用

心跳包其实就是一个普通的数据包；

一般每个几十秒，最长一般也就是 1 分钟【10 秒-60 秒之间】，有客户端主动发送给服务器；服务器收到之后，一般会给客户端返回一个心跳包；

三路握手，**tcp** 连接建立之后，才存在发送心跳包的问题—— 如果 **c** 不给 **s** 发心跳包，服务器会怎样；

约定 30 秒发送 一次； 服务器可能会在 90 秒或者 100 秒内，主动关闭该客户端的 **socket** 连接；

作为一个好的客户端程序，如果你发送了心跳包给服务器，但是在 90 或者 100 秒之内，你[客户端]没有收到服务器回应的心跳包，那么

你就应该主动关闭与服务器端的链接，并且如果业务需要重连，客户端程序在关闭这个连接后还要重新主动再次尝试连接服务器端；

客户端程序 也有必要提示使用者 与服务器的连接已经断开；

(2.2) 为什么引入心跳包

常规客户端关闭，服务器端能感知到；

有一种特殊情况，连接断开 **c/s** 都感知不到；

tcp 本身 **keepalive** 机制；不重点研究，大家可以自己百度，因为检测时间不好控制，所以不适合我们；

c /s 程序运行在不同的两个物理电脑上；**tcp** 已经建立；

拔掉 **c /s** 程序的网线； 拔掉网线导致服务器感知不到客户端断开，这个事实，大家一定要知道；

那位了应对拔网线，导致不知道对方是否断开了 **tcp** 连接这种事，这就是我们引入心跳包机制的原因；

超时没有发送来心跳包，那么就会将对端的 **socket** 连接 **close** 掉，回收资源；这就是心跳包的作用；

其他作用：检测网络延迟等等；大家以后遇到再研究；
我们这里讲解心跳包，主要目的就是检测双方的链接是否断开；

三：心跳包代码实战

(3.1) 接收心跳包与返回结果

心跳包 (ping 包)

规定消息代码为 0；一般心跳包也不需要包体，只有包头就够了；

(3.2) 处理不发送心跳包的客户端

30 秒；超过 $30 \times 3 + 10 = 100$ 秒，仍旧没收到心跳包，那么服务器端就把 tcp 断开；

增加配置 Sock_WaitTimeEnable, Sock_MaxWaitTime

修改 ReadConf()函数读取配置信息；

ngx_c_socket_time.cxx 专门存放一些跟时间有关的函数；

AddToTimerQueue() : 把一个连接的信息加入到时间队列中来；该函数由 ngx_event_accept()函数在连接成功连入时调用；

定时器队列：官方 nginx，红黑树做定时器，时间轮，老师用 multimap 来做定时器；

谁又来处理时间队列中的数据呢？咱们的思路是 创建一个新线程，专门处理事件队列这个事；介绍新线程 ServerTimerQueueMonitorThread()；

GetOverTimeTimer();RemoveFirstTimer();procPingTimeOutChecking();zdClosesocketProc()

改 造 inRecyConnectQueue() ;
ServerRecyConnectionThread();clearAllFromTimerQueue ()

(3.3) 测试代码的书写

7.2 控制连入数，黑客攻击防范及畸形包应对

一：控制并发连入数量

epoll 支持高并发：数万，数十万，百万【一台计算机】；

单纯探讨 epoll 支持多少并发连接，意义不是很大；

并发数量取决于很多因素：

(1)你采用的开发技术：epoll，支持数十万并发

(2)你这个程序收发数据的频繁程度，以及具体要处理的业务复杂程度

(3)你实际的物理内存；可用的物理内存数量，会直接决定你能支持的并发连接

(4)一些其他的 tcp/ip 配置项，以后有机会再谈；

一般，我们日常所写的服务器程序，支持几千甚至 1-2 万的并发，基本上就差不多了；
一个服务器程序，要根据我们具体的物理内存，以及我们具体要实现的业务等等因素，

控制能够同时连入的客户端数量;

如果你允许客户端无限连入, 那么你的服务器肯定会崩溃;

`m_onlineUserCount`

`void CSocekt::ngx_event_accept(lpngx_connection_t oldc) 连入人数+1`

`void CSocekt::inRecyConnectQueue(lpngx_connection_t pConn) 连入人数-1`

控制连入用户数量的解决思路: 如果同时连入的用户数量超过了允许的最大连入数量时, 我们就把这个连入的用户直接踢出去;

二: 黑客攻击的防范

攻击效果

(1)轻则: 服务器工作 延迟, 效率明显降低

(2)重则整个服务器完全停摆, 没有办法提供正常服务, 比如拒绝服务攻击就会导致这种状况【服务器失去响应】;

(3)最甚者, 如果你服务器程序写的有一些漏洞的话, 恶意黑客很可能利用给一些远程溢出攻击手段直接攻破你的服务器程序所在的计算机;

能拿到一定的权限, 甚至可能是 `root` 权限, 一旦拿到了 `root` 权限, 那么你的整个计算机都在黑客控制中了;

但是, 只要大家谨慎细心的写程序, 这个漏洞可以避免;

有些攻击是利用 `tcp/ip` 协议先天的一些设计问题来攻击;【比如 `syn flood` 攻击】

`DDOS` 攻击【`syn flood` 攻击也是 `DDOS` 攻击的一种】甚至防火墙等设备都可能防不住, 甚至得从数据路由器想办法;

我们的服务 程序, 对于 `DDOS` 攻击, 是没有办法解决的; 但是我们写的程序, 是能够解决一些用户三路握手连入进来后的一些网络安全问题;

(2.1) flood 攻击防范

以游戏服务器为例;

假设我们认为一个合理的客户端一秒钟发送数据包给服务器不超过 10 个;

如果客户端不停的给服务器发数据包, 1 秒钟超过了 10 个数据包, 那我服务器就认为这个玩家有恶意攻击服务器的倾向;

我们服务器就应该果断的把这个 `TCP` 客户端连接关闭, 这个也是服务器发现恶意玩家以及保护自身安全的手段;

代码上如何实现 1 秒钟超过 10 个数据包则把客户端踢出去;

增 加 了 `TestFlood()`; 改 造 了
`ngx_read_request_handler()`,`ngx_wait_request_handler_proc_p1()`,`ngx_wait_request_handler_proc_plast ()`

(2.2) 畸形数据包防范

意识: 客户端发送过来的数据并不可信, 因为这些数据有可能是造假的, 甚至可能是畸形的;

以游戏服务器为例;

(1)造假: 服务器端书写的时候要细致判断, 基本能够避免客户端造假;

(2)畸形数据包: 远程溢出攻击, 攻击成功的主要原因就是服务器程序书写不当, 比如

接收到的数据缺少边界检查;

以_HandleRegister()函数为例;

我们有必要在字符数组末尾 自己增加一个 \0 【字符串结束标记】, 以确保我们用这个字符数组的时候不会出问题;

```
p_RecvInfo->username[sizeof(p_RecvInfo->username) - 1] = 0;
```

```
p_RecvInfo->password[sizeof(p_RecvInfo->password) - 1] = 0;
```

三: 超时直接踢出服务器的需求

账号服务器;

账号服务器有一个需求: 超过 20 秒不主动断开与本服务器连接的, 那么本服务器就要主动的把这个用户踢下线 (断开 TCP 连接);

配置 Sock_TimeOutKick=1, 调整 GetOverTimeTimer() 函数 ;
procPingTimeOutChecking();

7.3 超负荷安全处理、综合压力测试

一: 输出一些观察信息

每隔 10 秒钟把一些关键信息显示在屏幕上;

(1)当前在线人数;

(2)和连接池有关: 连接列表大小, 空闲连接列表大小, 将来释放的连接多少个;

(3)当前时间队列大小

(4)收消息队列和发消息队列大小;

打印统计信息的函数: printTDInfo(), 每 10 秒打印一次重要信息;

二: 遗漏的安全问题思考

(2.1) 收到太多数据包处理不过来

限速: epoll 技术, 一个限速的思路; 在 epoll 红黑树节点中, 把这个 EPOLLIN 【可读】通知干掉;

在 printTDInfo()中做了一个简单提示, 大家根据需要自己改造代码;

(2.2) 积压太多数据包发送不出去

见 void CSocekt::msgSend(char *psendbuf)

.....大家多思考, 看有没有什么遗漏.....

(2.3) 连入安全的进一步完善

```
void CSocekt::ngx_event_accept(lpngx_connection_t oldc)
```

```
if(m_connectionList.size() > (m_worker_connections * 5))
```

.....大家多思考, 看有没有什么遗漏.....

三: 压力测试前的准备工作

(3.1) 配置文件内容和配置项确认

(3.2) 整理业务逻辑函数

四：压力测试

非常希望同学们好老师 一起测试。一般要测试很多天, 跑的时间长了可能 会暴露下次, 跑的时间短了可能还暴露不出来;

ScanThread

```
socket()
connect()
FuncsendrecvData()
    send()
    recv()
FunccloseSocket()
    closesocket();
FunccreateSocket()
    socket()
    connect();
```

老师建议:

(1)大家有设备, 有条件, 都来测试; 每个人都要 200%的用心测试;

收包, 简单的逻辑处理, 发包;

(2)建议如果有多个物理电脑; 客户端单独放在一个电脑;

建议用高性能 linux 服务器专门运行服务器程序

windows 也建议单独用一个电脑来测试;

(3)测试什么?

a)程序崩溃, 这明显不行, 肯定要解决

b)程序运行异常, 比如过几个小时, 服务器连接不上; 没有回应了, 你发过来的包服务器处理不了了;

c)服务器程序占用的内存才能不断增加, 增加到一定程度, 可能导致整个服务器崩溃;

top -p 3645 : 显示进程占用的内存和 cpu 百分比, 用 q 可以退出;

```
cat /proc/3645/status      -----VmRSS:      7700 kB
```

遇到错误, 及时更正; 最好放在不同的物理电脑上测试;

(4.1) 最大连接只在 1000 多个

日志中报: CSocet::ngx_event_accept()中 accept4()失败

这个跟 用户进程可打开的文件数限制有关; 因为系统为每个 tcp 连接都要创建一个 socet 句柄, 每个 socket 句柄同时也是一个文件句柄;

```
ulimit -n      -----1024    === 1018
```

我们就必须修改 linux 对当前用户的进程 同时打开的文件数量的限制;

(4.2) 学习忠告

如何修改 linux 对当前用户的进程 同时打开的文件数量的限制,留给大家;

依赖性; 老师作用不是做保姆, 不是当拐棍; 老师在关键时刻帮助大家应个急

首先: 百度, 求助其他同学; 老师是大家求助的最后一道防线;

程序人员每个人都必须要学会自己解决问题, 这种自行解决问题的能力比学习这门课程本身更重要; --授自己以渔

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_threadpool.Create(tmpthreadnums); 创建线程池中线程

(i) _socket.Initialize_subproc(); 初始化子进程需要具备的一些多线程能力相关的信息

(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 (;;) {

(i) ngx_process_events_and_timers(); 处理网络事件和定

时器事件

(i) g_socket.ngx_epoll_process_events(-1); -1 表示卡

着等待吧

(i) epoll_wait();

(i) g_socket.printTDInfo();

(i) }. 子进程开始在这里不断的死循环

(i) g_threadpool.StopAll(); 考虑在这里停止线程池;

(i) g_socket.Shutdown_subproc(); socket 需要释放的东西考虑释放;

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

7.4 惊群、性能优化大局观

一: cpu 占比与惊群

top -p pid,推荐文章: <https://www.cnblogs.com/dragonsuc/p/5512797.html>

惊群: 1 个 master 进程 4 个 worker 进程

一个连接进入, 惊动了 4 个 worker 进程, 但是只有一个 worker 进程 accept();其他三个 worker 进程被惊动, 这就叫惊群;

但是, 这三个被惊动的 worker 进程都做了无用功【操作系统本身的缺陷】;

官方 nginx 解决惊群的办法: 锁, 进程之间的锁; 谁获得这个锁, 谁就往监听端口增加 EPOLLIN 标记, 有了这个标记, 客户端连入就能够被服务器感知到;

3.9 以上内核版本的 linux, 在内核中解决了惊群问题; 而且性能比官方 nginx 解决办法效率高很多;

reuseport【复用端口】,是一种套接字的复用机制, 允许将多个套接字 bind 到同一个 ip 地址/端口上, 这样一来, 就可以建立多个服务器

来接收到同一个端口的连接【多个 worker 进程能够监听同一个端口】;

大家注意一点:

a)很多 套接字配置项可以通过 setsockopt()等等函数来配置;

b)还有一些 tcp/ip 协议的一些配置项我们可以通过修改配置文件来生效;

课后作业:

(1)在 worker 进程中实现 ngx_open_listening_sockets()函数;

(2)观察, 是否能解决惊群问题;

(3)如果在 master 进程中调用 ngx_open_listening_sockets()函数, 那么建议 master 进程中把监听 socket 关闭;

```
if (setsockopt(isock, SOL_SOCKET, SO_REUSEPORT,(const void *) &reuseport, sizeof(int))== -1)
```

二: 性能优化大局观

a)性能优化无止境无极限

b)没有一个放之四海皆准的优化方法, 只能够一句具体情况而定

c)老师在这里也不可能把性能优化方方面面都谈到, 很多方面, 大家都需要不断的探索和尝试;

从两个方面看下性能优化问题;

软件层面:

- a)充分利用 **cpu**，比如刚才惊群问题；
- b)深入了解 **tcp/ip** 协议，通过一些协议参数配置来进一步改善性能；
- c)处理业务逻辑方面，算法方面有些内容，可以提前做好；

硬件层面【花钱搞定】：

- a)高速网卡，增加网络带宽；
- b)专业服务器：数十个核心，马力极其强；
- c)内存：容量大，访问速度快；
- d)主板啊，总线不断升级的；

三：性能优化的实施

(3.1) 绑定 **cpu**、提升进程优先级

- a)一个 **worker** 进程运行在一个核上；为什么能够提高性能呢？

cpu：缓存；**cpu** 缓存命中率问题；把进程固定到 **cpu** 核上，可以大大增加 **cpu** 缓存命中率，从而提高程序运行效率；

worker_cpu_affinity【**cpu** 亲和性】，就是为了把 **worker** 进程固定的绑到某个 **cpu** 核上；
ngx_set_cpu_affinity,ngx_setaffinity;

- b)提升进程优先级,这样这个进程就有机会被分配到更多的 **cpu** 时间（时间片【上下文切换】），得到执行的机会就会增多；

setpriority();

干活时进程 **chuyuR** 状态，没有连接连入时，进程处于 **S**

pidstat -w -p 3660 1 看某个进程的上下文切换次数[切换频率越低越好]

cschw/s：主动切换/秒：你还有运行时间，但是因为你等东西，你把自己挂起来了，让出了自己时间片。

ncschw/s：被动切换/秒：时间片耗尽了，你必须切出去；

- c)一个服务器程序，一般只放在一个计算机上跑,专用机；

(3.2) **TCP / IP** 协议的配置选项

这些配置选项都有缺省值，通过修改，在某些场合下，对性能可能会有所提升；

若要修改这些配置项，老师要求大家做到以下几点：

- a)对这个配置项有明确的理解；
- b)对相关的配置项,记录他的缺省值，做出修改；
- c)要反复不断的亲自测试，亲自验证；是否提升性能，是否有副作用；

五：TCP / IP 协议的配置选项

(3.1) 绑定 **cpu**、提升进程优先级

(3.2) **TCP / IP** 协议的配置选项

(3.3) **TCP/IP** 协议额外注意的一些算法、概念等

- a)滑动窗口的概念
- b)**Nagle** 算法的概念
- c)**Cork** 算法
- d)**Keep - Alive** 机制
- e)**SO_LINGER** 选项

四：配置最大允许打开的文件句柄数
`cat /proc/sys/fs/file-max` : 查看操作系统可以使用的最大句柄数
`cat /proc/sys/fs/file-nr` : 查看当前已经分配的，分配了没使用的，文件句柄最大数
目

限制用户使用的最大句柄数
/etc/security/limit.conf 文件;
root soft nofile 60000 :setrlimit(RLIMIT_NOFILE)
root hard nofile 60000

ulimit -n : 查看系统允许的当前用户进程打开的文件数限制
ulimit -HSn 5000 : 临时设置，只对当前 session 有效;
n:表示我们设置的是文件描述符
推荐文章: <https://blog.csdn.net/xyang81/article/details/52779229>

五：内存池补充说明
为什么没有用内存池技术：感觉必要性不大
TCMalloc,取代 malloc();
库地址: <https://github.com/gperftools/gperftools>

8.1 结束语、课程总结

一：课程概要总结
二：课程技术总结
完整项目：通讯框架+业务逻辑框架【理解成框架课，也没问题】
项目包括
(1)项目是非常完整的多线程高并发服务器程序
(2)按照包头+包体格式 收，完美的解决了数据粘包的问题
(3)根据收到的数据包 来执行不同的业务逻辑
(4)把业务处理产生的结果数据包正确的返回给客户端

用到的主要开发技术
(1)epoll 高并发通讯技术，用的是水平触发模式【LT】，简单提及边缘触发模式【ET】
(2)通过线程池技术处理业务逻辑
(3)多线程，线程之间的同步技术包括互斥量、信号量等等
(4)次要技术：信号，日志打印，fork()创建子进程，守护进程怎么写

借鉴了官方 nginx 哪些精华代码
(1)一个 master 进程，多个 worker 进程的 进程框架;
(2)epoll 实现代码，但官方 epoll 的触发模式用的是 ET【我们的项目用的是 LT】
(3)借鉴了官方 nginx 的接收数据包以及发送数据包的核心代码;

哪些是我们没有借鉴官方 nginx 而独立实现的代码

(1)epoll LT 模式

(2)自己写了一套线程池来处理业务逻辑，调用适当的业务逻辑处理函数，只会处理完毕把数据发送回客户端；

(3)连接池中的连接的延迟回收；*****

(4)专门处理数据发送的一整套数据发送逻辑以及发送线程；

本课程对 windows 编程是否有帮助：帮助太大了；【epoll, iocp】

三：项目的用途总结

高级程序员，服务器主程序开发人员；

完整的项目代码；节省了你自己写通讯框架，业务逻辑框架的时间

就可以开口 【20-30k】，自带框架

本课程学到的不仅仅是知识，而是实实在在的经验，得到的是真真正正的工作经验；大家拿到了完整的商业框架；

四：学习方法总结

a)注意学习方法，只/优先 学习老师的这几门课程

b)无书在手不踏实,正确的方法：先看完老师的视频，然后你再决定是否有必要购买书籍；

c)做题；先把课程学完，回头再决定是否额外做题；

d)很多人 往往把一些工具书当成是常规书籍看；

很多书籍 泛读而是精读，大家只需要记一个大概内容、作用和一个大概位置，需要的时候去查阅就够了；

老师认为不值得花大量时间去啃工具书，需要的时候现学和查阅就够；

不要指望看一遍就能看懂

五：架构师之路

卷 1 是想成长为架构师的人所必须要学的知识，否则，你永远不可能成为一个合格的架构师；

在老师看来，架构师，首先应该是一个非常优秀的软件开发人员；

继续学分布式系统架构相关的知识和课程：zookeeper,dubbo,docker 容器，hadoop, Redis, memcached ,mysql;

提醒大家: 时刻不要忘记，提升自己的内功才是最关键的【编程能力，解决问题的思路】，内功练好，很多知识学起来触类旁通；

it 行业，永远需要学习，终生需要学习，永无止境；

六：卷 2 的计划

卷 2 的目的：老师亲自给大家示范 卷 1 中的服务器项目到底怎样在实际的工作中应用 cocos2d-x 作为客户端，卷 1 里的项目作为服务器端

《冒险之路》,steam 上:

七：再见

12 个 .h 头文件， 20 个.cxx 源文件

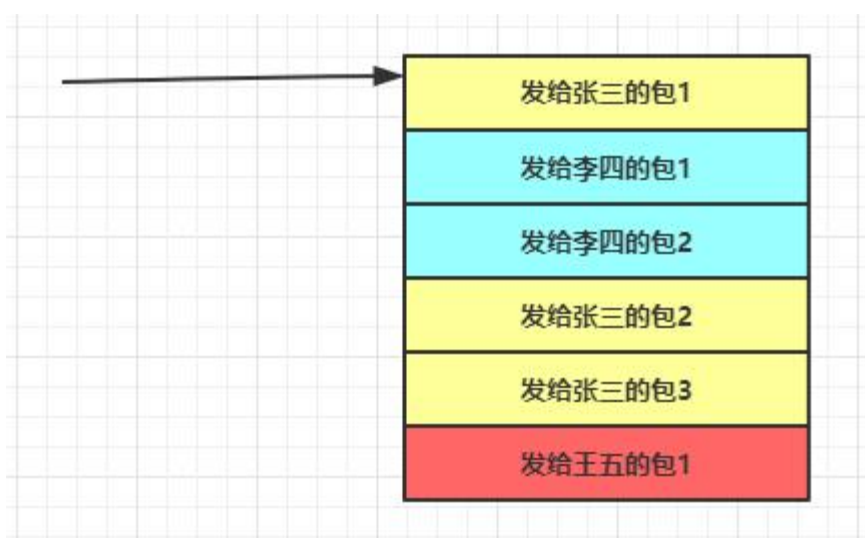
2200 行代码， 额外还有 1400 行注释； 代码行并不多， 注释量很大

老师期待同学学完这门课程靠这门课程拿到 30K， 请第一时间告诉老师这个好消息；

9.1 消息的跳跃发送

一： `ServerSendQueueThread()`:处理发送消息队列的线程

如果数据一次发送不完， 后续要通过 `ngx_write_request_handler()`函数来完成其余数据的发送；



包 1， 包 3， 包 2

解决：

a)不解决；

b)客户端； 一问一答；

实际业务上， 也根本没有控制哪个数据包先发给客户端的必要性

方案：

a)往每个连接中增加标记； 挺浪费性能；

b).....

最终方案：

无需处理